M T S

The Michigan Terminal System

Pascal in MTS

Volume 20

January 1989

University of Michigan Computing Center
Ann Arbor, Michigan

DISCLAIMER

The MTS manuals are intended to represent the current state of the Michigan Terminal System (MTS), but because the system is constantly being developed, extended, and refined, sections of this volume will become obsolete. The user should refer to the *U-M Computing News*, Computing Center Memos, and future Updates to this volume for the latest information about changes to MTS.

CONTENTS

PREFACE

The software developed by the Computing Center staff for the operation of the high-speed IBM 370-compatible computers can be described as a multiprogramming supervisor that handles a number of resident, reentrant programs. Among them is a large subsystem, called MTS (Michigan Terminal System), for command interpretation, execution control, file management, and accounting maintenance. Most users interact with the computer's resources through MTS.

The MTS Volumes are a series of manuals that describe in detail the facilities provided by the Michigan Terminal System. Administrative policies of the Computing Center and the physical facilities provided are described in other publications.

The MTS Volumes now in print are listed below. The date indicates the most recent edition of each volume; however, since volumes are periodically updated, users should check the file *CCPUBLICATIONS, or watch for announcements in the *U-M Computing News*, to ensure that their MTS volumes are fully up to date.

Volume 1	*The Michigan Terminal System*, November 1988
Volume 2	*Public File Descriptions*, January 1987
Volume 3	*System Subroutine Descriptions*, April 1981
Volume 4	*Terminals and Networks in MTS*, July 1988
Volume 5	*System Services*, May 1983
Volume 6	*FORTRAN in MTS*, October 1983
Volume 7	*PL/I in MTS*, September 1982
Volume 8	*LISP and SLIP in MTS*, June 1976
Volume 9	*SNOBOL4 in MTS*, September 1975
Volume 10	*BASIC in MTS*, December 1980
Volume 11	*Plot Description System*, August 1978
Volume 12	*PIL/2 in MTS*, December 1974
Volume 13	*The Symbolic Debugging System*, September 1985
Volume 14	*360/370 Assemblers in MTS*, May 1983
Volume 15	*FORMAT and TEXT360*, April 1977
Volume 16	*ALGOL W in MTS*, September 1980
Volume 17	*Integrated Graphics System*, December 1980
Volume 18	*The MTS File Editor*, February 1988
Volume 19	*Tapes and Floppy Disks*, November 1986
Volume 20	*Pascal in MTS*, January 1989
Volume 21	*MTS Command Extensions and Macros*, April 1986
Volume 22	*Utilisp in MTS*, May 1988
Volume 23	*Messaging and Conferencing in MTS*, August 1988

The numerical order of the volumes does not necessarily reflect the chronological order of their appearance; however, in general, the higher the number, the more specialized the volume. Volume 1, for example, introduces the user to MTS and describes in general the MTS operating system, while Volume 10 deals exclusively with BASIC.

The attempt to make each volume complete in itself and reasonably independent of others in the series naturally results in a certain amount of repetition. Public file descriptions, for example, may appear in more than one volume. However, this arrangement permits the user to buy only those volumes that serve his or her immediate needs.

Richard A. Salisbury
General Editor

PREFACE TO VOLUME 20

MTS Volume 20, *Pascal in MTS*, combines what was formerly available as separate documentation for Pascal/VS in MTS and portions of the *Pascal/JB User's Guide* along with the other Pascal related memos.

This volume supercedes Computing Center Memo 436, *Pascal/VS in MTS*, which was written by Douglas Orr. The revisions to Memo 436 were done by Chitra Ramanujan of the University of Michigan Computing Center staff and are included in this volume.

Portions of this volume were reproduced, with permission, from the IBM publication, *Pascal/VS Programmer's Guide*, SH20-6162.

Portions of this manual were reproduced, with permission, from *Pascal/JB User's Guide*, Version 1.0 (March 1985). The User's Guide is a publication of Plug Compatible Software, Inc.

The January 1989 revision of Volume 20 describes the implementation of version 2.0 of Pascal/JB on MTS and the differences between that version and the previous version 1.0.

INTRODUCTION

There are two different Pascal language compilers available on MTS. Both of these support a Pascal language, which has several extensions over the standard Pascal programming language developed by Nicholas Wirth in the late 1960's. These are *PASCALJB, a product of Plug Compatible Software, Inc., and *PASCALVS, which is a product of IBM. This MTS Volume is intended for those users who wish to use these extended Pascal languages in MTS. It is assumed that the reader is already familiar with the standard Pascal language.

This volume describes features of Pascal/VS and Pascal/JB as implemented on the Michigan Terminal System. The aspects of Pascal mentioned here have been customized for, or are specific to MTS. That means that this volume alone will not describe all available features of the language. For the standard features of the Pascal language, please refer to an introductory text book, for example, *Introduction to Computer Science: Using Pascal* by E. Glinert, or *Oh! Pascal!* by Doug Cooper and Michael Clancy.

For a detailed account of all the Pascal/VS language constructs refer to the IBM Publication, *Pascal/VS Language Reference Manual*, SH20-6168. Reference copies can be checked out from the output windows at NUBS and UNYN for use in the building.

Since the language accepted by the two compilers is essentially the same, programs written for use with one compiler will compile without change (or with minimal changes) under the other. However, Pascal/JB offers significant advantages over Pascal/VS. These advantages include greatly increased compilation speeds (typically 3-4 times faster), and better error detection, such as detection of references to uninitialized variables.

While the increased compilation speeds will result in reduced charges for all users, for the commercial user this reduction may be somewhat less significant since the terms of the *PASCALJB license require that external users be surcharged based on their CPU costs. This program product charge will be displayed in user's signoff statistics. The file *SOFTWARECHARGES contains current information on the exact amount of the surcharge.

The format of this volume is as follows. The sections that are to be used with Pascal/JB alone are presented first, followed by sections to be used with Pascal/VS and these two are followed by sections that are common to both the compilers. This format was chosen to make it easier on the reader to choose the sections that are relevant and skip the rest.

Certain syntactic primitives are sometimes surrounded by "<" and ">" for clarity. For instance, <compilation parameter> would represent a single syntactic item consisting of a compilation parameter.

SOURCE CODE READABILITY: PASCALTIDY

To enhance readability of source code files thus making it easier to amend or debug them, a "tidy" program exists for Pascal. This program makes a neat-looking, properly-indented copy of a syntactically correct program. The use of PascalTidy, is described in a later section of this document.

THE IBM VS PASCAL COMPILER

IBM recently replaced its Pascal/VS compiler with a new VS Pascal compiler. The VS Pascal language has some extensions and modifications to the previous Pascal/VS language. The language differences are minor but some incompatibilities exist between the two compilers. IBM has dropped support for the Pascal/VS compiler. The Computing Center is supporting only the old Pascal/VS compiler in the file *PASCALVS. Currently, there are no plans to support the new VS Pascal compiler.

Version 2.0 of Pascal/JB maintains compatibility with VS Pascal. Consequently the same incompatibilities are present between Pascal/JB 2.0 and Pascal/VS, and also between Pascal/JB versions 2.0 and 1.0. These are minor and are dealt with later in a separate section of this volume.

The IBM Language Reference Manual for VS Pascal is SC26-4320-0. This is the definitive document for the language used by version 2.0 of the Pascal/JB compiler offered on MTS. Copies are available for purchase at local book stores and reference copies can be checked out from the output windows at NUBS and UNYN for use in the building.

PASCAL/JB

COMPILING AND RUNNING PASCAL/JB

The general form of the command to invoke the Pascal/JB compiler is:

$RUN *PASCALJB [I/O assignments] [limits] [PAR=options]

where "I/O assignments" and "limits" are assignments of the MTS logical I/O units to specific files or devices, time and page limits, and other $RUN parameters (see MTS Volume 1, *The Michigan Terminal System*, for a full description of the $RUN parameters), and "options" is a list of Pascal/JB compilation or execution parameters. Some sample $RUN commands are:

```
$RUN *PASCALJB SCARDS=myprog SPRINT=list.output T=3
$RUN *PASCALJB SCARDS=prog SPUNCH=obj T=3 1=lib PAR=DECK
$RUN *PASCALJB SCARDS=wabc:herprog SPRINT=*PRINT* T=2
```

LOGICAL I/O UNIT ASSIGNMENTS

The logical I/O assignments for Pascal/JB are as follows:

SCARDS=<source stream>
SPRINT=<compiler listing and program output>
SERCOM=<compiler error and status messages>
SPUNCH=<object program> (if desired)
0=<library to merge object module with>
1=<include library 1>
.
.
.
8=<include library 8>

Typically, programs being translated in Compile-Load-and-Go mode require only the assignment of SCARDS to a file or device containing the source stream. In batch mode, when the source stream is on cards immediately following the $RUN card, even this assignment is unnecessary. Typical source streams consist of the program source followed by a /EXECUTE control statement followed by the data to be read by the compiled program. (See the section, "Components of the Source Stream," for a more detailed explanation.)

Typical programs being translated in DECK mode require only the assignment of SCARDS to the file or device containing the source program and the assignment of SPUNCH to the file that is to contain the object deck. If SPUNCH or logical I/O unit 0 is assigned, the compiler assumes that an object deck is desired. (See the section, "Compilation Parameters," for a complete description of the DECK parameter.)

Error messages for both compilation and execution errors are written to SERCOM, which defaults to the terminal.

It should also be noted that any of the logical I/O units may be accessed during the *execution* of the compiled program via the FILE or UNIT open options. See the sections, "I/O in Pascal/JB" and "Open Options", for a full description of this.

COMPONENTS OF THE SOURCE STREAM

When SPUNCH is not assigned, Pascal/JB defaults to "Compile-Load-and-Go" mode (See the DECK parameter in the section, "Compilation Parameters," to override this feature.) Compile-Load-and-Go (CLG) means that Pascal/JB will compile a program and (if there were no compilation errors) execute the program. In order to do this, Pascal/JB must have access to both the source for the program to be compiled and the data (if any) that the program reads during execution. The combination of the source program, a separator, and the program data is known as the source stream. Pascal/JB uses the MTS standard control statements for language processors to separate the source program from the data in CLG mode. In DECK mode (where an object deck is produced) no control statements are necessary. Most programs compiled in CLG mode need only the /COMPILE and /EXECUTE control statements as separators in the source stream. Other control statements may be used, for example, to control the format of the compilation listing. This section will only describe those aspects of the /COMPILE and /EXECUTE control statements needed to compile and execute a typical Pascal/JB program in CLG mode. See the section, "Control Statements," for a full description of control statements and their parameters.

The /COMPILE control statement designates the beginning of the source program to be compiled by Pascal/JB. Typically, the first line in a source stream is a /COMPILE statement. If the first line of the source stream is not a control statement of any kind, Pascal/JB assumes that it is the first line of a source program.

The /EXECUTE control statement designates the end of the source program and the beginning of the data to be read during the execution of the source program. By default, this data is read when the standard file variable INPUT is specified in an I/O operation such as READ or READLN. Given this default, EOF(INPUT) becomes true when another /COMPILE or /EXECUTE card is encountered in the source stream or when an actual end-of-file (or its equivalent such as $ENDFILE) is encountered. The /EXECUTE control statement is unnecessary if the program does not read from the standard file variable INPUT or if INPUT is explicitly reassigned using the RESET predefined procedure. An example, source stream is:

```
/COMPILE
program Testing;
  var
    I:Integer;
begin
  while not Eof do      {defaults to Eof(Input)}
    begin
      Read(i);          {reads from Input by default}
      Writeln('The number is:',I); {writes to Output by default}
    end
end.
/EXECUTE
  1
  2
  3
<end of file>
```

Most source streams, like the example above, will consist of only a single program and a single data set for that program. However, Pascal/JB allows multiple programs and/or multiple data sets to be specified in a single source stream. That is, the source stream may consist of any number of programs, each initiated by a /COMPILE control statement. Each source program may be followed by one or more /EXECUTE control statements and the data for each execution of that source program. Each source program is compiled only once. However, the program is then executed once for each

/EXECUTE statement encountered before the next source program or end-of-file in the source stream. This feature is particularly useful when testing programs using multiple sets of test data. The following example illustrates this feature:

```
/COMPILE
  <program source for program #1>
/EXECUTE
  <data for the first execution of program #1>
/EXECUTE
  <data for the second execution of program #1>
/COMPILE
  <program source for program #2>
/EXECUTE
  <data for the first (and only) execution of program #2>
<end-of-file>
```

EXECUTING OBJECT MODULES

An alternative to Compile-Load-and-Go mode is DECK mode. When DECK mode is used, an object module is produced on the MTS file or device attached to SPUNCH. Alternatively, the resulting object deck may be merged with an existing object library by assigning logical I/O unit 0 to the file containing that library. In DECK mode the resulting object module is *not* automatically executed at the completion of the compilation. In order to execute the just-compiled program, the user must enter an MTS command to invoke the program. The general form of the command to invoke a program compiled using Pascal/JB is:

$RUN object [I/O assignments] [limits]

where "I/O assignments" and "limits" are assignments of the MTS logical I/O units to specific files or devices, time and page limits, and other $RUN parameters (see MTS Volume 1 for a full description of the $RUN parameters). By default, the predefined file variables INPUT and OUTPUT are attached to the logical I/O units SCARDS and SPRINT, respectively. This may be overriden using the FILE or UNIT open options (see the section, "I/O in Pascal/JB," for details on these options). The FILE or UNIT open options may also be used to attach user-defined file variables to specific files or logical I/O units. Some sample $RUN commands are:

```
$RUN -obj  SCARDS=mydata SPRINT=myoutput T=3
$RUN obj1+obj2 T=3 PAR=HEAP=20P
$RUN -obj SCARDS=wabc:herdata SPRINT=*PRINT* T=2
$RUN -obj SCARDS=datafile SPRINT=outputfile T=2 PAR=parstring
```

In the last $RUN command, the "parstring" value is the value returned by the predefined function PARMS during execution.

PASCAL/JB CONTROL STATEMENTS

The /COMPILE and /EXECUTE statements discussed in the previous section are examples of *control statements*. Control statements are input to Pascal/JB that control the behaviour of the compiler. There are three classes of control statements available: those that delineate the various parts of the source stream, those that control the format of the compiler listing, and those that selectively activate or deactivate various compiler options. The latter two types are intended as an alternative to the Pascal/VS "%" facility. Pascal/JB accepts "%" statements, but the corresponding control statements are preferred.

All control statements begin with a "/" which *must* appear at the beginning of the line with no leading blanks. Blanks are *not* allowed between the "/" and the control statement name (e.g., "/ EXECUTE" is not allowed). Lines beginning with a "/" that are not control statements (that is, the line does not match any of the control statements or their abbreviations) are treated as ordinary program source or data as appropriate. Unlike Pascal/JB source statements, control statements may not be continued on subsequent lines. Furthermore, the entire line containing the control statement is considered part of the control statement. Control statements are not terminated or separated by semicolons.

Control statements may be written in upper- or lowercase or any combination thereof. The underlined portion for the statement in the following lists represent the minimum acceptable abbreviation.

CONTROL STATEMENTS THAT DELINEATE THE SOURCE STREAM

The following control statements delineate the various components of the source stream. These control statements are only necessary if the source stream requires delineation. That is, if a single program is being compiled in DECK mode, the source stream need only consist of the program source; no control statements are required. This class of control statements is most useful when the compiler is invoked in Compile-Load-and-Go mode or when multiple modules are being compiled with a single invocation of the compiler.

/COMPILE <compilation parameters>

The /COMPILE control statement marks the beginning of a source program to be compiled. Any compilation parameter may appear on the /COMPILE statement. These parameters are in effect only for the compilation of the source program immediately following the /COMPILE control statement and are overridden if they conflict with any parameters specified in the PAR field of the $RUN command.

Example: `/COMPILE NOSOURCE,CHECK`

/DEBUG <execution parameters>

The /DEBUG control statement marks the beginning of the data set to be read by the most recently compiled program in the source stream. It also instructs the compiler to enter the interactive debugger prior to the start of execution of the program. (See the section, "Pascal/JB Interactive Debugging System," for a complete description of the interactive debugger.) Except for this feature, the /DEBUG control statement is identical to the /EXECUTE control statement.

Example: `/DEBUG HEAP=4P`

/EXECUTE <execution parameters>

The /EXECUTE control statement marks the beginning of a data set to be read by the most recently compiled program in the source stream. /EXECUTE is invalid if it is encountered before the first source program in the source stream. The /EXECUTE control statement is ignored in DECK mode as is any data located between the /EXECUTE card and the next /COMPILE statement. Any execution parameter may appear on the /EXECUTE statement. These parameters are in effect only for the execution that uses the data set immediately following the /EXECUTE control statement and are overridden if they conflict with any parameters specified in the PAR field of the $RUN command.

Example: `/EXECUTE HEAP=4P`

/STOP

The /STOP control statement terminates the source stream. Pascal/JB will never read beyond a /STOP control statement.

CONTROL STATEMENTS THAT CONTROL THE SOURCE LISTING

The following control statements affect the source program listing produced by Pascal/JB. If no source listing is being produced (either because the NOSOURCE compilation parameter was set or defaulted), these control statements have no effect. Pascal/VS uses the "%" facility to accomplish the same tasks as these control statements. These control statements are preferred over the "%" statements (which are accepted by Pascal/JB only to maintain compatibility with Pascal/VS). See the IBM *Pascal / VS Language Reference Manual* for a description of the "%" facility.

/CPAGE <numlines>

The /CPAGE control statement forces the next line of the source listing to begin on a new page if there are fewer than <numlines> lines remaining on the current page. Otherwise, it has no effect. The CPAGE control statement is useful for ensuring that a particular section of the source program following the /CPAGE statement will appear on the same page of the compilation listing.

Example: `/CPAGE 6`

/PAGE

The /PAGE control statement forces the next line of the compilation listing to begin on a new page.

/PRINT {ON | OFF}

The /PRINT control statement activates or deactivates the compilation listing. /PRINT OFF will prevent source lines from appearing in the compilation listing until a /PRINT ON control statement is encountered. /PRINT ON will *not* activate the source listing if the NOSOURCE option was specified or defaulted.

Example: `/PRINT OFF`

/SKIP <numlines>

The /SKIP control statement inserts <numlines> of blank lines into the compilation listing. If fewer than <numlines> remain on the current page of the compilation listing, then /SKIP is equivalent to /PAGE. If <numlines> is not specified, it defaults to one.

Example: `/SKIP 10`

/SPACE <numlines>

The /SPACE control statement is a synonym for the /SKIP control command.

/TITLE <title>

The /TITLE control statement controls the title which appears in the header of every page of the compilation listing. It also forces the next line of the listing to begin on a new page. The <title> string may be up to 93 characters long.

OTHER CONTROL STATEMENTS

The following control statements request miscellaneous services from Pascal/JB. As with the preceding group of control statements, there are corresponding "%" statements for each control statement in this group.

/CHECK {PUSH | POP | ([<check parameter>] {ON | OFF} }

The /CHECK control parameter selectively activates or deactivates the production of object code to perform certain consistency checks. Deactivating checks decreases somewhat the resources (virtual memory and CPU time) needed by the program. However, it also could cause a program execution error to go undetected. If this occurs, the results of the program will be erroneous, unpredictable, and potentially disastrous. It is therefore recommended that this feature be used *only* after a program is thoroughly debugged and even then only around extremely time-critical portions of the program or when the programmer is sure that such checks are unnecessary. It should be noted that Pascal/JB will *not* generate object code to perform these checks whenever it can determine during compilation that such checks are unnecessary.

If PUSH is specified, the current value of each individual check listed below is saved. A subsequent POP restores each individual check to its value when the corresponding PUSH was encountered. /CHECK PUSH and /CHECK POP are generally most useful in INCLUDE sections when specific checking (or lack of checking) is desired within the INCLUDE section. By bracketing the INCLUDE section with /CHECK PUSH and /CHECK POP, the programmer can freely reset the values of individual checks without affecting the status of those checks in the portions of the module that follow the INCLUDE.

The <check parameter> may be any of the individual checks below followed by ON or OFF. If the check parameter is omitted, *all* of the following checks are either activated or deactivated (depending on whether ON or OFF was specified). The following check parameters are allowed on the /CHECK control statement:

POINTER	checks each use of the dereference operator "@" to ensure that the pointer being dereferenced is not nil.
SUBSCRIPT	checks the value of each subscript used to ensure that it is within the range specified in the array declaration. SUBSCRIPT checking implies SUBRANGE checking.
FUNCTION	ensures that a return value has been assigned to a function prior to the return from that function.
SUBRANGE	checks that a value being assigned is within the range declared for the target of the assignment. Also ensures that the PRED and SUCC functions do not produce a value that is not of the parent type of the PRED or SUCC operand. SUBRANGE checking implies SUBSCRIPT checking.
CASE	checks that the case selector value has a corresponding CASE label or an OTHERWISE clause.
TRUNCATE	checks that the target string in a string assignment is long enough to contain the entire source string.
INIT	checks that a value has been assigned to an operand prior to any reference to that operand. See the section, "Uninitialized Variable Checking," for a full description of this feature.

Example: `/CHECK SUBSCRIPT ON`

/INCLUDE {<filename> | <membername>}

The /INCLUDE control statement inserts statements into the source stream. These statements may either come from a file (if the /INCLUDE parameter is <filename>) or from a member of an INCLUDE library attached to logical I/O units 1 to 8. See the section, "Use of %Include Directive in MTS," for a detailed explanation of the INCLUDE library format. The INCLUDE libraries are searched in the order 1 to 8. If the parameter is not a member name of any of the include libraries, then (and only then) it is assumed to be a filename.

PASCAL/JB OPTIONS

Compilation options modify the behaviour of the Pascal/JB compiler. Execution options alter the behaviour of the run-time system that is available to programs compiled using Pascal/JB. Compilation options may appear either in the PAR field of the $RUN command or on the /COMPILE statement. Similarly, execution options may appear either in the PAR field or on the /EXECUTE statement. Any conflicts between options appearing in the PAR field of the $RUN command or on the /COMPILE or /EXECUTE statements are resolved in favor of the PAR optiions. Also note that execution options have no effect if the compiler is not being run in Compile-Load-and-Go (CLG) mode since the program is not executed in that case.

COMPILATION OPTIONS

The following are the compilation options accepted by Pascal/JB. The underlined portions represent the minimum acceptable abbreviations.

ARRAYINIT / NOARRAYINIT

Default: ARRAYINIT

This option activates or deactivates uninitialized variable checking for variables that are elements of arrays.

CHECK / NOCHECK

Default: CHECK

This option controls whether the compiler generates code to perform certain run-time consistency checks. For example, if CHECK is specified (or defaulted), the compiler will generate code to ensure that every variable is assigned a value before it is referenced in an expression. If an uninitialized variable is referenced during the execution of the compiled program, Pascal/JB will generate an execution error message followed by a symbolic dump of all active routines and variables. Other checks ensure that the values assigned to variables are within the allowable range for that variable and that pointer variables do not reference disposed records. See the description of the /CHECK control statement in the section, “Other Control Statements,” for a full description of the checks available and how to selectively (de)activate them.

CLG / DECK

Default: CLG

This option determines if the compiler is in Compile-Load-and-Go mode or DECK mode. In Compile-Load-and-Go mode, the compiler compiles the source program and, if there were no errors, executes the compiled program. In DECK mode, an object module is produced on logical I/O unit SPUNCH. NODECK is a synonym for CLG and NOCLG is a synonym for DECK. If either SPUNCH or logical I/O unit 0 is explicitly assigned on the $RUN command, the default is DECK; otherwise, the default is CLG. If SPUNCH is assigned, the object module is written on SPUNCH. If logical I/O unit 0 is assigned (but not SPUNCH), then it is assumed to be assigned

to a file containing an object library. The object module produced by Pascal/JB is added to or replaced in this object library. See the description of *OBJUTIL in MTS Volume 5, *System Services*, for a description of object libraries.

CPAGES=n

Default: Unlimited

This option limits the number of pages of program listing that will be produced during the compilation of the module. If this limit is exceeded, an error message is generated and compilation is terminated.

CROSSCHECK / NOCROSSCHECK

Default: NOCROSSCHECK

This option causes the Pascal/JB run-time support to check the declarations of variables and procedures across module boundaries. This extra type-checking is performed before program execution is initiated. An error message is generated and execution is suppressed if incompatible declarations are encountered. Intramodule declaration checking is performed automatically by the compiler at the time that the module was compiled.

CTIME=n

Default: Unlimited

This option limits the amount of time (in CPU seconds) available for compiling the module. If this limit is exceeded, an error message is generated and compilation is terminated.

DEBUG / NODEBUG

Default: DEBUG

This option controls whether or not the information required to produce a symbolic post-mortem dump or to invoke the interactive debugger is to be included as part of the object module produced by the compiler. If NODEBUG is specified and an error occurs during the execution of the program, the symbolic post-mortem dump will be suppressed. Specifying NODEBUG may reduce the size of the resulting object module significantly.

DISPOSECHECK / NODISPOSECHECK

Default: DISPOSECHECK

This option controls whether checks are generated that ensure that references to records via pointer variables do not refer to records that have been released through a call to DISPOSE or RELEASE. If DISPOSECHECK is specified (or defaulted), an execution error will occur if a pointer is used to reference a disposed or released record.

DPAGES=n

Default: Unlimited

This option limits the number of pages of output that may be produced during a post-mortem dump (which may occur as the result of an error detected during the execution of the program). If this limit is exceeded, an error message is generated and the post-mortem dump is terminated. This option is ignored in DECK mode.

INITCHECK / NOINITCHECK

Default: INITCHECK

This option activates or deactivates uninitialized variable checking. See the section, "Uninitialized Variable Checking," for a full description of this feature. INITCHECK implies ARRAYINIT and RECORDINIT. NOINITCHECK implies NOARRAYINIT and NORECORDINIT.

LANGLVL={STANDARD | STDRES | PVS | EXTENDED}

Default: EXTENDED

This option controls the language features available. If LANGLVL is set to STANDARD or STDRES, those language features incompatible with the Pascal standard are flagged with a warning message. Additionally, if LANGLVL is set at STDRES nonstandard reserved words are not recognized. This allows the nonstandard reserved words to be declared as identifiers within the program. If LANGLVL is set to PVS, those language features not available in the IBM Pascal/VS compiler are flagged with a warning message. If LANGLVL is set to EXTENDED, all available language extensions are supported.

LIBRARY=FDname

Default: None

This option specifies the name of a file or device which contains the object modules for routine(s) referenced by the program being compiled. In Compile-Load-and-Go mode, these routines are loaded prior to the execution of the program. This option is ignored in DECK mode.

LINECNT / LINECOUNT=n

Default: 60

This option sets the number of lines per page of the compilation listing.

MARGINS=(n,m)

Default: (1,255)

This option specifies the margins for the source statements for the program. Any characters not within the margins specified are ignored. The margins must be in the range of 1 to 255 and the left margin must be less than the right margin (n<m).

OPTIMIZE / NOOPTIMIZE

Default: NOOPTIMIZE

This option causes the resulting object code to be post-processed by the compiler in order to produce a more efficient and/or more compact program. The OPTIMIZE option implies NODEBUG and may increase slightly the cost of compilation.

PAGEWIDTH

Default: 132

This option specifies the width of the page for the compilation listing. The width must be in the range of 79 to 132.

RECORDINIT / NORECORDINIT

Default: RECORDINIT

This option activates or deactivates uninitialized variable checking for variables that are components of records.

SHORTRECORDS / NOSHORTRECORDS

Default: SHORTRECORDS

This option controls the effect of calls to the predefined procedures NEW and DISPOSE where explicit tag field values are specified. If SHORTRECORDS is specified (or defaulted), Pascal/JB will generate object code that allocates a record that is just long enough to contain the variant part selected by the tag field value specified in the call to NEW or DISPOSE. If NOSHORTRECORDS is specified, the compiler will generate object code that allocates records long enough to hold the maximum variant part regardless of the values specified for the tag fields. That is, if NOSHORTRECORDS is specified, the tag field values specified on calls to NEW or DISPOSE are ignored, and NEW and DISPOSE behave as if the tag fields had not been specified. This option is particularly useful for isolating errors in the use of records with variant parts. The NOSHORTRECORDS option corresponds to the way Pascal/VS handles NEW and DISPOSE.

SIZE=n

Default: 4P

This option sets the initial size of the compiler working storage. The compiler will automatically allocate more space when it is required so this option is always optional. For very large programs, it may be desirable to set the size to a value somewhat larger than the default in order to reduce the number of space acquisitions required by the compiler. This may result in a slight reduction in the overall cost of the compilation.

SOURCE / NOSOURCE

Default: See text below

This option controls whether a compilation source listing is to be produced on the logical I/O unit SPRINT. By default, the listing is always produced in batch mode. In conversational mode (that is, from a terminal) a listing is produced only if SPRINT has been explicitly assigned.

STATS / NOSTATS

Default: STATS

This option controls the printing of compilation statistics at the end of each compilation.

WARNING / NOWARNING

Default: WARNING

This option controls whether or not the compiler produces warning messages. It is strongly suggested that this option be *enabled* since warnings are usually errors (they are classified as warnings only because they are not *always* errors).

XPAGES=n

Default: Unlimited

This option limits the number of pages of TEXT output that may be produced during the execution of the program. If this limit is exceeded, an error message is generated and execution is terminated or control is returned to the interactive debugger as appropriate. This option is ignored in DECK mode.

XREF[={LONG | SHORT}] / NOXREF

Default: NOXREF

This option controls whether or not a cross-reference listing is produced after the compilation listing. If no compilation listing is produced, this option has no effect and no cross-reference listing is generated. The cross-reference listing contains an alphabetical list of each identifier in the module showing the attributes of the identifier and the line number of each reference to the identifier. If XREF=SHORT is specified, those identifiers that were declared but never referenced are omitted from the cross-reference listing. XREF is equivalent to XREF=LONG.

XSTACKLIMIT=n

Default: Unlimited

This option limits the amount of stack space (in bytes) that may be used during the execution of the program. If this limit is exceeded, an error message is generated and execution is terminated or control is returned to the interactive debugger as appropriate. The scale factors K or P may be used to specify kilobytes (1024 bytes) or pages (4096 bytes), respectively (e.g., XSTACKLIMIT=40P). This option is ignored in DECK mode.

XTIME=n

Default: Unlimited

This option limits the amount of time (in CPU seconds) that may be used during the execution of the program. If this limit is exceeded, an error message is generated and execution is terminated or control is returned to the interactive debugger as appropriate. This option is ignored in DECK mode.

EXECUTION OPTIONS

Execution options control the environment provided by the Pascal/JB system to the execution of a program compiled using Pascal/JB. In Compile-Load-and-Go mode, these options may appear on the /EXECUTE or /DEBUG control statement.

In DECK mode these options may be specified in the PAR field of the $RUN command when the object program is invoked. When this is done, the list of execution options must be terminated by a slash "/". The slash is used to separate the Pascal/JB execution options from the execution parameter string returned by the PARM predefined function. (See the IBM *Pascal/VS Language Reference Manual* for a complete description of the PARM predefined function.)

CROSSCHECK

Default: NOCROSSCHECK

This option causes the Pascal/JB run-time support to check the declarations of variables and procedures across module boundaries. This extra type-checking is performed before program execution is initiated. An error message is generated and execution is suppressed if incompatible declarations are encountered. Intra-module declaration checking is performed automatically by the compiler at the time that the module was compiled.

DPAGES=n

Default: Unlimited

This option limits the number of pages of output that may be produced during a post-mortem dump of a program. If this limit is exceeded, an error message is generated and the post-mortem dump is terminated.

DEBUG/NODEBUG

Default: NODEBUG

This option instructs the Pascal/JB run-time support system to invoke the Pascal/JB interactive debugger prior to the start of execution of the program. See the section, "Pascal/JB Interactive Debugging System," for a complete description of the interactive debugger.

HEAP=n

Default: 1P

This option controls the initial size of the heap in bytes. The heap is used for records allocated by calls to NEW and by the compiler internally to store certain string temporaries. Pascal/JB will automatically increase the size of the heap whenever necessary. For programs that require large heaps (e.g., programs that build extensive lists of dynamic records), it may be desirable to set the initial heap size to a value somewhat larger than the default in order to reduce the number of space acquisitions required. Unused heap space is released whenever the predefined procedure RELEASE is called. See the IBM *Pascal/VS Language Reference Manual* for a description of RELEASE and its companion procedure, MARK. The K or P scale factors may be used to specify kilobytes (1024 bytes) or pages (4096 bytes), respectively (e.g., HEAP=10P).

PAGES=n

Default: Unlimited

This option limits the number of pages of TEXT output that may be produced during the execution of the program. If this limit is exceeded, an error message is generated and execution is terminated or control is returned to the interactive debugger as appropriate.

STACK=n

Default: See text below

This option controls the initial size of the execution stack in bytes. The default size is set to a value at least 1/2 page (2048 bytes) larger than the storage required by the main body of the program (not including the storage required by any procedures that it may call). This option is always optional since Pascal/JB automatically extends the stack whenever necessary. For programs which require a large stack (e.g., programs with procedures that use large arrays locally), it may be desirable to set the initial stack size to a value somewhat larger than the default in order to reduce the number of space acquisitions required. The scale factors K or P may be used to specify kilobytes (1024 bytes) or pages (4096 bytes), respectively (e.g., STACK=10P).

STACKLIMIT=n

Default: Unlimited

This option limits the amount of stack space (in bytes) that may be used during the execution of the program. If this limit is exceeded, an error message is generated and execution is terminated or control is returned to the interactive debugger as appropriate. The scale factors K or P may be used to specify kilobytes (1024 bytes) or pages (4096 bytes), respectively (e.g., STACKLIMIT=40P).

TIME=n

Default: Unlimited

This option limits the amount of time (in CPU seconds) that may be used during the execution of the program. If this limit is exceeded, an error message is generated and execution is terminated or control is returned to the interactive debugger as appropriate.

Examples:

```
$RUN obj+*IG PAR=HEAP=10P/
$RUN -obj PAR=DEBUG/
$RUN obj1+obj2 SCARDS=dataset2 PAR=HEAP=2P,DEBUG/Data set 2
```

UNINITIALIZED VARIABLE CHECKING

Pascal/JB will optionally produce object modules that check for references to variables that have not yet been initialized. This feature is *not* available in Pascal/VS (see disclaimer). The Pascal/JB implementation of this feature does not rely on storage being initialized to a "magic number" or in any other manner that "almost always" works. Rather, the technique used by Pascal/JB ensures that all references to uninitialized variables are accurately detected. Uninitialized variable checks are performed on all variable types including enumerations, reals, array elements, and components of records.

Explicit or implicit references to the individual elements of arrays will result in an execution error if any of those elements has not been initialized. For example, if a PACKED ARRAY [1..n] OF CHAR variable is used in a WRITE or WRITELN statement or is converted to a STRING type, then *each* element of the PACKED ARRAY [1..n] of CHAR must have been initialized or an execution error will result.

Assignment of entire array or record structures (using a single assignment statement) is an exception to this. In this particular case, the uninitialized variable information is copied from the source to the target. For example, if A and B are compatible array types and B is assigned to A, then those elements of A which correspond to an element of B that is uninitialized will be considered uninitialized (even if the element of A had been initialized prior to the assignment of B to A).

PASCAL/JB LISTING INFORMATION

The source listing contains information about the source program and information about nesting of blocks and statement numbers.

PAGE HEADERS

The first line of every page contains the page number. The first line also contains the word PROGRAM or SEGMENT followed by the <program name> or <segment name> and the title, if a title exists. The title is set with the /TITLE statement and may be reset whenever necessary. If no title has been specified, the title field will be blank.

The second line contains the following column headers: MTS Line#, Stmt#, Lev, and Pascal/JB Version x, where the "x" in "version x" will vary depending on the current version. This is followed by the time and date of compilation. The MTS Line# refers to the line number in the source file and the statement number refers to the number that Pascal/JB has assigned to that statement. The Lev refers to the nesting level of the Begin block nesting.

STATEMENT NUMBERING

Pascal/JB numbers the statements of a routine. These numbers are referenced when a run-time error occurs and when break points are specified in the interactive debugger.

All non-empty statements are numbered except the "repeat" statement. However, the "until" portion of a "repeat" statement is numbered. A "begin" or "end" statement is not numbered because it serves only as a bracket for a sequence of statements and has no executable code associated with it.

PASCAL/JB INTERACTIVE DEBUGGING SYSTEM

The Pascal/JB interactive debugging system is designed to aid in the detection and isolation of bugs in Pascal/JB source programs. The interactive debugger allows the user to set breakpoints anywhere in the program, to display the contents of variables (including arrays and dynamic records), and to execute programs step by step. Effective use of the interactive debugger can be an invaluable aid in the development of Pascal programs.

The interactive debugger can be invoked in either Compile-Load-and-Go mode or in DECK mode. In CLG mode, the debugger is invoked when a /DEBUG control statment is encountered. The following is a typical source stream that would invoke the interactive debugger:

```
/COMPILE
        {Pascal/JB source statements}
/DEBUG
        {program data}
/STOP
```

In DECK mode, the interactive debugger may be invoked by specifying the DEBUG option when the resulting object module is executed. For example:

```
$RUN -OBJ PAR=DEBUG/
```

Once the interactive debugger is invoked, the user is prompted for debug commands. If, during the subsequent execution of the program, an execution error is encountered, the interactive debugger is automatically reinvoked. At that point, the user may enter any debug command, except those that do not make sense following an execution error. For instance, the current value of any identifier may be displayed or the value of all identifiers in one or more blocks may be dumped. The interactive debugger commands are described fully below.

STATEMENT SPECIFICATIONS

The BREAK and CLEAR debugger commands perform operations on specific Pascal/JB source statements. For instance, the BREAK command sets a breakpoint at one or more Pascal/JB statements. This section describes the syntax of the statement specifications accepted by these commands. These statement specifications usually include the statement number as it appears in the Pascal/JB compilation listing. Therefore, a current compilation listing of the program should be obtained before invoking the interactive debugger.

The statement specification is used to identify a particular statement to the debugger. The statement specification consists of three parts: the name of the module containing the statement, the name of the procedure (or function) containing the statement, and the statement number (or END). Each of these components is separated by a "/". If both the module name and the procedure name are omitted, the statement is assumed to be in the currently active procedure. If the module name (but not the procedure name) is omitted, the statement is assumed to be in the currently active module. If the procedure name (but not the module name) is omitted, the statement is assumed to be in the outermost level of the module. If END is specified in place of the line number, the command sets or resets a breakpoint following the last statement in the specified procedure or function.

The following are the legal forms of statement specifications and their interpretation:

M/P/S	statement S in procedure P in module M
M//S	statement S in the body of module M
P/S	statement S in procedure P in the active module
/S	statement S in the body of the active module
S	statement S in the active procedure and module

STACK-FRAME SPECIFICATIONS

The QUALIFY command expects a stack-frame specification as a parameter. A stack frame is the storage that contains the local variables associated with a particular call to a procedure or function. In most cases, the stack-frame specification consists of just the procedure or function name. However, in certain cases, such as when recursive procedures are used, a more elaborate specification may be required.

The stack-frame specification has two parts: a procedure or function name and an optional adjustment. The procedure name (or function name) selects the stack frame that was created when the named procedure was called. If there is no active instantiation of that procedure, which would occur if the procedure had not been called or had already returned, the debugger will issue an error message. If there is more than one instantiation of that procedure, the most recently activated instantiation is used, that is, the stack frame associated with the most recent call to that procedure. An asterisk "*" may be used in place of the procedure or function name to designate the most recently active procedure or function, that is, the procedure where the breakpoint or error was encountered.

The adjustment is a signed number that indicates how many stack frames prior to the named stack frame (if the number is negative) or following the named stack frame (if the number is positive) the desired stack frame is located. For instance, MYSUBR–1 would select the stack frame for the routine that called the routine MYSUBR. If only an adjustment is given, the adjustment is applied to the current stack frame. This is often useful for examining the previous or next stack frame in a chain of recursive calls.

The syntax for a stack-frame selector is:

[procedure name | function name | *] [{+ | –} number]

The following examples show some of the more common forms of stack-frame specifications:

```
*                 {most active stack frame}
*-4               {four frames prior to the most active frame}
-1                {the previous stack frame}
SUM_NUMBERS       {the frame for the function SUM_NUMBERS}
Tree_Search-3     {three frames prior to Tree_Search}
Get_Command+4     {four frames past Get_Command}
```

INTERACTIVE DEBUGGER COMMANDS

This section describes the various interactive debugger commands. If an error is detected in an interactive debugger command, an appropriate error message is written and the user is prompted for the next debug command. These errors have no effect on the underlying execution of the Pascal/JB program. All interactive debug messages and interactive debug statements use a "+" prefix character.

This aids in identifying when input is being requested by the interactive debugger and when input is being requested by the actual program execution.

Attention interrupts may be used to abort the operation of any interactive debugger command except the GO command. Control returns to the debugger, and the user is prompted for the next debugger command when this occurs. If the attention interrupt occurs during processing of the STEP command, control is passed to the interactive debugger prior to the execution of the NEXT statement encountered. Thus, the current statement is allowed to completely execute before the STEP command is aborted.

The following commands are accepted by the Pascal/JB interactive debugger. The underlined portions represent the minimum acceptable abbreviations.

BREAK <statement specification> ...

The BREAK debug command sets one or more breakpoints in the Pascal/JB program. If the execution of the underlying program proceeds to a statement for which a breakpoint has been set, the execution is suspended prior to the execution of that statement, and control is returned to the interactive debugger, the location of the breakpoint encountered is printed, and the user is prompted for a debug command. The STEP or GO debug statements can be used to resume execution. The CLEAR debug command, described below, resets breakpoints. The format of the <statement specification> is as described earlier in this section. There is no limit to the number of breakpoints that can be active at any time.

Examples:

```
BREAK Main/Summ_numbers/4
BREAK /Summ_numbers/end Util/Search_Tree/7
```

CLEAR [<statement specification> ...]

The CLEAR debug statement resets some or all of the active breakpoints set by a previous BREAK command. If no <statement specification> is given on the CLEAR command, *all* currently active breakpoints are removed. If one or more <statement specification> is given, only the breakpoints associated with those statements are reset. The format of a <statement specification> is as discussed earlier.

Examples:

```
CLEAR
CLEAR Main/Summ_Numbers/4 Util/Search_Tree/7
```

DISPLAY <variable> ...

The DISPLAY debug command displays the current value of one or more variables. The variable may be subscripted, dereferenced, or field qualified. Thus, it is possible to display the contents of a field in a dynamic record or an individual element in an array. Array indices must be constants that are compatible with the declared index type. The variable(s) being displayed must be accessible from the current scope (as defined by the normal Pascal rules governing the scope of an identifier). The QUALIFY debug command may be used to change the current scope to that of another stack frame.

Examples:

```
DISPLAY I SCHED[TUES,15] Head_Ptr@.Next_Ptr@.Student_Name
DISPLAY Current_Total
```

DUMP [<stack-frame specification> [<number>]]

The DUMP debug command dumps all of the variables associated with a given stack frame. The format of the dump is identical to the format of the post-mortem dump that Pascal/JB produces when an execution error is encountered. If no stack frame is specified, all currently active stack frames are dumped. The optional number specifies the number of stack frames to dump, starting with the specified stack frame and working back in the chain of routine calls.

Examples:

```
DUMP             {the current stack frame is dumped}
DUMP ALL         {all active stack frames are dumped}
DUMP * 3         {the active and previous two stack frames are dumped}
DUMP My_Routine  {all variables in the procedure My_Routine
                 are dumped}
```

EXPLAIN <debug command>

The EXPLAIN debug command gives a brief explanation of the specified <debug command>.

Example:

```
EXPLAIN DUMP
```

GO

The GO debug command resumes execution of the underlying Pascal/JB program. Control will not return to the interactive debugger unless a breakpoint or an error is encountered during the resumed execution of the program. No parameters are accepted by the GO command. See also the description of the STEP command in this section. RUN is a synonym for GO.

HEXDISPLAY <variable> [<length>] ...

The HEXDISPLAY debug command displays the hexadecimal value of one or more identifiers. The HEXDISPLAY command is similar to the DISPLAY command except that the DISPLAY command prints the values of the identifiers in a form that is consistent with the declared type of the identifier. Therefore, the DISPLAY command is usually preferred and the HEXDISPLAY command is generally useful only when using Pascal/JB in conjunction with routines written in other languages, for example 370/Assembler. The optional <length> is a positive number that is the length in bytes to be displayed. If <length> is omitted, the declared length of the variable is used.

Examples:

```
HEXDISPLAY I Head_Ptr@.NAME
HEXDISPLAY Head_Ptr@NAME 50
HEXDISPLAY Parts[13,10]
```

MTS

The MTS debug command returns control to the system. Other system commands (such as $EDIT) may then be issued. Control may be returned to the interactive debugger using the $RESTART command provided it has not been unloaded explicitly by the $UNLOAD command or implicitly by issuing a $RUN or $LOAD command. The MTS command does not accept any parameters. See also the "$" command description in this section.

QUALIFY <stack-frame specification>

The QUALIFY debug command resets the current scope to some other stack frame. The behaviour of the DISPLAY, DUMP, and HEXDISPLAY debug commands are all affected by changes in the current scope. The format of the <stack-frame specification> is as described earler in this section.

Examples:

```
QUALIFY Mysubr
QUALIFY *-4
```

RUN

The RUN debug command is a synonym for the GO debug command.

STEP [number]

The STEP debug command resumes execution of the underlying Pascal/JB program one statement at a time. The optional number parameter specifies the number of statements to be executed, which defaults to 1. The STEP command can be quite useful when trying to determine the flow of control of a Pascal/JB program. If a statement contains a procedure or function call, the STEP command will halt with the execution of the first statement in that procedure or function. Upon completion of the STEP command, control is returned to the debugger, a message indicating the next statement to be executed is printed, and the user is prompted for another debug command.

Examples:

```
STEP
STEP 4
```

STOP

The STOP debug command terminates execution of both the debugger and the underlying Pascal/JB program and returns control to MTS. The STOP command does not accept any parameters.

TRACEBACK

The TRACEBACK command gives a traceback of all currently active procedures and functions. The traceback starts with the current location, that is, the location of the breakpoint or error that caused the debugger to be invoked, followed by the location from which that routine was called, followed by the location from which the calling routine was called, and so on.

Example:

```
TRACEBACK
```

$<mts-command>

This command executes an MTS command. Control is returned to the interactive debugger upon completion of the command, unless the command unloads the program explicitly, e.g., $UNLOAD, or implicitly, e.g., $RUN or $LOAD.

CALLING SUBROUTINES FROM PASCAL/JB

This section describes how to call system subroutines from Pascal/JB. The techniques described later for Pascal/VS work equally well with Pascal/JB. The techniques described below make use of several special features of Pascal/JB that are not available in Pascal/VS which make calling system subroutines much easier. Therefore, Pascal/JB programmers should use these techniques unless compatibility with Pascal/VS is required.

Pascal/JB programs may easily call S-type (i.e., Fortran-callable) system subroutines. R-type (register called) subroutines may be called with a little more difficulty using the RCALL and ADROF system subroutines.

Each system subroutine called from a Pascal program must be declared either as a PROCEDURE or a FUNCTION with the attribute FORTRAN (not EXTERNAL). If the subroutine uses only the parameter list to accept and return values, it is declared as a PROCEDURE:

```
PROCEDURE subr(par1;par2;...;parn); FORTRAN
```

If the subroutine returns a value in general register 0 (an INTEGER function) or floating-point register 0 (a REAL or SHORT REAL function), it is declared as a FUNCTION:

```
FUNCTION subr(par1;par2;...;parn) : fcntype; FORTRAN
```

where “fcntype” is the data type of the value returned in register 0 (e.g., INTEGER, REAL).

The format of each parameter declaration “parn” is either

```
VAR name : datatype
```

or

```
CONST name : datatype
```

CONSTant parameters are used if the value of the parameter is not changed by the called system subroutine. If the system subroutine changes the value, a VARiable declaration must be used. In addition, the data type of the parameter must be declared to correspond to the data type expected by the system subroutine. The equivalences of data types for Pascal are given below:

Data Type	*Pascal Declaration*
Fullword integer	INTEGER
Halfword integer	PACKED –32768..32767
One-Byte integer	PACKED 0..255
8-Byte integer	ARRAY [1..2] OF INTEGER
Fullword real	SHORTREAL
Doubleword real	REAL[1]
Fullword logical	INTEGER (0 is FALSE, 1 is TRUE)
One-byte logical	BOOLEAN (0 is FALSE, 1 is TRUE)
Character string	PACKED ARRAY [1..n] OF CHAR ("n" is the length of the string)
Array	ARRAY [1..n] OF type ("n" is the number of elements, "type" is data type of each element)
Region	[PACKED] RECORD v1; v2; ... END ("v" are the declarations of the subfields)
Variable type	[PACKED] RECORD CASE INTEGER OF 1: v1; 2: v2; ... END ("v" are the declarations of the subfields)

[1]A doubleword REAL constant can be used where a fullword SHORTREAL is expected. However, the contents of the right-hand half of the doubleword is not guaranteed; this could affect the precision of the results up to a factor of 10**–16.

For a procedure, the call is made in the form

```
subr(p1,p2,...,pn);
```

where "pn" are the individual parameters to the subroutine. For a function, the call is made in the form

```
value := subr(p1,p2,...,pn);
```

where "value" is the function value returned. For both types of calls, either a constant, a variable, or an expression may be used in the parameter list if the parameter is declared as CONST in the PROCEDURE or FUNCTION declaration statement; if the parameter is declared as VAR, then only a variable may be used in the parameter list on the call.

The following programs illustrate Pascal declarations and calls to system subroutines.

```
Program Test;

Procedure MTS; Fortran;

Begin
  MTS;
End.
```

The above example calls the MTS subroutine which requires no parameters.

```
Program Test;

Type
  Char255 = Packed Array [1..255] of Char;
  Halfwrd = Packed -32768..32767;
Var
  String : Char255;
  Length : Halfwrd;
Procedure CMD(Const Cmdstg : Char255;
              Const Cmdlen : Halfwrd); Fortran;

Begin
  String := '$Display Timespelledout';
  Length := 23;
  CMD(String,Length);
End.
```

The above example calls the CMD subroutine to execute a $DISPLAY command. The subroutine requires two parameters, the first being a packed array of characters giving the command string and the second being a halfword command length (CMD also allows a fullword command length to be used). In this example, both parameters are declared as CONST (since they are not changed by the CMD subroutine) although they could also be declared VAR as long as STRING and LENGTH are also declared VAR.

```
Program Test(Input,Output);

Const
  Itemno = 2;
Type
  Char4  = Packed Array [1..4] of Char;
Var
  Userid : Char4;
Procedure GUINFO(Const Gufoitem : Integer;
                 Var   Gufoloc  : Char4); Fortran;

Begin
  GUINFO(Itemno,Userid);
  Writeln('User ID = ',Userid);
End.
```

The above example calls the GUINFO subroutine to obtain the current userID. This subroutine also requires two parameters, the first of which may be either a constant, a variable, or an expression of type integer and the second of which must be a variable packed array of characters.

```
Program Test(Input,Output);

Type
  Char18 = Packed Array [1..18] of Char;
Var
  Filename : Char18;
  Access   : Integer;
Function CHKFILE(Const Chkname : Char18) : Integer; Fortran;

Begin
  Filename := 'WABC:DATA ';
  Access   := CHKFILE(Filename);
  Writeln('Access = ',Access);
End.
```

The above example calls the CHKFILE subroutine to determine the program's access to the file WABC:DATA. Since the access is returned in register 0, the subroutine must be called as a FUNCTION.

```
Program Test(Input,Output);

Type
  Char4   = Packed Array [1..4]  of Char;
  Char8   = Packed Array [1..8]  of Char;
  Char16  = Packed Array [1..16] of Char;
  Char20  = Packed Array [1..20] of Char;
  Intgr2  = Array [1..2] of Integer;
  Intgr6  = Array [1..6] of Integer;
  Retrec  = Record
              Case Integer of
                1 : (Int : Intgr6);
                2 : (Chr : Char20)
              End;
  Catrec  = Packed Record
              Cial   : Integer;
              Cirl   : Integer;
              Cionid : Char4;
              Civol  : Char8;
              Ciuc   : Integer;
              Cilrd  : Integer;
              Cicd   : Integer;
              Cifo   : Integer;
              Cidt   : Integer;
              Ciflg  : Integer;
              Cilcd  : Integer;
              Cipkey : Char16;
              Cilcct : Intgr2;
              Cilncd : Integer;
              Cilnct : Intgr2;
              Cicdt  : Intgr2;
              Cilrdt : Intgr2
            End;
Var
  Unit    : Char8;
  Rtn     : Retrec;
  Flag    : Integer;
  Cinfo   : Catrec;
  I       : Integer;
Procedure GFINFO(Const Gfunit : Char8;
                 Var   Gfrtn  : Retrec;
                 Const Gfflag : Integer;
```

```
                Var   Gfcinf : Catrec;
                Const Gffinf : Integer;
                Const Gfsinf : Integer); Fortran;

Begin
  Unit       := 'SCARDS  ';       {Set unit name}
  Rtn.Int[6] := 0;                {Zero file name return region}
  Flag       := '00000002'X;      {Mark as unit name call}
  Cinfo.Cial := 25;               {Set Cinfo length}
  GFINFO(Unit,Rtn,Flag,Cinfo,0,0);
  Writeln(Rtn.Chr,'Owner = ',Cinfo.Cionid);
End.
```

The above example calls the GFINFO subroutine to obtain catalog information about the file attached to the logical I/O unit SCARDS. The subroutine requires that the region Cinfo (used for the returned catalog information) be defined as a packed record. The variable Rtn is defined using the Record Case Integer form so that on the call, it may be set to the integer zero (as required by GFINFO), and on the return, it may be used by the Pascal program to access the file name in character form. The GFINFO example below illustrates an alternate method of handling this particular situation.

FILES AND DEVICES

Many system subroutines refer to MTS files or MTS devices such as *PRINT*. With a few exceptions such as CREATE and DESTROY, all of these subroutines require that a Fdub-pointer or a logical I/O unit name be specified rather than a file or device name. Therefore, there are two options.

In order to use a Fdub-pointer, the GETFD subroutine must be first called to obtain the Fdub-pointer for the file or device. This Fdub-pointer can then be passed to the other subroutines that require them. The Fdub-pointer is released by calling the FREEFD subroutine.

In order to use logical I/O units, the file or device to be used is normally specified in the $RUN command, e.g.,

```
$RUN -OBJ SCARDS=A SPUNCH=B 0=C
```

The logical unit names are then passed to the the subroutines that require them. Alternatively, the SETLIO subroutine may be called to establish the relationship between the file/device name and the logical I/O unit. Note that unit names such as SCARDS and SPUNCH must be declared as Packed Array [1..8] of Char, whereas unit numbers, such as 0 or 99, may be declared either as Integer or Packed Array [1..8] of Char.

RETURN CODES

Most system subroutines return a value called the return code. This value is normally zero if the subroutine was called and returned properly, and nonzero if an error occurred.

In Pascal/JB, the predefined function FORTRANRC contains the return code from the most recent call to a Fortran procedure or function. The Pascal/JB code generator produces highly efficient code (a single instruction) to effect the FORTRANRC function call. This means that there is almost no overhead associated with using the FORTRANRC function.

The following example is similar to the GFINFO example above except that the return code is checked and the associated error message is accessed.

```
Program Test(Input,Output);

Type
  Char4   = Packed Array [1..4]  of Char;
  Char8   = Packed Array [1..8]  of Char;
  Char16  = Packed Array [1..16] of Char;
  Char20  = Packed Array [1..20] of Char;
  Char80  = Packed Array [1..80] of Char;
  Intgr2  = Array [1..2] of Integer;
  Retrec  = Packed Record
              Chr    :  Char20;
              Int    :  Integer
            End;
  Catrec  = Packed Record
              Cial   : Integer;
              Cirl   : Integer;
              Cionid : Char4;
              Civol  : Char8;
              Ciuc   : Integer;
              Cilrd  : Integer;
              Cicd   : Integer;
              Cifo   : Integer;
              Cidt   : Integer;
              Ciflg  : Integer;
              Cilcd  : Integer;
              Cipkey : Char16;
              Cilcct : Intgr2;
              Cilncd : Integer;
              Cilnct : Intgr2;
              Cicdt  : Intgr2;
              Cilrdt : Intgr2;
            End;
  Refname = Record End;
Var
  Unit    : Char8;
  Rtn     : Retrec;
  Flag    : Integer;
  Cinfo   : Catrec;
  Ercode  : Integer;
  Errmsg  : Char80;
  I       : Integer;
Procedure GFINFO(Const Gfunit : Char8;
                 Var   Gfrtn  : Retrec;
                 Const Gfflag : Integer;
                 Var   Gfcinf : Catrec;
                 Const Gffinf : Integer;
                 Const Gfsinf : Integer;
                 Const Gfecod : Integer;
                 Const Gfemsg : Char80); Fortran;

Begin
  Unit    := 'SCARDS  ';        {Set unit name}
  Rtn.Int := 0;                 {Zero file name return region}
  Flag    := '00000002'X;       {Mark as unit name call}
  Cinfo.Cial := 25;             {Set Cinfo length}
  GFINFO(Unit,Rtn,Flag,Cinfo,0,0,Ercode,Errmsg);
  If FORTRANRC > 0 Then
    Begin
```

```
      If FORTRANRC = 4 Then Writeln(Ercode,'  ',Errmsg);
      If FORTRANRC > 4 Then
         Writeln('Error return from GFINFO subroutine');
      Return
    End;
  Writeln(Rtn.Chr,'  Owner = ',Cinfo.Cionid);
End.
```

The following example illustrates the use of FORTRANRC with both a procedure and a function.

```
Program Test(Input,Output);

Const
  Mask    = '00000010'X;
Type
  Char4   = Packed Array [1..4] of Char;
  Char18  = Packed Array [1..18] of Char;
  Trplrec = Packed Record
              Ccid : Char4;
              Proj : Char4;
              Pkey : Char18
            End;
  Refname = Record End;
Var
  Access : Integer;
  Oldnam : Char18;
  Newnam : Char18;
  Triple : Trplrec;
Function  CHKACC(Const Chknam : Char18;
                 Const Chktrp : Trplrec):Integer;Fortran;
Procedure RENAME(Const Renold : Char18;
                 Const Rennew : Char18); Fortran;

Begin
  Oldnam := 'DATA1 ';               {Set old file name}
  Newnam := 'NEWDATA1 ';            {Set new file name}
  Triple.Ccid := 'WABC';            {Set triple}
  Triple.Proj := 'WXYZ';
  Triple.Pkey := '*EXEC ';
  Access := CHKACC(Oldnam,Triple);
  If FORTRANRC > 0 Then             {Test return code}
    Begin
      Writeln('File does not exist');
      Return
    End;
  If (Access & Mask) ~= Mask Then   {Test access bit}
    Begin
      Writeln('Rename access not allowed');
      Return
    End;
  RENAME(Oldnam,Newnam);
  If FORTRANRC > 0 Then             {Test return code}
    Begin
      Writeln('Error return from RENAME subroutine');
      Return
    End;
  Writeln('File successfully renamed');
 End.
```

The above example calls the CHKACC subroutine to determine if the file WABC:DATA1 exists and if

the user has rename access to the file. If both conditions are true, the program renames the file to NEWDATA1. In this example, the FORTRANRC function must be called following the call to both the CHKACC and RENAME subroutines.

R-TYPE SUBROUTINES

R-type subroutines can be called from Pascal/JB by using the RCALL and ADROF subroutines. The RCALL subroutine sets up a call to an R-type subroutine by inserting the parameters to the RCALL subroutine into the proper registers for the call to the R-type subroutine. The ADROF subroutine is used to obtain the address of a variable as required both for the RCALL subroutine and for other system subroutines such as GETFD.

The format of the declarations is as follows:

```
TYPE
  refname = RECORD END;
REF
  RCALL : refname;
  fcn   : refname;
FUNCTION  ADROF(CONST subr : refname) : INTEGER; FORTRAN;
PROCEDURE RCALL(CONST subr : refname;
                par1;par2;...;parn); FORTRAN;
```

The call is made in the following manner:

```
RCALL(ADROF(subr),r1,p1,...,r2,p2,...);
```

where "r1" is the number of registers to be set up on the call to "subr" and "p1,..." are the values to be inserted into the registers beginning with general register 0; "r2" is the number of registers to contain return values from "subr" and "p2,..." are the variables that will contain the returned values starting with general register 0. If the return codes are desired from the subroutine being called via RCALL, then FORTRANRC may be used.

The following example illustrates the use of RCALL and ADROF for a call to the GDINFO subroutine.

```
Program Test(Input,Output);

Type
  Char4  = Packed Array [1..4]  of Char;
  Hlfwrd = Packed -32768..32767;
  Byte   = Packed 0..255;
  Gdfrec = Packed Record
             Fdub   : Integer;
             Devtyp : Char4;
             Inlen  : Hlfwrd;
             Outlen : Hlfwrd;
             Use    : Byte;
             Device : Byte;
             Sws1   : Byte;
             Sws2   : Byte;
             Mods   : Integer;
             Beglnr : Integer;
             Prvlnr : Integer;
             Endlnr : Integer;
             Inclnr : Integer;
```

```
              Namptr : Integer;
              Msgptr : Integer;
              Iosave : Integer;
              Lastrc : Integer;
              Reglen : Hlfwrd;
              Width  : Hlfwrd;
              Macid  : Integer;
            End;
  Recptr  = @ Gdfrec;
  Refname = Record End;
Ref
  GDINFO  : Refname;
Var
  Unit1   : Char4;
  Unit2   : Char4;
  Dumy    : Integer;
  Info    : Gdfrec;
  Infoptr : Recptr;
Function  ADROF(Const Subr : Refname) : Integer; Fortran;
Procedure RCALL(Const Subr  : Refname;
                Const Gdnam : Integer;
                Const Rnum1 : Integer;
                Const Name1 : Char4;
                Const Name2 : Char4;
                Const Rnum2 : Integer;
                Const Dummy : Integer;
                Var   Regn  : Recptr); Fortran;

Begin
  Unit1 := 'SCAR';               {Set I/O unit name}
  Unit2 := 'DS  ';
  RCALL(ADROF(GDINFO),2,Unit1,Unit2,2,Dumy,Infoptr);
  If FORTRANRC > 0 Then
    Begin
      Writeln('Error return from GDINFO subroutine');
      Return
    End;
  Info := Infoptr@;              {Copy GDINFO region into Info}
  Writeln('Type = ',Info.Devtyp);
End.
```

In this example, the GDINFO subroutine is called by the RCALL subroutine. Two registers (general registers 0 and 1) are set up on the call to contain the eight-character logical I/O unit name. Two registers are also set up for the return. Register 1 will contain the *address* of the GDINFO information region; register 0 is not used, hence a dummy argument must be inserted into the RCALL parameter list. This example also illustrates the case where a system subroutine returns a pointer to an area of storage acquired by the subroutine itself. Hence the variable INFOPTR, which upon return will contain the address of the acquired storage, must be declared as a pointer variable. The statement following the subroutine call is then used to copy the contents of that storage into the Pascal record variable Info so that the individual items of GDINFO information can be accessed by the program. Note that the GDINF alternative entry to the GDINFO subroutine also could have been called; this would circumvent the problem of using pointer variables in the Pascal program.

SPECIAL CASES

The following example illustrates a special case that occurs when an external subroutine is called with more than one type of parameter list. In this case, the ADROF subroutine is called in two

different ways, once to get the address of a file name (of type Char20) and once to get the address of the external subroutine GETFD (normally of type Refname). Since Pascal does not allow ADROF to be declared as having two different types of parameter lists, GETFD is declared as Char20 to circumvent the problem.

```
Program Test(Input,Output);

Const
  First   = 1000;
  Last    = 1000000000;
  Beg     = 1000;
  Inc     = 1000;
Type
  Char18  = Packed Array [1..18] of Char;
  Refname = Record End;
Ref
  GETFD   : Char18;
Var
  Filenam : Char18;
  Fdub    : Integer;
Function  ADROF(Const Fname : Char18) : Integer; Fortran;
Procedure RCALL(Const Subr  : Integer;
                Const Rnum1 : Integer;
                Const Zero  : Integer;
                Const Faddr : Integer;
                Const Rnum2 : Integer;
                Var   Fdub  : Integer); Fortran;
Procedure RENUMB(Const Renfdb : Integer;
                 Const Renfrt : Integer;
                 Const Renlst : Integer;
                 Const Renbeg : Integer;
                 Const Reninc : Integer); Fortran;

Begin
  Filenam := 'DATA1 ';                    {Set file name}
  RCALL(ADROF(GETFD),2,0,ADROF(Filenam),1,Fdub);  {Get FDUB}
  RENUMB(Fdub,First,Last,Beg,Inc); {Renumber}
  If FORTRANRC > 0 Then                   {Test return code}
    Begin
      Writeln('Error return from RENUMB subroutine');
      Return
    End;
  Writeln('File successfully renumbered');
End.
```

Several system subroutines can be called with a variable number of parameters. If this variability is to be used within a Pascal program, separate procedure or function declarations must be made for each case. Due to Pascal language restrictions, these declarations must use external names for each case.

There is a simple solution. Namely, for each declaration a suffix may be added to the name to differentiate it. Then the external (loader) name of the subroutine may be placed in parentheses following the FORTRAN directive. For example, the TAPEINIT subroutine can be called with either four or five parameters. Therefore the declarations should be given in the form

```
PROCEDURE TAPEINIT1(par1,par2,par3,par4); FORTRAN (TAPEINIT)
PROCEDURE TAPEINIT2(par1,par2,par3,par4,par5); FORTRAN (TAPEINIT)
```

From the viewpoint of the Pascal program, TAPEINIT1 and TAPEINIT2 are separate subroutines.

However, from the viewpoint of the MTS loader, both resolve to one subroutine, TAPEINIT.

The following example illustrates the case of variable-parameter calls to the COMMAND subroutine.

```
Program Test(Input,Output);

Const
  Sws     = 0;
Type
  Char255 = Packed Array [1..255] of Char;
Var
  Cmdtext : Char255;
  Length  : Integer;
  Summary : Integer;
  Ercode  : Integer;

Procedure COMMAND1(Const Cmdstg : Char255;
                   Const Cmdlen : Integer;
                   Const Switch : Integer); Fortran (COMMAND);

Procedure COMMAND2(Const Cmdstg : Char255;
                   Const Cmdlen : Integer;
                   Const Switch : Integer;
                   Var   Sumry  : Integer;
                   Var   Code   : Integer); Fortran (COMMAND);

Begin
  Cmdtext := '$Display Timespelledout ';
  Length  := 23;
  Command1(Cmdtext,Length,Sws);
  Cmdtext := '$Display Timemisspelledout ';
  Length  := 26;
  Command2(Cmdtext,Length,Sws,Summary,Ercode);
  If Summary > 0 Then
    Begin
      Writeln('Command Error Code = ',Ercode);
      Return
    End;
End.
```

Some system subroutines, such as CALC and EDIT, allow the calling program to pass an external subroutine as a parameter which will be called by the system subroutine. This external subroutine may be a Pascal routine if a standard S-type linkage is used between the system subroutine and the external subroutine. Any Pascal/JB routine may be passed in this manner provided that the routine declarations of the Pascal routine are identical to the routine declarations in the formal parameter list of the system routine.

*PASCALJBINCLUDE

The declarations required by Pascal/JB to call many of the system subroutines described above are contained in the file *PASCALJBINCLUDE. These declarations may be accessed during program compilation by assigning the file *PASCALJBINCLUDE to one of the logical I/O units 1 through 8; for example,

```
$RUN *PASCALJB SCARDS=SOURCE SPUNCH=OBJECT 1=*PASCALJBINCLUDE
```

*PASCALJBINCLUDE contains definitions for the following system subroutines:

BSRF	CANREPLY	CFDUB
CHGFSZ	CHGMBC	CHGXF
CHKFDUB	CLOSEFIL	CNTLNR
COST	CSGET	CSSET
DISMOUNT	EMPTYF	ERROR
FSIZE	FSRF	GETFST
GETLST	GRJLDT	GRJLTM
GUINFUPD	LOCK	LODMAP
MOUNT	MTS	NOTE
POINT	QUIT	RENUMB
REWIND	RSSAS	SETPFX
SYSTEM	TAPEINIT	TRUNC
UNLK	URAND	WRITEBUF

See MTS Volume 3, *System Subroutine Description*, for more information about these routines.

To make use of these declarations, a statement of the form

```
%INCLUDE rrrr
```

where "rrrr" is the name of the routine whose declarations are to be accessed must be placed in the declaration section of the program. In addition, the user must be aware of the following:

(1) If the routine has a parameter that refers to a logical I/O unit or Fdub-pointer, there must be an %INCLUDE Fdub_Type statement in the program before the other %INCLUDEs.

(2) For each routine "rrrr" used, there must be an %INCLUDE rrrr statement in the program. This will define a Procedure or Function (as appropriate) with the name "rrrr". COST, GRJLDT, GRJLTM, and URAND are Functions; all others are Procedures.

(3) %INCLUDE NOTE and %INCLUDE POINT contain the same definitions. They both define NOTE and POINT. They also define a type ARRAY_NOTEPOINT, which is an array of four integers suitable for holding NOTE/POINT information. In other words, there may be either %INCLUDE NOTE or %INCLUDE POINT, *but not both*.

(4) %INCLUDE FSIZE defines a type ARRAY_FSIZE, which is an array of sixteen integers suitable for use by FSIZE.

(5) %INCLUDE GRJLDT defines a type GRJLDT_CHAR8, which is a Packed Array of 8 characters suitable for holding a date string of the form "mm/dd/yy".

(6) %INCLUDE GRJLTM defines a type GRJLTM_CHAR16, which is a Packed Array of 16 characters suitable for holding a time/date string of the form "mm/dd/yyhh:mm:ss".

(7) %INCLUDE TAPEINIT defines the types TAPEINIT_CHAR4, TAPEINIT_CHAR6, and TAPEINIT_CHAR10, which are Packed Arrays of four, six, and ten characters, respectively, suitable for holding values used by the TAPEINIT routine.

(8) Since %INCLUDE SETPFX is defined as a Procedure, this definition does not allow the original prefix to be retrieved.

To call the routine, place a statement of the form

```
pppp(param1,param2,...paramn);
```

in the program, where “pppp” is the routine name and “param1”, “param2”, etc., are the parameters of the routine in question. The function FortranRC may be used to test the return code.

Note: Any parameter which holds a Fdub-pointer or an I/O unit must either be a variable of type Fdub_Type or a string.

The following three examples illustrate the use of *PASCALJBINCUDE.

```
%INCLUDE COST
BEGIN
  WRITELN('Your cost is',COST);
```

```
%INCLUDE FDUB_TYPE
%INCLUDE CHGFSZ
VAR RC : INTEGER;
    F  : FDUB_TYPE
BEGIN
  {Get FDUB}
  ...
  CHGFSZ(F,200,0);
  IF FORTRANRC~=0 THEN WRITELN(' Error in CHGFSZ.');
```

```
%INCLUDE FDUB_TYPE

{You only should have one of NOTE and POINT}

%INCLUDE NOTE
%INCLUDE ERROR
VAR RC : INTEGER;
    AR : ARRAY_NOTEPOINT;
BEGIN
  {Remember where we are now, so we can go
   back there later}
  NOTE('1',AR);
  IF FORTRANRC~=0 THEN ERROR;
  ...
  POINT('1',AR,15);
  IF FORTRANRC~=0 THEN ERROR;
```

PASCAL/JB EXTENSIONS AND INCOMPATIBILITIES

This section discusses differences between the language accepted by the IBM VS Pascal compiler and Pascal/JB and may be considered an addendum to the IBM *VS Pascal Language Reference Manual*. Refer also to the sections on compilation and execution parameters for other differences in features. These differences are minor and most Pascal/VS programs will run under Pascal/JB with no modifications. If other source language incompatiblities are encountered, please report them to the Computing Center.

EXTENSIONS

The following are extensions of the Pascal/VS language or represent limitations of the Pascal/VS compiler that have been relaxed in the Pascal/JB compiler.

(1) Pascal/JB has no limit on the nesting of procedures (the Pascal/VS limit is 8).

(2) Pascal/JB limits the size of the object code produced for an individual function or procedure to 256K. Pascal/VS limits this to 8K.

(3) Identifier names in Pascal/JB are not truncated at 16 characters as they are in Pascal/VS. The length of an identifier in Pascal/JB is restricted only by the maximum line length of 255 characters.

(4) Pascal/JB allows comparisons between an expression of type STRING and an expression of type PACKED ARRAY [1..n] OF CHAR.

(5) Pascal/JB allows source lines up to 255 characters in length.

(6) DEF variables with an associated VALUE declaration to set an initial value are made entry points in the routine containing the VALUE declaration. Pascal/VS produces a separate CSECT for these. Using the Pascal/VS technique, multiple VALUE declarations in separate modules for the same DEF variable go undetected. Using the Pascal/JB technique, such erroneous declarations result in a loader error for multiple definitions of the same entry point.

DIFFERENCES AND INCOMPATIBLITIES

(1) The size of the file control block (FCB) in Pascal/JB is different than the size in Pascal/VS. This only matters if an FCB is passed to a non-Pascal/JB routine (a dubious practice).

(2) The size of variables of type STRINGPTR is only 4 bytes in Pascal/JB. Pascal/VS uses 8 bytes. Again, this is only important if a STRINGPTR is passed to a non-Pascal/JB routine.

(3) Output from calls to the TRACE predefined procedure is formatted slightly differently in Pascal/JB.

(4) Calls to standard I/O functions with multiple parameters (e.g., WRITE(a,b,c)) are treated

as separate calls (e.g., WRITE(a); WRITE(b); WRITE(c);) as the Pascal standard requires. Pascal/VS evaluates each parameter before calling the given procedure (in violation of the Pascal standard). These two techniques will produce identical results unless the value of a parameter is changed as a side effect of the evaluation of a subsequent parameter. (It is generally considered a bad programming practice to write programs with such dependencies).

(5) When allocating space for variant records dynamically, (for e.g., with NEW(p) or NEW(p,t) where "p" is a pointer to a variant record and "t" is a tag constant) Pascal/VS allocates space to accomodate the maximum variant despite tag specifications. Pascal/JB makes use of the tag specifications, if present. The compiler option NOSHORTRECORDS makes Pascal/JB allocate variant records with maximum variant size and also allows DISPOSE, of pointers to variant records, to be used without the tag specifications. Without this option, in the above example, if "p" is called with NEW(p,t), then "p" has to be disposed by calling DISPOSE(p,t).

(6) The procedures (or functions) Rpad, Lpad, Itohs, and Picture are built in and do not need to be %INCLUDEd in the user program. However, for compatibility, these may be %INCLUDEd and one of units 1 through 8 can be assigned to *PASCALJBINCLUDE. But these procedures (or functions) should not be redeclared by the user.

(7) When using logical I/O units 1 through 8, for INCLUDE sections, Pascal/JB searches all the units whereas Pascal/VS stops with the first unassigned unit.

PASCAL/JB VERSION 2.0

Pascal/JB 2.0 maintains compatibility with VS Pascal, the IBM replacement for Pascal/VS. VS Pascal has slight modifications and improvements over the language used by Pascal/VS. There are also some incompatibilities between the two. Consequently, the incompatibilities that exist between Pascal/VS and VS Pascal also exist between Pascal/JB 2.0 and Pascal/JB 1.0 as well as Pascal/JB 2.0 and Pascal/VS.

INCOMPATIBILITIES BETWEEN PASCAL/JB 1.0 AND 2.0

The following incompatiblities exist between Pascal/JB versions 1.0 and 2.0:

(1) Program parameters are now processed. All program parameters must be declared as global variables (except INPUT and OUTPUT). Since the program parameters are ignored by Pascal/VS (and Pascal/JB 1.0), it is always possible to fix programs that do not conform to this restriction by merely deleting the offending program parameters.

(2) If INPUT is a program parameter, a RESET(INPUT) is automatically issued. If it is *not* present, INPUT is implicitly opened upon its first use (the old Pascal/VS behaviour).

(3) I/O now conforms to the ANSI/IEEE standard. In particular, writing a real zero is not a special case (zero was always written as "0.0" regardless of the field length specified; all other numbers were written using the specified field length). This means that some programs will produce output that is formated slightly differently.

(4) The processing of global variables has changed slightly. This has no effect on programs that consist of a single compilation. However, programs composed of multiple compilation units should all be compiled using the same version of the compiler (that is, if one of the modules is recompiled then all modules should be recompiled).

Pascal/JB has maintained upward compatibility of the execution library between versions 1.0 and the 2.0. Therefore, it is *not* necessary to recompile old programs (although some programs will execute faster if they are recompiled using version 2.0).

However, if one module in a multi-module program is recompiled using version 2.0, then *all* of the modules comprising that program should be compiled using that version. This is the most significant of the incompatibilities listed above.

PASCAL/VS

COMPILING & EXECUTING PASCAL/VS PROGRAMS

A compiler is a program that translates from one language to another. Pascal is a language that a human can use to describe actions which a computer is to take. The program Pascal/VS is a compiler that translates a program written in Pascal into a machine language program that can be run on MTS. The Pascal program that the compiler reads is typically referred to as the "source", and the machine language program produced is frequently referred to as the "object" program.

COMPILING PASCAL/VS PROGRAMS

In general to invoke the Pascal/VS compiler on MTS:

$RUN *PASCALVS [I/O assignments] [limits] [PAR=options]

where "I/O assignments" and "limits" consist of logical I/O unit assignments, time and page limits, and other parameters which can be specified on an MTS $RUN command (see the description of the $RUN command in MTS Volume 1, *The Michigan Terminal System*, for a complete list), and "options" is a list of compiler options (see the section, "Compiler Options," for a complete list of compiler options).

Some examples of running the Pascal/VS compiler:

```
$RUN *PASCALVS SCARDS=wabc:pauls.pgn SPUNCH=-obj T=2
$RUN *PASCALVS SCARDS=treesearch SPUNCH=*PUNCH* PAR=debug
$RUN *PASCALVS SCARDS=my.pgm SPUNCH=-obj 1=mylib PAR=nos,opt
$RUN *PASCALVS SPUNCH=wabc:puffy.object SPRINT=*PRINT*
      .
      . Pascal/VS program (read from *SOURCE* by default)
      .
$ENDFILE
```

Logical I/O Unit Assignments

The logical I/O assignments for Pascal/VS are as follows:

SCARDS=<source-fdname>
SPUNCH=<object-fdname>
SPRINT=<listing-fdname>
SERCOM=<error-fdname>
0=<objutil-fdname>
1=<include-library-1>
.
.
.
8=<include-library-8>

The Pascal program is read from <source-fdname> and compiled. A descriptive listing of the program, error messages, and related information is written to <listing-fdname>. Error messages are also written to <error-fdname>. In batch mode SERCOM is not used, unless it is explicitly assigned. In conversational mode SPRINT is not used, unless it is explicitly assigned.

If no errors are detected by the compiler, an object module is produced. If logical I/O unit 0 is not assigned, the object module is written to <object-fdname> by way of logical I/O unit SPUNCH. If logical I/O unit 0 is assigned, OBJUTIL is used to update <objutil-fdname> (for further information, see the descriptions of *OBJUTIL in MTS Volumes 2 and 5).

If logical I/O units 1-8 are assigned, the files to which they are assigned are treated as include libraries, which contain the information to be inserted in the source program by way of the %INCLUDE directive (see the section, "Use of the %Include Directive in MTS," for a complete description).

Compiler Options

Compile-time options indicate which features are to be enabled or disabled when the compiler is invoked. Compile-time options are specified in the PAR field on the $RUN command and are delimited by commas.

The following table lists all compiler options. Underlining indicates their abbreviated forms. Their default values are on the right. A description of each option follows.

Compiler Option	Default
CHECK / NOCHECK	CHECK
DEBUG / NODEBUG	DEBUG
GOSTMT / NOGOSTMT	GOSTMT
LANGLVL={STANDARD | STDRES | EXTENDED}	LANGLVL=EXTENDED
LINECOUNT=n	LINECOUNT=60
LIST / NOLIST	NOLIST
MARGINS=(m,n)	MARGINS=(1,100)
OPTIMIZE / NOOPTIMIZE	NOOPTIMIZE
PAGEWIDTH=n	PAGEWIDTH=132
PXREF / NOPXREF	NOPXREF
SEQUENCE=(m,n) / NOSEQUENCE	NOSEQUENCE
SOURCE / NOSOURCE	SOURCE
WARNING / NOWARNING	WARNING
XREF({LONG | SHORT}) / NOXREF	NOXREF

CHECK / NOCHECK

Default: CHECK

If the CHECK option is enabled, the Pascal/VS compiler will generate code to perform run-time error checking. When a run-time checking error occurs, a diagnostic message will be displayed on SERCOM followed by a traceback of the routines which were active when the error occurred. The diagnostic message and traceback will be sent to the file or device assigned to SERCOM. The %CHECK feature can be used to enable or disable particular checking code at specific

locations within the source program. If NOCHECK is specified, all run-time checking will be suppressed and all %CHECK statements will be ignored.

Note: The compiler makes no explicit attempt to diagnose the use of uninitialized variables; however, to help detect such errors, the SETMEM run-time option has been provided.

The run-time errors which may be checked are listed as follows:

Case-Statements

A 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 function call is checked to verify that it returns a value.

Pointers

A pointer dereference is checked to make sure that the pointer is not Nil.

Subranges and Subscripts

Assignments to subrange variables and subscripts are checked to guarantee that they are within the subrange bounds. This check is also made after some procedure calls which pass a subrange as a Var parameter, for example, Read.

String Truncation

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

DEBUG / NODEBUG

Default: DEBUG

A debugging facility is available for Pascal/VS programs. If the DEBUG option is enabled, the compiler will provide the information the debugger needs.

Use of the DEBUG option also requires that the GOSTMT option be active. Note that the DEBUG option causes object programs to be somewhat larger and slower. If NODEBUG is specified, the debugger cannot be used fully for that segment.

GOSTMT / NOGOSTMT

Default: GOSTMT

The GOSTMT option enables the inclusion of a statement table within the object code. The entries within this table allow the run-time environment to identify the source statement causing an execution error. This statement table also permits the interactive debugger to place breakpoints based on source statement numbers. NOGOSTMT will prevent the statement table from being generated.

Note that the GOSTMT option causes object programs to be somewhat larger but no slower.

LANGLVL={STANDARD | STDRES | EXTENDED}

Default: EXTENDED

If LANGLVL=STANDARD is specified, the compiler will diagnose all constructs and features that do not conform to standard Pascal. Violations of the standard will appear as warnings. In addition, many of the predeclared identifiers which are unique to Pascal/VS will not be recognized when LANGLVL=STANDARD is specified. If LANGLVL=STDRES is specified the compiler will turn LANGLVL=STANDARD on and also will not recognize any of the non-ANSI-Standard Pascal/VS reserved words. The following is a list of these nonstandard reserved words: ASSERT, CONTINUE, DEF, LEAVE, OTHERWISE, RANGE, REF, RETURN, SPACE, STATIC, VALUE, and XOR. These can now be used as identifiers. LANGLVL=EXTENDED specifies that the full Pascal/VS language is to be supported.

LINECOUNT=<n>

Default: 60

The LINECOUNT option specifies the number of lines to appear on each page of the output listing.

LIST / NOLIST

Default: NOLIST

The LIST option generates an assembler-style listing of the object program. The NOLIST option suppresses this listing. Note: NOLIST will cause any %LIST statement within the source program to be ignored.

MARGINS=(m,n)

Default: (1,100)

The MARGINS(m,n) option sets the left and right margins for the source program. The compiler scans each line of the program starting at column “m” and ending at column “n”. Any data outside these limits is ignored. The maximum right margin allowed is 100. The specified margins must not overlap the sequence field, which is defined by the SEQUENCE option, described below.

OPTIMIZE / NOOPTIMIZE

Default: NOOPTIMIZE

The OPTIMIZE option causes the compiler to generate improved object code. NOOPTIMIZE suppresses object code improvement. Note that OPTIMIZE causes the compiler to run about ten percent slower. The effect of OPTIMIZE on an object program is highly dependent upon the program.

PAGEWIDTH=n

Default: 132

The PAGEWIDTH option specifies the maximum number of characters that may appear on a single line of the output listing. The number specified in the PAGEWIDTH option does not include carriage-control characters. The value of the Pagewidth parameter must be in the range of 120 to 210.

PXREF / NOPXREF

Default: NOPXREF

The PXREF option specifies that the right margin of the output listing is to contain information about the identifiers used in that line. NOPXREF suppresses these entries.

SEQUENCE=(m,n) / NOSEQUENCE

Default: NOSEQUENCE

The SEQUENCE=(m,n) option specifies which columns within the program being compiled are reserved for a sequence field. The starting column of the sequence field is "m"; the last column of the field is "n". The compiler does not process sequence fields, which serve only to identify lines in the source listing. NOSEQUENCE indicates that there is to be no sequence field. Note: The sequence field must not overlap the source margins. If the value for "n" specified is greater than 100, it defaults to 100. This essentially means that the MARGINS option must be used along with the SEQUENCE option.

SOURCE / NOSOURCE

Default: SOURCE

The SOURCE option generates the source program portion of the listing. NOSOURCE suppresses this listing. Note: NOSOURCE will cause any %PRINT statement within the source program to be ignored.

WARNING / NOWARNING

Default: WARNING

The WARNING option causes warning messages to be produced. NOWARNING suppresses these messages.

XREF({LONG | SHORT}) / NOXREF

Default: NOXREF

The XREF option generates the cross-reference portion of the listing. NOXREF suppresses this. Either a short or long cross-reference can be generated. A long cross-reference contains all identifiers declared in the program. A short cross-reference contains only those identifiers which were referenced.

To specify a particular listing mode, either the word LONG or SHORT is placed after the XREF specification and enclosed within parentheses. If no such specification exists, SHORT is defaulted. For example, the specification XREF(LONG) would cause a long cross-reference to be generated.

RUNNING PASCAL/VS PROGRAMS

When the Pascal/VS program successfully compiles, an object module is produced. To run that program, a $RUN command of the following form is used:

$RUN program [mts-par] [PAR=[pascal-par/][program-par]]

where “program”

is the name of the file or device containing the object module to be run.

“mts-par”

consists of logical I/O unit assignments, time and page limits, and other parameters which can be specified on an MTS $RUN command (see MTS Volume 1, *The Michigan Terminal System*, for a description of the $RUN command).

“pascal-par”

is a list of Pascal/VS run-time system options (see the section, “Run-Time Options,” for a complete list.)

and “program-par”

is the program-specific parameter list for “program”, and the predefined function PARMS returns this string as value, at execution time.

Some examples of running Pascal/VS produced programs:

```
$RUN -obj SCARDS=mydata T=3
$RUN -obj 1=-inp1 2=-inp2 SPRINT=*PRINT* T=2 PAR=COUNT/full
$RUN wabc:bigfile SCARDS=-data T=4 PAR=debug/
$RUN object T=3 PAR=program parameter string
$RUN -obj1+-obj2+-obj3 SCARDS=data T=3 PAR=/find/words/
```

Note that if <prog-par> contains “/”, <pas-par> must be given, although it may be null.

Run-Time Options

Features within the Pascal/VS run-time environment may be enabled or disabled by passing options to the Pascal/VS run-time system. These options are passed to a Pascal/VS program through the PAR field. To distinguish run-time options from the parameter string intended to be processed by the program, the options must precede the parameter string (if any) and be terminated with a slash “/”.

The following is a list of supported run-time options:

COUNT
DEBUG
ERRCOUNT=number
MAINT
NOCHECK
NOSPIE
SETMEM
STACK=number
HEAP=number

COUNT

The COUNT option specifies that instruction frequency information is to be collected during program execution. After the program is completed, the information is written to the predefined file variable Output. This option will have an effect only if the program was compiled with the DEBUG option.

DEBUG

The DEBUG option specifies that the interactive debugger is to gain initial control when the program is invoked. Note: This option is valid only if the object module was generated with the DEBUG option. The DEBUG option causes the SETMEM option to become effective.

ERRCOUNT=n

The ERRCOUNT option puts an upper limit on the number of nonfatal run-time errors that can occur. After "n" errors have occurred, execution is terminated.

HEAP=k

The HEAP keyword specifies the minimum number of kilobytes (1024 bytes) by which the heap is to be "extended" each time the heap overflows. The heap is where memory is allocated when the procedure NEW is called. When the end of the heap is reached, the MTS GETSPACE procedure is invoked to allocate more memory for the heap. An integral number of pages (4096 bytes) is always allocated. The fewest pages not less than the space required by NEW and "k" kilobytes is allocated each time the heap is extended.

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

MAINT

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

NOCHECK

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

NOSPIE

The NOSPIE option specifies that the Pascal/VS run-time environment is not to intercept program interrupts.

SETMEM

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

STACK=k

The STACK keyword specifies the minimum number of kilobytes (1024 bytes) by which the run-time stack is to be "extended" each time the stack overflows. The run-time stack is where the dynamic storage areas (DSA) of routines are allocated when the routines are invoked. When the end of the stack is reached, the MTS GETSPACE procedure is invoked to allocate more memory for the stack. An integral number of pages (4096 bytes) is always allocated. The fewest pages not less than the DSA size and "k" kilobytes is allocated each time the stack is extended.

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

PASCAL/VS LISTING INFORMATION

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

PAGE HEADERS

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

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

(1) The module name is given, followed by a colon. This name becomes the name of the control section (CSECT) in which the generated object code will reside.

(2) Following the colon may be the name of the procedure/function definition which was being compiled when the page boundary occurred.

(3) The time and date of the compilation.

(4) The page number.

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

NESTING INFORMATION

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

```
B P C I        STMT
```

where

B indicates the depth of Begin block nesting.

P indicates the depth of Procedure nesting.

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

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

STMT is the subsection of a column that numbers the executable statements of each routine. If the source line originated from an include file, the include number and a colon ":" precede the statement number.

STATEMENT NUMBERING

Pascal/VS numbers the statements of a routine. These numbers are referenced when a run-time error occurs and when break points are specified in the interactive debugger.

All non-empty statements are numbered except the Repeat statement. However, the Until portion of a Repeat statement is numbered. A Begin or End statement is not numbered because it serves only as a bracket for a sequence of statements and has no executable code associated with it.

PAGE CROSS-REFERENCE FIELD

If the PXREF compiler option is active, the right margin of the listing contains a cross reference field. This field contains an indicator for each identifier that appears in the associated line. The indicators have the following meanings:

(1) A number indicates a page number on which the corresponding identifier was declared.

(2) An "*" indicates that the corresponding identifier is being declared.

(3) A "P" indicates that the corresponding identifier is predefined.

(4) An "R" indicates that the corresponding identifier is a reserved keyword.

(5) 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.

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.

PREDEFINED PROCEDURES IN PASCAL/VS

Declarations for these procedures, except the PVCALLRC routine are provided in *PASCALVSINCLUDE. These can be included in the user program by making use of the %INCLUDE statement and assigning one of the logical units 1 through 8 to *PASCALVSINCLUDE.

The procedures themselves are contained in *PASCALVSLIB, which must be explicitly concatenated to the object program in the $RUN command. For example, to compile

```
$RUN *PASCALVS SCARDS=sou SPUNCH=obj 1=*PASCALVSINCLUDE T=t
```

and to execute,

```
$RUN obj+*PASCALVSLIB SCARDS=data SPRINT=output T=t
```

PROCEDURE DESCRIPTIONS

PVPFXGET

Procedure PVPFXGET(var pfx: String);

This procedure is an interface to the system to get the current execution prefix. A call to this procedure has the form PVPFXGET(pfx), where “pfx” is a String variable declared by the user to receive the value of the current prefix character. The execution prefix can be of length up to 120 characters. Hence “pfx” should be declared as a variable of type String with length of at least 120.

PVPFXSET

Procedure PVPFXSET(const pfx:String);

This procedure is an interface to the system to alter the execution prefix. A call to this procedure has the form PVPFXSET(pfx), where “pfx” contains the value of the new prefix. The execution prefix can be of length up to 120 characters, so “pfx” in this case has to be a String constant of length less than or equal to 120.

PVCALLRC

This routine interfaces to OS/I(S) routines which set return codes. Normally when a non-Pascal routine is called from a Pascal program, it is not possible to access the return code. This procedure enables calling non-Pascal routines and also checking the return code on exit from the routine.

To access a non-Pascal procedure "p" the declaration has the following form:

```
PVCALLRC(const p:Proctype; var rc:Integer;...
    {parameters...for p}); Fortran;
```

Name of the procedure PVCALLRC... :

It is not possible to declare a Pascal/VS procedure with a variable number of parameters. So, to be able to call PVCALLRC more than once, or to be able to access different non-Pascal routines each of which uses a different number of parameters, we need more than one declaration of the procedure. This is achieved by making use of the fact that while only the first eight letters are significant to the loader, the first 16 letters are significant to the compiler. So, to call different routines, e.g., procedures "p1", "p2", function "p3" (of type Integer for example), etc., the user can now declare,

```
Procedure PvcallrcP1(const p1: ...); Fortran;
Procedure PvcallrcP2(const p2: ...); Fortran;
Function PvcallrcP3(const p3: ...):Integer; Fortran;
    .
    .
    .
```

Type of the parameter "p":

"p" is declared as a Ref variable, to be resolved to a routine of the same name defined elsewhere. The Ref declaration implies that storage for that variable already exists and the compiler does not allocate any storage, so the type definition here is not used for determining the storage requirement for variables of that type. It is used because the Pascal compiler requires association of a type with a variable declaration. So, a dummy type is used and "p" is declared to be of that type, e.g.,

```
Type Proctype = record end;
Ref p: Proctype;
```

Note: Here the type Proctype is a dummy type which defines zero storage. While the type specification is unimportant, this type keeps it distinct from the types of any of the other parameters in the procedure declaration. This is further protection against accidentally overwriting "p".

Rc is a variable of type integer; it receives the return code. Note that it is the second parameter in the procedure declaration.

The other parameters that follow are those required by the non-Pascal routine that is called.

Examples of declarations and calls:

To call the system routines SETLIO or function SETPFX:

```
Type proctype = record end;
Ref setlio, setpfx : proctype;
    (any other declarations needed by the procedure
     or function ...see example 3)

Procedure PVCALLRCsetlio(CONST p:proctype;
  Var rc:INTEGER;CONST unit:a8; CONST fd : a16); Fortran;

Function PVCALLRCsetpfx(CONST p:proctype;
  Var rc:INTEGER;CONST pfx:CHAR): CHAR; Fortran;
```

are the declarations and calls are of the form

```
PVCALLRCsetlio(setlio,rc,unit,fname);
PVCALLRCsetpfx(setpfx,rc,c);
```

EXAMPLES

Example 1

Use of PVPFXSET and PVPFXGET

```
Program Change_Pfx;

{First saves the current prefix, then sets it to '**', and
 then restores it.}

  %Include pvpfxget;
  %Include pvpfxset;
Var
  Savedpfx : String(120);
Begin
  Pvpfxget(savedpfx); {Save}
  Pvpfxset('**');     {Set}
     ...
  Pvpfxset(savedpfx); {Restore}
End.
```

Example 2

Use of the PVCALLRC Routines

```
Program Callrc;

{Callrc reads two lines.  The second line is copied to the
 file -f at the line number given on the first line.  MTS
 logical I/O unit 99 is used.}

Const
  ati = '00000002'X; { I/O modifier - indexed write}
Type
  a8 = packed array(.1..8.) of char;
  a80 = packed array(.1..80.) of char;
  a16 = packed array(.1..16.) of char;
  halfword = packed -32768..32767;
  whatever = record end; {A dummy for routine names}
Ref
  WRITE, SETLIO : whatever;

Procedure pvcallrcWRITE(const p:whatever; var rc: Integer;
  const reg:a80; const len:halfword; const mods:Integer;
  const lnum:Integer; const unit:a8); fortran;

Procedure pvcallrcSETLIO(const p:whatever; var rc: Integer;
  const unit:a8; const fd:a16); fortran;

Var
  rc,num: integer;
  r: real;
  line: a80;
Begin
  readln(r); readln(line);
  num := round(r*1000);
  pvcallrcSETLIO(setlio,rc,'99','-f');
  if rc <> 0 then Writeln(' Error from SETLIO');
  pvcallrcWRITE(write,rc,line,80,ati,num,'99');
  if rc <> 0 then Writeln(' Error from WRITE')
End.
```

PASCAL/VS INTERACTIVE DEBUGGER

When debugging a program, it is very convenient to be able to monitor the flow of control within the program, and the values of variables at selected points. It is possible to do this by inserting "write" statements at appropriate places; however, this method requires the user to recompile the program whenever new information is required, or old information is no longer necessary.

A better way to debug is available to Pascal/VS users. Programs compiled with the DEBUG option can be interactively monitored by the user. The debugger allows the user to stop execution at selected points, display the values of variables, display statement frequency counts, execute MTS commands, and resume execution.

GETTING STARTED WITH THE DEBUGGER

To use the Pascal/VS debugger, the program must first be compiled with the DEBUG option. This tells the compiler to include in the object code information that the debugger will need to be able to display the values of variables, set breakpoints, etc.

By default the Pascal/VS program is compiled with the DEBUG option ON.

To debug the program, the run-time option DEBUG must be specified. The user should also obtain a current copy of the compile listing, to assist in choosing statement numbers for setting breakpoints, etc. The user initially is given control in the Pascal/VS debug environment, where breakpoints can be set, the values of variables can be displayed, etc. The remainder of the $RUN command (logical I/O unit assignments, etc.) is the same as it would be if the program were being run without the debugger.

For example:

```
$RUN myobj SCARDS=mydata T=3 PAR=DEBUG/
```

will allow the user to use the Pascal/VS debugging system on the program found in the file "myobj" which reads data from the file "mydata".

The Pascal/VS debug system reads from the MTS pseudo-device *SOURCE* and writes to the MTS pseudo-device *SINK*.

QUALIFICATION

A qualification consists of a module name and a routine name (module meaning "segment" or "program," and routine meaning "procedure"). The debugger uses the *current qualification*, by default, when retrieving information for commands. The current qualification is the module and routine name of the routine which was interrupted when the debugger last gained control. The current qualification is initially the name of the module containing the main program, and the main program itself.

INTERACTIVE DEBUGGER COMMANDS

This section describes the commands that a user may issue within the debug environment. Minimum abbreviations of commands are underlined, in the command description. Optional parts of commands are enclosed within brackets ([...]). Semicolons can be used to separate multiple commands on a line.

BREAK [[module/] [routine]/] location

where

module	is the name of a Pascal/VS module,
routine	is the name of a Pascal/VS routine, and
location	is a statement number or END.

The BREAK command sets a breakpoint at the specified location, within the specified routine and module. If a statement number is specified, the program will be stopped, and execution will be returned to the debugger, *before* that statement is executed. Statement numbers can be found on the program listing. If END is specified, the program will stop after the last statement in the specified routine is executed, and before control is returned to the statement which called that routine.

A maximum of 32 breakpoints may be set at any one time.

If the module and/or routine names are not specified, they take a default value from the current qualification, as follows:

Form	*Module*	*Procedure*
B S	current	current
B /S	current	main program
B P/S	current	P
B M//S	M	main program
B M/P/S	M	P

where

current	is the currently qualified procedure,
M	is a module name,
P	is a procedure name, and
S	is a statement number or END.

CLEAR

The CLEAR command removes all breakpoints.

CONTINUE

The CONTINUE command resumes program execution at the point where it was last terminated. Initially, this command starts program execution at the first statement in the main program.

DISPLAY [{BREAKS | EQUATES}]

The DISPLAY command, when given with no parameters, displays

the current qualification,
the next statement to be executed,
the current status of "counts", and
the current status of "tracing".

The BREAKS option displays a list of all breakpoints which are currently set.

The EQUATES option displays a list of all equated symbols and their current definition.

DUMP [{LOCAL | SHORT | HEX} ...]

One or more of the options may be specified, separated by at least one blank or a comma. They can be given in any order.

DUMP without any options displays the variables in their normal form, with the exception of variables of type SPACE which are displayed in hexadecimal representation. The variables declared in the currently active routine are displayed, followed by the variables declared in the preceeding levels of procedure calls.

The LOCAL option displays only the variables in the currently active routine.

The SHORT option restricts array variable dumps to the first 10 elements and prints a message about the other array indices being skipped. If the array has 10 or less elements the whole array is displayed.

The HEX option displays the values of the variables in hexadecimal representaion. The HEX and SHORT options given together display a maximum of 48 bytes of the array in hexadecimal form. A message about the bytes being skipped is also printed.

Note 1: Because of insufficient information in the Pascal/VS debugging tables, array indices are always printed as integers, even when they were declared as an enumerated type.

Note 2: Since ";" is used as the command separator, the character ";" in Dump HEX; LOCAL will not be flagged as invalid. Here LOCAL will be read as a separate command and hence will be flagged as invalid.

EQUATE identifier [data]

where

identifier	is a Pascal/VS identifier, and
data	is the string it is to represent.

The EQUATE command equates an identifier with a data string. When that identifier is encountered in a debugger command, it is replaced within the command by the data string with which it is equated. For example:

```
EQUATE Pc PRINT Head@.count
```

will cause the command "PRINT Head@.count" to be executed every time "Pc" is entered.

Equated strings can contain equated identifiers within them. For example, after:

```
EQUATE Idx Head@.Next@.Index
EQUATE Arr Index_Array[Idx]
```

the command

```
PRINT Arr
```

will cause the command

```
PRINT Index_Array[Head@.Next@.Index]
```

to be executed.

The effect of debugger commands (with the exception of the EQUATE command itself) can be changed through use of the EQUATE command.

When the EQUATE command is specified with no data operand, the effect is to remove any existing equated definition of the given identifier.

GO

The GO command is synonymous with the CONTINUE command. Program execution is resumed at the point where it was last interrupted. Initially, program execution begins at the first statement in the main program.

HELP

The HELP command prints a brief list of the currently available debugger commands and their descriptions.

HEXPRINT variable [{ATTR | NOATTR}]

The HEXPRINT option is similar to the PRINT option, except that the variable is displayed in hexadecimal and character form, with periods substituted for nonprintable characters in the character section.

If the variable is uninitialized, a hexadecimal value FE will be displayed for each uninitialized byte.

MTS

The MTS command returns control to MTS command mode. The currently executing program is not unloaded, and can be restarted at the point where it was left through the MTS $RESTART command.

PRINT variable [{ATTR | NOATTR}]

where "variable" is the name of a Pascal/VS variable.

The PRINT command displays the current value of the specified variable.

If the variable exists within the scope of the currently qualified routine, its value is displayed; otherwise, an error message is produced.

The output is formatted according to the type of the variable. The user may view an array slice by specifying fewer indices than are in the declared dimension of the array. All variable values are displayed in their standard forms (real numbers look like real numbers, integers look like integers, etc.), except for variables of type Space which are displayed in hexadecimal.

If no option is given, the default is the current ATTR setting (the initial ATTR setting is OFF and can be changed through the SET ATTR command). If attributes are requested, either through the default or by specifying the option ATTR, the data type, memory class, length (if relevant), and routine where the variable was declared are displayed along with the value of the variable. If attributes are to be suppressed, either through the default or by specifying the option NOATTR, only the value of the variable is displayed.

If the variable is uninitialized and of a simple type, the word "uninitialized" will be printed as its value. In the case of arrays and fields of records, uninitialized variables are displayed as either a "?", as "uninitialized", or as a truncated form of "uninitialized". For example in a Packed Array of Char, a "?" is printed to denote an uninitialized element.

Variables of type space are displayed in hexadecimal and hence the value FE is printed for each uninitialized byte of a Space variable.

Array subscripts must be variables or constants. Expressions are not allowed.

Some Print examples:

```
PRINT Index
PRINT Car_Prices[123]
PRINT Head@
PRINT Head@.Value
PRINT Worksheet[1,Index] ATTR
PRINT Head@[x,y].Acct_Ptr@.Acct[Index]
```

PRINT hexstring [: length]

where

hexstring	is a number in hexadecimal notation, and
length	is a decimal length.

This form of the PRINT command displays the contents of specific memory locations. The first memory location to be dumped is specified by the hexstring operand. If a length operand is specified, that number of bytes is displayed. If no length operand is specified, 16 bytes are displayed.

Both the hexadecimal and character representations of the contents of memory are displayed. In the character representation, nonprinting characters are displayed as periods.

The hexstring must be a hexadecimal number, surrounded by single quotes, and followed by the character "x" (e.g., '602536'x). The length is specified in decimal.

QUAL [module/] [routine]

where

module	is the name of a module, and
routine	is the name of a routine.

The QUAL command changes the current qualification to the specified qualification.

If no module is specified, the current module name is used. If no routine is specified, the main program or outermost lexical level of the current module is used (depending on whether the current module is a program or a segment module).

If no activation of a routine is available (no invocations have taken place), variables from that routine cannot be displayed.

RESTORE [[module/] [routine]/] location

where

module	is the name of a module,
routine	is the name of a routine, and
location	is a statement number or END.

The RESTORE command removes the breakpoint which is set at the specified location. The defaults are the same as in the BREAK command.

SET option

where "option" may be

ATTR {ON | OFF}
COUNT {ON | OFF}
TRACE} {ON | OFF}

The ATTR option sets the default way in which variables are displayed. The ON parameter specifies that information about the variable's attributes is to be displayed by default. The OFF parameter specifies that information about the variable's attributes is not to be displayed by default.

The COUNT option initiates and terminates statement counting. The ON parameter specifies that the number of times each statement is executed during program execution is to be counted. The OFF parameter specifies that this counting is to stop.

The TRACE option initiates or terminates statement tracing. The ON parameter specifies that a list of every statement number executed is to be output to the user's terminal, (or to <filename>

if specified in SET TRACE ON<filename>) as the execution occurs. The OFF parameter specifies that this tracing is to stop.

STOP

The STOP command terminates program execution, unloads the program, and returns control to MTS.

TRACE

The TRACE command produces a list of the names of procedures in the current invocation chain, along with the most recently executed statement in each.

WALK

The WALK command executes the next statement in the program. After execution of that statement control is returned to the Pascal/VS debug system. This command can be used to "single step" through a program.

$...

Commands that begin with a "$" are passed to MTS to be interpreted as MTS commands. Control returns to the Pascal/VS debug system after the execution of the command (with the exception of the commands $RUN, $LOAD, $DEBUG, $RERUN, and some forms of the $RESTART and $START commands).

CALLING SUBROUTINES FROM PASCAL/VS

This section describes how to call system subroutines from Pascal/VS. Since Pascal/JB is compatible with Pascal/VS, the techniques described here work equally well with Pascal/JB. Thus, these techniques may also be used by Pascal/JB programmers wishing to remain compatible with Pascal/VS. In addition, Pascal/JB includes several extensions which make it significantly easier to call system subroutines. Use of these extensions is described in the Section on, "Calling Subroutines from Pascal/JB".

Pascal/VS programs may easily call S-type (i.e., Fortran-callable) system subroutines. R-type (register called) subroutines may be called with a little more difficulty using the RCALL and ADROF system subroutines.

Each system subroutine called from a Pascal program must be declared either as a PROCEDURE or a FUNCTION with the attribute FORTRAN. If the subroutine uses only the parameter list to accept and return values, it is declared as a PROCEDURE:

```
PROCEDURE subr(par1;par2;...;parn); FORTRAN
```

If the subroutine returns a value in general register 0 (an INTEGER function) or floating-point register 0 (a REAL or SHORT REAL function), it is declared as a FUNCTION:

```
FUNCTION subr(par1;par2;...;parn) : fcntype; FORTRAN
```

where "fcntype" is the data type of the value returned in register 0 (e.g., INTEGER, REAL).

The format of each parameter declaration "parn" is either

```
VAR name : datatype
```

or

```
CONST name : datatype
```

CONSTant parameters are used if the value of the parameter is not changed by the called system subroutine. If the system subroutine changes the value, a VARiable declaration must be used. In addition, the data type of the parameter must be declared to correspond to the data type expected by the system subroutine. The equivalences of data types for Pascal are given below:

Data Type	*Pascal Declaration*
Fullword integer	INTEGER
Halfword integer	PACKED –32768..32767
One-Byte integer	PACKED 0..255
8-Byte integer	ARRAY [1..2] OF INTEGER
Fullword real	SHORTREAL
Doubleword real	REAL[1]
Fullword logical	INTEGER (0 is FALSE, 1 is TRUE)
One-byte logical	BOOLEAN (0 is FALSE, 1 is TRUE)
Character string	PACKED ARRAY [1..n] OF CHAR ("n" is the length of the string)
Array	ARRAY [1..n] OF type ("n" is the number of elements, "type" is data type of each element)
Region	[PACKED] RECORD v1; v2; ... END ("v" are the declarations of the subfields)
Variable type	[PACKED] RECORD CASE INTEGER OF 1: v1; 2: v2; ... END ("v" are the declarations of the subfields)

[1]A doubleword REAL constant can be used where a fullword SHORTREAL is expected. However, the contents of the right-hand half of the doubleword is not guaranteed; this could affect the precision of the results up to a factor of 10**–16.

For a procedure, the call is made in the form

```
subr(p1,p2,...,pn);
```

where "pn" are the individual parameters to the subroutine. For a function, the call is made in the form

```
value := subr(p1,p2,...,pn);
```

where "value" is the function value returned. For both types of calls, either a constant, a variable, or an expression may be used in the parameter list if the parameter is declared as CONST in the PROCEDURE or FUNCTION declaration statement; if the parameter is declared as VAR, then only a variable may be used in the parameter list on the call.

The following programs illustrate Pascal declarations and calls to system subroutines.

```
Program Test;

Procedure MTS; Fortran;

Begin
  MTS;
End.
```

The above example calls the MTS subroutine which requires no parameters.

```
Program Test;

Type
  Char255 = Packed Array [1..255] of Char;
  Halfwrd = Packed -32768..32767;
Var
  String : Char255;
  Length : Halfwrd;
Procedure CMD(Const Cmdstg : Char255;
              Const Cmdlen : Halfwrd); Fortran;

Begin
  String := '$Display Timespelledout';
  Length := 23;
  CMD(String,Length);
End.
```

The above example calls the CMD subroutine to execute a $DISPLAY command. The subroutine requires two parameters, the first being a packed array of characters giving the command string and the second being a halfword command length (CMD also allows a fullword command length to be used). In this example, both parameters are declared as CONST (since they are not changed by the CMD subroutine) although they could also be declared VAR as long as STRING and LENGTH are also declared VAR.

```
Program Test(Input,Output);

Const
  Itemno = 2;
Type
  Char4  = Packed Array [1..4] of Char;
Var
  Userid : Char4;
Procedure GUINFO(Const Gufoitem : Integer;
                 Var   Gufoloc  : Char4); Fortran;

Begin
  GUINFO(Itemno,Userid);
  Writeln('User ID = ',Userid);
End.
```

The above example calls the GUINFO subroutine to obtain the current userID. This subroutine also requires two parameters, the first of which may be either a constant, a variable, or an expression of type integer and the second of which must be a variable packed array of characters.

```
Program Test(Input,Output);

Type
  Char18 = Packed Array [1..18] of Char;
Var
  Filename : Char18;
  Access   : Integer;
Function CHKFILE(Const Chkname : Char18) : Integer; Fortran;

Begin
  Filename := 'WABC:DATA ';
  Access   := CHKFILE(Filename);
  Writeln('Access = ',Access);
End.
```

The above example calls the CHKFILE subroutine to determine the program's access to the file WABC:DATA. Since the access is returned in register 0, the subroutine must be called as a FUNCTION.

```
Program Test(Input,Output);

Type
  Char4   = Packed Array [1..4]  of Char;
  Char8   = Packed Array [1..8]  of Char;
  Char16  = Packed Array [1..16] of Char;
  Char20  = Packed Array [1..20] of Char;
  Intgr2  = Array [1..2] of Integer;
  Intgr6  = Array [1..6] of Integer;
  Retrec  = Record
              Case Integer of
                1 : (Int : Intgr6);
                2 : (Chr : Char20)
              End;
  Catrec  = Packed Record
              Cial   : Integer;
              Cirl   : Integer;
              Cionid : Char4;
              Civol  : Char8;
              Ciuc   : Integer;
              Cilrd  : Integer;
              Cicd   : Integer;
              Cifo   : Integer;
              Cidt   : Integer;
              Ciflg  : Integer;
              Cilcd  : Integer;
              Cipkey : Char16;
              Cilcct : Intgr2;
              Cilncd : Integer;
              Cilnct : Intgr2;
              Cicdt  : Intgr2;
              Cilrdt : Intgr2
            End;
Var
  Unit    : Char8;
  Rtn     : Retrec;
  Flag    : Integer;
  Cinfo   : Catrec;
  I       : Integer;
Procedure GFINFO(Const Gfunit : Char8;
                 Var   Gfrtn  : Retrec;
```

```
                Const Gfflag : Integer;
                Var   Gfcinf : Catrec;
                Const Gffinf : Integer;
                Const Gfsinf : Integer); Fortran;

Begin
  Unit       := 'SCARDS  ';       {Set unit name}
  Rtn.Int[6] := 0;                {Zero file name return region}
  Flag       := '00000002'X;      {Mark as unit name call}
  Cinfo.Cial := 25;               {Set Cinfo length}
  GFINFO(Unit,Rtn,Flag,Cinfo,0,0);
  Writeln(Rtn.Chr,'Owner = ',Cinfo.Cionid);
End.
```

The above example calls the GFINFO subroutine to obtain catalog information about the file attached to the logical I/O unit SCARDS. The subroutine requires that the region Cinfo (used for the returned catalog information) be defined as a packed record. The variable Rtn is defined using the Record Case Integer form so that on the call, it may be set to the integer zero (as required by GFINFO), and on the return, it may be used by the Pascal program to access the file name in character form. The GFINFO example below illustrates an alternate method of handling this particular situation.

FILES AND DEVICES

Many system subroutines refer to MTS files or MTS devices such as *PRINT*. With a few exceptions such as CREATE and DESTROY, all of these subroutines require that a Fdub-pointer or a logical I/O unit name be specified rather than a file or device name. Therefore, there are two options.

In order to use a Fdub-pointer, the GETFD subroutine must be first called to obtain the Fdub-pointer for the file or device. This Fdub-pointer can then be passed to the other subroutines that require them. The Fdub-pointer is released by calling the FREEFD subroutine.

In order to use logical I/O units, the file or device to be used is normally specified in the $RUN command, e.g.,

```
$RUN -OBJ SCARDS=A SPUNCH=B 0=C
```

The logical unit names are then passed to the the subroutines that require them. Alternatively, the SETLIO subroutine may be called to establish the relationship between the file/device name and the logical I/O unit. Note that unit names such as SCARDS and SPUNCH must be declared as Packed Array [1..8] of Char, whereas unit numbers, such as 0 or 99, may be declared either as Integer or Packed Array [1..8] of Char.

RETURN CODES

Most system subroutines return a value called the return code. This value is normally zero if the subroutine was called and returned properly, and nonzero if an error occurred.

Normally, when a standard procedure or function call is made to a non-Pascal external or system subroutine, the return code returned by the subroutine is not accessible to a Pascal/VS calling program. However, the return code can be obtained if the subroutine is called by using the Pascal/VS routine PVCALLRC.

The format of the declaration for procedures and functions is as follows:

```
TYPE
  refname = RECORD END;
REF
  pcd : refname;
  fcn : refname;
VAR
  rcode : INTEGER;
  value : fcntype;
PROCEDURE PVCALLRCpcd(CONST subr : refname;
                      VAR   rc   : INTEGER;
                      par1;par2;...;parn); FORTRAN;
FUNCTION  PVCALLRCfcn(CONST subr : refname;
                      VAR   rc   : INTEGER;
                      par1;par2;...;parn) : fcntype; FORTRAN;
```

where “refname” is a dummy variable declared as a reference variable to be used for resolving the names of the external subroutines called. PVCALLRCpcd and PVCALLRCfcn are used to generate distinct calls to the PVCALLRC routine for each of external subroutine calls made (the MTS loader will treat all of the PVCALLRC items as one external symbol since it truncates external names at 8 characters). The suffixes “pcd” and “fcn” are usually the names of the external subroutines that are being called (note that the suffix is unnecessary if PVCALLRC is only being used to call one external subroutine). The subroutine parameters “par” are declared in the same manner as described in the above examples.

The calls to the external subroutines are made in the following manner:

```
PVCALLRCpcd(pcd,rcode,p1,p2,...,pn);
value := PVCALLRCfcn(fcn,rcode,p1,p2,...,pn);
```

The following example is similar to the GFINFO example above except that the subroutine is now called using PVCALLRC so that the return code and associated error message can be accessed.

```
Program Test(Input,Output);

Type
  Char4   = Packed Array [1..4]  of Char;
  Char8   = Packed Array [1..8]  of Char;
  Char16  = Packed Array [1..16] of Char;
  Char20  = Packed Array [1..20] of Char;
  Char80  = Packed Array [1..80] of Char;
  Intgr2  = Array [1..2] of Integer;
  Retrec  = Packed Record
              Chr    :  Char20;
              Int    :  Integer
            End;
  Catrec  = Packed Record
              Cial   : Integer;
              Cirl   : Integer;
              Cionid : Char4;
              Civol  : Char8;
              Ciuc   : Integer;
              Cilrd  : Integer;
              Cicd   : Integer;
              Cifo   : Integer;
              Cidt   : Integer;
              Ciflg  : Integer;
              Cilcd  : Integer;
```

```
              Cipkey : Char16;
              Cilcct : Intgr2;
              Cilncd : Integer;
              Cilnct : Intgr2;
              Cicdt  : Intgr2;
              Cilrdt : Intgr2;
            End;
  Refname = Record End;
Ref
  GFINFO  : Refname;
Var
  Rcode   : Integer;
  Unit    : Char8;
  Rtn     : Retrec;
  Flag    : Integer;
  Cinfo   : Catrec;
  Ercode  : Integer;
  Errmsg  : Char80;
  I       : Integer;
Procedure PvcallrcGFINFO(Const Subr   : Refname;
                         Var   Rc     : Integer;
                         Const Gfunit : Char8;
                         Var   Gfrtn  : Retrec;
                         Const Gfflag : Integer;
                         Var   Gfcinf : Catrec;
                         Const Gffinf : Integer;
                         Const Gfsinf : Integer;
                         Const Gfecod : Integer;
                         Const Gfemsg : Char80); Fortran;

Begin
  Unit    := 'SCARDS  ';        {Set unit name}
  Rtn.Int := 0;                 {Zero file name return region}
  Flag    := '00000002'X;       {Mark as unit name call}
  Cinfo.Cial := 25;             {Set Cinfo length}
  PvcallrcGFINFO(GFINFO,Rcode,Unit,Rtn,Flag,Cinfo,0,0,Ercode,Errmsg);
  If Rcode > 0 Then
    Begin
      If Rcode = 4 Then Writeln(Ercode,'  ',Errmsg);
      If Rcode > 4 Then
         Writeln('Error return from GFINFO subroutine');
      Return
    End;
  Writeln(Rtn.Chr,'  Owner = ',Cinfo.Cionid);
End.
```

The following example illustrates the use of PVRCALLRC with both a procedure and a function.

```
Program Test(Input,Output);

Const
  Mask    = '00000010'X;
Type
  Char4   = Packed Array [1..4] of Char;
  Char18  = Packed Array [1..18] of Char;
  Trplrec = Packed Record
              Ccid : Char4;
              Proj : Char4;
              Pkey : Char18
            End;
  Refname = Record End;
```

```
Ref
  CHKACC : Refname;
  RENAME : Refname;
Var
  Rcode  : Integer;
  Access : Integer;
  Oldnam : Char18;
  Newnam : Char18;
  Triple : Trplrec;
Function  PvcallrcCHKACC(Const Subr   : Refname;
                         Var   Rc     : Integer;
                         Const Chknam : Char18;
                         Const Chktrp : Trplrec):Integer;Fortran;
Procedure PvcallrcRENAME(Const Sub    : Refname;
                         Var   Rc     : Integer;
                         Const Renold : Char18;
                         Const Rennew : Char18); Fortran;

Begin
  Oldnam := 'DATA1 ';               {Set old file name}
  Newnam := 'NEWDATA1 ';            {Set new file name}
  Triple.Ccid := 'WABC';            {Set triple}
  Triple.Proj := 'WXYZ';
  Triple.Pkey := '*EXEC ';
  Access := PvcallrcCHKACC(CHKACC,Rcode,Oldnam,Triple);
  If Rcode > 0 Then                 {Test return code}
    Begin
      Writeln('File does not exist');
      Return
    End;
  If (Access & Mask) ~= Mask Then   {Test access bit}
    Begin
      Writeln('Rename access not allowed');
      Return
    End;
  PvcallrcRENAME(RENAME,Rcode,Oldnam,Newnam);
  If Rcode > 0 Then                 {Test return code}
    Begin
      Writeln('Error return from RENAME subroutine');
      Return
    End;
  Writeln('File successfully renamed');
 End.
```

The above example calls the CHKACC subroutine to determine if the file WABC:DATA1 exists and if the user has rename access to the file. If both conditions are true, the program renames the file to NEWDATA1. In this example, the subroutine PVCALLRC must be called with declarations for both the CHKACC and RENAME subroutines, hence the use of the name PVCALLRCCHKACC and PVCALLRCRENAME. Other names for PVCALLRC, such as PVCALLRC1 and PVCALLRC2, could also have been used.

R-TYPE SUBROUTINES

R-type subroutines can be called from Pascal/VS by using the RCALL and ADROF subroutines. The RCALL subroutine sets up a call to an R-type subroutine by inserting the parameters to the RCALL subroutine into the proper registers for the call to the R-type subroutine. The ADROF subroutine is used to obtain the address of a variable as required both for the RCALL subroutine and

for other system subroutines such as GETFD.

The format of the declarations is as follows:

```
TYPE
  refname = RECORD END;
REF
  RCALL : refname;
  fcn   : refname;
FUNCTION  ADROF(CONST subr : refname) : INTEGER; FORTRAN;
PROCEDURE PVCALLRCRCALL(CONST subr : refname;
                        VAR   rc   : INTEGER;
                        par1;par2;...;parn); FORTRAN;
```

The call is made in the following manner:

```
PVCALLRCRCALL(RCALL,rcode,ADROF(subr),r1,p1,...,r2,p2,...);
```

where “r1” is the number of registers to be set up on the call to “subr” and “p1,...” are the values to be inserted into the registers beginning with general register 0; “r2” is the number of registers to contain return values from “subr” and “p2,...” are the variables that will contain the returned values starting with general register 0. If the return codes are not desired from the subroutine being called via RCALL, then RCALL may be called in a more direct manner (see the GETFD example below).

The following example illustrates the use of RCALL and ADROF for a call to the GDINFO subroutine.

```
Program Test(Input,Output);

Type
  Char4  = Packed Array [1..4]  of Char;
  Hlfwrd = Packed -32768..32767;
  Byte   = Packed 0..255;
  Gdfrec = Packed Record
             Fdub   : Integer;
             Devtyp : Char4;
             Inlen  : Hlfwrd;
             Outlen : Hlfwrd;
             Use    : Byte;
             Device : Byte;
             Sws1   : Byte;
             Sws2   : Byte;
             Mods   : Integer;
             Beglnr : Integer;
             Prvlnr : Integer;
             Endlnr : Integer;
             Inclnr : Integer;
             Namptr : Integer;
             Msgptr : Integer;
             Iosave : Integer;
             Lastrc : Integer;
             Reglen : Hlfwrd;
             Width  : Hlfwrd;
             Macid  : Integer;
           End;
  Recptr  = @ Gdfrec;
  Refname = Record End;
Ref
  GDINFO  : Refname;
```

```
  RCALL   : Refname;
Var
  Unit1   : Char4;
  Unit2   : Char4;
  Dumy    : Integer;
  Info    : Gdfrec;
  Rcode   : Integer;
  Infoptr : Recptr;
Function  ADROF(Const Subr : Refname) : Integer; Fortran;
Procedure PvcallrcRCALL(Const Subr  : Refname;
                        Var   Rc    : Integer;
                        Const Gdnam : Integer;
                        Const Rnum1 : Integer;
                        Const Name1 : Char4;
                        Const Name2 : Char4;
                        Const Rnum2 : Integer;
                        Const Dummy : Integer;
                        Var   Regn  : Recptr); Fortran;
Begin
  Unit1 := 'SCAR';               {Set I/O unit name}
  Unit2 := 'DS  ';
  PvcallrcRCALL(RCALL,Rcode,ADROF(GDINFO),2,Unit1,Unit2,2,Dumy,Infoptr);
  If Rcode > 0 Then
    Begin
      Writeln('Error return from GDINFO subroutine');
      Return
    End;
  Info := Infoptr@;              {Copy GDINFO region into Info}
  Writeln('Type = ',Info.Devtyp);
End.
```

In this example, the GDINFO subroutine is called by the RCALL subroutine. Two registers (general registers 0 and 1) are set up on the call to contain the eight-character logical I/O unit name. Two registers are also set up for the return. Register 1 will contain the *address* of the GDINFO information region; register 0 is not used, hence a dummy argument must be inserted into the RCALL parameter list. This example also illustrates the case where a system subroutine returns a pointer to an area of storage acquired by the subroutine itself. Hence the variable INFOPTR, which upon return will contain the address of the acquired storage, must be declared as a pointer variable. The statement following the subroutine call is then used to copy the contents of that storage into the Pascal record variable Info so that the individual items of GDINFO information can be accessed by the program. Note that the GDINF alternative entry to the GDINFO subroutine also could have been called; this would circumvent the problem of using pointer variables in the Pascal program.

SPECIAL CASES

The following example illustrates a special case that occurs when an external subroutine is called with more than one type of parameter list. In this case, the ADROF subroutine is called in two different ways, once to get the address of a file name (of type Char20) and once to get the address of the external subroutine GETFD (normally of type Refname). Since Pascal does not allow ADROF to be declared as having two different types of parameter lists, GETFD is declared as Char20 to circumvent the problem.

```
Program Test(Input,Output);

Const
  First   = 1000;
```

```
  Last    = 1000000000;
  Beg     = 1000;
  Inc     = 1000;
Type
  Char18  = Packed Array [1..18] of Char;
  Refname = Record End;
Ref
  GETFD   : Char18;
  RENUMB  : Refname;
Var
  Filenam : Char18;
  Fdub    : Integer;
  Rcode   : Integer;
Function  ADROF(Const Fname : Char18) : Integer; Fortran;
Procedure RCALL(Const Subr  : Integer;
                Const Rnum1 : Integer;
                Const Zero  : Integer;
                Const Faddr : Integer;
                Const Rnum2 : Integer;
                Var   Fdub  : Integer); Fortran;
Procedure PvcallrcRENUMB(Const Subr   : Refname;
                         Var   Rc     : Integer;
                         Const Renfdb : Integer;
                         Const Renfrt : Integer;
                         Const Renlst : Integer;
                         Const Renbeg : Integer;
                         Const Reninc : Integer); Fortran;

Begin
  Filenam := 'DATA1 ';                    {Set file name}
  RCALL(ADROF(GETFD),2,0,ADROF(Filenam),1,Fdub);  {Get FDUB}
  PvcallrcRENUMB(RENUMB,Rcode,Fdub,First,Last,Beg,Inc); {Renumber}
  If Rcode > 0 Then                       {Test return code}
    Begin
      Writeln('Error return from RENUMB subroutine');
      Return
    End;
  Writeln('File successfully renumbered');
End.
```

Several system subroutines can be called with a variable number of parameters. If this variability is to be used within a Pascal program, separate procedure or function declarations must be made for each case. Due to Pascal language restrictions, these declarations must use different external names for each case.

If the system subroutine name is eight characters long, there is a simple solution. Namely, for each declaration a suffix may be added to the name to differentiate it. For example, the TAPEINIT subroutine can be called with either four or five parameters. Therefore the declarations should be given in the form

```
PROCEDURE TAPEINIT1(par1,par2,par3,par4); FORTRAN
PROCEDURE TAPEINIT2(par1,par2,par3,par4,par5); FORTRAN
```

From the viewpoint of the Pascal program, TAPEINIT1 and TAPEINIT2 are separate subroutines. However, from the viewpoint of the MTS loader which truncates all external names to eight characters, both resolve to one subroutine, TAPEINIT.

If the system subroutine name is less than eight characters long, one of two solutions must be employed:

(1) The name is padded to eight characters before the suffix is added; e.g.,

```
PROCEDURE COMMANDX1(par1,par2,par3); FORTRAN
PROCEDURE COMMANDX2(par1,par2,par3,par4,par5); FORTRAN
```

The object module from the Pascal compilation is then processed by the *OBJUTIL program to generate an "alias" name for the subroutine COMMANDX, e.g.,

```
$RUN *OBJUTIL 0=object
RENAME COMMANDX=COMMAND
STOP
```

(2) Each declaration of the external subroutine is embedded in a separate local procedure in the Pascal program which is then called from the main program.

The following example illustrates the second method in the case of variable-parameter calls to the COMMAND subroutine.

```
Program Test(Input,Output);

Const
  Sws     = 0;
Type
  Char255 = Packed Array [1..255] of Char;
Var
  Cmdtext : Char255;
  Length  : Integer;
  Summary : Integer;
  Ercode  : Integer;

Procedure Command1;
  Procedure COMMAND(Const Cmdstg : Char255;
                    Const Cmdlen : Integer;
                    Const Switch : Integer); Fortran;
  Begin
    COMMAND(Cmdtext,Length,Sws);
    Return
  End;

Procedure Command2;
  Procedure COMMAND(Const Cmdstg : Char255;
                    Const Cmdlen : Integer;
                    Const Switch : Integer;
                    Var   Sumry  : Integer;
                    Var   Code   : Integer); Fortran;
  Begin
    COMMAND(Cmdtext,Length,Sws,Summary,Ercode);
    Return
  End;

Begin
  Cmdtext := '$Display Timespelledout ';
  Length  := 23;
  Command1;
  Cmdtext := '$Display Timemisspelledout ';
  Length  := 26;
```

```
  Command2;
  If Summary > 0 Then
    Begin
      Writeln('Command Error Code = ',Ercode);
      Return
    End;
End.
```

The solution is much more direct if the variable-parameter subroutine is called using PVCALLRC to obtain the return code, since separate names do not need to be generated for the external subroutine. The following example illustrates the case of a call to the SETKEY subroutine using PVCALLRC.

```
Program Test(Input,Output);

Type
  Char18  = Packed Array [1..18] of Char;
  Char80  = Packed Array [1..80] of Char;
  Refname = Record End;
Ref
  SETKEY  : Refname;
Var
  Filenam : Char18;
  Progkey : Char18;
  Errcode : Integer;
  Errmsg  : Char80;
  Rcode   : Integer;
Procedure Pvcallrc1(Const Subr : Refname;
                    Var   Rc   : Integer;
                    Const Name : Char18;
                    Const Pkey : Char18;
                    Const Info : Integer;
                    Var   Ecod : Integer;
                    Var   Emsg : Char80); Fortran;
Procedure Pvcallrc2(Const Subr : Refname;
                    Var   Rc   : Integer;
                    Const Name : Char18;
                    Const Pkey : Char18;
                    Const Info : Integer); Fortran;

Begin
  Filenam := 'DATA1 ';              {Set file name}
  Progkey := 'DATAKEY ';            {Set pkey to DATAKEY}
  Pvcallrc1(SETKEY,Rcode,Filenam,Progkey,0,Errcode,Errmsg);
  If Rcode > 0 Then                 {Test return code}
    Begin
      Writeln('Error return from SETKEY subroutine');
      Writeln(Errcode,Errmsg);
      Return
    End;
  Writeln('Program key = DATAKEY');
  Progkey  := '*EXEC ';             {Restore pkey to *EXEC}
  Pvcallrc2(SETKEY,Rcode,Filenam,Progkey,0);
  Writeln('Program key = *EXEC');
End.
```

Another case where this is likely to happen is when the RCALL subroutine is needed to call several different R-type subroutines, each with a different length parameter list. For example, in the previous GDINFO example, the FREESPAC subroutine could be called to release the storage acquired by the GDINFO subroutine. Here again, separate declarations would be required for PVCALLRC for

each different call to RCALL.

```
Program Test(Input,Output);

Type
  Char4  = Packed Array [1..4]  of Char;
  Hlfwrd = Packed -32768..32767;
  Byte   = Packed 0..255;
  Gdfrec = Packed Record
             Fdub   : Integer;
             Devtyp : Char4;
             Inlen  : Hlfwrd;
             Outlen : Hlfwrd;
             Use    : Byte;
             Device : Byte;
             Sws1   : Byte;
             Sws2   : Byte;
             Mods   : Integer;
             Beglnr : Integer;
             Prvlnr : Integer;
             Endlnr : Integer;
             Inclnr : Integer;
             Namptr : Integer;
             Msgptr : Integer;
             Iosave : Integer;
             Lastrc : Integer;
             Reglen : Hlfwrd;
             Width  : Hlfwrd;
             Macid  : Integer;
           End;
  Recptr  = @ Gdfrec;
  Refname = Record End;
Ref
  GDINFO   : Refname;
  FREESPAC : Refname;
  RCALL    : Refname;
Var
  Unit1   : Char4;
  Unit2   : Char4;
  Dumy    : Integer;
  Info    : Gdfrec;
  Rcode   : Integer;
  Infoptr : Recptr;
Function  ADROF(Const Subr : Refname) : Integer; Fortran;
Procedure Pvcallrc1RCALL(Const Subr  : Refname;
                         Var   Rc    : Integer;
                         Const Gdnam : Integer;
                         Const Rnum1 : Integer;
                         Const Name1 : Char4;
                         Const Name2 : Char4;
                         Const Rnum2 : Integer;
                         Const Dummy : Integer;
                         Var   Regn  : Recptr); Fortran;
Procedure Pvcallrc2RCALL(Const Subr  : Refname;
                         Var   Rc    : Integer;
                         Const Gdnam : Integer;
                         Const Rnum1 : Integer;
                         Const Rzero : Integer;
                         Const Regn  : Recptr;
                         Const Rnum2 : Integer); Fortran;
```

```
Begin
  Unit1 := 'SCAR';               {Set I/O unit name}
  Unit2 := 'DS  ';
  Pvcallrc1RCALL(RCALL,Rcode,ADROF(GDINFO),2,Unit1,Unit2,2,
                 Dumy,Infoptr);
  If Rcode > 0 Then
    Begin
      Writeln('Error return from GDINFO subroutine');
      Return
    End;
  Info := Infoptr@;              {Copy GDINFO region into Info}
  Writeln('Type = ',Info.Devtyp);
  Pvcallrc2RCALL(RCALL,Rcode,ADROF(FREESPAC),2,0,Infoptr,0);
End.
```

Some system subroutines, such as CALC and EDIT, allow the calling program to pass an external subroutine as a parameter which will be called by the system subroutine. This external subroutine may be a Pascal routine if a standard S-type linkage is used between the system subroutine and the external subroutine. The Pascal external subroutine should be encoded in the following form:

```
SEGMENT segname;
PROCEDURE subr(par1;par2;...;parn); MAIN;
PROCEDURE subr;
BEGIN
   {Program statements}
END; .
```

The MAIN declaration forces the Pascal subroutine to have a standard S-type linkage. If the main program calling the system subroutine which then calls the Pascal routine is not a Pascal program, then it should also call the Pascal subroutine PSCLHX (with a zero in the parameter list) after the call to the system subroutine in order to clean up the Pascal environment. For example (using a Fortran main calling program),

```
  ...
CALL CALC(...)
  ...
CALL PSCLHX(0)
  ...
```

System subroutines such as ATTNTRP, PGNTTRP, and SVCTRP cannot be called using this scheme since the exit routines are not called with an S-type linkage. In these cases, the exit routines must be encoded in 370-assembler language or another suitable lower-level language. on the following pages.

*PASCALVSINCLUDE

The file *PASCALVSINCLUDE contains declarations for calling many of the system subroutines. These declarations may be accessed during program compilation by assigning the file *PASCALVSINCLUDE to one of the logical I/O units 1 through 8, for example,

```
$RUN *PASCALVS SCARDS=SOURCE SPUNCH=OBJECT 1=*PASCALVSINCLUDE
```

*PASCALVSINCLUDE contains definitions for the following routines:

BSRF	CANREPLY	CFDUB

CHGFSZ	CHGMBC	CHGXF
CHKFDUB	CLOSEFIL	CNTLNR
COST	CSGET	CSSET
DISMOUNT	EMPTYF	ERROR
FSIZE	FSRF	GETFST
GETLST	GRJLDT	GRJLTM
GUINFUPD	LOCK	LODMAP
MOUNT	MTS	NOTE
POINT	QUIT	RENUMB
REWIND	RSSAS	SETPFX
SYSTEM	TAPEINIT	TRUNC
TWAIT	UNLK	URAND
WRITEBUF		

See MTS Volume 3, *System Subroutine Descriptions*, for more information about these routines.

To make use of these declarations, statements of the form %Include *name* need to be placed in the declarations section of the program. The following points must be noted:

(1) SETPFX cannot be called using the PVCALLRC mechanism outlined below. To call SETPFX, place the line

```
%INCLUDE SETPFX
```

in the declaration section of the program. The call statement has the form

```
SETPFX(setpfx_char)
```

where the "setpfx_chr" is the character to be used as the prefix character. This definition does not allow the old prefix to be retrieved.

(2) SYSTEM, QUIT, ERROR, MTS, and DISMOUNT can be called using the PVCALLRC mechanism outlined below or by a simpler mechanism. Use a

```
%INCLUDE rrrr
```

statement in the declaration section of the program, where "rrrr" is SYSTEM, QUIT, ERROR, MTS, or DISMOUNT. The call statement has the form

```
rrrr;
```

except for DISMOUNT which takes a parameter. The call statement for this is

```
DISMOUNT(dismount_string);
```

(3) For all remaining subroutines, use PVCALLRC and follow steps 3 through 11. There must be a %INCLUDE Procedure_Type statement in the program which is placed physically before any %INCLUDE PVCALLRC... statements.

(4) If the routine refers to a logical I/O unit or a Fdub-pointer, there must be an %INCLUDE Fdub_Type statement in the program which is placed physically before any %INCLUDE

PVCALLRC... statements.

(5) For each routine "rrrr" to be called, there must be an %INCLUDE PVCALLRCrrrr statement in the program. This will define a Ref with the name "rrrr") and a Procedure or Function (as appropriate) with the name PVCALLRCrrr. PVCALLRCCOST, PVCALLRCGRJLDT, PVCALLRCGRJLTM, PVCALLRCLETGO, and PVCALLRCURAND are Functions; all others are Procedures.

(6) %INCLUDE PVCALLRCNOTE and %INCLUDE PVCALLRCPOINT contain the same definitions. They both define PVCALLRCNOTE, PVCALLRCPOINT, NOTE, and POINT. They also define a type ARRAY_NOTEPOINT, which is an array of four integers suitable for holding NOTE/POINT information. In other words, there may be either %INCLUDE PVCALLRCNOTE or %INCLUDE PVCALLRCPOINT, *but not both*.

(7) %INCLUDE PVCALLRCFSIZE defines a type ARRAY_FSIZE, which is an array of sixteen integers suitable for use by FSIZE.

(8) %INCLUDE PVCALLRCGRJLDT defines a type GRJLDT_CHAR8, which is a Packed Array of 8 characters suitable for holding a date string of the form "mm/dd/yy".

(9) %INCLUDE PVCALLRCGRJLTM defines a type GRJLTM_CHAR16, which is a Packed Array of 16 characters suitable for holding a time/date string of the form "mm/dd/yyhh:mm:ss".

(10) %INCLUDE PVCALLRCTAPEINIT defines the types TAPEINIT_CHAR4, TAPEINIT_CHAR6, and TAPEINIT_CHAR10, which are Packed Arrays of four, six, and ten characters, respectively, suitable for holding values used by the TAPEINIT routine.

(11) %INCLUDE PVCALLRCTWAIT defines the two types: TIMEINTERVAL which is an array of two fullword integers suitable for holding a time interval, and TIMEARRAY16 which is a Packed Array of 16 characters suitable for holding a time/date of the form "hh:mm:ssmm/dd/yy". %INCLUDE PVCALLRCTWAIT also defines two routines, PVCALLRCTWAIT8 and PVCALLRCTWAITS. PVCALLRCTWAIT8 requires the second parameter to be of type TIMEINTERVAL, and PVCALLRCTWAITS requires the second parameter to be of type TIMEARRAY16.

To call the routine, place a statement of the form

```
pppp(rrrr,ccc,param1,param2,...paramn);
```

in the program, where "pppp" is usually "PVCALLRC" followed by the routine name, for example "PVCALLRCCLOSEFIL" or "PVCALLRCRENUMB". The only exceptions are PVCALLRCTWAITS and PVCALLRCTWAIT8 as explained above. "rrrr" is the name of the MTS subroutine, "ccc" is an Integer variable into which the return code will be placed, and "param1", "param2", etc., are the parameters of the routine in question.

Note: Any parameter which holds a Fdub-pointer or an I/O unit must be either a variable of type Fdub_Type or a string.

The following five examples illustrate the use of *PASCALVSINCLUDE.

```
(1)  %INCLUDE PROCEDURE_TYPE
     %INCLUDE PVCALLRCCOST
     VAR RC:INTEGER;
     BEGIN
       WRITELN('Your cost is',PVCALLRCCOST(COST,RC));


(2)  %INCLUDE PROCEDURE_TYPE
     %INCLUDE FDUB_TYPE
     %INCLUDE PVCALLRCCHGFSZ
     VAR RC : INTEGER;
         F  : FDUB_TYPE
     BEGIN
       {Get FDUB}
       ...
       PVCALLRCCHGFSZ(CHGFSZ,RC,F,200,0);


(3)  %INCLUDE PROCEDURE_TYPE
     %INCLUDE FDUB_TYPE

     {You only should have one of PVCALLRCNOTE
      and PVCALLRCPOINT}

     %INCLUDE PVCALLRCNOTE
     VAR RC : INTEGER;
         AR : ARRAY_NOTEPOINT;
     BEGIN

       {Remember where we are now, so we can go
        back there later}

       PVCALLRCNOTE(NOTE,RC,'1',AR);
       ...
       PVCALLRCPOINT(POINT,RC,'1',AR,15);


(4)  CONST LARGEST_SIZE = 3000;
     VAR  SIZE : INTEGER;
     %INCLUDE ERROR
     %INCLUDE SYSTEM
     BEGIN
       ...
       IF SIZE > LARGEST_SIZE THEN ERROR
         ELSE SYSTEM;
       ...
     END


(5)  %INCLUDE SETPFX
     BEGIN
       SETPFX('>');
       ...
     END
```

PASCAL/JB AND PASCAL/VS

I/O WITH PASCAL/VS AND PASCAL/JB

FILES AS DATA STRUCTURES IN PASCAL

Files consist of groups of logical records. The end of the file is delimited by an "end-of-file" mark (see the description of the predefined I/O function Eof). Files can be one of two general types: Text files or Record files.

Text Files

Files of the predefined type Text consist of groups of logical records, each containing a variable number of characters. It is possible for a program, when reading from a Text file, to determine if the file variable points to the end of the current line. Logical records, within Text files, can be thought of as being terminated by an "end-of-line" mark.

Automatic conversion of integers and real numbers to their character representation, and the reverse, are available when using Text files.

Record Files

All other files consist of physical records that contain one or more logical records. Each logical record contains the internal representation of an element of the file. For example, a file declared as

```
f : file of Integer
```

will consist of logical records that are four-byte binary numbers, since the internal representation of integers in Pascal/VS and Pascal/JB is a four-byte binary number. A physical record must contain an integral number of logical records (see the section, "Open Options," for a complete description of the BLOCK and NOBLOCK options).

Since some file operations in MTS trim trailing blanks by default (the MTS $COPY command, for example), particular care must be taken when using files not created directly by Pascal programs. This trimming of blanks can change the contents of the physical records in a file and produce blocking errors. Files of Arrays of characters are particularly susceptible.

INTERNAL FILES

In order to communicate with the outside world, Pascal file variables can be associated with MTS logical I/O units or with specific MTS files or devices. But some programs use files as a means of storing large, varying amounts of data. These programs have no need for the long-term availability of information stored in an MTS file or on an MTS device. In such cases internal files should be used. A file variable will access an internal file if no external file, device, or logical I/O unit is associated with it when it is initialized (see the section, "Open Options," for a complete description of the FILE and UNIT options).

Internal files can be used by programs to temporarily store information which is to be used internally by those programs. Information written to internal files is stored in virtual memory. The virtual memory used to store the information is released when the file is closed, and the file contents

are lost (indicating clearly why they are only useful for temporary storage of information). Since all information in an internal file must have been generated during program execution, it usually makes no sense to open an internal file for reading before it has been written to.

PREDEFINED FILE VARIABLES

There are two predefined file variables, Input and Output, which are used as the defaults for input and for output. Both are of type Text. By default when reading from Input, information is read from MTS logical I/O unit SCARDS. Similarly, when writing to Output, information is written to MTS logical I/O unit SPRINT. If no file variable is specified when calling the predefined procedures Read or Readln, Input is used. If no file variable is specified when calling the predefined procedures Write or Writeln, Output is used. Both Input and Output are implicitly opened for use when the first call that uses the I/O (other than Eof or Eoln) is made. The calls that initialize the default file variables have the same effect as if they did the following:

```
Reset (Input, 'Unit=SCARDS')
Rewrite (Output, 'Unit=SPRINT')
```

Note: With Pascal/JB, the PAR=STANDARD option makes this work in the standard way. That is, the default files are opened, if and only if they appear as progam parameters. For example,

```
Program Example(Input,Output)
```

PREDEFINED I/O PROCEDURES AND FUNCTIONS

The file variable “f” in the following sections refers to a Pascal variable, declared in the declaration section of the user program. For example,

```
Var f: text;
```

The open options, “File” or “Unit”, can be used to attach a MTS file or device to a Pascal file variable. The open options are discussed in a later section.

Reset

The predefined procedure Reset is used to open a file for reading. A file must be opened for reading before it can be read, except in the case of the default file Input.

A call to the Reset procedure has the form:

```
Reset(f)          or
Reset(f,opts)
```

where “f” is a File variable, and “opts” is a string that contains a list of options to be used when opening the file (see the section, “Open Options,” for a complete list of available options).

If no option string is specified and the file has been previously opened, the FILE and UNIT options remain in effect and all other options are reset.

Normally, after resetting a file, the value of the file pointer is defined as being either the first element in the file, or undefined if the file is empty. In order to achieve this, the first record of the file must be read at the time the file is Reset. In order to eliminate synchronization problems that this may produce, the INTERACTIVE open option is provided for interactive input that eliminates this initial read and defines the initial value of the file pointer as undefined. See "Open Options" for a complete description of the INTERACTIVE option.

Rewrite

The predefined procedure Rewrite initializes a file for output. Files must be initialized through a call to Rewrite before they can be used for output, except in the case of the default file, Output.

A call to the Rewrite procedure has the form:

```
Rewrite(f)          or
Rewrite(f,opts)
```

where "f" is a File variable, and "opts" is a string that contains a list of options to be used when opening the file.

If no option string is provided and the file has already been opened, the FILE and UNIT options remain in effect and all other options are reset. *By default, Rewrite causes the file to be emptied of its previous contents.* See "Open Options" for a description of the NOEMPTY option.

Update

The predefined procedure Update opens a file for both input and output. Files must be initialized through a call to Update before they can be updated.

A call to the Update procedure has the form:

```
Update(f)           or
Update(f,opts)
```

where "f" is a File variable, and "opts" is a string that contains a list of options to be used when opening the file. If no option string is specified and the file has been previously opened, the FILE and UNIT options remain in effect and all other options are reset.

When a file is opened for updating, the file is first Reset, and the first element is placed in the buffer variable. A subsequent Put operation will place the contents back into the file at the location where it was read from.

Each Get operation reads the next logical record into the buffer and each Put operation replaces the file component obtained by the last Get operation. The contents of a record can thus be read into the buffer using Get, modified and then replaced at the same location by using Put. Note: The Put operation works this way only when the file is opened for Update. Update is not supported for Text files.

Get

The predefined procedure Get is used to advance a File variable so that it points to the next element within a file.

A call to the Get procedure has the form:

```
Get(f)
```

where "f" is a File variable.

Record File Get

With Record files, a call to the procedure Get sets the file variable to point to the next logical record in the file. If, prior to the call, the file variable was pointing to the last logical record in the file, the end-of-file condition will become true, and the file variable will be undefined.

Text File Get

With Text files, a call to the procedure Get sets the file variable to point to the next character within the current logical record. If, prior to the call, the file variable pointed to the last character in the logical record, the file variable is set to point to a blank, and the end-of-line condition is set to true.

If, prior to the call, the end-of-line condition was true, the file variable is set to the first character of the next logical record. If no further logical records exist (the file variable is pointing to the end of the last logical record), the end-of-file condition is set.

Attempting to call the procedure Get on a file whose end-of-file condition is true or on a file opened by the procedure Rewrite will produce a run-time error.

Put

The predefined procedure Put is used to advance a file variable within a file to point at the next location where information is to be stored.

A call to the Put procedure has the form:

```
Put(f)
```

where "f" is a File variable.

A call to the procedure Put causes the file variable to be advanced to the next location in the file, eliminating the possibility of changing the element to which it previously pointed, except when the file is opened for Update.

Attempting to call the procedure Put on a file which has been opened by the procedure Reset will produce a run-time error.

Read

The predefined procedure Read is used to move information from a file into program variables.

A call to the Read procedure has the form:

```
Read(f,v1)
Read(f,v1,...,vn)
```

where "f" is an optional File variable, and "v1,...,vn" are program variables. If no file variable is given, the predefined file variable Input is used. The second form is equivalent to:

```
Read(f,v1);
Read(f,v2);
    .
    .
    .
Read(f,vn)
```

A call to Read performs two functions:

(1) it assigns the element to which the file pointer currently points, to the program variable specified, and

(2) it performs a "Get(f)", for example, Read(f,v1) implies v1 := f@; Get(f);

If the file is opened with the INTERACTIVE option (see "Open Options" for a complete description of the INTERACTIVE option), and the file variable is not yet set at the time of the read, an initial "Get (f)" will be performed.

Text File Read

When reading from a Text file, an alternate form is available:

```
Read(f,v1:n)
```

where "n" is the width of the field to be read.

Variables whose values are to be read from Text files must be of one of the following types:

Char (or subrange thereof)
Integer (or subrange thereof)
Packed Array of Char
Real (or Shortreal)
String

Text files are defined (loosely) to consist of Packed Arrays of Char. It is possible, however, to read from a Text file into an Integer or a Real variable, if the data in the Text file is of the correct form. The compiler will perform automatic conversion from the character form of a number into its internal representation, making numeric input more convenient.

When reading Integer or Real data, the file variable is advanced until a nonblank character is found or the end of the file is reached. Logical record boundaries are ignored. If the end of the file is reached before a nonblank is encountered, a run-time error will be produced. If a nonblank character is found, the compiler will attempt to convert the characters at and following that position to the internal representation of the correct type. It will stop conversion when either an illegal character is found, the optional field width is satisfied, a blank is encountered, or the end of the record is reached. If no value can be extracted from the characters found (the first character is illegal or the number consists only of a sign), then a run-time error is signalled.

Note: Since a blank stops the scanning, Eof will never be True following a successful read of an Integer or Real, because of the Eoln blank.

Integer data consists of an optional sign ("+" or "–") followed by a series of digits. Real data consists of an optional sign ("+" or "–") followed by a series of characters which conform to the syntax of real numbers in Pascal.

If a field width is specified, the file variable will be left positioned at the end of the field, even if the variable read was a number and that number only occupied part of the field (if the number was terminated by an illegal character that occurred within the field, for example).

When reading data into variables that are declared as Packed Array of Char or String, data is read until the variable is filled to its declared (maximum) length, the end of the line is reached, or the field width is exhausted.

The length of String variables will be the number of characters read (which varies as described above). Variables declared as Packed Array of Char will be padded with sufficient blanks to reach their declared length, if necessary.

Readln

The predefined procedure Readln is used to skip all remaining characters within a logical record. It can optionally read a list of variables in the same fashion as Read, before skipping the remaining characters in the current logical record.

A call to the Readln procedure has the same form as a call to the Read procedure, with two added forms:

```
Readln(f)
Readln(f,v1,...,vn)
```

where "f" is an optional File variable, the file that is to be read from. If no file variable is given, the predefined file variable Input is used. "v1,...,vn" are program variables. The parentheses are not required if the file variable and program variables are omitted.

A call of this form skips over the remaining characters in the current logical record after reading in the values of any requested variables.

Normally, after reading in the values of any requested variables, a call to this procedure reads in the next logical record and positions the file variable at its beginning. If the last logical record is exhausted through a call to this procedure, the end-of-file condition becomes true.

A call to Readln on a file opened with the INTERACTIVE option reads in any requested variables, causes the end-of-line condition to become true, and positions the file variable at the end of the current logical record.

On a file opened with the INTERACTIVE option, a call to Readln such that

(1) it reads no variables,

(2) the file variable is already positioned at the end of the current logical record, and

(3) the position is not the result of a previous call to Readln,

has no effect. See "Open Options" for a complete description of the INTERACTIVE option.

A call to Readln when positioned at the end of the file will produce a run-time error.

Write

The predefined procedure Write is used to move information from program variables into a file.

A call to the Write procedure has the form:

```
Write(f,v1)              or
Write(f,v1,...,vn)
```

where "f" is an optional File variable, and "v1,...,vn" are program variables. If no file variable is given, the predefined file variable Output is used. "v1,...,vn" cannot be file variables. The parentheses are not required if the file variable and program variables are omitted. The second form of the call is equivalent to:

```
Write(f,v1);
Write(f,v2);
    .
    .
    .
Write(f,vn)
```

A call to Write performs two functions:

(1) it assigns to the file variable the program variable specified, and

(2) it performs a "Put (f)", that is, Write(f,v1) implies f@ := v1; Put(f);

Text File Write

When writing to a Text file, two forms are available:

```
Write(f,v1:n1)           or
Write(f,v1:n1:n2)
```

where "n1" and "n2" are optional and are the lengths of the field and number of decimal places,

respectively. Both "n1" and "n2" are integer expressions.

If "f" is omitted, it is assumed to be the predefined file Output.

The first character of each logical record, which is reserved for carriage control, is normally inaccessible (see the description of the predefined procedure Page in this section) unless the file is opened with the NOCC option. See "Open Options" for a complete description of the NOCC option.

If, during a call to Write, the width of the file or device being written to is exceeded, an error diagnostic is, by default, generated and the output continues on the next logical record. The use of the WRAP option when opening the output file will inhibit the generation of this error diagnostic. See "Open Options" for a complete description of the WRAP option.

If the value specified for the length of a field (n1 of our examples) is positive, the data will appear right-justified within the field. If the value is negative, the data will appear left-justified within the field.

String data that have a length greater than the width of the field specified are truncated to that length. If the width of the character representation of numeric data is greater than the specified field width, the data are written using the actual width of the character representation.

If no field width is specified, a default width is used:

Data Type	*Default Width*
Boolean	10
Char	1
Integer	12
Real	20
Shortreal	20
String	current string length
Packed Array of Char	declared length

Writeln

The predefined procedure Writeln is used on Text files to complete the current logical record and begin a new logical record, or to write out a list of program variables and then complete the current logical record and begin a new logical record.

Writeln takes the same form as Write for Text files, with the additional forms:

```
Writeln(f)
Writeln(f,v1,...,vn)
```

where "f" is an optional File variable, the file that is to be written to. If no file variable is provided, the predefined file variable Output is used. "v,...,vn" are program variables. Parentheses are not required if the file variable and program variables are omitted.

A call of this form writes the values of any specified variables to the current logical record, ends the current logical record, and begins a new logical record.

By default, on MTS files, a call to Writeln when the current record is empty produces a line with one blank on it. See the description of the NULLINE option in "Open Options" for information on writing a zero-length line to a Text file. Zero-length lines are allowed in internal files, so such a call will produce a null line.

Note: Attempting to call this procedure using a file which has been opened by a call to Reset will produce a run-time error.

Page

The predefined procedure Page is used on Text files to cause output to those files to start on a new page.

A call to the Page procedure has the form:

```
Page(f)
```

where "f" is a File variable of type Text. The page procedure inserts the logical carriage-control character for "page eject" as the first character in the next logical record to be output. If a partially completed logical record exists at the time of the call to the Page procedure, an implicit call to Writeln is made to complete that logical record and start a new one, in which the page carriage control will be inserted.

Eoln

The predefined function Eoln returns a Boolean result which indicates whether or not the end-of-line condition for a given File variable is True.

A call to the Eoln procedure has the form:

```
Eoln(f)
```

where "f" is an optional File variable of type Text. If "f" is not specified, the predefined file variable Input is used.

It is an error to call Eoln when Eof is True.

Eof

The predefined function Eof returns a Boolean result which indicates whether or not the end-of-file condition for a given file variable is true.

A call to the Eof procedure has the form:

```
Eof(f)
```

where "f" is an optional File variable. If "f" is not specified, the predefined file Input is used.

The end-of-file function returns True when:

(1) Reset is called and the file is empty,

(2) the file is open for output,

(3) Get is called when the file pointer is positioned at the end of the last logical record in the file (at which time the end-of-file condition becomes true), or

(4) Read is called and all of the last logical record has been read.

When the end-of-file condition is True, the value of the file variable is nil.

Calls to Get or Read for a file for which the end-of-file condition is True produce a run-time error. Eof and Eoln are not correctly set until the file is opened explicitly (by Reset) or implicitly by a call to Read, Readln, or Get (in the case of input files).

Eof becomes True for an input Text file when the last line of the file is read or under certain circumstances when MTS reads a line which consists of the string "$ENDFILE", or in the CLG mode in Pascal/JB, the control statement /COMPILE or /EXECUTE is encountered. By default, when $ENDFILE is read from *SOURCE*, it is treated as an end-of-file. The default behavior in batch mode, when reading from data cards within a batch deck, is to treat $ENDFILE as an end-of-file. See the section on commands and delimiters in MTS Volume 1, *The Michigan Terminal System*, and the section, "Pascal/JB Control Statements," in this volume.

Seek

The predefined procedure Seek permits Pascal programs to perform direct access I/O on Text or Record files.

A call to the Seek procedure has the form:

Seek(f,n)

where "f" is a File variable, and "n" is an integer expression.

The Seek procedure has two modes in which it can work,

(1) MTS line-number mode, and

(2) IBM mode.

The mode in which Seek will work is determined when the file is opened for I/O (see the section, "Open Options," for a complete description of the MTSSEEK and IBMSEEK options). If Seek is working in MTS line-number mode, the integer expression "n" represents the MTS line number, multiplied by 1000, at which the file variable "f" is to be positioned.

If Seek is working in IBM mode (the default), the integer expression "n" represents the record number at which the file variable "f" will be positioned. In this mode, if the file is unblocked, the record number 1 corresponds to line number 1 in MTS files or devices, record number 2 corresponds to line number 2, etc. Lines which have fractional line numbers cannot be accessed.

Note: Only IBM-seek is valid for internal files.

Seek with Unblocked Files

By default, all MTS files (non-internal files) are unblocked. An unblocked file contains one logical record per physical record (line) of the file. Each record number will correspond to a line number of that file. In IBM-seek mode, the record number "n" corresponds to the line number "n" (e.g., record 5 will be at line 5) and in MTS-seek the record number "n" corresponds to the line n/1000 (e.g., record 5 will be at line number 0.005).

Seek with Blocked Files

When the file is opened with the BLOCK option, as many logical records are stored in a physical record as is possible. If the size of a logical record is "b" bytes and that of the physical record is "m", and if $3b \leq m < 4b$, then 3 records will be stored per line. The maximum length of an MTS line is 32767 bytes.

In IBM-seek mode, the integer expression "n" in Seek(f,n) refers to the record number "n". In MTS-seek mode, specifying n*1000 does not refer to record number "n", but refers to line number "n" of the file "f". The following examples illustrate clearly the differences in the IBM- and MTS-seek modes with unblocked *MTS files*.

Both program fragments do a Seek and put 8 records in a record file; Example A seeks and puts alternately and Example B seeks once and puts twice.

Example A

The program fragment is:

```
For i := 1 to 8 Do Begin
    Seek(f,i);
    ...
    Put(f);
    End;
```

The tables give the line numbers and the record numbers of the records stored in that line.

IBM-seek		MTS-seek	
Records	MTS Line No.	Records	MTS Line No.
rec1	1.000	rec1	0.001
rec2	2.000	rec2	0.002
rec3	3.000	rec3	0.003
rec4	4.000	rec4	0.004
rec5	5.000	rec5	0.005
rec6	6.000	rec6	0.006
rec7	7.000	rec7	0.007
rec8	8.000	rec8	0.008

With unblocked files, only one logical record is stored per MTS line. With IBM-seek, Seek(f,i) followed by Put(f) puts record "i" in line number "i" for i = 1 to 8. With MTS-seek, Seek(f,i) followed by Put(f) puts record "i" in line number i/1000 for i = 1 to 8.

Example B

This example does a Seek and a Put and follows it up with a Put. That is, an indexed write followed by a sequential write. This is not recommended, especially for MTS-seek, unless the user is conversant with the MTS rules for mixing sequential and indexed writes. The program fragment is:

```
For i := 1 to 4 Do Begin
    Seek(f,i);
    ...
    Put(f);
    ...
    Put(f);
    End;
```

IBM-seek		MTS-seek	
Records	MTS Line No.	Records	MTS Line No.
rec1	1.000	rec1	0.001
rec3	2.000	rec3	0.002
rec5	3.000	rec5	0.003
rec7	4.000	rec7	0.004
rec8	5.000	rec8	1.000

Here, we have Seek(f,i) followed by Put(f) twice.

With IBM-seek, Seek(f,1) followed by 2 Puts stores rec1 and rec2 in lines 1 and 2. Seek(f,2) and two Puts store rec3 in line 2 (overwriting rec2) and rec4 in line 3. The process continues; finally, Seek(f,4) and two Puts store rec7 and rec8 in lines 4 and 5, overwriting rec6.

With MTS-seek, in Example B the Seek(f,1) seeks to line 0.001 and the first Put puts rec1. The next Put, for rec2, goes to line number 1.000 The sequential operation (Put without the Seek) following the indexed operation (Seek followed by Put) is done at a line number chosen according to the following MTS rule:

> For a file which has a default beginning line number and a default increment, lines written sequentially after an indexed write will be written at the next integral line number. For other cases, see "Sequential Operations with Line Files" in MTS Volume 1, *The Michigan Terminal System*.

Thus, in the example, rec3 goes to .002 and rec4 again to 1, by the same rule. Eventually rec7 goes to line number .004 and rec8 to 1.

Note: The choice of the line number where the first sequential write is done, after a line with an indexed write was completed, follows the MTS rules and is not a peculiarity of the Pascals on MTS.

While using MTS-seek it is a good idea not to mix sequential operations and indexed operations. Users who do this must be aware that subsequent line number choice after indexed operations depends on whether or not the file is blocked and on the MTS rule for sequential operations following indexed operations.

Seek performs no I/O, so the positioning is not actually completed until the next Get or Put is executed. As a result, a call to Seek immediately followed by a call to Read (without an intervening Get) will not produce what might be construed as the desired results. See Example 7 in the section, "I/O Examples," for a full example of how to correctly Seek within a file. Seek behaves this way to be compatible with the original IBM implementation.

Close

The predefined procedure Close will explicitly close a Pascal file, terminating I/O which is being done on it.

A call to the Close procedure has the form:

```
Close(f)
```

where "f" is a File variable.

All open files, which are simple, local variables within a procedure, are automatically closed upon termination of that procedure. All files used by a program are closed automatically upon termination of that program.

If the variable associated with an open file is destroyed before program termination, unpredictable results may arise. This could occur in the following cases:

(1) When the file variable is an element of an array.

(2) When the file variable is a field of a record.

(3) When the file variable is pointer qualified (exists on the heap).

(4) When a routine that contains local file variables is exited with a "goto" statement.

In these cases, a file could be left locked or there could be data left on the output buffer that did not get written to the file, etc. To avoid this, in these cases the file must be explicitly closed with a call to the Close procedure.

OPEN OPTIONS

There are a number of options which affect how I/O is done to Pascal files. These options can be specified by way of an option string given to the routine which is used to initialize the File variable associated with that file (see the section, "Predefined I/O Procedures and Functions," for complete descriptions of the predefined procedures Reset, Rewrite, and Update). This string is a variable or a constant, which contains one or more options, separated by commas.

The open options are:

FILE=fdname
UNIT=unitname
IFUNIT=unitname
INTERACTIVE
NOCC
NULLINE
UCASE
NOERROR
NOEMPTY
MAXLEN=length
IBMSEEK / MTSSEEK
BLOCK / NOBLOCK
WRAP

FILE=fdname

The FILE option specifies that the Pascal File variable is to be associated with the MTS file or device, "fdname". "fdname" may include MTS line-number ranges and I/O modifiers.

UNIT=unitname

The UNIT option specifies that the Pascal File variable is to be assigned to MTS logical I/O unit "unitname" (e.g., SPRINT, SCARDS, 1, ...).

IFUNIT=unitname

The IFUNIT option is equivalent to the UNIT option, unless "unitname" is not assigned, in which case all Pascal I/O will in effect be ignored and no MTS I/O will be done. Thus, "IFUNIT=unitname" is equivalent to "UNIT=unitname" or "FILE=*DUMMY*". Note: Unassigned units include those which have been defaulted, but for which no I/O has yet been done.

INTERACTIVE

The INTERACTIVE option specifies that the file is to be opened for interactive input. With this option, no initial Get is done when the file is Reset, and the value of the file pointer is initially set to nil (see the section, "Predefined I/O Procedures and Functions," for a complete description of the predefined procedure Reset and more information about interactive I/O). This option is useful for interactive conversations when the initial Get, which is normally done implicitly when a file is Reset, would cause the input and output done by the program to become out of synchronization.

This option has no meaning when used with internal files or files opened for output.

NOCC

Normally, the first character of each logical record in a Text file is inaccessible, having been reserved for a carriage-control character. The first character is normally a blank, but can be changed to the carriage control for page eject by the predefined procedure Page (see the section, "Predefined I/O Procedures and Functions," for a complete description of the Page procedure). If the NOCC option is specified, the first character of each logical record is no longer reserved. The first character the user writes becomes the first character of the logical record. This option is important for programs where carriage control is unwanted or where the user wants to take responsibility for providing his or her own carriage-control character.

This option has meaning only when used for output to files of type Text.

NULLINE

Normally, writing a null line (a line of length zero) results in a single blank being written to the output file. The use of this option allows zero-length lines to be written to a Text file. This can be used for deleting lines from a Text file. The use of NULLINE activates the NOCC option.

UCASE

The UCASE option causes I/O to or from an MTS file or device to be performed with the @UC modifier (see MTS Volume 1, *The Michigan Terminal System*, for a complete description of I/O modifiers). All lowercase characters input or output when using this option are translated to uppercase.

This option is only meaningful when used with files which contain character data (Text files, files of Char or packed array of Char) and can produce incorrect results if used for I/O to other file types. This option is ignored for internal files.

NOERROR

When the NOERROR option is specified, if errors such as "file xxx does not exist" are encountered, the errors are suppressed and the file variable is not opened. If this option is used, then IOSTATUS must be checked for the file variable before doing any I/O on it. For example,

```
REWRITE(f,'FILE=xxx,NOERROR');
```

If the file "xxx" does not exist, the file variable "f" is not opened. The function IOSTATUS(f) returns zero in this case.

This option applies only to Pascal/JB.

NOEMPTY

If a file variable is associated with an MTS file or device, that file is, by default, emptied when the file variable is initialized through a call to Rewrite (see the section, "Predefined I/O Procedures and Functions," for a description of the procedure Rewrite). If the NOEMPTY option is specified, the file is not emptied on a call to Rewrite.

MAXLEN=length

Default: MAXLEN=min(133,outlen)

The MAXLEN option is used to specify the maximum logical record length possible for output to a Text file. Normally, the maximum logical record length for a Text file is the width of the file or device to which it is attached, or 133 (whichever is smaller). "length" is an integer expression that becomes the maximum logical record length. When the maximum logical record length is exceeded on output, an error diagnostic is generated, and the excess text is placed on the next logical record of the file. This option allows the user's program to write text lines longer than the Pascal default length.

Note: MAXLEN includes the carriage control character.

IBMSEEK / MTSSEEK

Default: IBMSEEK

The IBMSEEK and MTSSEEK options control the behavior of the predefined procedure Seek when accessing MTS files (see the section, "Predefined I/O Procedures and Functions," for a description of the predefined procedure Seek). The Seek procedure requires an Integer argument to specify which logical record is sought. When using the IBMSEEK option (the default), the record number corresponds to the MTS line number which is to be sought. It is impossible to access fractional line numbers when using this option. When using the MTSSEEK option, the record number corresponds to the MTS line number times 1000 (e.g., if the number provided is 12, MTS line number 0.012 will be sought).

This option is relevant only when using the Seek procedure. Only IBMSEEK is valid when using internal files (see the section, "Internal Files"). See the description of BLOCK and NOBLOCK options for more information relevant to record files.

BLOCK / NOBLOCK

Default: NOBLOCK

The BLOCK option affects the behavior of output to record files. Program efficiency is improved if fewer I/O operations can be done to MTS files and devices, and fewer calls are made to allocate space for internal files (see the section, "Internal Files," for a complete description). As a result, when the BLOCK option is used in record files, more than one logical record is stored in each physical record. The NOBLOCK option causes only one logical record to be put on each physical record of a record file. The default is NOBLOCK for MTS files.

WRAP

The WRAP option causes lines which are longer than the maximum record length of a file (see the description of the MAXLEN option) to be written on several physical records. The use of this option eliminates the error diagnostic normally associated with this situation. This option is only applicable to Text files.

GUIDELINES FOR BEGINNERS

Here are some guidelines on I/O for beginners:

(1) Always do a Reset before reading in input.

(2) Use Readln wherever possible instead of Read. Note: Eof is not set to True when a call to Read of Interger or Real encounters an end-of-file.

(3) Always do a Get before checking Eof in interactive mode.

I/O IN COMPILE-LOAD-AND-GO MODE

In both Pascal/JB and Pascal/VS, calls to the predefined procedure Reset normally result in a Rewind being issued for the file or device attached to the Pascal file variable specified in the Reset call. However, this is inappropriate in Compile-Load-and-Go mode (available only with Pascal/JB), where the program data may be part of the source stream; the Rewind would rewind the source stream to the beginning of the source *program*. Therefore, in CLG mode, a call to Reset for the standard file variable Input, which defaults to the source stream, will result in the input file being repositioned to the first line following the /EXECUTE or /DEBUG control statement. This special repositioning is suppressed if the Reset call (or a prior Reset call for the file variable Input) specifies the UNIT, IFUNIT, or FILE open options or if Input has been redirected away from the source stream in some other manner, for instance, using a $CONTINUE WITH line in the source stream.

Similarly, the Rewrite predefined procedure normally results in an Empty of the file or device attached to the Pascal file variable specified in the Rewrite call. This is inappropriate in CLG mode where the compilation listing and output from the standard file variable Output are normally directed to the same file attached to the logical I/O unit SPRINT; the Empty would erase the compilation listing. Therefore, in CLG mode, Pascal/JB suppresses the EMPTY on Rewrite calls for the standard I/O unit Output unless the UNIT, IFUNIT, or FILE open options have been specified in a call to Rewrite for Output.

I/O EXAMPLES

Example 1

Reading and Writing Using Input and Output

```
Program SimpleIO(Input,Output);

{SimpleIO echoes Input to Output until it finds the
 sentinel.  It assumes there are no long lines.}

Var
  Data: String(255);
Begin
Read(Data);
While Data <> '**END OF DATA**' do begin
   Writeln(Data);
   Read(Data);
   End;
End.
```

By default, data read from the predefined file Input are read from the MTS logical I/O unit SCARDS, and data written to the predefined file Output are written to the MTS logical I/O unit SPRINT. By default, SCARDS is assigned to the MTS pseudo-device *SOURCE*, and SPRINT is assigned to the MTS pseudo-device *SINK*.

If this example were compiled and the object put into the file –OBJ, the following command:

```
$RUN -OBJ T=3
```

would read lines from *SOURCE* and write them to *SINK*. The default behavior of this command, when used in batch, is to read from the input records (cards) which follow the $RUN command, and write to the printer where the rest of the batch job is printed.

The logical I/O units SCARDS and SPRINT can be reassigned on the program $RUN command. For example:

```
$RUN -OBJ SCARDS=Allens.data SPRINT=*PRINT* T=3
```

would read lines from the MTS file “Allens.data” and write lines to the MTS pseudo-device *PRINT*, sending the output to a printer.

Example 2

Reading and Writing Using Declared File Variables

```
Program MergeFiles;

{MergeFiles appends lines from the file attached to MTS
 logical I/O unit 2 to those from the file attached to MTS
 logical I/O unit 1.  The concatenated lines are written
 to the file CCID:MergedLines.  It stops when it reaches
 the end of either file.  It assumes short lines.}

Var
  line1, line2: string(255);
  file1, file2, out: text;
Begin
  Reset(file1,'unit=1');
  Reset(file2,'unit=2');
  Rewrite(out,'file=CCID:MergedLines,nocc');
  While not (eof(file1) or eof(file2)) do begin
    Readln(file1,line1);
    Readln(file2,line2);
    Writeln(out,line1||'|'||line2);
    End;
End.
```

If this program were compiled and the object put into the file –OBJ, the following $RUN command:

```
$RUN -OBJ 1=CCID:Leftside 2=CCID:Rightside T=3
```

would read from the files “CCID:Leftside” and “CCID:Rightside,” and produce output in the file “CCID:MergedLines.”

Note that when using this scheme, the names of the input files can be different every time the program is run, but the name of the output file is constant and can only be changed by changing and recompiling the program. The input files must be specified every time, however, and the output file need never be specified. This illustrates a basic trade-off between the use of MTS logical I/O units and the use of constant file names that are built into a program.

Example 3

Interactive Terminal I/O

```
Program ImmaturityPersonified;

{A silly little program}

Var
  inp, out: text;
  count: 1..5;
  name: string(255);
Begin
  Reset(inp,'file=*msource*,interactive');
  Rewrite(out,'file=*msink*');
  Writeln(out,'What is your name?');
  Get(inp);
  While not eof(inp) do begin
    Readln(inp, name);
    For count := 1 to 5 do
    Writeln(out,name,'''s Pascal programs never compile');
    Writeln(out, 'nya nya nya nya nyaa nya');
    Writeln(out, 'give me another name ...');
    Get(inp);
    End;
End.
```

This program makes use of the INTERACTIVE option, when reading input from the user. Since the default behavior of the Pascal compiler is to read in the first record from a file at the time that it is reset (so that the file variable may have valid data to point to), problems arise when the data read has to be in response to a prompt that is written by the program. If the call to Reset, which initializes the file being read from, comes before the prompt, the program tries to read from the user before the user has been told what data to provide. Using the INTERACTIVE option inhibits this initial read and sets the initial value of the file variable to nil, providing the correct synchronization of input and output.

Pascal/VS and Pascal/JB read or write an entire record at a time. As a result, calls to Write for Text files put information into a buffer that is written out all at once when the logical record is terminated (through a call to Writeln). This program uses Writeln to write its prompts to ensure that the prompt is written at the correct time.

The initial value of a file variable opened with the INTERACTIVE option is nil, so the end-of-file condition cannot be checked for until after the first input operation is done on the file. Therefore, the first input operation "Get(inp)" is done outside of the While-loop.

A call to "Readln(inp,name)" causes a "Get(inp)" to be issued followed by an assignment to "name." If the previous "Get(inp)" caused an end-of-file condition, the attempted assignment to "name" will result in an error condition (the end-of-file condition causes the file variable to be assigned the value of nil and the whole thing gets very ugly). Since a call to Readln on a file opened with the INTERACTIVE option causes the end-of-line condition to become true, but does not read in the next record, the next record can be read in and the end-of-file condition tested for through a "Get(inp)." After this Get, the file variable will be left pointing at the first character in the new line (assuming the end of file was not found) and "Readln(inp,name)" can be used to read in the value of "name."

"*MSOURCE*" is defined to be the master source (or input) file or device. Here it refers to the user's terminal. "*MSINK*" is defined to be the master sink (or output) file or device. Here it refers to the display screen of the terminal. For more information on these, please see MTS Volume 1, *The Michigan Terminal System*.

Note: A certain amount of trouble must be taken to write a prompt and have the response printed on the same line. Simple use of the procedure Write will not produce the desired results. The MTS subroutine CUINFO can be used to change the user's execution prefix to the desired prompt or an "&" carriage control can be used in some situations (the latter solution produces less consistently correct results).

In the above example, to use the "&" carriage control to have the response read from the same line as the prompt, the following changes will be required:

Replace the Rewrite statement with

```
rewrite(out,'file=*msink*,nocc');
```

and replace the two prompt statements respectively with

```
writeln(out,'& What is your name?  ');
writeln(out,'& Give me another name ...  ');
```

This will read the response from the same line as the prompt.

Example 4

Use of Get and Put With Text Files

```
Program BlankRobber(Input,Output);

{BlankRobber reads one character at a time and writes
 the nonblanks. It produces one Output line per Input
 line.}

Begin
Reset(input);
While not eof(input) do begin
  If input@ <> ' ' then write(input@)
     else if eoln     then writeln;
  Get(input);
  End;
End.
```

In this program, characters are read, one by one, by way of the predefined procedure Get. Since the file Input is opened by default only when the first call to Read, Readln, or Get is made, initial testing of Eof or Eoln, before any such calls, would be an error. So, in this example, the file Input is opened explicitly without the INTERACTIVE option (see the section, "Predefined File Variables," for a description of the predefined file variable Input).

The program terminates when the end-of-file condition occurs for the file Input. If the program were compiled and the resultant object put in a file –OBJ, the following cards, within a batch job:

```
$RUN -OBJ T=1
     All Work And No Play Makes Jack A Dull Boy.
     A Stitch In Time Saves Nine.
$ENDFILE
```

would produce the following output:

```
AllWorkAndNoPlayMakesJackADullBoy.
AStitchInTimeSavesNine.
```

Example 5

Reading and Writing Numeric Data

```
Program ReadABunchOfNumbers(Input,Output);

{Read and echo two numbers per line using input field
 widths.  Check that there is nothing else on the lines.}

Var
  num1,num2: integer;
  garbaggio: char;
Begin
  Reset(input);
  While not eof do begin
    Read(num1:3, num2:4);
    Write('num1=',num1:1, ', num2=',num2:1);
    If not eoln then begin
      Read(garbaggio);
      Write(' and the remaining garbage was ',garbaggio);
      End;
    Readln;
    Writeln;
    End;
End.
```

If the following data were given for this example:

```
1234567890
1234567abc
12345a
 1234567890
```

The following output would be produced:

```
Num1=123, Num2=4567 and the remaining garbage was 8
Num1=123, Num2=4567 and the remaining garbage was a
Num1=123, Num2=45
Num1=12, Num2=3456 and the remaining garbage was 7
```

This example demonstrates the several ways in which the use of field widths affects input. The first two lines read produce fairly predictable results. A field of three characters produces the value of Num1 on the third line. When reading Num2, a nonnumeric is encountered, ending conversion at that point. In order to satisfy the field width, the file variable is advanced to the end of the four character field, setting the end-of-line condition to true. As a result, no trailing garbage is read in. The output produced after reading the fourth line indicates that leading blanks are significant. The first character of the three-character field from which Num1 is read is a blank which is ignored. The remaining two characters are read into Num1. The rest of the line is processed in a predictable fashion.

Example 6

Using Record Files

```
Program FindExpensiveParts;

{Scan the inventory file, MTS unit 0, for parts
 costing over $100.  Copy these to the expensive
 file, MTS unit 1).}

Type
  partfile = file of record
    cost: real;
    name: alpha;
    pieces: integer;
    end;
Var
  inventory,
  expensive: partfile;
Begin
  Reset(inventory,'unit=0');
  Rewrite(expensive,'unit=1');
  While not eof(inventory) do begin
    If inventory@.cost>100.00 then
       write(expensive,inventory@);
    Get(inventory);
    End;
End.
```

This example illustrates basic use of record files. An inventory of parts is read from MTS logical I/O unit 0 and those records which contain parts whose cost is greater than $100.00 are written to MTS logical I/O unit 1.

Example 7

Use of the Seek Procedure

```
Program DidTheyRegister(output);

{Read a list of student numbers from SCARDS and see if they
 have registered. This is read non-interactively, from a
 file assigned to SCARDS at run-time.
 Use the student data file assigned to unit 0
 and indexed by student number, which is also the MTS line
 number.}

Type
  char50 = packed array [1..50] of char;
  data = record
    sex: (male, female);
    name: char50;
    hold,
    registered: boolean;
    end;
Var
  check: text;
  student: file of data;
  number: integer;
Begin
  Reset(student, 'unit=0,mtsseek');
  Reset(check, 'unit=scards');
  While not eof(check) do begin
    Readln(check, number);
    Seek(student, number);
    Get(student);       { seek does no I/O, sigh }
    If eof(student) then begin
      writeln('! error !');
      writeln(number:1,' does not exist');
      end
    else begin
      write(number:1,': ');
      if student@.sex=female then write('s');
      write('he is ');
      if not student@.registered then write('NOT ');
      writeln('registered.');
      end;
    End;
End.
```

This example illustrates basic use of the Seek procedure. Records are stored in a data file, indexed by social security number. For example, a person with social security number 275-60-7683 would be stored at MTS line number 275607.683 in this data file. A list of social security numbers is read from MTS logical I/O unit SCARDS, and their corresponding entry in this file is sought. If no entry is made at that position, the end-of-file flag will be set. An appropriate error message is printed out in this case. Otherwise, the information contained in that record is extracted and operated upon.

Example 8

Use of the Seek Procedure with Text Files

```
Program Getinfo;

{Information stored is a list of names - of lengths
 20 or less - and is accessed by specifying the MTS
 line numbers. The file with the information is assigned to
 SCARDS at run-time. Use of seek with text files.}

Var
  name : string(20);
  line_nr : integer;
  infolist : text;

Begin
  Reset(infolist,'unit=scards,MTSSEEK');
  Reset(Input,'File=*Msource*,Interactive');
  Writeln(' Enter line   Number');
              /* Enter MTS line number*1000 for MTSSeek*/
  Get(input);
              /* Since Input is interactive Get is necessary
                before testing EOF */
  While not eof(input) do begin
    Readln(line_nr);
    Seek(infolist,line_nr);
    Readln(infolist);
              /* Position the file pointer  */
    If not EOF(infolist) then begin
      Read(infolist,name);
      Writeln(' The name on the list is  ', name);
      end
    else writeln('no entry for that number');
    Writeln('enter next number');
    Get(input);
  End;
End.
```

This example illustrates the use of the Seek procedure with Text files. Information is stored in a Text file assigned to SCARDS at run-time. The line numbers are read in interactively and the information in line-number*1000 is retrieved.

PREDEFINED FUNCTIONS AND PROCEDURES

For Pascal/JB, the following functions and procedures are built into the compiler and can be called directly from the user program. No procedure declarations or include statements are needed except for OnError, which needs a /Include Onerror statement, to include its definition from *PASCALJBINCLUDE.

For Pascal/VS, declarations for these predefined functions and the procedure OnError are provided in *PASCALVSINCLUDE. These can be included in the user program by using the %INCLUDE statement and assigning the next available (that is, unassigned) unit among logical units 1 through 8 to *PASCALVSINCLUDE.

For Pascal/VS, the procedure/function bodies for the following, except for OnError, are contained in *PASCALVSLIB, which must be explicitly concatenated to the object program in the $RUN command. OnError does not need this concatenation. For Pascal/JB, the procedure and function bodies are automatically loaded when the object program is executed ($RUN).

DESCRIPTIONS

Itohs

The Itohs function is declared as

```
Function Itohs(i:Integer):String(8); external;
```

This procedure takes an expression of type Integer as input and returns the hexadecimal representation of that integer. For example,

```
Itohs(100)
```

returns the value “00000064”.

Lpad

The Lpad procedure is declared as

```
Procedure Lpad(var s:String; len:Integer; c:Char); external;
```

This procedure pads or truncates the String variable “s” on the left until the length is equal to “len”. If the length is greater than “len”, the characters to the left are truncated; if the length is less than “len”, “s” is padded on the left with the padding character “c”. For example:

```
Lpad(s,10,'*')
```

where

```
s := 'abcde'                gives '*****abcde', and
s := 'abcdefghijklmn'       gives 'efghijklmn'
```

Rpad

The Rpad procedure is declared as

```
Procedure Rpad(var s:String; len:Integer; c:Char); external;
```

This procedure pads or truncates the String variable "s" on the right until the length is equal to "len". If the length is greater than "len", the characters to the right are truncated; if the length is less than "len", "s" is padded on the right with the padding character "c". For example:

```
Rpad(s,10,'*')
```

where

```
s := 'abcde'                  gives 'abcde*****', and
s := 'abcdefghijklmn'         gives 'abcdefghij'
```

Picture

The Picture function is declared as

```
Function Picture(const p:String; r:Real): String(100); external;
```

This procedure is used for formatting of real numbers. The feature is similar to that of Cobol and PL/I.

The parameter "r" contains the real number which is to be formatted. The parameter "p" gives the picture specification.

The picture specification consists of 2 fields, the decimal and the exponent fields. Of these, the exponent field is optional and the decimal field is required. The decimal field is made up of an integer part and a fractional part. Of these, the fractional part is optional. For example,

```
's99.v99'          {integer and fractional parts}
'9v.99es99'        {integer, fractional, and exponent parts}
'9999'             {integer part only}
```

The picture specification characters may be in upper- or lowercase.

Picture Specifications

Digits and Decimal-Point Characters:

9 The position in the string where 9 occurs is to correspond to a decimal digit in the output. For example,

```
Picture('9999',1.0) gives '0001'
```

V This divides the decimal field into the integer part and the fractional part. A decimal point will be assumed at this position. The decimal point itself is not inserted in the output by the use of V.

The assigned value is truncated or extended on either side with zeros, if necessary.

The absence of a V character is equivalent to an implied V at the right end of the decimal field.

```
'999V.99'          123.456   gives '123.46'
'999V.99'          123.4     gives '123.40'
'999V.99'          23.4      gives '023.40'
'999V99'           123.456   gives '12346'
'99V.99'           123.456   gives '23.46'
```

Zero-Suppression Characters:

These are used to replace leading zeros. Z specifies that a blank is to replace a leading zero, if one is found in that position. "*" specifies that a "*" is to be used as replacement character for a leading zero, if one exists at that position.

```
'ZZZZ9'            8.0       gives '    8'
'****9'            8.0       gives '****8'
'****9'            323.8     gives '**324'
'*****'            0.0       gives '*****'
'ZZZZ9'            323.4     gives '  323'
```

Insertion Characters:

The character is inserted at the specified position in the string. Insertion is done if no zero suppression is taking place at that position.

A comma, period, or B (for a blank) is used as an insertion character.

The period is not in any way connected with decimal point alignment, which is the sole function of the V character.

```
'ZZZ,ZZ9'          1.0       gives '      1'
'ZZZ,ZZ9'          1234.0    gives '  1,234'
'9B9B9B9B9'        1234.0    gives '0 1 2 3 4'
'9B9B9B9'          1234.0    gives '1 2 3 4'
'9.9.9.9'          1234.0    gives '1.2.3.4'
'ZZZZV.99'         0.45      gives '    .45'
'ZZZZ.V99'         0.45      gives '     45'
```

Sign and "$" Characters:

The indicated characters are to appear in the specified positions under the following conditions:

+ specifies that a plus sign is to be used in that position if the number is nonnegative, and a blank is to be used if the number is negative.

– specifies that a blank is to be used in that position, if the number is nonnegative, and a minus sign is to be used if the number is negative.

S specifies that a plus sign is to be used in that position if the number is nonnegative, and a minus sign is to be used if the number is negative.

$ specifies that a dollar sign is to be used in that position.

Multiple use of these characters indicates that leading zeroes are to be suppressed.

```
'$$$,$$9V.99'      456.78    gives '    $456.78'
'-zz,zz9V.99'      -456.78   gives '-   456.78'
'---,--9V.99'      -456.78   gives '   -456.78'
'---,--9V.99'      456.78    gives '    456.78'
'S9999'            123.0     gives '+0123'
'+9999'            123.0     gives '+0123'
'+9999'            -123.0    gives ' 0123'
```

Exponentiation Characters:

E specifies the exponent field delimiter. The corresponding position is to contain the character E and it indicates the start of the exponent field.

K The exponent field is to the right of the fraction and no character is included.

```
S9V.9999ES99      1.2345    gives +1.2345E+00
S9V.9999KS99      1.2345    gives +1.2345+00
```

The exponent field must be the last field in the picture specification parameter.

OnError

When an execution time error occurs, the procedure OnError is called to perform any necessary action prior to generating a diagnostic. A default OnError routine, which does nothing, is provided in the Pascal library.

One may write one's own version of the routine and declare it as an external procedure. This procedure will be invoked when an error occurs, so a decision as to how the error is to be handled can be made depending on the error.

*PascalVSinclude and *PascalJBinclude contain the necessary procedure declaration and datatype declarations for using Onerror. The contents of the include file that has the information relevant to formulating the OnError routine are shown in the table on the next page.

The Pascal/VS error diagnostic messages are of the form AMPX, followed by a 3-digit number, followed by an S, E, or I. The Ferror parameter refers to the 3-digit number that is associated with the specific run-time error. A list of all the errors, numbers and the messages can be found in the IBM publication, *Pascal / VS Programmer's Guide*, SH20-6162.

In Pascal/JB, there are no error numbers associated with the error messages but the run-time error numbers used with OnError coincide with the corresponding ones used by Pascal/VS.

Upon entry to OnError, the parameter Ferror contains the number that corresponds to the error encountered. Fmodname, Fprocname, and Fstmtno refer, respectively, to the segment, the procedure/function, and the statement number of the location of the error. Faction determines the course of action to be taken. Upon invocation of OnError, Faction will contain the value of the default action that will take place after OnError returns. This information can be examined and a decision made regarding whether the error is to be handled by the user or whether the default action is to be taken.

The Faction parameter may be modified as desired. If Faction is set to Xumsg, Fretmsg must also be assigned the text of the required message.

See Example 1 at the end of this section.

Contents of the Include section OnError:

```
     { RUN-TIME ERROR INTERCEPTION ROUTINE }
Type
  Errortype = 1 .. 90;       { range of execution errors }
  Erroractions = (           { action to be performed }
    Xhalt,                   { terminate program }
    Xpmsg,                   { print Pascal's diagnostic }
    Xumsg,                   { print user's message }
    Xtrace,                  { produce traceback }
    Xdebug,                  { invoke the debugger}
    Xdecerr);                { decrement error counter }
  Errorset = Set of Erroractions;
Procedure OnError(
  const Ferror : Errortype;  { error number }
  const Fmodname : Alpha;    { module name where occurred}
  const Fprocname : Alpha;   { procedure where occurred }
  const Fstmtno :  Integer;  { statement  where occurred }
  var Fretmsg : String;      { returned user's message }
  var Faction : Errorset);   { action modified by ONERROR }
External;
```

In Pascal/JB, this list is preceeded by the two statements

```
/CHECK PUSH
/CHECK NOINIT
```

and succeeded by the following statement

```
/CHECK POP
```

This is done to make OnError independent of the current setting of the INIT CHECK compile option. For an explanation of these CHECK options, see the section, “Pascal/JB Control Statements.”

Example 1

User Error Interception with OnError Routine

```
Program Intercept_Error;

 {Intercept_Error reads two integers per line.  In case
 of invalid data, it prints a message and ignores the
 line.  Errors 54 ... 56 refer to end-of-file while reading
 integer and encountering invalid data in place of an Integer.}

%Include OnError;
Var
  i,j: Integer;
  valid : Boolean;

Procedure OnError;
Begin
  if ferror in (.54, 55, 56.) then begin
    faction := (.xumsg.);
    fretmsg := ' illegal or insufficient data ';
    valid := false;
    end;
End;

Begin
  reset(input);
  while not eof do begin
    valid := true;
    Readln(i,j);
    if valid then begin
      { ... process input ... }
      end
    end;
End.
```

STORAGE MAPPING

This section describes the rules that the Pascal compilers employ in mapping variables to storage locations.

AUTOMATIC STORAGE

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

INTERNAL STATIC STORAGE

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

DEF STORAGE

Each Def variable which is initialized by means of the value declaration will generate a named control section (csect). Each Def variable which is not initialized will generate a named common section. Each Def variable becomes a named common block 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.

DYNAMIC STORAGE

Pointer-qualified variables are allocated dynamically from heap storage by the procedure New. Such variables are always aligned on a doubleword 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

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

PREDEFINED TYPES

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

Datatype	*Size (bytes)*	*Boundary Alignment*
Alfa	8	byte
Alpha	16	byte
Boolean	1	byte
Char	1	byte
Integer	4	fullword
Shortreal	4	fullword
Real	8	doubleword
String(len)	len+2	halfword
Stringptr (with P/VS)	8	fullword
Stringptr (with P/JB)	4	fullword

ENUMERATED SCALERS

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

SUBRANGE SCALARS

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

Given a type definition T as:

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

and

```
const
        i = ord(i);
        j = ord(j);
```

Range of I .. J	*Size (bytes)*	*Boundary 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

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.

RECORDS

An unpacked record is aligned on a boundary in such a way that every field of the record is properly aligned on its required boundary. That is, records are aligned on the boundary required by the field with the largest boundary requirement.

For example, record A below will be aligned on a fullword because its field A1 requires a fullword alignment; record B will be aligned on a doubleword 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 {doubleword aligned}
                B1 : a;
                B2 : Real;
                B3 : Boolean
        end;

        C = record {byte aligned}
                C1 : packed 0..255;
                C2 : alpha
        end;
```

Packed records are always aligned on a byte boundary;

ARRAYS

Consider the following type definition:

```
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 elements 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
```

FILES

File variables occupy 64 bytes in Pascal/VS and 68 bytes in Pascal/JB and are aligned on a fullword boundary with both the compilers.

SETS

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

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

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

has the base type defined by '0'..'9', and a fundamental base type of Char.

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

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

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

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

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

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

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

```
var
    C : set of (c01,c02,c03,c04,c05,c06,c07,c08,
                c09,c10,c11,c12,c13,c14,c15,c16);
    D : set of c01..c08;
    E : packed set of c01..c08;
```

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

Thus, variables U and V below will both occupy 256 bits; variable W will occupy 11 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 below indicates how the set will be mapped in storage.

Range of M	*Size (bytes)*	*Boundary Alignment*
$1 \le M \le 8$	1	byte
$9 \le M \le 16$	2	halfword
$17 \le M \le 24$	3	byte
$25 \le M \le 32$	4	byte
$33 \le M \le 256$	(M+7) div 8	byte

SPACES

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

```
var     s : space[1000] of Integer;
```

SEPARATE COMPILATION ISSUES

When an entire Pascal program is not self contained, there must be some communication between the separate compilations. This communication can be broadly divided into two categories:

(1) Calls to or from separately compiled units

  (a) A Pascal procedure calls a Pascal procedure
  (b) A Pascal procedure calls a non-Pascal procedure
  (c) A non-Pascal procedure calls a Pascal procedure

(2) Access to data in separately compiled units

  (a) Use of Def variables to communicate between Pascal units
  (b) Use of Ref variables to communicate with non-Pascal units
  (c) Variables defined in the outer environment

In case (1a) above, a Pascal procedure calls a separately compiled Pascal procedure. The basic separate compilation units in Pascal are the program and the segment. A program corresponds to the main program in which execution is initiated. A segment contains data and/or procedures, but no main program.

To access the procedures in a separate compilation unit, it is necessary to declare the procedure header in each unit that calls it. A simple example follows:

Example 1

```
Program example;
  Function cube(x: Integer): Integer; external;
  Var z : Integer;
Begin
  Reset(Input);
  While not Eof(Input) do begin
    Readln(Input,z);
    Writeln(Output, cube(z))
    End;
  End.

////////// A separate compilation unit ////////////

Segment cubeseg;
  Function cube(x: Integer): Integer; external;
  Function cube;
Begin
  Cube := x * x * x;
  End;. {note the semicolon followed by a period}
```

External names refer to names of variables and routines that are accessed across different compilation units.

Loader names are limited to eight characters by the MTS loader. Pascal/VS, however, allows external names to be longer than eight characters. Thus, the loader name associated with an external variable or routine consists of the first eight characters of the declared name of the variable or routine. Such name truncation may produce unexpected results. For this reason, Pascal/JB gives a warning at compilation time that the external name will be truncated to form the loader name.

In Pascal/JB, the default loader name may be overridden for Fortran-type external routines by placing the loader name within parentheses, following the FORTRAN directive. For example,

```
Procedure MTSCOMMAND; FORTRAN(CMD);
```

The loader name should conform to the Pascal identifier syntax and will be translated to uppercase.

Alternatively, the loader name may be a string or a hex-string constant, in which case the loader name will be *exactly* as specified by the string. For example,

```
Procedure OddBall; FORTRAN('XYZ@A');
```

Pascal/JB will generate a warning if the loader name is truncated to eight characters, regardless of the manner in which the loader name was specified.

Rule 1: External names should be no more than eight characters.

Program and segment names are used for the control section names. These names should be unique. Both Pascal/VS and Pascal/JB have a strange little quirk that allows a segment name to be the same as an external procedure within that segment. This "feature" should not be used as it can cause severe problems if an error is made in declaring the procedure.

Rule 2: External names should be unique.

Note that the same external procedure declaration must appear in both the calling compilation unit and the called compilation unit. If these declarations should differ in any significant way (e.g., the number or types of parameters or return value), the program will not run correctly. Unfortunately, there are no facilities provided for checking such consistency across compilation unit boundaries. To ensure that the declarations are the same, and to save typing, it is best to make one version of the declarations and then copy the declarations into each compilation unit by means of the %INCLUDE directive.

Rule 3: Use %INCLUDE to copy shared declarations.

Def Variables

It is possible to share variables between compilation units in much the same way that Fortran does with Common. If a variable is declared as a Def variable in more than one compilation unit, there will be only one copy of it and all compilation units will refer to that one copy. Since the variable name will become an external symbol, it is important to observe the advice given above; external names should be no more than eight characters, external names should be unique, and %INCLUDE should be used to copy shared declarations. The preceding example is repeated here, but in this case it passes the parameter as a Def variable instead of as a parameter. A real program should use %INCLUDE to copy the declarations, but for clarity, this example shows the common declarations directly.

Example 2

```
Program example;
  Function cube: Integer; external;
    def z : Integer;
  Begin
    reset(input);
    While not Eof(Input) do begin
      Readln(Input,z);
      Writeln(Output, cube)
      End
    End.
 /////////// A separate compilation unit ////////////

 Segment cubeseg;
   Function cube: Integer; external;
   Def z : Integer;
   Function cube;
 Begin
   Cube := z * z * z;
 End;.
```

Use of Ref Variables

Ref variables are used to make Pascal variables point to some storage that is not defined in the Pascal program or segments. There are very few occasions to use Ref, and Ref should not be used to communicate between Pascal programs. There is one use of Ref, however, that can be handy; that is with Pascal/VS, when calling non-Pascal procedures which yield a return code. See the description of PvcallRC for a discussion of how to use it. With Pascal/JB, there is a predefined function, FortranRC, which gives the return code from the previous call to a Fortran routine.

Rule 4: Use Def (not Ref) to communicate between Pascal units.

Sharing Global Variables Between Compilation Units

One of the most error-prone features of Pascal is that Var declarations at the outer level in segments overlay any outer level Var declarations in the program unit and in any other segments. This feature is useful in writing reentrant programs, generally only of interest to systems programmers installing a Pascal program in shared memory. If all of the outermost declarations match, then this method could be used to share variables between compilation units. Since Def variables already do this quite well and since this feature is extremely error-prone, it should not be used.

Pascal/JB issues a warning about outer level Var declarations in segments.

Rule 5: Never make outer-level Var declarations in a segment.

USE OF THE %INCLUDE DIRECTIVE IN MTS

When the Pascal compiler encounters a %INCLUDE directive, it replaces the %INCLUDE directive with the text specified by the identifier that follows the "%INCLUDE" string.

The general form of the %INCLUDE directive is:

%INCLUDE identifier

where "identifier" specifies the text that is to be inserted into the program the compiler is reading.

The text to be inserted is stored in a library file. The format of %INCLUDE libraries is similar to that of 370/Assembler macro libraries.

The beginning of the library contains a dictionary. Each line of the dictionary contains an identifier, followed by an MTS line number. The line number indicates the first line of the text to be associated with the identifier. The string "00000000" terminates the dictionary. The remainder of the file contains the various sections of text which are to be %INCLUDEd, located at the correct line numbers within the file, and terminated by a $ENDFILE.

Alternately, the identifier may refer to an MTS file name. In that case the contents of the entire file are inserted into the text, and the file should not contain a dictionary with line numbers.

The text to be included can contain anything, but the %INCLUDE directive is particularly useful for ensuring that declarations of routines and types are consistent between separately compiled sections of code. When an external procedure has parameters that are declared differently in the segment where that procedure is defined and in the segment where it is called, the compiler usually produces incorrect results or a program exception, followed by an unhelpful error message. Use of the %INCLUDE directive can save a lot of time and headaches when working with separately compiled modules.

As noted in the sections, "Compiling and Running Pascal/JB" and "Compiling and Executing Pascal/VS Programs," the %INCLUDE libraries are assigned to logical I/O units 1 to 8 when compiling programs which use them. When using units 1 to 8, Pascal/VS stops with the first unassigned unit. Pascal/JB searches all the units.

Example 3

```
 Line    Line
Number Contents

  1.000 search    200.000
  2.000 gasp      300.000
  2.500 types     450.100
  3.000 00000000
    ...
200.000 Procedure Search (target : Alpha;
201.000                   tree : Treeptr;
202.000                   var found : Boolean);
202.500   external;
203.000 $ENDFILE
    ...
300.000 Procedure Gasp (epithet : char255;
301.000                 var epithet_file : Text);
302.000   external;
303.000 $ENDFILE
    ...
450.100 Type
451.000   Tree_node = record
452.000     contents : char255;
453.000     lchild   : @ Tree_node;
454.200     rchild   : @ Tree_node;
455.000     end;
456.000   Head_node = @ Tree_node;
457.000 $ENDFILE
```

INTER-LANGUAGE COMMUNICATION

It can be desirable to invoke subprograms (procedures) written in other programming languages in order to obtain services not available directly in the Pascal compilers. It also can be desirable to have a Pascal procedure called from a non-Pascal program in order to take advantage of Pascal in an existing application without rewriting the entire application. This chapter will discuss the options available and what must done to have this flexibility.

We can divide inter-language communication into two classes:

(1) The Pascal procedure is the calling procedure and the non-Pascal procedure is being called.

(2) The Pascal procedure is called from a non-Pascal calling procedure.

Note: Modules and procedures written in Pascal/VS and Pascal/JB cannot be mixed. All the modules must be compiled using the same compiler. If it is desired to access the routines that already exist with one of the Pascals from the other, the desired procedure must be declared as a Fortran routine in the calling program. The routine itself must be declared with the Main or Reentrant directive.

FORTRAN

With Pascal as the calling language:

Define procedures and functions in Pascal using the Fortran directive. This enables a subprogram written in Fortran to be called.

With Pascal as the called language:

Use a CALL statement in Fortran to call the Pascal procedure. The Pascal procedure must be defined with the Main or Reentrant directive. After the last call to a Pascal procedure, Psclhx(0) (Pascal halt execution) must be called in order to release the virtual memory implicitly obtained by invoking a Pascal procedure.

370/ASSEMBLER

With Pascal as the calling language:

Define procedures and functions using the Fortran directive.

With Pascal as the called language:

Use a V-type constant in the 370/Assembler routine to define the Pascal entry point. The Pascal procedure must be defined as Main or Reentrant. After the last call to a Pascal procedure, Psclhx(0) must be called in order to release the virtual memory implicitly obtained by invoking a Pascal procedure.

PL/I

With Pascal as the calling language:

Define the PL/I procedures and functions in the Pascal program using the Fortran directive. This enables a subprogram written in PL/I to be called. The PL/I procedure must be defined with the Fortran option. Consult MTS Volume 7, *PL/I in MTS*, for more details.

With Pascal as the called language:

Declare the Pascal procedure with the Main directive. Use a call statement in PL/I to call a Pascal procedure. The PL/I procedure should specify the Pascal procedure as External. After the last call to a Pascal procedure, Psclhx(0) must be called in order to release the virtual memory implicitly obtained by invoking a Pascal procedure.

PASSING PROCEDURE OR FUNCTION PARAMETERS

With Pascal/VS, a procedure that is declared as a Fortran procedure cannot have procedures or functions passed as parameters. This restriction does not apply to Pascal/JB.

Note: With both compilers, there are restrictions about passing Standard procedures as parameters to any procedure or function. Please refer to the IBM publication, *Pascal/VS Programmer's Guide*, SH20-6168, for more information.

EXAMPLES OF INTER-LANGUAGE COMMUNICATION

Some examples of communication between Fortran and Pascal follow. More examples and examples of communication between Pascal and Assembler and PL/I can be found in the IBM *Pascal/VS Programmer's Guide*.

Example 1

Calling Fortran from Pascal

```
Program Change_That_Prefix;

{Change the execution prefix to a character of the
 user's choice.  Although it's easier with Pvpfxget
 and Pvpfxset, we'll use SETPFX from MTS Volume 3.}

Var
  newpfx : Char;
  result : Char;
Function Setpfx (const newpfx : Char) : Char; Fortran;
Begin
  Writeln('Enter new prefix:');
  Read(newpfx);
  Result := Setpfx(newpfx); {set the new pfx}
    ...
  Result := setpfx(result); {restore old pfx}
End.
```

When an external function is declared with the Fortran directive, the function result is assumed to be returned in GR0 or FR0 (as with Fortran functions) and can only be of a simple scalar type.

In order to form a correct parameter list, the parameters to Setpfx must be declared as Var or Const. Typically, Const is used for parameters which are not to be changed by the called Fortran routine and Var is used for those parameters which are not.

When a parameter is declared to be Var and of type String, an extra entry in the parameter list is produced. The maximum length of the string is placed before the address of the string in the parameter list. This frequently causes a program exception if the called Fortran routine inadvertently accesses this extra parameter. Since it is strongly inadvisable to alter String parameters with a Fortran procedure, declaring a String parameter to a Fortran procedure as Var is usually an error.

Multi-dimensional arrays are stored in row-major order in Pascal and column-major order in Fortran. When an array is passed as a parameter from Pascal to Fortran, its transpose actually gets passed.

It is not currently possible to directly access the return code from a Fortran or 370/Assembler routine from Pascal/VS. Although Pascal/VS does set the VL-bit on its last parameter, routines which can use a variable number of parameters cannot be directly used by Pascal/VS programs. However, the procedure PvcallRC can be used to call a non-Pascal routine and also access the return code (see the description of PvcallRC in the section, “Predefined Procedures in Pascal/VS”).

Pascal/JB has an integer function FortranRC which can be used to access the return code from a Fortan or Assembler routine.

Example 2

Calling Fortran from Pascal

```
Program Execute_A_Command;

{Some people's idea of fun is to read commands from the
 terminal and execute them!!...}

Type
  char255 = packed array[1..255] of Char;
  halfword = packed -32768..32767;
Var
  cmdstr : String(255);
Procedure Cmd(const cmd: char255; const len : halfword); Fortran;

Begin
  Reset(Input, 'file=*msource*,interactive');
  Writeln(' Enter an MTS Command:');
  Get(input);
  While not Eof(input) do begin
    Readln(cmdstr);
    CMD(cmdstr, length(cmdstr));
    Writeln(' Enter an MTS Command:');
    Get(Input)
    End
End.
```

Note here that the Pascal compiler will automatically convert from String to Packed Array of Char and from fullword integer to halfword integer (although, in this case the fullword integer would have worked just as well).

Example 3

Calling Pascal from Fortran

```
      SUBROUTINE  MAIN
C
C Try calling a banal Pascal subroutine, just to prove
C that you can do it.
C
      INTEGER  I
C
      I = 1
C Add 3 to I (ho hum)
      CALL ADD3(I)
C Kill what remains of Pascal
      CALL PSCLHX(0)
      WRITE(6,100) I
  100 FORMAT('And the new value of I is: ', I6)
      STOP
      END
```

```
Segment Test;
Procedure Add3(var i: Integer); main;
Procedure Add3;
Begin
  i := i + 3;
  End;.
```

Note: Here we need *PASCALVSLIB (with Pascal/VS) or *PASCALJBLIB (with Pascal/JB) concatenated to the object on the $RUN command.

DATA-TYPE EQUIVALENCES

The table on the following page compares Pascal data types with their equivalents in Fortran and PL/I.

The Pascal compilers make no attempt to remap any storage when an inter-language call is made. This means that because Fortran stores its arrays in column-major order and Pascal stores its arrays in row-major order, a call between Fortran and Pascal procedures appears to transpose the array.

Data-Type Equivalences Between Different Languages

Pascal	*FORTRAN*	*PL/I*
Char	CHARACTER*1	CHAR
Boolean	LOGICAL*1	FIXED BINARY(1,0)
Integer	INTEGER*4	FIXED BINARY(31,0)
Packed –32768..32767	INTEGER*2	FIXED BINARY(15,0)
Packed 0..65536	na	na
Packed –128..127	na	FIXED BINARY(7,0)
Packed 0..255	na	na
Real	REAL*8	REAL FLOAT DEC(16)
Shortreal	REAL*4	REAL FLOAT DEC(6)
Packed Array[1..n] of Char	CHARACTER*n	CHAR(n)
String(m)	na	CHAR(m) VARYING
Set of 0..n	na	BIT(n+1)
@Id	na	POINTER
Array	Dimensioned Variable	Dimensioned Variable
Record	na	STRUCTURE
Space	na	AREA

PASCAL ADAPTER ROUTINES

INTRODUCTION

Most of the system subroutines described in MTS Volume 3, *System Subroutine Descriptions*, can be called from both PASCAL/VS and PASCAL/JB. Calling MTS system subroutines from Pascal is, in general, complicated by the fact that Pascal is a strongly typed language, which implies that each external routine (subroutine or function) called by a Pascal program must be declared within that program. Earlier sections in this volume describe how to construct Pascal declarations and calls for MTS system subroutines for both Pascal/VS and Pascal/JB. Even using these methods, the process is often complicated and time consuming. For example, GUINFO is a commonly used system subroutine; a typical invocation of the GUINFO subroutine might look like this:

```
Program Example;
Type PAC_8 = Packed Array[1..8] of Char;
     PAC_256 = Packed Array[1..256] of Char;
     PAC_48 = Packed Array[1..48] of Char;
     Guinfo_Type = (GU_Fixed, GU_Int1, GU_Int2, GU_Var);
     Gresult_type = Record case GUINFO_TYPE of
             GU_Fixed: (Fixed_Char: PAC_48);
             GU_Int1:  (Integer_Val : Integer);
             GU_Int2:  (Integer_Val1, Integer_Val2: Integer);
             GU_Var:   (Max_Length, Length : Integer;
                        Chars: PAC_256);
             End;
Procedure Guinfo(const item_name : PAC_8;
                 var guinf_result : gresult_type);FORTRAN;
Var Temp:gresult_type;
    Res1:PAC_48;
    Res2, Res3, Res4:INTEGER;
    Res5:PAC_256;
Begin
    Guinfo('SIGTMUT',Temp);
    Res1 := Temp.Fixed_Char;
    Guinfo('RUNTIME',Temp);
    Res2 := Temp.Integer_Val;
    Guinfo('RUNETIME',Temp);
    Res3 := Temp.Integer_Val1;
    Res4 := Temp.Integer_Val2;
    Temp.Max_Length := 256;
    Guinfo('PROJSIGF',Temp);
    Res5 := Temp.Chars;
End;
```

This call to the GUINFO subroutine illustrates some of the difficulties involved when declaring and calling from Pascal a routine that returns a mixture of fullword integer, doubleword integer, and character information in a single argument.

As an alternative to the above method, the user may instead call a set of routines that are functionally equivalent to many MTS system subroutines but are designed to be Pascal-callable without constructing complicated declarations and calling sequences. These routines are located in the subroutine libraries *PASCALVSSYSLIB and *PASCALJBSYSLIB and are much easier to use. The calling sequences for these routines are simpler than the corresponding MTS Volume 3 subroutines.

The program described above could be written much more easily by using the routines PGUINFOI, PGUINFO2, and PGUINFOS which are contained in both subroutine libraries. These routines

perform the same functions as GUINFO.

```
Program Test;
%Include PGuinfoI
%Include PGuinfoS
Var Res1 : String(18);
    Res2, Res3, Res4 :Integer;
    Res5 : String(30);
Begin
   PGuinfoS('SIGTMUT',Res1);
   PGuinfoI('RUNTIME',Res2);
   PGuinfo2('RUNETIME',Res3,Res4);
   PGuinfoS('PROJSIGF',Res5);
End.
```

THE ORGANIZATION OF THE %INCLUDE AND SUBROUTINE LIBRARIES

All of the information necessary to call the routines described in this section is contained in the two subroutine library files and their corresponding %INCLUDE library files. The files *PASCALVSSYSLIB and *PASCALVSINCLUDE are used together with PASCAL/VS. The files *PASCALJBSYSLIB and *PASCALJBINCLUDE are used together with PASCAL/JB.

Both of the %INCLUDE library files contain Procedure or Function declarations for the routines in the subroutine library files which are equivalent to the MTS system subroutines.

Some of the routines in each subroutine library are called “adapter” routines and perform functions that are similar to the MTS system subroutines. Each of these routines passes one or more parameters to specific system subroutines, returns results back to the calling program, and processes error conditions. Other routines are called “support” routines and either alter how adapter routines behave when they detect an error or retrieve information about an error when an error condition occurs. These libraries also contain an “error monitor” routine which is not called directly by the Pascal program, but is invoked indirectly whenever an error condition occurs.

Also included in the %INCLUDE libraries are declarations for MTS subroutines that, by their nature, can be called directly without using *PASCALVSSYSLIB and *PASCALJBSYSLIB and thus are not included in the libraries. If the subroutine being called is not in these libraries, declarations must be constructed and calls made for them according to the procedures described in the sections on Pascal/VS and Pascal/JB.

HOW TO USE THE %INCLUDE AND SUBROUTINE LIBRARIES

To use a %INCLUDE library in Pascal, the correct %INCLUDE library file must be attached when running the Pascal compiler. With *PASCALVS, one of the units 1 through 8 must be assigned to the file *PASCALVSINCLUDE. With *PASCALJB, one of the units 1 through 8 must be assigned to the file *PASCALJBINCLUDE. In addition, the source program must contain %INCLUDE statements so that the necessary declarations for the routines that are called are retrieved from the %INCLUDE library. %INCLUDE statements can be placed whenever declaratives are legal. In general, at least one %INCLUDE statement will be needed for each subroutine that is used. For example,

```
Program Test;

{Example of program that calls MTS, QUIT, and SYSTEM}
```

```
%INCLUDE MTS
%INCLUDE QUIT
%INCLUDE SYSTEM
VAR DONE, ERROR, SUSPEND:BOOLEAN;
 ...
IF DONE THEN SYSTEM;
IF SUSPEND THEN MTS;
IF ERROR THEN QUIT;
 ...
```

Extra %INCLUDE statements will be needed if %INCLUDE items that refer to Types that are defined in other %INCLUDE items. Such %INCLUDE statements must be before other %INCLUDE statements. All the %INCLUDE statements must appear in a part of the program where it is legal to have declaratives, i.e., after a Procedure or Function statement, but before the associated Begin statement.

Each subroutine description in the following sections lists all the %INCLUDEs required to use that routine. All of the required %INCLUDES must be placed in the declaration section of the procedure or function. The order of the %INCLUDES is important if a specific subroutine requires more than one %INCLUDE. Also, if two or more subroutines are being used, the same %INCLUDE statement may not be used more than once. Failure to use the %INCLUDEs correctly will result in Pascal error messages.

For example, suppose we wish to use the PGETFD, PREAD, and PWRITE subroutines. The description of the PGETFD subroutine lists

```
%INCLUDE PGETFD
%INCLUDE Fdub_Type
```

The description of PWRITE lists

```
%INCLUDE PWRITE
%INCLUDE Fdub_Type
%INCLUDE IO_Modifiers
```

The description of PREAD lists

```
%INCLUDE PREAD
%INCLUDE Fdub_Type
%INCLUDE IO_Modifiers
```

Since duplicates of %INCLUDE Fdub_Type and %INCLUDE IO_Modifiers are not allowed, the program should look like this:

```
PROGRAM XXX;
%INCLUDE FDUB_TYPE
%INCLUDE IO_MODIFIERS
%INCLUDE PGETFD
%INCLUDE PWRITE
%INCLUDE PREAD
VAR ...
BEGIN;
 ...
END;
```

After the source program is compiled, the object program must be run. If one or more routines are needed from the subroutine libraries, concatenate the correct subroutine library file to the file containing the object program. The subroutine library file for *PASCALJB is *PASCALJBSYSLIB and the subroutine library file for *PASCALVS is *PASCALVSSYSLIB. For *PASCALVS, a typical command sequence might look like this:

```
$RUN *PASCALVS SCARDS=sou 1=*PASCALVSINCLUDE SPUNCH=-obj
$RUN -obj+*PASCALVSSYSLIB SCARDS=data
```

For *PASCALJB, a typical command sequence might look like this:

```
$RUN *PASCALJB SCARDS=sou 1=*PASCALJBINCLUDE SPUNCH=-obj
$RUN -obj+*PASCALJBSYSLIB SCARDS=data
```

Note that it is very important to use the correct file names. Using either *PASCALVSINCLUDE or *PASCALVSSYSLIB with *PASCALJB or vice-versa may result in obscure run time errors.

USING FILES FROM A SYSTEM SUBROUTINE

Many of the routines described in the following sections use, access, or manipulate MTS files. In most cases, instead of referring to the file by name, the file is referred to by giving either a unit or a Fdub-pointer. A Fdub-pointer can be obtained for a specific file or device by calling the routine PGETFD and placing the resultant Fdub-pointer in a variable of type Fdub_Type. This Fdub-pointer can subsequently be given to the routine in question. Alternately, the name of a logical I/O unit represented as a string can be supplied, i.e., “SCARDS”, “SPUNCH”, “SPRINT”, “SERCOM”, “GUSER”, or an integer unit between 0 and 99. A unit between 0 and 9 is represented as a one-character string, for example, “0” or “1”. A unit between 10 and 99 is represented as a two-character string, for example, “10” or “15”. Examples of this are given in the individual subroutine descriptions.

ADAPTER ROUTINE DESCRIPTIONS

This section describes each of the adapter routines. Unless otherwise indicated, each of these routines is used as a procedure.

A *return code* is a value that is returned by each of these routines. A zero return code indicates that there were no errors. A nonzero return code indicates something went wrong. Normally, most conditions that cause a nonzero return code will generate an error message and will cause the program to stop with an error return. This is not always desirable. For example, a program may try create a file using PCREATE. If for some reason, the file cannot be created, an error message is produced and the program will stop. It may sometimes be desirable for the program to react in some reasonable fashion to the fact that the file could not be created. To have the program do this, see the section, "Support Routines," for more information.

Attention Trapping Routines

Subroutine Description

Purpose: To allow a Pascal program to intercept attention interrupts.

Location: *PASCALVSSYSLIB and *PASCALJBSYSLIB

Module Name: TRAP

Includes: %INCLUDE TRAP

Calling Sequences: Pascal:
```
TRAPON;
TRAPOFF;
TRAP;
TRPRESET;
```

Values Returned:

Trap is a Boolean function that returns True if an attention interrupt has occurred and False otherwise.

Return Codes:

Each of these routines set the return code to zero.

Description: TRAPON turns attention trapping on and TRAPOFF turns it off. Attention trapping defaults off, which means that if an attention interrupt occurs, control returns to MTS. If attention trapping is turned on, anytime an attention interrupt occurs, a flag is set. The function TRAP normally returns False, but will return True if an attention interrupt has occurred since the last time TRPRESET was called (or since the time program was loaded). TRAP does not turn the flag off; to turn the flag off, call TRPRESET. Neither TRAP, TRPRESET, nor the occurrence of an attention interrupt, affects the status of attention trapping itself. It is possible, if several attention interrupts occur in rapid succession, that the program may stop execution and return to MTS. This happens rather infrequently; normally, an attention interrupt simply causes the flag to be set.

Error Conditions: These routines do not detect any error conditions or alter the value returned by SYSRC.

Example: The following example illustrates the use of the attention-trapping routines.

```
PROCEDURE ...;
%INCLUDE TRAP
BEGIN
  TRAPON;
  WHILE ~TRAP DO
    BEGIN
    ...
    END;
  IF TRAP WRITELN(' Attention!');
```

PCFDUB

Subroutine Description

Purpose: To allow a Pascal program to call the CFDUB system subroutine.

Location: *PASCALVSSYSLIB and *PASCALJBSYSLIB

Module Name: PCFDUB

Includes: %INCLUDE Fdub_Type
%INCLUDE CFDUB

Calling Sequences: Pascal: PCFDUB(Fdub1,Fdub2)

Parameters:

Fdub1, Fdub2 : Fdub_Type;

The Fdub-pointer or units to be compared. This must be either a variable of type Fdub_Type or a string.

Values Returned:

PCFDUB is a Boolean function which returns True if the two units or Fdub-pointers correspond to the same file or device, and False if they do not correspond.

Return Codes:

0 Both Fdub-pointers are valid.
8 One or both Fdub-pointers are invalid.

Description: This function tests if two units or Fdub-pointers represent the same file or device by calling the CFDUB subroutine. If the units compare equal, PCFDUB returns True. If the two units are not equal, PCFDUB returns False. If either unit is invalid, PCFDUB gives return code 8. See MTS Volume 3, *System Subroutine Descriptions*, for information on the CFDUB subroutine.

Error Conditions: All error conditions belong to error class ERR_FILE.

Example: The following example illustrates the use of the PCFDUB routine.

```
PROCEDURE ...;
%INCLUDE FDUB_TYPE
%INCLUDE PCFDUB
BEGIN
  IF PCFDUB('SCARDS','SPUNCH') THEN
     WRITELN(' SCARDS and SPUNCH are the same!');
```

PCOMMAND

Subroutine Description

Purpose: To allow a Pascal program to call the COMMAND system subroutine.

Location: *PASCALVSSYSLIB and *PASCALJBSYSLIB

Module Name: PCOMMAND

Includes: %INCLUDE PCOMMAND

Calling Sequences: Pascal:
```
PCOMMAND(Command);
PCOMMAND1(Command,Switches);
PCOMMAND2(Command,Switches,CS_Summary,
    CS_Code,CS_Origin);
```

Parameters:

Command : String;

The MTS command to be executed which must be between 1 and 255 characters in length. If the length is greater than 255, a return code of 8 will be generated.

Switches : Command_Switch;

A set containing any combination of the values Always_Echo, Never_Echo, Always_Summ, and Never_Summ. If this parameter is omitted, the command is echoed and a summary is printed if and only if $SET ECHO=ON is in effect.

If Always_Echo is on, the command will always be echoed. If Never_Echo is on, the command will never be echoed. If neither Always_Echo nor Never_Echo is set on, the command will be echoed depending on the setting of the $SET ECHO option in MTS.

If Always_Summ is on, a summary will always be printed. If Never_Summ is on, a summary will never be printed. If neither Always_Summ nor Never_Summ is set on, a summary will be printed if the command was echoed.

CS_Summary, CS_Code, CS_Origin : Integer;

See the description of the COMMAND subroutine in MTS Volume 3, *System Subroutine Descriptions*. If any or all of these parameters are omitted, the corresponding values are not returned.

Return Codes:

0 Successful return.
4 MTS command execution failed.
8 Illegal length for MTS command.
12 Invalid parameter(s).

Description: The characters in Command are executed as an MTS command. For more details, see the description for the COMMAND subroutine in MTS Volume 3.

Error Conditions: By default, when the "MTS command execution failed" situation occurs, execution will continue without an error message (MTS itself will normally print out an error message explaining why the command failed.) This error belongs to error class ERR_MTS_COM_FAIL. All other errors belong to class ERR_COMMAND.

Example: The following example illustrates the use of the PCOMMAND routine.

```
PROCEDURE ...;
%INCLUDE PCOMMAND
VAR STRINGITEM:STRING(30);
    COMMAND:STRING(255);
    CS_CODE, CS_SUMMARY, CS_ORIGIN: INTEGER;

BEGIN
  PCOMMAND('$DISPLAY TIMESPELLEDOUT');
  WRITELN(' What item would you like displayed?')
  READLN(STRINGITEM);
  PCOMMAND1('$DISPLAY ' || STRINGITEM,[Never_Echo]);
  ...
  READLN(COMMAND);
  PCOMMAND2(COMMAND,[],CS_CODE,CS_SUMMARY,CS_ORIGIN);
```

PCONTROL

Subroutine Description

Purpose: To allow a Pascal program to call the CONTROL system subroutine.

Location: *PASCALVSSYSLIB and *PASCALJBSYSLIB

Module Name: PCONTROL

Includes:
```
%INCLUDE Fdub_Type
%INCLUDE PCONTROL
```

Calling Sequences: Pascal: PCONTROL(Command,Fdub);

Parameters:

Command : String;

The control command string as explained in MTS Volume 3.

Fdub : Fdub_Type;

Fdub is either a variable of type Fdub_Type or a String.

Return Codes:

0	Successful return.
<1000	Control command failed. The number represents the DSR return code as explained in the description of the CONTROL subroutine in MTS Volume 3.
1000	Missing second parameter.
1004	Invalid parameter (or parameters).
>1004	This value is the return code from CONTROL plus 1000.

Description: This subroutine performs a control operation on the indicated device. For more details, see the description for the CONTROL subroutine in MTS Volume 3, *System Subroutine Descriptions*.

Notes: When issuing control operations that reposition a magnetic tape (such as FSR, BSR, or REW) and that tape is also being processed by Pascal I/O, one must be aware of how Pascal processes files. Pascal I/O normally reads a new record whenever an end-of-line is detected. This "reading ahead" may cause unexpected results. For a more detailed explanation, see the sections on PREWIND, PSKIP, PFSRF, and PBSRF.

If Pascal I/O has determined that an end-of-file condition exists on a magnetic tape, then a control operation to reposition that tape (such as FSR, BSR, or REW) will not change the end-of-file condition. A RESET or REWRITE is necessary to clear the end-of-file condition.

Error Conditions: All error conditions belong to error class ERR_CONTROL.

Example: The following example illustrates the use of the PCONTROL routine.

```
PROCEDURE ...;
%INCLUDE FDUB_TYPE
%INCLUDE PCONTROL
VAR FDUB : FDUB_TYPE;
BEGIN

  PCONTROL('BLANK','SCARDS');
  ...
  PCONTROL('FSR 1','0');
  ...
  PCONTROL('EMPTY',FDUB);
  ...
  END;
```

PCREATE

Subroutine Description

Purpose: To allow a Pascal program to call the CREATE system subroutine.

Location: *PASCALVSSYSLIB and *PASCALJBSYSLIB

Module Name: PCREATE

Includes: %INCLUDE PCREATE

Calling Sequences: Pascal: PCREATE(Filename);

PCREATE1(Filename,Size);

PCREATE2(Filename,Size,Filetype);

Parameters:

Filename : String;

The name of the file to be created. This string must not be longer than 256 characters. The file name may have trailing blanks as long as these blanks do not make the complete string longer than 256 characters. It must not have imbedded blanks, or imbedded file terminator characters. Violations of these conditions will result in return code 20.

Size : Integer;

This is the initial size of the file in pages. If omitted, the initial size of the file will be one page.

Filetype : File_Types;

A value of Line_File indicates a line file, and a value of Sequential_File indicates a sequential file. If this parameter is omitted, the file will be a line file.

Return Codes:

0 Successful return.
4 File already exists.
8 Illegal file type.
12 Invalid size parameter.
16 Insufficient disk space for file.
20 Illegal file name.
24 Hardware error or software inconsistency.
28 Disk allotment has been exceeded.

Description: A file with the given name is created. Any available disk volume is used. The maximum file size is set to 32767 pages. The initial file size and file type (either line or sequential) are set as indicated above. For more details, see the description for the CREATE subroutine in MTS Volume 3, *System Subroutine Descriptions*.

Error Conditions: All errors belong to error class ERR_CREATE.

Example: The following example illustrates the use of the PCREATE routine.

```
PROCEDURE ...;
%INCLUDE PCREATE
BEGIN
  PCREATE('DATAFILE1');
  PCREATE1('-HUGEFILE',5000);
  PCREATE2('MYFILE',1,SEQUENTIAL_FILE);
  ...
```

PEMPTY, PTRUNC

Subroutine Description

Purpose: To allow a Pascal program to call the EMPTY and TRUNC system subroutines.

Location: *PASCALVSSYSLIB and *PASCALJBSYSLIB

Module Names: PEMPTY and PTRUNC

Includes:
%INCLUDE Fdub_Type
%INCLUDE rrrr

where "rrrr" is one of PEMPTY or PTRUNC.

Calling Sequences: Pascal:
```
PEMPTY(Fdub);

PTRUNC(Fdub);
```

Parameters:

Fdub : Fdub_Type;

The Fdub-pointer or unit for the file to be emptied or truncated. This must be either a variable of type Fdub_Type or a character string.

Return Codes:

0	Successful return.
>0	See MTS Volume 3.

Description: PEMPTY calls the system subroutine EMPTY with the given Fdub-pointer or unit. PTRUNC calls the system subroutine with the given Fdub-pointer or unit. For further details, see the descriptions of the EMPTY and TRUNC subroutines in MTS Volume 3, *System Subroutine Descriptions*.

Error Conditions: All error conditions belong to error class ERR_FILE.

Example: The following example illustrates the use of the PEMPTY and PTRUNC routines.

```
PROCEDURE ...;
%INCLUDE FDUB_TYPE
%INCLUDE PTRUNC
%INCLUDE PEMPTY
%INCLUDE PGETFD
VAR FDUB : FDUB_TYPE

BEGIN
  PGETFD('FILE1',FDUB);
  ...
  PEMPTY(FDUB);
  ...
  PTRUNC(FDUB);
  ...
```

PFREEFD

Subroutine Description

Purpose: To allow a Pascal program to call the FREEFD system subroutine.

Location: *PASCALVSSYSLIB and *PASCALJBSYSLIB

Module Name: PFREEFD

Includes: %INCLUDE Fdub_Type
%INCLUDE PFREEFD

Calling Sequence: Pascal: PFREEFD(Fdub);

Parameters:

Fdub : Fdub_Type;

A variable of type Fdub_Type representing the Fdub-pointer to be freed.

Return Codes:

0 Successful return.
4 Invalid Fdub-pointer.

Description: This routine releases a Fdub. For more details, see the description of the FREEFD subroutine in MTS Volume 3, *System Subroutine Descriptions*.

Error Conditions: All error conditions belong to error class ERR_GETFREEFD.

Example: The following example illustrates the use of the PFREEFD routine.

```
PROCEDURE ...;
%INCLUDE FDUB_TYPE
%INCLUDE PFREEFD
VAR FDUB : FDUB_TYPE;

BEGIN;
  ...
  PFREEFD(FDUB);
  ...
```

PGETFD

Subroutine Description

Purpose: To allow a Pascal program to call the GETFD system subroutine.

Location: *PASCALVSSYSLIB and *PASCALJBSYSLIB

Module Name: PGETFD

Includes:
```
%INCLUDE Fdub_Type
%INCLUDE PGETFD
```

Calling Sequence: Pascal: PGETFD(Fdname,Fdub);

Parameters:

Fdname : String;

The file or device name. It may have any number of trailing blanks (including zero), but it must not have imbedded blanks.

Fdub : Fdub_Type;

A variable of type Fdub_Type which will contain the Fdub-pointer.

Return Codes:

0	Successful return.
4	Invalid address.
8	Device is busy.
12	Device is not operational.
1000	Missing second parameter.
1004	Invalid Fdname (blanks not allowed).
1008	Uninitialized file name string.
1012	FREESPAC failure (should not occur).

PGETFD will give a zero return code for non-existent, non-accessible, or invalid file or device names. Other system routines such as GDINFO, GDINFO2, or GDINFO3 can be used to determine the status of the file or device.

Description: This routine obtains a Fdub-pointer when given a string of characters that represent the Fdname. For more details, see the description for the GETFD subroutine in MTS Volume 3, *System Subroutine Descriptions*.

Error Conditions: All error conditions belong to error class ERR_GETFREEFD.

Example: The following example illustrates the use of the PGETFD routine.

```
PROCEDURE ...;
%INCLUDE FDUB_TYPE
%INCLUDE PGETFD
VAR FDUB1, FDUB2, FDUB3 : FDUB_TYPE;
    FILENAME : STRING(30);

BEGIN
  PGETFD('MYFILE',FDUB1);
  PGETFD('MYFILE1+MYFILE2@INDEXED' ||
         '+HISFILE(1,45,8)@-IC', FDUB2);
  PGETFD('MYFILE+' || FILENAME, FDUB3);
  ...
```

PGUINFOI

Subroutine Description

Purpose: To allow a Pascal program to call the GUINFO system subroutine and obtain information as integer values.

Location: *PASCALVSSYSLIB and *PASCALJBSYSLIB

Module Name: PGUINFOI

Includes: %INCLUDE PGUINFOI

Calling Sequences: Pascal: PGUINFOI(Item,Value1);

PGUINFO2(Item,Value1,Value2);

Parameters:

Item : String;

One of the Item names as described in the GUINFO/CUINFO subroutine description in MTS Volume 3, *System Subroutine Descriptions*. The name must be left-justified, in uppercase, and may have trailing blanks. This information item must correspond to an integer data type. If it does not, a return code 16 will be generated.

Value1 : Integer;

The first (or only) word returned by GUINFO.

Value2 : Integer;

The second word returned by GUINFO. If omitted, this value is not returned.

Return Codes:

0	Successful return.
8	Item not on list.
16	This GUINFO item is not an integer.
1000	Missing second parameter.

Description: This routine retrieves information about the system. For more details, see the description for the GUINFO/CUINFO subroutines in MTS Volume 3, *System Subroutine Descriptions*.

Error Conditions: All error conditions belong to error class ERR_GUINFO.

Example: The following example illustrates the use of the PGUINFOI routine.

```
PROCEDURE ...;
%INCLUDE PGUINFOI
VAR M,C : INTEGER;

BEGIN
  PGUINFOI('MAXDISK',M); PGUINFOI('CURRDISK',C);
  WRITELN(' You have ', M-C:0,
          ' pages of disk space left.');
```

PGUINFOS

Subroutine Description

Purpose: To allow a Pascal program to call the GUINFO system subroutine and obtain information as a string value.

Location: *PASCALVSSYSLIB and *PASCALJBSYSLIB

Module Name: PGUINFOS

Includes: %INCLUDE PGUINFOS

Calling Sequences: Pascal: PGUINFOS(Item,Value1);

Parameters:

Item : String;

One of the Item names as described in the GUINFO/CUINFO subroutine description in MTS Volume 3. The name must be left-justified, in uppercase, and may have trailing blanks.

Value1 : String;

The result from GUINFO. If it is character-type information (either fixed-character format or variable format), it is converted to a string. If the value represents a simple number, it is converted to a string. If it represents one of a discrete list of values, such as “on” or “off”, a character representation of that value is returned. If it represents an amount of money, it is converted to a monetary amount (“$” followed by the value in dollars and cents). If the value represents a time, it is converted to an external time or time/date. The result is then returned in Value1.

Return Codes:

0	Successful return.
8	Invalid item name.
12	String too short to hold result from GUINFO.
1000	Missing second parameter.

Description: This routine retrieves information about the system. The value from GUINFO is converted to a string (as explained above) and placed in the second parameter.

Error Conditions: If the string provided by the user is not long enough to hold the converted string, PGUINFOS truncates the string on the right as needed and gives a return code 12. For all other error occurs, Value1 will contain the string “?”. For more details, see the description for the GUINFO/CUINFO subroutines in MTS Volume 3, *System Subroutine Descriptions*. All error conditions belong to error class ERR_GUINFO.

Example: The following example illustrates the use of the PGUINFOS routine.

```
PROCEDURE ...;
%INCLUDE PGUINFOS
VAR SIG : STRING(4);

BEGIN
  PGUINFOS('SIGNONID',SIG);
  ...
```

PLOCK, PUNLK, PCLOSEFL, PWRITEBF

Subroutine Description

Purpose: To allow a Pascal program to call the LOCK, UNLK, CLOSEFIL, and WRITEBUF system subroutines.

Location: *PASCALVSSYSLIB and *PASCALJBSYSLIB

Module Names: PLOCK, PUNLK, PCLOSEFL, and PWRITEBF

Includes:
%INCLUDE Fdub_Type
%INCLUDE rrrr

where "rrrr" is one of PLOCK, PUNLK, PCLOSEFL, or PWRITEBF.

Calling Sequences: Pascal:

```
PLOCK(Fdub,How,Waitflag);

PUNLK(Fdub);

PCLOSEFL(Fdub);

PWRITEBF(Fdub);
```

Parameters:

Fdub : Fdub_Type;

The Fdub-pointer or unit for the file to be locked, unlocked, or closed. This must be either a variable of type Fdub_Type or a character string.

How, Waitflag : Integer;

Two integer values as explained in the description of the LOCK subroutine in MTS Volume 3, *System Subroutine Descriptions*.

Return Codes:

0	Successful return.
>0	See MTS Volume 3.

Description: The LOCK, UNLK, CLOSEFIL, or WRITEBUF subroutine is called with the appropriate arguments.

Error Conditions: All error conditions belong to error class ERR_FILE.

Note: The Pascal I/O library will lock a file when it reads or writes to that file. The file is normally unlocked when it is subsequently closed by Pascal.

If a Pascal file was opened by a REWRITE or a RESET with the UNIT

parameter specified and the same unit was used by PLOCK and PUNLK, then the locking and unlocking by Pascal and the locking and unlocking by PLOCK and PUNLK occur in a straightforward manner.

If a Pascal file was opened by a REWRITE or a RESET with the FILE parameter specified or PLOCK/PUNLK used a FDUB-pointer rather than a logical I/O unit, the Pascal I/O library and the PLOCK/PUNLK routines will act independently since they are using different FDUBs. If either Pascal or PLOCK/PUNLK have a locking request, then the file is locked at the highest locking request. If neither Pascal nor PLOCK/PUNLK have a locking request, the file is unlocked.

In the following example, the file DATA will remain locked after the following code sequence:

```
Var f:Text;
    fdub:fdub_type;
Begin
  Rewrite(f,'FILE=DATA');
  Writeln(f,' This locks the file "DATA"');
  PGETFD(fdub,'DATA');
  PUNLK(fdub);
```

However, if the file DATA has been assigned to logical I/O unit 99, it will be unlocked after the following sequence:

```
Var f:Text;
Begin
  Rewrite(f,'UNIT=99');
  Writeln(f,' This locks unit 99');
  PUNLK('99');
```

Example: The following example illustrates the use of the PLOCK and PUNLK routines.

```
PROCEDURE ...;
%INCLUDE FDUB_TYPE
%INCLUDE PLOCK
%INCLUDE PUNLK

BEGIN
  PLOCK('1',1,0);
  ...
  PUNLK('1');
  ...
```

PMOUNT

Subroutine Description

Purpose: To allow a Pascal program to call the MOUNT system subroutine.

Location: *PASCALVSSYSLIB and *PASCALJBSYSLIB

Module Name: PMOUNT

Includes: %INCLUDE PMOUNT

Calling Sequences: Pascal: PMOUNT(Request,Options);

Parameters:

Request : String;

One or more mount requests separated by semicolons ";" as described in the MOUNT subroutine description in MTS Volume 3, *System Subroutine Descriptions*.

Options: Option_Type;

A set with elements taken from the following:

MOUNT_NOECHO
: This option suppresses echoing of the mount request.

MOUNT_NOPRINT
: This option suppresses the printing of error messages.

MOUNT_NOPROMPT
: This option suppresses the prompting for replacement of an erroneous mount request.

MOUNT_PROCESSALL
: This option causes the MOUNT subroutine to attempt to process all the mount requests, rather than stopping when an erroneous mount request is detected.

MOUNT_NOVERIFY
: This option suppresses the message that is normally printed when a mount request has been successful.

MOUNT_NOATTN
: This option turns off attention interrupts during the operator wait. If this option is used, the user cannot attention out of an operator wait.

MOUNT_NOPDNRN
: This option prevents the pseudo-device name and rack number from appearing in any messages that MOUNT produces.

MOUNT_WAIT
: This option causes the user to wait in a tape mount queue, if necessary, without any prompting message. If MOUNT_NOQUEUE is selected or this is a batch job, this

option is ignored.

MOUNT_NOQUEUE
: This option prevents a request from being queued if there are no devices to satisfy the request.

Return Codes:

0	Successful return.
<1000	The value obtained from the ERCODE parameter.
≥1000	The value obtained from the MOUNT return code plus 1000. Normally, the return code will be less than 1000 (i.e., it will be the value obtained from the ERCODE parameter, or zero).

Description: This routine calls the MOUNT subroutine with the indicated mount string and option bits.

Error Conditions: An array is set up for both the ERCODE and ERRMSG parameters. The first nonzero ERCODE is used for the return code. The corresponding ERRMSG is used for the error message. PSETERR does not affect whether or not an error message is produced (this is selected by the Options parameter). All error conditions belong to error class ERR_MOUNT.

Example: The following example illustrates the use of the PMOUNT routine.

```
PROCEDURE ...;
%INCLUDE PMOUNT

BEGIN
  PMOUNT('C0007 *T1* VOL=VOLUME WRITE=NO' ||
         ' ''LABELONE'';C0009 *T2* VOL=VOLUME2 ' ||
         ' WRITE=YES ''LABELTWO''',[]);
  PMOUNT('MNET *MSU* D=MS;MNET *WSU* D=WU;'
         ,[MOUNT_NOECHO]);
```

PREAD

Subroutine Description

Purpose: To allow a Pascal program to call the READ system subroutine.

Location: *PASCALVSSYSLIB and *PASCALJBSYSLIB

Module Name: PREAD

Includes:
```
%INCLUDE Fdub_Type
%INCLUDE IO_Modifiers
%INCLUDE PREAD
```

Calling Sequences: Pascal: PREAD(Input,Modifiers,Linenumber,Fdub);

Parameters:

Input : String;

The location where the input record will be placed. Note that this routine forces @MAXLEN and uses the maximum length of the string as the maximum length.

Modifiers : IO_Modifiers;

Modifiers are modifiers as explained in MTS Volume 3, *System Subroutine Descriptions*, with a few exceptions:

(1) Both @MAXLEN and @FDUBCONT are always set on regardless of the modifiers set by the user.
(2) If @NOTIFY is set, and that condition is raised, PREAD gives return code 1008 and returns the null string.
(3) If @NOPROMPT is set and that condition is raised, PREAD gives return code 1004 and returns the null string.

The modifiers are expressed as a SET with values taken from the following list:

```
MOD_NOTIFY
MOD_ERRRTN
MOD_NOATTN
MOD_NOEC
MOD_MAXLEN
MOD_NOPROMPT
MOD_FDUBCONT
MOD_ENDFILE / MOD_NOENDFILE
MOD_BACKWARDS / MOD_FORWARDS
MOD_IC / MOD_NOIC
MOD_SP / MOD_NOSP
```

MOD_TRIM / MOD_NOTRIM
MOD_MCC / MOD_NOMCC
MOD_PEEL / MOD_NOPEEL
MOD_PREFIX / MOD_NOPREFIX
MOD_CC / MOD_NOCC
MOD_UC / MOD_LC
MOD_BINARY / MOD_EBCD
MOD_INDEXED / MOD_SEQUENTIAL
MOD_MFR / MOD_NOMFR
MOD_MACRO / MOD_NOMACRO
MOD_LOG / MOD_NOLOG

Linenumber : Integer;

As explained in MTS Volume 3.

Fdub : Fdub_Type;

Fdub is either a variable of type Fdub_Type or a String which defines the file or device to be read from.

Value Returned:

PREAD is a Boolean function that returns True if there is no end of file detected on this call to READ, and False otherwise. If @INDEXED is used, then True means that the requested line exists and False means that it does not. Note after most nonzero return code conditions, this value will be False. The two exceptions are when a line gets trimmed and when the @NOTIFY condition occurs.

Return Codes:

0	Successful return.
2	Line trimmed as explained below.
4	End of file or line not in file.
1004	@NOPROMPT used and bad file/device name given.
1008	@NOTIFY used and Fdub opened for first time.
Others	See “I/O Subroutine Return Codes” in MTS Volume 3.

Description: The MTS READ subroutine is called with the parameters as explained above. See MTS Volume 3 for more details about the READ subroutine. A length triple is set up and @MAXLEN is used. The max-length of the length triple is set to the maximum length of the string before the READ routine is called. The third word of the length triple the physical length is not returned.

Note: Normally, it is best not to use PREAD/PWRITE and Pascal I/O on the same file or device at the same time.

Error Conditions: The “line trimmed” condition indicates that the physical length of the line was longer than the maximum size of the string. It is possible that the trimmed characters were blanks (which is possible even if @TRIM was in effect.) PREAD will generate a return code zero rather than a return code 2 if @TRIM is

in effect and the trimmed length of the line is at least one shorter than the maximum size of the string. If the trimmed length of the line is exactly the same length as the maximum length of the string and characters were lost, a return code 2 is generated *even if all the discarded characters were blanks*.

Return codes 2, 4, 1004, and 1008 never generate error messages. Also as explained in MTS Volume 3, return codes from READ greater than 4 normally cause an error message to be printed and generate an error return to MTS. If this happens, the error monitor never has an opportunity to intercept the error. See MTS Volume 3 for information on how to alter this behaviour. All error conditions (that is, return codes >4 and <1000) correspond to ERR_READWRITE.

If the unit or Fdub-pointer is invalid, MTS will produce an error message and execution of the program will terminate.

Example: The following example illustrates the use of the PREAD routine.

```
%INCLUDE FDUB_TYPE
%INCLUDE IO_MODIFIERS
%INCLUDE PREAD
VAR STRINGX,CORRECTANSWER:STRING(30);
  I : INTEGER;
  FDUB : FDUB_TYPE;

BEGIN
  WHILE PREAD(STRINGX,[],0,FDUB) DO
  IF STRINGX=CORRECTANSWER
     THEN WRITELN(' You are correct.');
```

PREWIND, PSKIP, PFSRF, PBSRF

Subroutine Description

Purpose: To allow a Pascal program to call the REWIND, SKIP, FSRF, and BSRF system subroutines.

Location: *PASCALVSSYSLIB and *PASCALJBSYSLIB

Module Names: PREWIND, PSKIP, PFSRF, and PBSRF

Includes:
%INCLUDE Fdub_Type
%INCLUDE rrrr

where "rrrr" is one of PREWIND, PSKIP, PFSRF, or PBSRF.

Calling Sequences: Pascal:

```
PREWIND(Fdub);

PSKIP(Fdub,Nfiles,Nrec);

PFSRF(Fdub,Nrec);

PBSRF(Fdub,Nrec);
```

Parameters:

Fdub : Fdub_Type;

The Fdub-pointer or unit for the file or device to be positioned by SKIP, REWIND, FSRF, or BSRF. This must be either a variable of type Fdub_Type or a character string.

Nfiles, Nrec : Integer;

See the description of the SKIP, FSRF, and BSRF subroutines in MTS Volume 3.

Return Codes:

0	Successful return.
>0	See MTS Volume 3.

Description: This subroutine calls REWIND, SKIP, FSRF, or BSRF. See MTS Volume 3, *System Subroutine Descriptions*.

Notes: If two or more Fdubs refer to a *line* file and a positioning operation such as REWIND or FSRF takes place on one of the Fdubs, the other Fdubs are not affected. In particular, if Pascal I/O takes place on a Pascal file opened with "file=f", a positioning operation using a different Fdub has no effect on the Pascal I/O. If the Pascal I/O and the adapter routine both use the same I/O unit, the positioning will affect the Pascal I/O. This does not appply to either

sequential files or magnetic tapes; positioning operations will affect all Fdubs that refer to a given magnetic tape or sequential file.

Pascal I/O normally reads a new record whenever an end-of-line is detected. This "reading ahead" may result in unexpected results. Suppose the following statements are used to process a tape:

(1) Readln(...,s1);
(2) Readln(...,s2);
(3) PREWIND(...);
(4) Readln(...,s3);
(5) Readln(...,s4);

The sequence of event occurs as follows: The first record is read from the tape, variable "s1" is obtained using that record. The second record is read from the tape, "s2" is assigned using that record. The third record is read from the tape, the tape is rewound, the variable "s3" is assigned using the *third* record, and *not* the first. The first record is read from the tape, and variable "s4" is assigned using that record.

If Pascal I/O has determined that an end-of-file condition exists on a file or device, repositioning that file or device does not change the end-of-file condition. A RESET or REWRITE is necessary to clear the end-of-file condition.

Error Conditions: All error conditions belong to error class ERR_FILE.

Example: The following example illustrates the use of the PFSRF, PBSRF, PREWIND, and PSKIP routines.

```
PROCEDURE ...;
%INCLUDE FDUB_TYPE
%INCLUDE PFSRF
%INCLUDE PBSRF
%INCLUDE PREWIND
%INCLUDE PSKIP
%INCLUDE PGETFD
VAR FDUB : FDUB_TYPE

BEGIN
  PGETFD('*T*',FDUB);
  ...
  PREWIND(FDUB);
  ...
  PSKIP(FDUB,1,0);
  ...
  PFSRF(FDUB,1);
  ...
  PBSRF(FDUB,3);
```

PWRITE

Subroutine Description

Purpose: To allow a Pascal program to call the WRITE system subroutine.

Location: *PASCALVSSYSLIB and *PASCALJBSYSLIB

Module Name: PWRITE

Includes:
```
%INCLUDE Fdub_Type
%INCLUDE IO_Modifiers
%INCLUDE PWRITE
```

Calling Sequences: Pascal: PWRITE(Output,Modifiers,Linenumber,Fdub);

Parameters:

Output : String;

The characters to be sent to the output device.

Modifiers : IO_Modifiers;

Modifiers are modifiers as explained in MTS Volume 3, *System Subroutine Descriptions*, with a few exceptions:

(1) @FdubCONT is always set on regardless of the modifiers set by the user.
(2) If @NOTIFY is set, and that condition is raised, the return code is 1008.
(3) If @NOPROMPT is set and that condition is raised, the return code is 1004.

The modifiers are expressed as a SET with the values taken from the following:

```
MOD_NOTIFY
MOD_ERRRTN
MOD_NOATTN
MOD_NOEC
MOD_MAXLEN
MOD_NOPROMPT
MOD_FDUBCONT
MOD_ENDFILE / MOD_NOENDFILE
MOD_BACKWARDS / MOD_FORWARDS
MOD_IC / MOD_NOIC
MOD_SP / MOD_NOSP
MOD_TRIM / MOD_NOTRIM
MOD_MCC / MOD_NOMCC
MOD_PEEL / MOD_NOPEEL
```

MOD_PREFIX / MOD_NOPREFIX
MOD_CC / MOD_NOCC
MOD_UC / MOD_LC
MOD_BINARY / MOD_EBCD
MOD_INDEXED / MOD_SEQUENTIAL
MOD_MFR / MOD_NOMFR
MOD_MACRO / MOD_NOMACRO
MOD_LOG / MOD_NOLOG

Linenumber : Integer:

As explained in MTS Volume 3.

Fdub : Fdub_Type;

Fdub is either a variable of type Fdub_Type or a string which defines the file or device to be written to.

Return Codes:

0	Successful return.
1004	@NOPROMPT used and bad file/device name given.
1008	@NOTIFY used and Fdub opened for first time.
Others	See "I/O Subroutine Return Codes" in MTS Volume 3.

Description: The MTS WRITE subroutine is called with the parameters as explained above. The length of the information transmitted is simply the length of Output. For more details, see the description of the WRITE subroutine in MTS Volume 3.

Note: It is normally best not to use both PREAD/PWRITE and Pascal I/O on the same file or device at the same time.

Error Conditions: Return codes 1004 and 1008 never generate error messages. As explained in MTS Volume 3, most return codes from WRITE cause an error message to be printed and generate an error return to MTS. If this happens, the error monitor never has an opportunity to intercept the error. See MTS Volume 3 for information on how to alter this behavior. All error conditions (that is conditions with return codes >0 and <1000) correspond to ERR_READWRITE.

If an invalid unit or Fdub-pointer is used, MTS will produce an error message and execution of the program will terminate.

Example: The following example illustrates the use of the PWRITE routine.

```
PROCEDURE ...;
%INCLUDE FDUB_TYPE
%INCLUDE IO_MODIFIERS
%INCLUDE PWRITE
VAR FDUB : FDUB_TYPE;
   I : INTEGER;

BEGIN
  {Write to line 1.000 of a file}

  I := 1000;
  PWRITE('Hello',[MOD_INDEXED],I,FDUB);
  ...
```

SUPPORT ROUTINES

This section lists the support routines available for the Pascal adapter routines.

A given error condition belongs to one of several error classes. A given error class either has the attribute PRINT or NOPRINT. It also either has the attribute FATAL or NONFATAL. The following describes these attributes in more detail:

PRINT — An error message will be printed whenever that error is detected (the default for most errors).

NOPRINT — If an error class has the attribute NOPRINT and does not have the attribute FATAL, then no indication is made that there was an error. The existence of an error must be detected by means of the SYSRC routine.

FATAL — An error message is printed and the current error count is checked whenever an error in that class occurs (the error count defaults to zero). One is subtracted from the error count and, if it becomes negative, the MTS system subroutine ERROR# is called and execution is terminated; otherwise, execution continues.

NONFATAL — Execution is not terminated, and an error message is printed or not printed depending on the setting of the PRINT attribute.

Unless there is a statement to the contrary for a particular routine, all error classes have FATAL and PRINT as defaults. The correspondence between error classes and specific errors is explained in the individual routine descriptions that follow.

The settings of the FATAL/NONFATAL and PRINT/NOPRINT attributes can be changed with the support routine PSETERR. PSETERR also allows the current error count to be changed. By default, all error messages are printed on SERCOM. However by calling PSETERR, errors can produced on any file or device. The routines SYSRC, PMODULE, and PERRMESS can be used to retrieve information about errors that adapter routines have detected.

SYSRC

Subroutine Description

Purpose: Whenever any of the Pascal adapter routines are called, the return code is stored so that it can be retrieved later. This routine retrieves that value.

Location: *PASCALVSSYSLIB and *PASCALJBSYSLIB

Module Name: SYSRC

Includes: %INCLUDE SYSRC

Calling Sequences: Pascal: SYSRC

Value Returned:

SYSRC returns the value of the return code after the last adapter routine returned. If no routines have been called, zero is returned.

Return Codes:

This routine does not update the return code as returned by SYSRC. It returns zero in GR15. No errors are possible.

Example: The following example illustrates the use of the SYSRC routine.

```
%INCLUDE SYSRC
%INCLUDE PCREATE
PCREATE1('FILE',30);
IF SYSRC~=0
  THEN WRITELN(' Couldn''t create file,'
       ' return code is ', SYSRC:0);
```

PERRMESS

Subroutine Description

Purpose: To allow a Pascal program to retrieve the last error message used by an adapter routine.

Location: *PASCALVSSYSLIB and *PASCALJBSYSLIB

Module Name: PERRMESS

Includes: %INCLUDE PERRMESS

Calling Sequences: Pascal: PERRMESS(ErrMsg);

Parameters:

ErrMsg : String;

The value of the last error message is placed in ErrMsg. If ErrMsg is too short to hold the string, the string is truncated without warning. (No messages are longer than 100 characters.) Note that an error message is recorded even if it is not printed. If there have been no error messages so far, a null string is returned.

Return Codes:

This routine does not update the return code as returned by SYSRC. It returns zero in GR15. No errors are possible.

PMODULE

Subroutine Description

Purpose: To allow a Pascal program to determine the name of the last routine that detected an error.

Location: *PASCALVSSYSLIB and *PASCALJBSYSLIB

Module Name: PMODULE

Includes: %INCLUDE PMODULE

Calling Sequences: Pascal: PMODULE(Module);

Parameters:

Module : String;

The module name of the last routine to detect an error. This is always an 8-character string. If no errors have been detected, eight blanks are returned. If the string is greater than eight characters, it is truncated without warning.

Return Codes:

This routine does not update the return code as returned by SYSRC. It gives a zero in GR15 on return. No errors are possible.

Description: The name of a module for the routine that detected the error is returned.

PSETERR

Subroutine Description

Purpose: To allow a Pascal program to control the behavior of the adapter routines on detection of an error condition.

Location: *PASCALVSSYSLIB and *PASCALJBSYSLIB

Module Name: PSETERR

Includes:
```
%INCLUDE Fdub_Type
%INCLUDE PSETERR
```

Calling Sequences: Pascal:
```
PSETERR(ErrList,Selections,ErrLimit,Fdub);
```

Parameters:

ErrList : Errnum_Type;

A list of error conditions that are to be controled. This list must be expressed as a set with elements selected from the following:

```
ERR_READWRITE
ERR_CONTROL
ERR_COMMAND
ERR_MTS_COM_FAIL
ERR_GETFREEFD
ERR_CREATE
ERR_GUINFO
ERR_MOUNT
ERR_FILE
```

Selections : Options_Type;

The error conditions mentioned in ErrList are controlled. If one of the options is ERR_FATAL, then all the errors in the list are made fatal errors. If one of the options is ERR_NONFATAL, then all errors in ERROR_LIST become nonfatal. If one of the options is ERR_PRINT, an error message is printed when any of the errors in ERROR_LIST occur. If one of the options is ERR_NOPRINT, then when any of the errors in error list are detected, no error message is printed. (Exception: A fatal error always prints an error message regardless of the setting of the PRINT option). Other error conditions remain as before. If either ErrList or Selections are the empty set, no change occurs.

ErrLimit : Integer;

If ErrLimit is positive, the error limit is set to ErrLimit. The error limit is only used for “FATAL” errors. If an error is FATAL, the error limit is checked for zero. If it is greater than zero, one is

subtracted from the error limit and execution continues. If the error limit is less than or equal to zero, execution terminates. If the ErrLimit parameter is negative, the error limit remains unchanged.

Fdub : Fdub_Type;

There are four options:

(1) A blank string.
(2) Fdub is a Fdub-pointer as returned by GETFD or PGETFD (or perhaps another system routine).
(3) Fdub is the character string for one of the units SCARDS, GUSER, SPRINT, SERCOM, or SPUNCH.
(4) Fdub is an integer unit number (0-99) represented as a string. If the unit is between 0 and 9, it must be represented as a one character string such as "0" or "1". If the unit is between 10 and 99, it must be represented as a two character string such as "50" or "60".

If Fdub is blank, no change occurs, otherwise all future error messages will be printed using that Fdub or I/O unit.

Return Codes: This routine does not update the return code as returned by SYSRC. It gives a zero in GR15 on return. No errors are possible.

Example: The following example illustrates the use of the PSETERR routine.

```
%INCLUDE FDUB_TYPE
%INCLUDE PSETERR
BEGIN

{Alter the error tables so that whenever an
error involving either PGUINFOI or PGUINFOS
occurs, an error message will be printed
and execution will continue.}

PSETERR([ERR_GUINFO],
        [ERR_PRINT,ERR_NONFATAL],-1,
        ' ');
```

SUPPLEMENTAL PASCAL PROGRAMS

PASCALTIDY

The PascalTidy program is based on the original version by Michael N. Condict of Lehigh University, written in 1975 and updated in 1978, and published in *Pascal News*, No. 13, December 1978. It has been modified for MTS to support the extended Pascal language used by Pascal/VS and Pascal/JB.

PascalTidy is a Pascal program formatter. A program formatter creates a copy of a program, adding spacing and indentation in appropriate ways to make the program source file easier to read. Such "tidying" aids in debugging programs that compile correctly, but need to be proofread closely to detect logical errors. It also helps make later alterations or additions to a program easier to perform.

To format a program as the logical structure of the code requires, PascalTidy must analyze the syntax of the program, which is why the tidy operation does not work on a program that does not compile correctly. The program takes a syntactically correct Pascal program for input and writes a reformatted program to an output file assigned to SPUNCH. The output operation is carried out as each "token" (identifier, keyword, number, etc.) is assembled. PascalTidy introduces uniform spacing between tokens. It inserts blank lines, as needed, to distinguish different sections of the program. It uses uniform indentation to keep the declarations and different levels of nested statements properly aligned. The copy of the program thus created is clear and easy to understand.

The original version of PascalTidy was written to format programs written in standard Pascal. The MTS version of PascalTidy has been modified and extended to work on Pascal/VS and Pascal/JB programs.

To run PascalTidy, issue the command

```
$RUN *PASCALTIDY SCARDS=source SPUNCH=output
```

where

source	is the Pascal program to be formatted, and
output	is the formatted output produced by PascalTidy.

ERROR REPORTING

PascalTidy has very limited error recovery facilities. In most cases, after a syntax error is encountered, the formatting is abandoned and the program simply echoes the input program until end of file.

The few error messages reported are the following:

(1) '***Error***!! Check for Syntax errors!!'

Program syntactically incorrect.

(2) '***NO PROGRAM FOUND TO FORMAT.'

Input file is empty.

(3) '***STRING TOO LONG'

A character string in the program including the quotes, is greater in length than the output margins.

(4) '***Error! SPUNCH and SCARDS cannot refer to the same file.'

Cannot read and write to the same file.

DIRECTIVES

PascalTidy directives are options specified by the user to control the way formatting is done. These options all have default values which the user may override as desired. Directives are specified by the user in the form of comments, imbedded in the program.

A good place for a directive is the very first line of the program. The options may be varied for portions of the program by giving alternative directives wherever needed.

The directives are supplied to PascalTidy from within the program by using comments with a specific syntax.

Items inside brackets ([ ]) are optional in the following definitions. Items denoted by ([ ] ...) may be repeated as many times as needed. The vertical bar (|) denotes alternate values, e.g., A | B | C means A or B or C. Note that only uppercase letters are to be used for directive options.

No blank space is allowed between "{" and "[" (or the equivalent symbols) at the start of the directive comment.

Directivecomment = '{[' Directivelist ']' CommentText '}'
Directivelist = Directive [',' Directive] ...
Directive = Letter Setting
Letter = 'A' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'L' | 'N' | 'O' | 'P' | 'R' | 'W'
Setting = Switch | Value | Range
Switch = '+' | '–'
Value = '=' UnsignedInteger
Range = '=' UnsignedInteger '–' UnsignedInteger ['<' | '>']
UnsignedInteger = Digit[Digit]...
CommentText = [Any character or characters except comment closing characters]

Note: In case "{" or "[" is not available on the terminal:

[is equivalent to (.
] is equivalent to .)
{ is equivalent to (*
} is equivalent to *)

An example of a direct comment is

```
{[A=10, W=1-68, N=1-1< ]}
```

Here the A option specifies that declarations should be aligned. For example, the following line:

```
i,j:integer; area: real;
```

would be formatted as:

```
 i,
 j:        integer;
 area:     real;
```

The W=1–68 specifies that output margins are to be set at columns 1 and 68. The N directive specifies that sequence numbers starting at 1 and with increment of 1 are to be written to the right of the output margins.

Another example is

```
(*(.  G=2, E=2 .)*)
```

Here the E option is used for appending comments to "end" statements. The G option specifies that 2 spaces are to be put between symbols. See the description of the options below, and the program examples for further help in using the directives.

The PascalTidy directives are given below.

A=n

Default: A=0

This directive controls alignment of declarations. If A is set to a value greater than 0, then "n" should be greater than or equal to the maximum length of the longest identifier in the program. In this case, each identifier is placed on a separate line and alignment is to the right of the colon, equal sign, etc. If "n" is less than the number of spaces required, then "n" is ignored for that statement.

This directive visually clarifies the declaration part of the program. When A=0 (the default), the identifiers which are part of one declaration under the same type id (e.g., i, j, k: integer;) are placed on the same line, and spaces equal to the number of spaces specified by the G directive (or a default value of 1) are placed between the comma and the next identifier.

D+ or D–

Default: D+

The D directive turns the output operation on or off. D allows selective output of portions of the program during formatting. It can be used for obtaining portions of a program.

E=n

Default: E=2

The E directive generates comments containing the program name (or function or procedure name) after the "end" symbols, if no comments are there. "n" may be 0, 1, 2, or 3.

E=0 means no comments are introduced.

E=1 creates comments that follow the “end” symbols in compound statements within structured statements. In this case, the comment takes the form of the name of statement. For example, the end statement corresponding to the compound statement within a “while do” loop has the comment {While}.

E=2 creates comments after the “end” symbols constituting procedure, function, and program bodies.

E=3 means both the options, E=1 and E=2, are effective.

F+ or F–

Default: F+

The F directive turns formatting on or off. F– copies the source file without any change. Therefore, by setting F to off first and then later to on with the F directive, text (such as comments) that is already in the required format can be preserved.

G=n

Default: G=1

The G directive determines the number of spaces to be inserted between Pascal symbols while formatting. G=0 still allows one space between identifiers and reserved words, etc. The symbols “(”, “)”, “,”, “:”, and “:=” are handled independently of G.

I=n

Default: I=2

The I directive specifies the indentation count, the number of spaces by which the next line is indented. This allows indenting of each nesting level of statements and declarations by a given number of column spaces.

L=n

Default: L=4

The L directive specifies the indentation count for lines that wrap around. L determines the indentation from the first part of the statement (or declaration) for any remainder that is too long to fit on one line.

N=x–y< or N=x–y>

Default: N=0–0>

The N directive generates sequence numbers on the left or right of the write margins. “x” specifies the starting number and “y” specifies the increment. If “y” is greater than 0, then sequence numbers are written outside the specified write margins for the formatted program starting at “x”. y=0 shuts off sequence number generation. “<” writes up to 4-digit, right-justified sequence numbers with a trailing space to the left of each line. “>” writes 6-digit, zero-filled sequence numbers to the right of each line. The N directive should be combined with

the W directive. (See also the R directive.) The default is no sequence numbers.

O+ or O–

Default: O+

PascalTidy supplies blank lines for visual clarification of the program, if the blank lines are not already there. The O– option deletes blank lines other than these supplied for clarification in declarations and procedure or function bodies. This option can also be turned on and off selectively.

P=n

Default: P=2

The P directive specifies the number of blank lines to appear between procedure and function declarations and other sections of the program. "n" greater than 2 makes procedures and functions very prominent.

R=x–y

Default: R = 1–999

The R directive indicates which columns are significant for reading from the input file. R allows PascalTidy to accept programs that have sequence numbers on the right or left margins. (See also the N directive.)

W=x–y

Default: W=1–100

The W directive specifies output margins, which indicate the columns to be used for writing on the output file. Any line numbers generated with the N directive are written outside of these margins.

CONCLUSION

By default for a program without sequence numbers, PascalTidy produces a more readable program with no further effort beyond issuing the $RUN *PASCALTIDY command. PascalTidy supplies its own blank lines for visual clarification of the program. It introduces one blank line after each set of declarations like Type, Const, or Var. Procedure and function declarations are separated by two lines. Indents are two spaces and wrap-arounds are four spaces.

Comments that are appended to the ends of the statements or after keywords, etc., are output "as is", following the statements, while stand-alone comments are placed on a separate line without indentation. The F directive, which turns formatting off, can be used selectively to preserve large sections of comment text as they are. Although the directives may look complex, Pascal users should find PascalTidy easy to use.

EXAMPLES

Example 1

Here is a sample program fragment before processing by PascalTidy.

```
program sample1;
Const N=10; ABC = 'Letter'; Four =4;
Type matrix = Array(.1..N,1..N.) of real;
st_range = 1..100;
pern  = Record fname,lname :string(12) end;
color = (red,blue,green);
static this_pern : pern;
Value this_pern.fname := 'John';
       this_pern.lname := 'Doe';
Var I1,I2,I100: st_range;
Yes,No,Maybe :boolean;

Begin
 {Statements}
end.
```

Here is the output from PascalTidy with the default directive values.

```
program sample1;

const
  N = 10;
  ABC = 'Letter';
  Four = 4;

type
  matrix = array (.1 .. N, 1 .. N.) of real;
  st_range = 1 .. 100;
  pern = record
           fname, lname: string(12)
         end;
  color = (red, blue, green);

static
  this_pern: pern;

value
  this_pern.fname := 'John';
  this_pern.lname := 'Doe';

var
  I1, I2, I100: st_range;
  Yes, No, Maybe: boolean;

begin {Statements}
  end {sample1}.
```

Here is the output after adding a line with A=10 directive.

```
{(.A=10.) Align declarations }
program sample1;

const
  N =          10;
  ABC =        'Letter';
  Four =       4;

type
  matrix =     array (.1 .. N, 1 .. N.) of real;
  st_range =   1 .. 100;
  pern =       record
                 fname, lname: string(12)
               end;
  color =      (red, blue, green);

static
  this_pern:   pern;

value
  this_pern.fname := 'John';
  this_pern.lname := 'Doe';

var
  I1,
  I2,
  I100:        st_range;
  Yes,
  No,
  Maybe:       Boolean;

begin {Statements}
  end {sample1}.
```

Example 2

The following example shows how PascalTidy handles some Pascal statements. The first part is a program before processing by PascalTidy, the second part shows it after processing with the default directive values, and the third part shows it with E=3 and P=1 as options.

```
program primes;
var n: integer;
   valid        : boolean{ true if n is between 2 and 1000};
function check_primes(const n:integer):boolean;
   (*  checks if the parameter value is
       a prime *)
  var  divisor : integer;
 begin
 check_primes := true;
 for divisor := 2 to round(sqrt(n)) do
 if (n mod divisor) = 0 then begin
 check_primes := false;
 leave
 end
 end;
begin
reset(input,'unit=scards');
while not eof(input) do begin
valid := false;
repeat
read(n);
  if (n >= 2) and (n <= 1000) then valid := true
else begin
  writeln(' input value is out of range.');
get(input);
end;
until valid or eof(input);

if valid then begin
if check_primes(n) then writeln(n:4,' is a prime ')
else writeln(n:4,' is not a prime');
get(input)
end;
end;
end.
```

```
program primes;

var
  n: integer;
  valid: boolean { true if n is between 2 and 1000};


function check_primes(const n: integer): boolean;
(*  checks if the parameter value is
       a prime *)

  var
    divisor: integer;

  begin
    check_primes := true;
    for divisor := 2 to round(sqrt(n)) do
      if (n MOD divisor) = 0
        then begin
          check_primes := false;
          leave
          end
    end {check_primes};


begin {primes}
  reset(input, 'unit=scards');
  while NOT eof(input) do begin
    valid := false;
    repeat
      read(n);
      if (n >= 2) AND (n <= 1000)
        then
          valid := true
        else begin
          writeln(' input value is out of range.');
          get(input);
          end;
      until valid OR eof(input);

    if valid
      then begin
        if check_primes(n)
          then
            writeln(n: 4, ' is a prime ')
          else
            writeln(n: 4, ' is not a prime');
        get(input)
        end;
    end;
  end {primes}.
```

```
program primes;
{(.E=3, P=1.)}

var
  n: integer;
  valid: boolean { true if n is between 2 and 1000};

function check_primes(const n: integer): boolean;
(*  checks if the parameter value is
       a prime *)
  var
    divisor: integer;

  begin
    check_primes := true;
    for divisor := 2 to round(sqrt(n)) do
      if (n MOD divisor) = 0
        then begin
          check_primes := false;
          leave
          end {then}
    end {check_primes};

begin {primes}
  reset(input, 'unit=scards');
  while NOT eof(input) do begin
    valid := false;
    repeat
      read(n);
      if (n >= 2) AND (n <= 1000)
        then
          valid := true
        else begin
          writeln(' input value is out of range.');
          get(input);
          end {else};
      until valid OR eof(input);

    if valid
      then begin
        if check_primes(n)
          then
            writeln(n: 4, ' is a prime ')
          else
            writeln(n: 4, ' is not a prime');
        get(input)
        end {then};
    end {while};
  end {primes}.
```

TURBO PASCAL

MTS 20: Pascal in MTS

January 1989

INTRODUCTION

This section deals with a comparison of Turbo Pascal with Pascal/VS and Pascal/JB.

Turbo Pascal is a Pascal compiler that runs on microcomputers, implemented for the CP/M-80, CP/M-86, MS-DOS, Z-DOS, and PC-DOS operating systems. It contains some useful extensions to the Standard Pascal language. It generates fast machine code, supports editing in memory, and gives fairly good error diagnostics. The programs compile very fast. It supports facilities for running fairly large programs. Documentation for Turbo Pascal can be found in the *Turbo Pascal Reference Manual* that comes with the Turbo Pascal compiler. Turbo Pascal is a product of Borland International.

Turbo-87 is a special version of Turbo Pascal which uses the 8087 processor for real arithmetic. This results in greater speed and precision in real-number processing. Programs written for Turbo Pascal will compile and run under Turbo-87. Turbo-BCD is yet another version of Turbo Pascal, which uses binary coded decimal real numbers for higher accuracy in real-number processing. Programs written for Turbo Pascal or Turbo-87 will compile and run under Turbo-BCD. The only differences in these three versions is in the real-number processing and format.

Pascal/VS is the MTS version of the IBM Pascal compiler, which has several extensions to the Standard Pascal language. This version of the compiler is in *PASCALVS.

Pascal/JB is another Pascal compiler available on MTS. It is a product of Plug Compatible Software, Inc. Since Pascal/JB and Pascal/VS use the same language definitions of Pascal, almost everything in this documentation that applies to Pascal/VS also applies to Pascal/JB. Where there is a difference, the difference is noted. This compiler is available in *PASCALJB.

This part focuses on differences between Turbo Pascal and the Pascals on MTS, with the view of porting a program from Pascal/VS or Pascal/JB to Turbo Pascal. To do this effectively, one would need the *Turbo Pascal Reference Manual* in addition to this volume.

Turbo Pascal has some minor and some major deviations from the Pascal Standards. For this reason, compiling Pascal/VS or Pascal/JB programs with the PAR=STANDARD option will not ensure portability to Turbo Pascal.

The next three sections describe the language elements. After reading these, pay special attention to the sections, “Major Differences,” “Predefined Files,” and “Compiler Directives.” The rest of the sections, especially the section, “Standard Procedures and Functions,” can be used as a reference.

In some cases, even when the routine names and their functions coincide, there are some minor differences. So the reader is urged to look up the descriptions under the routine names. To facilitate this, a Symbol Index with the routine names has been provided. For a discussion of other available functions and procedures, one should consult the respective language reference manuals.

SYNTAX DIFFERENCES AND MINOR VARIATIONS

PASCAL/VS, PASCAL/JB	TURBO PASCAL
Line Length	
Maximum length of a program line is 100 with Pascal/VS.	Maximum line length is 127.
It is 255 with Pascal/JB.	
PROGRAM statement is necessary.	Optional.
String Declarations	
Parentheses used in syntax, e.g., Name: String(n)	Square brackets used in syntax, e.g., Name: String[n]
Pointer Declarations	
Pointer declarations and dereferencing use the "@" symbol.	The caret "^" symbol is used.
The "^" symbol may be used as an alternate.	No alternate symbol.
Comments	
{, (*, /* } *), */	{, (* (comment starting) }, *) (comment terminating)
"{" and "}" as well as "(*" and "*)" are valid comment starting and terminating symbols.	Same.
The above two notations are considered equivalent, hence may not be nested.	Not equivalent.
Nested comments use "/*" and "*/" with one of the other two forms.	Comments may be nested within comments if they use the different notations. "/*" and "*/" are not recognized. Note: If an illegal nested comment is started before the pending comment is terminated, Turbo Pascal does not flag it as an error.

PASCAL/VS, PASCAL/JB	TURBO PASCAL
Identifiers Restricted to 16 significant characters in Pascal/VS. Longer names are allowed, but they will have to be unique in the first 16 characters.	Up to 127 characters allowed.
Restricted only by line length in Pascal/JB.	Restricted only by line length.
Dollar sign "$" and underscore "_" are valid characreters in identifier syntax.	Underscore is valid, but not "$" sign.
Integer Hexconstants Up to 8 hexdigits enclosed in quotes followed by X (or x) for integers.	"$" sign prefixed to the hexdigits (up to 4 allowed).
String and Real Hexconstants Up to 16 hexdigits in quotes followed by XR (or xr) for reals.	No equivalents.
Even number of hexdigits in quotes followed by XC (or xc) for strings.	

OPERATORS AND STATEMENTS

PASCAL/VS, PASCAL/JB	TURBO PASCAL
Operators	
¬, NOT	NOT (not valid for set variables)
&, AND	AND
| |	+
<<	SHL
>>	SHR
| , OR	OR
&&, XOR	XOR
@, ^	^
Statements	
ASSERT	No equivalent.
LEAVE, CONTINUE	No equivalent. The GOTO statement can be used for leaving a loop or jumping to the end of a loop.
RETURN RETURN returns from the procedure/function block.	EXIT (Version 3.0 only) Equivalent to a GOTO statement addressing a label just before end of the current procedure/function block.
GOTO statements can transfer to statements outside the procedure as long as the transfer is within the scope of the label with certain restrictions.	GOTO statements cannot leave the current procedure/function block.
CASE statements use an OTHERWISE clause as one of the case options.	ELSE is used instead of OTHERWISE.
Declarations	
VAR	Similar. It is the only kind of variable declaration used by Turbo Pascal. Constant declarations are discussed later.
STATIC and VALUE	Typed-constant declarations are discussed later.
DEF and REF	Absolute variables are discussed later.

DATA TYPES

SIMPLE DATATYPES

PASCAL/VS, PASCAL/JB	TURBO PASCAL
Integer Integer variables use 4 bytes of memory. The allowable range is –2147483648 to 2147483647. Type Halfword=Packed –32768..32767 can be used for Integer variables that use 2 bytes of memory.	Integer variables use 2 bytes of memory. The allowable range for integers is –32768 to 32767. Arithmetic expressions should watch out for partial results causing overflow, even if the end result is within the range.
Packed 0..255 A subrange of integer. Variables of this type occupy one byte of memory.	**Byte** The predefined type Byte is a subrange of Integer of the range 0..255. Byte variables occupy one byte of memory.
Packed –128..127 Subrange of Integer. Variable occupies one byte.	No equivalent.
Real Real variables use 8 bytes of memory.	Real variables occupy 6 bytes. (Turbo-87 uses 8 bytes and Turbo-BCD uses 10 bytes for reals.)
Short Real Short Real variables occupy 4 bytes.	No equivalent type.
Boolean Boolean variables occupy 1 byte.	Same.

PASCAL/VS, PASCAL/JB	TURBO PASCAL
Char	
Character variables occupy one byte.	Same.
Refers to the characters in EBCDIC code. For example, ord('a') = hex 81	Refers to the characters in ASCII code. ord('a') = hex 61
Pointer	
Pointer variables use 4 bytes.	Also uses 4 bytes. Refers to the offset (the two least significant bytes, stored first in memory) and the base address.
Uses "@" for pointer dereferencing.	Uses caret "^" for dereferencing.

STRUCTURED DATATYPES

PASCAL/VS, PASCAL/JB	TURBO PASCAL
Packed Array of Char	
Packed Array of Char and Array of Char are two different types. The first is a string of constant length and the second is an array type.	The word Packed does not have a special significance. Packing is done where possible.
Cannot assign a String variable/constant to a variable of type Array of Char.	Can assign a String constant to a variable of type Array of Char, if lengths are same. An Array of Char can participate in string expressions, where it is treated as a String.
Can assign String variables to Packed Array of Char. If the string is longer than the packed array, then string truncation checking error will occur.	Cannot assign String variables to variables of type Array of Char, but may assign String constants if the lengths match exactly.
A Packed Array of Char cannot be assigned to a variable of type String (Pascal/VS only).	An Array of Char can be assigned to a String variable.
Lower bound has to be 1.	No bounds restriction.
Direct I/O on a Packed Array of Char variable allowed.	Read on a variable of type Array of Char is not valid.
String	
Parentheses used in String declarations, e.g., Var St: String(12);	The square brackets are used instead of parentheses, e.g., Var St: String[12];
A String variable of length "n" occupies n+2 bytes. The first two bytes store the length.	A String variable of length "n" occupies n+1 bytes. The first byte holds the length.
Maximum length for a String variable is 32767. Default length is 255 for String variables.	Maximum string length possible is 255. There is no default length associated with String variables.

PASCAL/VS, PASCAL/JB	TURBO PASCAL
Operations on Strings:	
Concatenation operator is | |.	The "+" sign is used for concatenation.
Assignment: Attempt to assign a string longer than the maximum length of the target string results in a string truncation checking error.	If the length of the string is longer than the target string's maximum length, then the string is truncated on the right. If the length is greater than 255, then a run-time error occurs.
String Comparison: The shorter string is always padded with blanks on the right before comparing strings.	The shorter string is not padded with blanks before comparison.
Records Similar in both.	
Set There is a NOT operator to denote Set Complement and A&&B denotes exclusive union of A and B.	There are no corresponding operators. A&&B is same as A+B–(A*B).
The packed sets (Packed Set of ...) occupy less storage.	There are no packed sets. Elements occupy one bit each. The bytes that are statically zero (that is, no bits in those bytes are being used) are not stored.
File Types Text is a predefined file type with component type being Char.	Same.
File of {type}, where {type} is a predefined type other than file type.	Same.
A File cannot be declared without a component type.	Type File with no type specification for the component type is used for direct access to disk files using a component size of 128 bytes.

PASCAL/VS, PASCAL/JB	TURBO PASCAL
Arrays	Similar. Exception: Array of Char.
ALFA is a predefined type which is a Packed Array[1..8] of Char;	No predefined type. But can be declared as an Array[1..8] of Char;
ALPHA is Packed Array[1..16] of Char	No predefined type. Can be declared as Array[1..16] of Char;
Space	No equivalent.
Stringptr	No equivalent.
No equivalents.	**MEM, PORT** Arrays of bytes for accessing Memory and Dataports. **MEMW, PORTW** Components of 2 bytes (for CP/M-86 and MS-DOS versions.)

PASCAL/VS, PASCAL/JB	TURBO PASCAL
Typed Constants	
Simple Constant Declarations No explicit reference to type.	Same. For example, Const Interest=12.50;
A Structured Constant Declaration There are two kinds of structured constants, Arrays and Records. Constant values are assigned to these structured constants at declaration time.	Typed Constants can be thought of as variables of the declared type, initialized with a constant value. These can be of Array, Record, or Set types.
The constants cannot be passed as Var parameters.	Typed constants can also be passed as Var parameters to procedures and functions. Compiler does not flag it as an error if these constants are reassigned values. These constants can be of simple types or structured types.

Example:
`Type Point = Record`
`  x,y,z : Integer; End;`
`Const Origin = Point(0,0,0);`
	Example:
	`Const Interest : Real = 12.50;`
	`Type Point = Record`
	`  x,y,z : Integer; End;`
	`Const Origin : Point = (x:0;y:0;z:0);`

	Typed Constants are similar to the Static variables that are initialized with the Value statement in Pascal/VS and Pascal/JB.

STANDARD PROCEDURES AND FUNCTIONS

STRING ROUTINES

PASCAL/VS, PASCAL/JB	TURBO PASCAL
Function DELETE(st,pos,num)	**Procedure DELETE(st,pos,num)**
"st" is a string variable and "pos" and "num" are integer expressions. "num" number of characters are deleted from "st" starting at position "pos", and returned as function value.	DELETE is a procedure and not a function.
If "pos+num" is greater than the length of the string, it is a run-time error.	If "pos" or "pos+num" is greater than the length of the string, then only the characters inside the string, if any, are deleted. If "pos" or "pos+num" is greater than 255, a run-time error occurs.
Can also be called without the "num" parameter to delete characters starting with "pos" to the end of the string.	
	Procedure INSERT(obj,target,pos)
No corresponding procedure. Can use the SUBSTR function and the concatenation operator to achieve the same result.	"obj" is a string expression, "target" is a string variable, and "pos" is an integer expression. "obj" is inserted into "target" at position "pos". If "pos" is greater than string length, then "obj" is concatenated to "target". If the resulting string is longer than the target's maximum length, it is truncated on the right. If it is longer than 255, a run-time error occurs.

PASCAL/VS, PASCAL/JB	TURBO PASCAL
Procedure WRITESTR(st,v) This converts "v" to a String and returns it in "st" of type String. "v" can be of any type that is legal for WRITE. "v" can be qualified with a field-width expression. There can be more than one parameter following "st". Note: The STR function of Pascal/VS (and Pascal/JB) is different. It takes a Char or Packed Array of Char variable and converts it into string.	**Procedure STR(v,st)** This converts the numeric value in "v" to a String and returns it in "st". "st" is of type String, and "v" can be real or integer and can be qualified with a field width expressions.
Procedure READSTR(st,v) This reads data from a source string "st" and returns it in "v". "v" does not have to be numeric, can consist of more than one parameter, and can be qualified with field length.	**Procedure VAL(st,v,code)** This takes a string expression in "st", with no trailing or leading blanks, and returns a numeric value for "v", real or integer, depending on the type of "v". The "code" is set to 0 or the position of the error in "st".

STRING FUNCTIONS

PASCAL/VS, PASCAL/JB	TURBO PASCAL
	Function CONCAT(s1,s2{,s3})
The concatenation operator is used. No predefined function.	Returns a string with "s1,s2,s3,..." concatenated. Arguments may be any number of strings. If the length is greater than 255, a run-time error occurs.
Function SUBSTR(st,pos,num)	**Function COPY(st,pos,num)**
Returns a substring starting at "pos" with "num" characters.	Same.
If "pos" is greater than the length of "st", a run-time error occurs. If "pos+num" is greater than the current length of "st", a run-time error occurs.	If "pos" is greater than the length of "st", a null string is returned. If "pos+num" is greater than the length of "st", only characters inside the string are returned. If "pos+num" is greater than 255, a run-time error occurs.
Can also be called without "num". Returns the substring from "pos" to the end of the string.	
Function INDEX(target,obj)	**Function POS(obj,target)**
Gives the position, where "obj" starts in "target". If no occurrence is found, then 0 is returned as value.	Same, with parameters reversed.
TRIM, LTRIM, COMPRESS, and MAXLENGTH functions	No equivalent functions.

I/O PROCEDURES AND FUNCTIONS

PASCAL/VS, PASCAL/JB	TURBO PASCAL
	ASSIGN(filevar,name)
Assigning a file is incorporated into the RESET and REWRITE statements.	This procedure asssociates the variable "filevar" with a disk file with name equal to the string expression "name". Except in the case of predefined file variables, ASSIGN must be called before any I/O operation can be performed on a file.
REWRITE(filevar,'FILE=str')	**REWRITE(filevar)**
Associates the file variable "filevar" with "str" and opens the file for output.	Creates a new disk file of the name assigned to "filevar", if one does not exist. Any existing file of that name will be erased. The file pointer is set to the beginning of the file.
RESET(filevar,s)	**RESET(filevar)**
where "s" is a string expression that gives the information about the file to be associated with "filevar" and the open options.	RESET along with ASSIGN opens the file associated with "filevar" for reading. Many of the open options of Pascal/VS do not have equivalents. Some options like the INTERACTIVE option for RESET are supported with different predefined file variables for input.
READ and READLN Procedures	**READ and READLN**
	These work differently when used with predefined file INPUT and the B+ compiler option. See "Predefined Files."
Can specify field-width parameter, e.g., READLN(I:4,S:12);	No such provision.

PASCAL/VS, PASCAL/JB	TURBO PASCAL
WRITE and WRITELN Procedures	**WRITE and WRITELN**
Use default field widths, e.g., 12 spaces for integers. Can specify field-width parameter, e.g., WRITELN(I:4), for left justification, and WRITELN(I:–4), for right justification.	By default, all variables except reals use as many spaces as needed. Field widths can be specified, similar to Pascal/VS. Negative field widths are ignored. Reals use 18 spaces and the E format, by default.

FILE HANDLING PROCEDURES

PASCAL/VS, PASCAL/JB	TURBO PASCAL
SEEK(f,n)	**SEEK(f,n)**
SEEK moves the file pointer to the nth component of "f".	Same.
In Pascal/VS, Turbo Pascal's SEEK corresponds to the default (IBMSEEK) SEEK on Record files.	An existing disk file can be expanded only by adding components to the end of the file. This is done by using SEEK(f,FILESIZE(f)). Since FILESIZE gives the total number of records in "f", and the position of first component is 0 in the file, this will put the file pointer at end of file. Note: REWRITE(f) will empty the file. There is no option corresponding to the NOEMPTY open option of Pascal/VS and Pascal/JB.
SEEK must be followed by a GET or a PUT (or a READ/WRITE) depending on whether the program is doing Input or Output.	
CLOSE(filevar)	**CLOSE(filevar)**
	Same.
UPDATE(f,opts)	
This is used for doing both input and output on the file "f".	No equivalent procedure.
GET(f)	
File pointer moves to the next record.	No separate GET procedure. This is incorporated into the READ procedure.
PUT(f)	
Writes the record assigned to the file pointer variable (f@) to the file and moves the file pointer to the next record.	This is incorporated into the WRITE procedure.

PASCAL/VS, PASCAL/JB	TURBO PASCAL
	RENAME(filevar,str)
No equivalent. Can call system routine.	Changes the name of the disk file associated with "filevar" to the name given by the string expression in "str".
	ERASE(f)
No equivalent. REWRITE(f) will empty the file. Can also call system routine.	Erases the diskfile associated with F.
	BLOCKREAD(f,v,n) and BLOCKWRITE(f,v,n)
There is no untyped file.	where "f" is an untyped file. These are high-speed transfer procedures which transfer "n" records of 128 bytes each between "f" and "v" starting at the first byte occupied by "v". These can be used for copying one disk file to another. All file-handling procedures and functions except READ, WRITE, and FLUSH can be used with untyped files.

STANDARD FUNCTIONS

PASCAL/VS, PASCAL/JB	TURBO PASCAL
	FILEPOS(f)
No corresponding function.	Gives the current position (component number) of file pointer.
	FILESIZE(f)
No corresponding function	Gives the number of component items in the file.
EOF(f) and EOLN(f)	**EOF(f) and EOLN(f)**
	Same as Pascal/VS and Pascal/JB functions when attached to files other than logical devices, such as a terminal or printer. In the default mode, Input is assigned to the CON: device and EOF and EOLN work differently in these cases. See "Predefined Files."
	SEEKEOLN(f) (Version 3.0)
No corresponding function.	Similar to EOLN(f), except that it skips tabs and blanks before testing for EOLN.
SEEKEOF(f) (Pascal/JB only)	**SEEKEOF(f)** (Version 3.0)
No corresponding function in Pascal/VS. SEEKEOF in Pascal/JB is different. It positions the file pointer to the end of the file.	Similar to EOF(f), except that it skips tabs, blanks, and end-of-line markers before testing for EOF.

MEMORY HANDLING ROUTINES

PASCAL/VS, PASCAL/JB	TURBO PASCAL
NEW(p)	**NEW(p)**
Space is allocated to represent a value of the type referred to by the dynamic variable "p".	Same.
If "p" is a variant record, space is allocated to accomodate the largest variant.	Space cannot be allocated with NEW(p) where "p" is a pointer to a variant record.
NEW can be called with tag-field specifications, if it is known which variant (and subvariants) will be active. With Pascal/VS space is allocated to accommodate the maximum variant irrespective of tag specifications. Pascal/JB uses the required amount of space for the specified tag. With the NOSHORTRECORDS compile option, Pascal/JB also allocates space to accommodate the maximum variant. Example: NEW(p,t1,t2,...), where "t1" and "t2,..." are the active variant, subvariant, etc.	In case of a variant record, GETMEM(p,n) can be used, specifying the required number of bytes to be allocated for the pointer variable "p".
DISPOSE(p)	**DISPOSE(p)**
This releases the storage obtained by NEW(p) (or New(p,t1,t2..)). With Pascal/JB, if "p" was called with NEW(p,t1,t2,..), it has to be disposed with DISPOSE(p,t1,t2,..) as required by the standards, unless the NOSHORTECORDS option is specified.	Releases storage obtained by NEW(p).

PASCAL/VS, PASCAL/JB	TURBO PASCAL
MARK(p) and RELEASE(p)	**MARK(p) and RELEASE(p)**
MARK(p) allocates a new area (heap) from which dynamic variables are allocated. "p" is a pointer variable. RELEASE(p) destroys the heap allocated by MARK(p). The parameter "p" contains the address of the associated heap control block and cannot be called with NEW(p) or DISPOSE(p).	MARK(p) assigns the value of the heap pointer to "p" and RELEASE(p) sets the heap pointer back to the address in "p". These two together are used to deallocate the memory starting from the specific dynamic variable "p" onwards. Since MARK/RELEASE and DISPOSE use different approaches to memory management, it is advisable not to mix them in a single program.
	GETMEM(p,n)
No equivalent procedure. Note that NEW can be used with variant records. Refer also to predefined datatype SPACE.	Allocates the number of bytes specified by integer expression "n" to a pointer variable "p". Must use for variant records.
	FREEMEM(p,n)
No equivalent procedure. Note that DISPOSE can be used with variant records also.	Releases the memory allocated with GETMEM(p,n).
	Function MAXAVAIL
No equivalent. Not needed on MTS, since availability of memory is not a problem.	Gives an integer result equal to the largest consecutive block of free space available on the heap. In 16-bit systems, the result is in paragraphs (16 bytes) and in 8-bit systems, it is in bytes. If more than 32767 paragraphs/bytes are available, then the result is a negative integer, which is then added to 65536.0 to get the correct value.

PASCAL/VS, PASCAL/JB	TURBO PASCAL
	Function PTR(cseg,offset)
No equivalent.	Assigns a value to a pointer. This function returns a 32-bit pointer consisting of a segment address and an offset. If P:=PTR(cseg,offset), then the functions SEG(P^) and OFS(P^) give the segment part and offset part of the pointer P.
Function ADDR(name)	**ADDR(name)**
Returns the address of the variable specified in "name". "name" can be subscripted if an array, and field qualified if a record. The address returned is an integer.	The address returned is a pointer, except in CP/M-80 implementation, where it is an integer.

DATA ACCESS AND DATA MOVEMENT ROUTINES

PASCAL/VS, PASCAL/JB	TURBO PASCAL
Functions LOWEST(s) and HIGHEST(s)	
Give the lowest and highest values in the scalar type "s". "s" is a type name or a variable.	No equivalent.
Functions HBOUND(v,i) (or HBOUND(t,i)) and LBOUND(v,i) (or LBOUND(t,i))	
where "v" is a variable of Array type (or "t" is an Array type), and "i" is an integer expression giving the array dimension. Give the upper and lower bound for indices of the array type. The value returned is same as the index type.	No equivalent functions.
The function SIZEOF(s)	**SIZEOF(s)**
where "s" is a variable name or a type name, returns the number of bytes needed for "s".	Same.
Scalar Functions PRED and SUCC	**PRED and SUCC** Same.
Procedure PACK(a,i,packed_a)	
where "a" is an array, "i" is an index into the array, and "packed_a" is a packed array of same type as components of "a", fills "packed_a" with elements of "a", starting from a[i].	No equivalent procedure.
UNPACK(packed_a,a,i)	
fills the array "a", starting from a[i] to the end of the array, with elements from "packed_a".	No equivalent procedure.

CONVERSION ROUTINES

PASCAL/VS, PASCAL/JB	TURBO PASCAL
Functions ORD, CHR	**ORD, CHR**
	Same.
The Type Conversion Routine	
This is the reverse of the ORD function. Assuming, Type Days=(Sun,Mon,Tue,Wed,Thurs,Fri,Sat); then Days(0)=Sun and Ord(Sun)=0. Integer(Sun) is not valid.	Same. Days(0)=Sun and Ord(Sun)=0. Integer(Sun) is same as ORD(Sun).
Function STR(v)	
Takes a Char or a Packed Array of Char and returns a string.	STR function converts integer/real number to a string.
Function FLOAT(n)	
Converts an integer to a real.	No predefined function. Can use INT(n) instead.

ARITHMETIC FUNCTIONS

PASCAL/VS, PASCAL/JB	TURBO PASCAL
ABS, ARCTAN, SIN, COS, LN, EXP, SQR, SQRT, ROUND, and TRUNC	These functions are same.
MAX and MIN Functions Return the maximum and minimum values of two or more expressions of a scalar type (Integer and Real expressions can be mixed).	No functions provided.
No functions provided. These are easily duplicated using the TRUNC function.	**INT(num), FRAC(num)** These functions give the integer and fractional parts of "NUM". ("NUM" can be integer or real. The value returned is real.)
RANDOM(seed) "seed" is an integer and the value returned is a real number in the range 0.0 and 1.0. The way this function is used, is to call it with a nonzero seed the first time and call it with zero, thereafter.	**RANDOM** Returns a random real number in the range 0.0 and 1.0. **RANDOM(num)** "NUM" is an integer and the value returned is an integer, in the range 0 and "NUM".

PREDEFINED FILES

When no file variable is provided, the default files INPUT and OUTPUT are used by Turbo Pascal as well as Pascal/VS and Pascal/JB. This section deals with the use of these predefined files.

INTERACTIVE INPUT

Pascal/VS and Pascal/JB:

The file is opened with RESET(Input,'Interactive');

Turbo Pascal:

It is illegal to ASSIGN or RESET any of the predefined files. The use of the default file INPUT effects the interactive mode. The B– directive corresponds to the interactive Reset of Pascal/VS and Pascal/JB.

I/O WITH PREDEFINED FILES

Turbo Pascal:

Turbo Pascal uses other predefined files such as KBD, LST, etc. When input is not to be echoed, the logical device KBD is used in place of INPUT. LST refers to the printer device. With the B+ compiler directive (the default), INPUT is assigned to the device CON: and with B–, it is assigned to TRM:. The CON: device provides buffered input, while TRM: does not.

The following hold with both the B+ and B– compiler directives: (B+ allows editing while entering input, whereas B– does not. With B–, input entries may follow the Standard Pascal formats.)

(1) Numeric variables should be followed by a blank if followed by any other variables.

(2) With Numeric input, if EOF (CTRL-Z) is true at beginning of READ, no new value is assigned. The variable retains old value.

The following hold as noted in each case. If the input line does not have the number of data items to correspond to the number of variables in READ's parameter list then:

(1) String variables in excess will be empty (with both B+ and B–).

(2) Character variables, in excess, will be set to CTRL-Z (with both B+ and B–).

(3) Numeric variables retain previously assigned values or remain uninitialized (with B+; with B–, end-of-lines are skipped when reading numeric input).

Internally, input line is stored with a CTRL-Z (end-of-file character) appended to it. So, with the construct:

```
While Not Eof(Input) Do Begin READ(variable); ...  End;
```

the following points should be noted:

(1) Numeric variables should be followed by one or more blanks, if more values are to be read; otherwise, EOF is set to true at the end of the READ operation.

(2) String variables should not be shorter than declared length; otherwise, EOF is automatically set to true after the READ operation.

With the B– directive activated, the read operation skips over EOLN to the next line of input, until EOF is true.

Pascal/VS and Pascal/JB:

The read operation skips over EOLN to next line of input, until EOF is true.

READ AND READLN

Turbo Pascal:

With the default directive (B+), when the READ procedure is used without specifying a file name, it always inputs a new line. Any remaining characters in the line buffer, including CTRL-Z, are ignored. That is, a second READ statement does not read from the same input line. A carriage return is necessary after the input values for the parameters for READ and READLN. For example,

```
Var a,b : Char;
Read(a); Read(b);
```

and the input line is "xyzCRpqr" (where CR refers to a Carriage Return). The result is a=x and b=p. For example,

```
Read(i); Read(j);
```

where "i" and "j" are integer and "1234 5678CR" gives i=1234 and leaves "j" uninitialized.

READ and READLN do not take Packed Array of Char as a valid parameter type. There is no provision for specifying format parameters with READ/READLN that determine how many characters are to be read from the input line for each variable.

Pascal/VS and Pascal/JB:

READLN(f,i:5); will read 5 characters from "f", for computing the value of "i". Packed Array of Char is a valid parameter type for READ/READLN.

WRITE AND WRITELN

The WRITE parameters can be specified with format.

Turbo Pascal:

If no field width is specified, Turbo Pascal uses minimum required number of spaces for integers and booleans and an 18-character width with E notation for reals. It ignores negative field width specifications. Characters, strings, and arrays of characters use minimum required width, by default.

The default file OUTPUT refers to the user's terminal device. The predefined file LST can be used to send the output to the printer. Note that LST is predefined and hence need not and must not be opened with an ASSIGN statement.

Pascal/VS and Pascal/JB:

Negative field widths can be used for "n" in "v:n" for left-justified output using "n" columns. The default field widths used are 12 for integers, 10 for boolean, and 20 for reals (with floating-point notation). Character variables, strings, and arrays of characters use the minimum required width, by default.

Turbo Pascal Version 3.0 has 3 additional logical devices provided, for MS-DOS systems. INP: for MS-DOS standard input file, OUT: for the MS-DOS standard output file, and ERR: for the standard error output file. For more details, see the *Turbo Pascal Reference Manual*.

COMPILER DIRECTIVES

Turbo Pascal:

Compiler directives are given as comments, e.g., {$I–}. No blanks are allowed before the "$" sign. The "+" sign activates a feature. The "–" sign deactivates a feature. Some of these directives, once set, cannot be changed in the course of a program. These are referred to as global directives. The default directives are picked to optimize execution speed, etc. So, one might want to activate some features like, R and U directives, explained later.

Pascal/VS and Pascal/JB:

The PAR field of the compiler-invocation command specifies the compiler options. For example,

```
$RUN *PASCALVS ...  PAR={compiler options}
```

Some of these options, such as LIST, PRINT, and CHECK can be turned on (or off) with the %option ON (or %option OFF} statements in the source program.

Pascal/JB allows the use of "/" symbol as an alternate to the "%" symbol, if the symbol is in column 1.

The Turbo Pascal compiler directives are as follows:

B - Global directive

The B directive has to go at the beginning of the program. {$B+} assigns console device CON: for standard file INPUT. This option allows buffered input; that is, editing is possible while entering an input line. {$B–} assigns the TRM: device. This does not allow editing while entering input. With this option, the I/O corresponds to the Standard Pascal I/O. The default is B+.

C - Global directive

This controls whether the CONTROL character is to be effective ({$C+}) or is to be ignored ({$C–}) during CONSOLE I/O. For example, CTRL-C in response to READ/READLN will interrupt execution, if C is active. The default is C+.

I

I controls I/O error handling. If deactivated ({$I–}), it is the user's responsibility to check for errors using the function IORESULT. See the Reference Manual for details. The default is I+.

I filename

{$I filename} includes the file "filename" in the source code.

For Pascal/VS and PASCAL/JB, %INCLUDE filename

R

This controls range checking. If R is active ({$R+}), the array index checks and subrange checks are made. The default is R–.

Pascal/VS and Pascal/JB allow separate enabling (or disabling) of checking, e.g., %CHECK subrange ON (or %CHECK subrange OFF). All checking is enabled by default. Individual features can be disabled as in the example above. The %CHECK OFF statement suppresses all checking as does the NOCHECK compiler option.

V

This activates type checking on strings passed as Var parameters. With V active ({$V+}), the lengths of actual and formal parameters must match. The default is V+.

Note: In a procedure declaration, the type field of a string parameter must have a type ID, i.e., STRING[n] cannot be used in the type field. Instead, a type STRG=STRING[n] must be declared and used. With V–, the actual parameter and formal parameter need not have matching string lengths.

For Pascal/VS and Pascal/JB, conformant string parameters are used. That is, strings can be declared in the procedure declaration as of type String and any string, irrespective of length, will be compatible.

U

If U is active ({$U+}), CTRL-C can be used to interrupt the execution. The default is U–

There are other directives some of which apply only to CP/M-80, such as the A, W, and X directives. A+ is the default, and with A+ active, only nonrecursive code is generated. A– allows recursive calls. For the X and W directives and the other directives that apply only to MS-DOS or PC-DOS operationg systems, refer to the *Turbo Pascal Reference Manual*.

USE OF INCLUDE FILES & EXTERNAL PROGRAMS

There is no facility to support simple use of external compilation in Turbo Pascal. There are predefined functions/procedures provided to access the operating system's routines. External programs, which contain executable machine code can be called from a Turbo Pascal program.

INCLUDE DIRECTIVE

Turbo Pascal:

Source from another file can be inserted in the file by including the following line where it is to be inserted:

```
{$I INCFILE}
```

This inserts INCFILE in place of the include statement.

Pascal/VS and Pascal/JB:

The %statement is used instead:

```
%INCLUDE INCFILE
```

This can include the entire file or the include files can use a directory and include only some sections of the file.

Note: Pascal/JB allows /INCLUDE in addition to %INCLUDE.

Turbo Pascal:

The OVERLAY feature can also be used for breaking up source files (see the section, "Major Differences").

CHAIN AND EXECUTE PROCEDURES

These procedures are used for calling one Turbo Pascal program from another.

Before compiling, use of the O command lets you select a compiler option to direct the compiled code

(1) either into memory—choose M,

(2) or into a .COM file (.CMD with CP/M versions)—choose C,

(3) or into a .CHN file—choose H.

The main program code goes into the .COM file. The chained program code goes into .CHN file.

The main program uses the CHAIN(filename) statement to activate the .CHN file. EXECUTE(filename) activates any Turbo Pascal .COM (.CMD) file.

On selecting H (or C), a memory usage menu is displayed. For the main program, the minimum code and data-segment size should be set to at least the same value as the largest of the code and data-segment sizes from the chained programs. This can be done by saving the code and data-segment size information displayed at the end of the compilation for the chained programs.

A .COM file can be executed directly from the operating system. A .CHN file can only be activated from another Turbo Pascal program, since it does not include the Turbo Pascal library. Data can be shared through GLOBAL variables and by ABSOLUTE ADDRESS variables. If data is shared through the GLOBAL variables, then these will have to be declared as the very first variables in both the main and chained programs and listed in the same order, to ensure overlapping. This can be best achieved with the use of include files. Also both programs must be compiled to the same size of code and data segments.

ABSOLUTE VARIABLES are declared to reside at specific addresses. This is done by adding to the VAR declaration, the word ABSOLUTE followed by two integer constants (separated by the colon ":" symbol), a segment address, and an offset.

A variable can be declared to reside starting at the same address as another variable by specifying the second variable for the address. For example,

```
VAR strg : STRING[16];
strlen : BYTE ABSOLUTE strg;

i : INTEGER ABSOLUTE $000:$00F0;
```

The standard identifiers CSEG and DSEG, which refer to the code segment and the data segment, can be used as segment addresses.

USE OF EXTERNAL SUBPROGRAMS

Turbo Pascal:

External procedures contain executable machine code. Parameters can be passed as with regular procedures. The procedure declaration is followed by the word EXTERNAL and a string constant with the name of the file where the external program is located, the file being a .COM or .CMD file.

These files are loaded and placed into the object code during the compilation of a program. The object produced by the external procedure must be relocatable, and also the external code must save and restore the BP, CS, DS, and SS registers on entry and before return.

System routines in Turbo Pascal are callable using the MS-DOS function (BDOS or BIOS in CP/M systems). Refer to the *Turbo Pascal Reference Manual* and the operating system manual for the specific operating system.

Inline Code: Turbo Pascal allows use of machine-code instructions directly into the program text with the INLINE statement. See the *Turbo Pascal Reference Manual* for details.

Pascal/VS and Pascal/JB:

The FORTRAN directive can be used to call external non-Pascal routines and system routines. The EXTERNAL directive is used to call Pascal routines that are part of another compilation module. The DEF declaration can be used for generating common storage for variables shared across compilation units.

MAJOR DIFFERENCES

DEVIATIONS

The following are the major differences between the Pascals on MTS and Turbo Pascal. Some of these cases are also deviations from the Pascal Standards.

(1) The GET and PUT procedures are not implemented. There is no file pointer (f@) defined corresponding to a file "f".

(2) EOF and EOLN work differently when using the default file INPUT with the B+ (default) directive.

(3) Functions and procedures cannot be passed as parameters.

(4) With CP/M-80 versions, recursive procedures cannot use local variables as VAR parameters.

(5) GOTO statements cannot leave the procedure/function block.

(6) There is no facility for doing modular compilation.

(7) Even with the V– option, the STRING type in the formal parameter of a procedure declaration needs a predeclared TYPE id. (for e.g., declare TYPE Strg=STRING[50]; use Strg in place of STRING[50]). A function which returns a string also needs to use a TYPE id.

(8) PACK, UNPACK, and PAGE procedures are not implemented.

ADDITIONAL FEATURES

(1) There are many screen-related routines that are not applicable to Pascal/VS or Pascal/JB.

(2) Turbo Pascal has an OVERLAY system to conserve memory usage for large programs.

(3) Turbo Pascal has Graphics, Window, and Sound procedures for the IBM-PC and compatibles.

(4) Turbo Pascal version 3.0 has Turtlegraphics for the IBM-PC and procedures to manipulate MS-DOS directories. Turbo-BCD has a FORM function which corresponds to the PICTURE function of Pascal/VS and Pascal/JB.

OVERLAY System

This allows programs to use larger memory than is available on the computer. The code for procedures/functions preceded by the word OVERLAY are placed in an OVERLAY file. Consecutive OVERLAY routines are placed in the same OVERLAY file. The main program code reserves an

OVERLAY area equal to the memory needed by the largest of the overlay routines. At execution time, the OVERLAY procedure is loaded into memory if it is not already there. OVERLAYS can be nested. There can be more than one OVERLAY file associated with a program.

Restrictions

OVERLAY procedures cannot call each other. Programs using OVERLAY cannot be compiled with the MEMORY compiler option.

In conclusion, Pascal/VS or Pascal/JB programs that

(1) do not make use of system routines that do not have equivalent routines in Turbo Pascal,

(2) do not call programs written in other languages, and

(3) do not use long strings,

can be, with some changes, ported to Turbo Pascal. In the other direction, Turbo Pascal programs that do not make use of the machine-dependent features of the language (such as absolute variables specifying addresses), and do not use the Sound or screen-related procedures (Color, Graph, etc.) can be ported. Since Turbo Pascal does deviate from the standards in some cases, even programs written in Standard Pascal might need conversion before compiling and running under Turbo Pascal.

APPENDIX A
SYMBOL INDEX

INDEX

Reader's Comment Form

Pascal in MTS

January 1989

Errors noted in publication:

Suggestions for improvement:

Date _____

Name _____

Address _____

Your comments will be greatly appreciated. Please fold the completed form as shown on the reverse side, seal or staple, and drop in Campus Mail or in the Suggestion Box at any Campus Computing site.

fold here

Publications
Computing Center, User Services
The University of Michigan
Ann Arbor, Michigan 48109
USA

---

fold here