HONEYWELL

# MULTICS PASCAL USER'S GUIDE

SOFTWARE

# MULTICS PASCAL
# USER'S GUIDE

SUBJECT

Description of the Multics Implementation of Pascal

SOFTWARE SUPPORTED

Multics Software Release 10.2

ORDER NUMBER

GB62-00

March 1984

Honeywell

# PREFACE

This user's guide describes Pascal on Multics. It describes the various Multics extensions to standard Pascal and the few ways in which Multics Pascal deviates from standard Pascal. It is intended as a user's guide to the Multics implementation of Pascal rather than a reference manual of the Pascal language. This manual does not attempt to provide the reader with extensive knowledge of the Multics system. The reader is referred to the Multics Programmer's Reference Manual or Introduction to Programming on Multics for details on programming in the Multics environment.

The software product identified as the Multics Pascal Compiler is the property of the Centre Interuniversitaire de Calcul de Grenoble and the Institut National de Recherche en Informatique et en Calcul. Authorship of the Multics Pascal Compiler is attributed to the Centre Interuniversitaire de Calcul de Grenoble and the Centre de Recherche en Informatique pour les Sciences Sociales - Grenoble.

Section 1 is an introduction to the manual.

Section 2 explains how to compile a Pascal program on Multics.

Section 3 details the Multics extensions to standard Pascal.

Section 4 describes the implementation of various Multics Pascal features.

Appendix A lists the Multics deviations from standard Pascal and includes a table of Multics Pascal implementation restrictions.

Appendix B lists the French translation of Pascal reserved symbols.

Appendix C describes the Pascal commands.


File No.: 1L23, 1U23 GB62-00

The following symbols are used in this manual:

- Braces {} indicate an optional argument entry.
- Lowercase letters enclosed in angle brackets <> indicate a symbolic variable whose exact value you must supply.
- The vertical bar (|) is used as an "or" conjunctor.

Each section/appendix of this document is structured according to the heading hierarchy shown below. Each heading indicates the relative level of the text that follows it.

| LEVEL | HEADING FORMAT |
|---|---|
| **1 (highest)** | **ALL CAPITAL LETTERS, BOLD TYPE FACE** |
| **2** | **Initial Capital Letters, Bold Type Face** |
| *3* | *ALL CAPITAL LETTERS, ITALICS TYPE FACE* |
| *4* | *Initial Capital Letters, Italics Type Face* |

# CONTENTS

*Tables*

# SECTION 1

# INTRODUCTION

Multics Pascal is based on the standard ISO Pascal. In addition, extensions to the standard make Multics Pascal a truly integrated part of the Multics programming environment (see Section 3).

Pascal is a popular language because of its carefully chosen control structures and powerful data structuring capabilities. Because of these, programs written in Pascal are easy to read. Pascal is recommended for teaching introductory programming and well-structured programming in general.

# SECTION 2

# COMPILING AND EXECUTING A PASCAL PROGRAM

A Pascal source segment is compiled by issuing the "pascal" command. The command line:

```
pascal prog1
```

compiles a source segment named prog1.pascal. The ".pascal" suffix is assumed by the command and does not have to be specified. See Appendix C for a description of all of the available compiler options.

## CHARACTER SET AND IDENTIFIERS

Multics Pascal lets you use the full ASCII character set but there is no difference in identifiers between uppercase characters and lowercase characters. Variable and type names, program names, file names, and the names of all "imported" and "exported" variables and procedures are converted to lowercase by the Pascal compiler. The name used to run a Pascal program *must* be in lowercase (character string constants and comments are not converted).

You can specify up to 32 characters for Pascal identifiers. Programs compiled in "nonstandard" mode can use the underscore (_) character in identifiers.

In the Multics Pascal character set, the following symbols are equivalent:

```
# is a synonym for <>
@ is a synonym for ^
(* is a synonym for {
*) is a synonym for }
```

## PASCAL COMMANDS

The following Pascal commands are available (for details, refer to Appendix C of this manual).

| Command Name | Command Description |
|---|---|
| pascal_area_status | Displays information about the maximum size and location of a Pascal area. |
| pascal_create_area | Creates a Pascal area. |
| pascal_delete_area | Deletes a Pascal area. |
| pascal_file_status | Displays the status of active Pascal files. |
| pascal_indent | Improves the readability of a Pascal program by indenting it according to standard conventions. |
| pascal_reset_area | Frees the blocks allocated in a Pascal area. |
| pascal_set_prompt | Sets the Pascal prompt string. |

## COMPILING INTERACTIVE PROGRAMS

A pascal program that requires the user to enter input in response to a prompt issued by the program is called an "interactive program".

A program that accepts input from the keyboard must have a program heading of the form:

```
program progname(input);
```

A program that directs output to the terminal must have a program heading of the form:

```
program progname(output);
```

A program that accepts input from the keyboard and directs output to the terminal must have a program heading of the form:

```
program progname(input,output);
```

The program heading can also specify other user files.

Interactive programs must be compiled with the -interactive (or -int) control argument. The format is:

```
pascal prog_name -interactive
```

For more information on interactive programs, refer to Section 4.

**Example**

The following simple interactive pascal program provides the square of a number that the user specifies in response to a program prompt:

```
program square(input,output);
var
  number:integer;
  sqrvalue:integer;
begin
  repeat
     writeln('Enter a number:');
     readln(number);
     sqrvalue := number*number;
     writeln('square equals ',sqrvalue);
  until number = 0
end.
```

Compile this interactive program, square, as follows:

```
pascal square -interactive
```

Execute the program by typing:

```
square
```

The program then prompts the user for input, as follows:

```
Enter a number:
? 12
square equals 144
      .
      .
      .
```

## FORMATTING A PASCAL PROGRAM

The pascal_indent command lets you enter your Pascal program without regard to standard Pascal formatting conventions. Once you have created a source program, invoke the pascal_indent command. This command formats the program, indenting where necessary. Although adherence to a specific format is not required in a Pascal program for it to compile successfully, proper formatting makes a Pascal program easy to read and understand. If you use the "-highlight" control argument to the pascal_indent command, all Pascal keywords are converted to uppercase for legibility.

### Example

The following program, entitled div_mod, prints out the results of the *div* and *mod* Pascal operators applied to two specified integer arguments. The program is designed to be used as a Multics command: arguments are typed on the command line, results are printed on user_output, and error messages are printed on error_output.

Note the use of the *argc* and *argv* functions which are Multics extensions to standard Pascal. Also note the use of the nonstandard *sread* function which is used to convert arguments (character strings) to integers.

The text of the program is entered without regard to any formatting conventions:

```
program div_mod(output, error);

var
index, it,
n_args:integer; (*number of arguments*)
v:array[1..2] of integer; (*values of arguments*)
str:packed array[1..256] of char; (*buffer for argument strings*)
err_in_args:boolean;

begin
(*process arguments*)
err_in_args:=false;
n_args:=argc;
if n_args <> 2 then
err_in_args:=true
else
for it:=1 to 2 do
```

```
begin
argv(it,str);
index:=sread(str, 1, v[it]);
if index = -1 then err_in_args:=true;
end;
(*print results*)
if not err_in_args then
begin
if v[2] = 0 then
writeln('DIV: division by zero not allowed.')
else writeln(v[1], ' DIV ', v[2], '=', v[1]div v[2]);
if v[2]<= 0 then
writeln('MOD: negative or null right arg is not allowed.')
else writeln(v[1], 'MOD', v[2], '=', v[1]mod v[2]);
end
else
writeln(error, 'div_mod: usage: div_mod <integer> <integer>');
end.
```

To format the above program, issue the pascal_indent command as follows:

```
pascal_indent div_mod -hl -com 55
```

The formatted program looks like this:

```
PROGRAM div_mod(output, error);

VAR
    index, it,
    n_args: integer;                          (* number of args *)
    v: ARRAY [1..2] OF integer;               (* values of args *)
    str: PACKED ARRAY [1..256] OF char;       (* buffer - arg strings*)
    err_in_args: boolean;

 BEGIN
                                                (* process args *)
    err_in_args := false;
    n_args := argc;
    IF n_args <> 2 THEN
       err_in_args := true
    ELSE
       FOR it := 1 TO 2 DO
          BEGIN
             argv (it, str);
             index := sread (str, 1, v [it]);
             IF index = -1 THEN err_in_args := true;
          END;
                                                (* print results *)
```

```
   IF NOT err_in_args THEN
      BEGIN
         IF v [2] = 0 THEN
            writeln ('DIV: division by zero not allowed.')
         ELSE writeln (v [1], ' DIV ', v [2], '=', v [1] DIV v [2]);
         IF v [2] <= 0 THEN
            writeln ('MOD: negative or null right arg is not allowed.')
         ELSE writeln (v [1], 'MOD', v [2], '=', v [1] MOD v [2]);
      END
   ELSE
      writeln (error, 'div_mod: usage: div_mod <integer> <integer>');
END.
```

# SECTION 3

# MULTICS PASCAL EXTENSIONS

This section contains descriptions of the Multics extensions to the ISO Pascal standard. These extensions are meant to augment the standard Pascal language and make programming in Pascal on Multics easier and more versatile. The extensions tailor standard Pascal to the Multics environment. The Multics extensions are not mandatory and programs written using the standard features of Pascal can be run without modification on Multics.

The Multics extensions to the Pascal standard include:

- Underscores in identifiers
- Assignment between character strings of different length
- Dynamic allocation reset
- Importing and Exporting Variables, Functions, and Procedures
- $value directive
- $include directive
- File extensions
- Conditional compilation ($options)
- Arguments passed to a Pascal program
- *log10* function
- *otherwise* extension in *case* statement
- Predefined constants *maxreal* and *minreal*
- Octal, hexadecimal, and binary notation for integers
- Clock, date, and time functions
- *sread* and *swrite* functions

## UNDERSCORES IN IDENTIFIERS

Multics Pascal allows the underscore (_) character as part of identifier names. This is not allowed in standard Pascal.

## ASSIGNMENT BETWEEN CHARACTER STRINGS OF DIFFERENT LENGTH

This option permits you to specify a character string array of a specific length and then assign it the value of a shorter string. The target string is padded with blanks. In standard Pascal, this operation is not allowed and assignment must involve two strings of equal length.

**Example**

```
string4 : packed array [1..4] of char;
  .
  .
  .
string4 := 'Ab';    (* string4 is padded on the right with blanks *)
```

## DYNAMIC ALLOCATION RESET

Dynamic allocation reset (*reset*) deallocates the block pointed to by block_pointer (as for the *dispose* statement) and deallocates *all* the blocks allocated since the allocation of this block.

As you make successive calls to alloc, it moves the "next-free" pointer logically upwards. If you issue a *reset*, the pointer is returned to the value that it had just before you did the alloc(foo) for which you are now doing a reset(foo). This allows you to clean up all the allocs that you do in, say, a procedure invocation when you are about to exit the procedure.

The syntax is:

```
<reset_area_statement> =
   reset (<variable_access>)
```

where variable_access is a reference to a variable of pointer type.

**Example**

```
reset (block_pointer)
```

## IMPORTING AND EXPORTING VARIABLES, FUNCTIONS, AND PROCEDURES

Pascal programs can call routines written in any Multics-supported language. Pascal programs and programs written in languages other than Pascal can access Pascal procedures and variables that have been defined in an $export section (see below).

The following two compiler directives let you import or export variables, functions, or procedures from other Multics supported languages:

- *$import* - import variables, functions, or procedures
- *$export* - export variables, functions, or procedures

The $import directive must appear before the $export directive and both must appear immediately after the program header.

Table 4-2 list the data equivalences for importing or exporting PL/1 and FORTRAN variables.

### $Import Directive

The $import directive lets you import procedures and variables defined in any Multics supported language.

To import procedures and variables defined in programs, define all external procedure and variable names in the $import directive immediately following the program heading. Note that you must terminate the list of external names in the $import directive with a $.

The syntax for the $import directive is:

```
<import_directive> =
     $import <imported_list> {;<imported_list>} $

<imported_list> =
     <external_segment_description_string>: <identifier> {,<identifier>}
     |   <external_entry_description_string>: <identifier>
<external_segment_description_string> =
     ' <segment_name> (<generator_name>) '
     |   ' <external_static> '
<external_entry_description_string> =
     ' <segment_name> $ <entry_name> '
```

### Example

```
PROGRAM example;
$import
   'pl1_program (pl1)': proc1, funct1;
   'fortran_program (fortran)': proc3;
   'external_static': v1, v2, v3; (* allocated in external
                                     static standard area *)
   'pascal_program (pascal)': proc5, v4, v5;
   'segment_x$procedure_y (pl1)': proc_xy;
   'static_data (cds)': v6, v7 $ (* allocated in a data segment
                                    created by the cds command *)
```

You must declare imported names in the var section in the standard manner. Procedures and functions in the $import section must be declared as external, where external takes the place of the body of the procedure.

In the following example, declarations are made for the $import section described above.

**Example**

```
var
    v1, v2, v3, v4, v5, v6, v7: integer;
procedure proc1; external;
function funct1: real; external;
procedure proc3 (var i, j: integer); external;
procedure proc5; external;
procedure proc_xy (var a: real); external;
```

**$Export Directive**

A Pascal program that wants to export (make public) procedures or variables for another program (written in either Pascal or any other Multics supported language) must use the nonstandard $export directive. The $export section must appear immediately after the $import section if there is one), otherwise specify it immediately after the program heading.

The maximum size of internal (not imported or exported) globals in a Multics Pascal program is 16384 words. The $export directive lets you export large arrays and large variables.

The syntax for the $export directive is:

```
<export_directive> =
     $export <identifier> {,<identifier>} $
```

where <identifier> is the name of an exported variable, function, or procedure.

**Example**

```
$export

  proc6, v8, v9 $

var
  v1, v2, v3, v4, v5, v6, v7, v8, v9:  integer;
procedure proc1; external;
function funct1:  real; external;
procedure proc3 (var i, j:  integer); external;
```

```
procedure proc5; external;
procedure Proc_xy (var a:  real); external;

procedure proc6;
  begin
  end;

begin
end.
```

## $VALUE DIRECTIVE

The $value compiler directive initializes the values of specified variables. After the *var* section, and before procedure declarations, you can insert a $value section that lets you initialize global variables declared in the preceding *var* section. The $value section is not allowed for internal procedures.

Initializations declared in the $value section are performed the first time and each additional time that the program segment is made known. If these variables are modified, they keep their new value for subsequent executions, until the program segment is terminated. For each execution, initialization must be done explicitly by assignments within the program.

Reinitialization can be forced by terminating and re-initiating the segment. Reinitialization is not performed for global variables.

The syntax for the $value section is:

```
<value_directive> =
     $value <identifier> = <value> {;<identifier> = <value>} $
```

When using the $value section, observe the following rules:

- The variables must appear in the order of their declarations in the *var* section.
- Use only single constants.
- The form N*constant can be used to initialize an array or subarray of N elements.
- Initialization of records is not allowed.
- Initialization of any packed data structure (other than packed array [i..j] of char) is not allowed.

**Example**

```
program value (output);
  var
    i, j: integer;
    t: array [1..3] of real;
    ch: array [boolean, 1..3] of integer;
    xx: packed array [1..23] of integer;
    str: packed array [1..4] of char;

$value
    i = 3;
    t = (2, 3.4, 5.002);
    ch = (4 * 999, 888, 777);
    str = 'abcd' $

begin
    j := 3;
    writeln ('I =', i);
    writeln ('J =', j);
    i := i + 1;
    if i = 5 then
    for j := 1 to 3 do
    write (ch [false, j], ch [true, j]);
end.
```

The above program can be compiled and executed as follows:

```
pascal demo -ns
PASCAL 8.00
r 10:19 1.240 157

demo
I =           3
J =           3
r 10:20 0.092 6

demo
I =           4
J =           3
999         999       999         888         999         777
r 10:20 0.074 0
```

(The run command causes temporary reinitialization. See the *Commands and Active Functions* manual.)

```
run demo
I =            3
J =            3
r 10:20 0.419 42

demo
I =            5
J =            3
r 10:20 0.061 0

tmr demo
r 10:20 0.137 2

demo
I =            3
J =            3
r 10:20 0.081 2

demo
I =            4
J =            3
999          999       999          888          999          777
r 10:20 0.083 0
```

## $INCLUDE DIRECTIVE

You can insert stored portions of text in a program at compilation time by using the $include compiler directive. Include files are particularly useful when the same set of declarations is to be used in several programs. They ensure that the declarations are identical in all programs.

Include files eliminate redundant work and reduce the likelihood of errors whenever more than one program references the same structured data. Include files can also be used to guarantee identical assumptions about naming conventions and systems of encoded values. If an include file exists that describes a given data structure, that include file should be used rather than creating a different one describing the same structure.

The syntax of the $include directive is:

```
<include_directive> =
  $include ' <file_name> ' {,<begin_string>, <end_string>} $

<begin_string> = <include_file_delimiter>
<end_string> = <include_file_delimiter>
<include_file_delimiter> = <character_string> | *
```

where begin_string and end_string are either quoted character strings to specify starting and ending positions within the included file, or * to indicate the beginning and end of the file.

If begin_string is 'foo', for example, the included portion of file_name begins with the character immediately following the first occurrence of the literal string foo. If end_string is bar, for example, the included portion of <file_name> ends with the character immediately preceding the first occurrence of bar. These strings are NOT interpreted as qedx regular expressions. They are Pascal character strings, with the standard interpretation of single quote ('); two consecutive single quotes ('') inserts a single quote.

The included file is named file_name.incl.pascal and is found via the translator search list.

Occurrences of $include in the include file itself are expanded recursively.

**Example 1**

```
$include 'foo' $
```

or:

```
$include 'foo', * , * $
```

includes the entire file foo.incl.pascal

**Example 2**

```
$include 'foo', 'AAA', * $
```

includes the file foo.incl.pascal from the first character following the first occurrence of the string 'AAA' to the end.

**Example 3**

```
$include 'foo', * , 'BBB' $
```

includes the file foo.incl.pascal from the beginning to immediately before the first occurrence of the string 'BBB'.

## FILE EXTENSIONS

Multics Pascal provides several facilities that let you connect to sequential files or direct access files from within a Pascal program or from command level. For details on file input and output, see Section 4.

## $OPTIONS DIRECTIVE

The $options compiler directive accepts a variety of keywords that control the format of the output compilation listing, the portion of the source that is compiled, and the generation of debugging checks in the code as produced by pascal -debug.

The syntax for the $options directive is:

```
<options_directive> =
   $options option_name = <option_value>
   {;<option_name = <option_value>} $

<option_value> = true | false | {not} <switch_name>
```

where:

switch_name
: is the identifier of a compilation switch that has been previously assigned a value by a compilation switch assignment directive or by the -cond argument to the pascal command.

option_name
: is one of the following:

  listing
  : Determines whether source text appears in the listing. (default is true)

  debug
  : Determines whether generated code includes debugging checks. (default is true)

  page
  : Determines whether listing output skips to a new page.

  compile
  : Determines the part of the source text subject to conditional compilation. (default is true)

The syntax of the compilation switch assignment directive is:

```
<compilation_switch_assignment_directive> =
  $options <switch> <switch_name> {:= <switch_value>}
  {,<switch_name> {:= <switch_value>}} $

<switch_value> = true | false.
```

$options <switch> trace $

is equivalent to:

$options <switch> trace := false $

### Page Breaks in Listings

The following directive causes the compilation listing to skip to a new page:

```
$options page $
```

### Listing Source Text

The following directive controls whether subsequent source lines are written to the compilation listing. The default is true:

```
$options listing = true $
or:
$options listing = false $
or:
$options listing = foo $
or:
$options listing = not foo $
```

### Debugging Checks

The following directive controls whether special code is to be generated to cause harmless faults for uninitialized pointers and to provide other safeguards against program errors. These checks are the same ones that are generated by the -debug control argument to the pascal command. The default value is true. All values for debug are overridden by -debug or -no_debug on the command line:

```
$options debug = true $
or:
$options listing = false $
or:
$options debug = foo $
or:
$options debug = not foo $
```

### Conditional Compilation

The following directive controls whether succeeding text is to be included in the compilation. This feature allows a source segment to contain multiple versions of program text, which are selected based on the values of switches as assigned by the -cond control argument to the pascal command. The default value is true.

```
$options compile = true $
or:
$options compile = false $
or:
$options compile = foo $
or:
$options compile = not foo $
```

### Example

The following sample conditional compilation program prints a terse message if the compiler is invoked with "-cond brief true", otherwise it prints a longer message:

```
procedure print_not_found (name:packed array [a..b:integer] of char);
  begin
  $options switch brief := false $    (* default is not-brief *)
  $options compile = brief $
     writeln ('Not found: ', name);
  $options compile = not brief $
     writeln ('Unable to find program, check library for: ', name);
  $options compile = true $
  end;
```

## PASSING ARGUMENTS TO A PASCAL PROGRAM

You are allowed to pass arguments to a Pascal program.

The predeclared function *argc* returns the number of arguments passed to the main procedure by the command processor (similar to cu_$arg_count).

The predeclared procedure *argv* (<expression>, <parameter>) where <expression> evaluates to an integer, returns in the character string <param> the <expression>'th argument. Args are numbered from 1 to argc. (similar to cu_$arg_ptr).

A fatal error occurs if an argument list passed to the main procedure has no descriptors or if the referenced argument is not a character string or does not exist.

## LOG10 FUNCTION

The log10 function returns the base 10 logarithm of a specified real argument.

### Example

```
log_value := log10 (expression);
```

## OTHERWISE EXTENSION IN CASE STATEMENT

The case statement permits the use of the *otherwise* extension, as follows:

Standard syntax:

```
<case_statement> =
   case <case_index> of
    <case_list_element> {;<case_list_element>}
    {;<otherwise_statement> {;<statement>}} {;} end.
```

Extended syntax:

```
<extended_case_statement> =
   case <case_index> of
   <case_list_element> {;<case_list_element>}
   {;otherwise_statement {;<statement>}} {;} end.
```

## MAXREAL AND MINREAL CONSTANTS

*Maxreal* and *minreal* are predefined constants in Multics Pascal. The *maxreal* constant contains the largest positive real number allowed in Multics Pascal, and *minreal* contains the smallest nonnull real number allowed in Multics Pascal.

## INTEGER NOTATION

Multics Pascal lets you define integers in octal, hexadecimal, and/or binary notation. For example, J := 45; is equivalent to:

```
J := '2d'x;       (hexadecimal)
J := '55'o;       (octal)
J := '101101'b;   (binary)
```

## CLOCK FUNCTION, DATE FUNCTION, AND TIME PROCEDURE

The *clock* function returns virtual cpu process time in milliseconds.

The *date* procedure returns an 8-character date of the form MM/DD/YY.

The *time* procedure returns an 8-character time of day of the form HH:MM:SS.

### Examples

```
var cpu_time: real;
date_string, time_string: packed array [1..8] of char;
...
cpu_time := clock;
date (date_string);
time (time_string);
```

## SREAD AND SWRITE FUNCTIONS

The *sread* and *swrite* functions operate on strings in the same way that the standard *read* and *write* operations manipulate file variables.

The syntax of the *sread* function is:

```
<sread_function_designator> =
   sread (<string_variable_access>, <integer_expression>,
   <variable_access> {,<variable_access>})
```

where:

string_variable_access
: is a reference to a variable of string type (packed array of char9) from which values are read.

integer_expression
: is the integer value of the index of the first character to read from the string.

variable_access
: is a reference to a variable of type real, integer, or character that receives the read value.

The returned value, of type integer, is the current index position in the string after the operation is finished.

**Example**

```
i := sread (string, index, foo, bar)
```

In this example, *sread* reads the values of the variables foo and bar from the character string, starting at the position designated by index. When it is finished, "i" is assigned the value of the index position of the next character in the string.

The syntax of the *swrite* function is:

```
<swrite_function_designator> =
   swrite (<string_variable_access>, <integer_expression>,
   <swrite_parameter> {,<swrite_parameter>})
```

where:

string_variable_access
: is a reference to a variable of string type (packed array of char) into which the parameters are written.

integer_expression
: is the integer value of the index of the first character to be written into the string.

swrite_parameter
is any parameter allowed in the standard write statement, or a parameter of the form:

string_variable: length: start

to specify a substring of the string_variable, where length and start are integer expressions.

The returned value of type integer is the current index position in the string after the operation is finished.

**Example**

The Pascal assignment:

```
i := swrite (s1, 10, s2:3:5)
```

is equivalent to the PL/1 assignment:

```
substr (s1, 10, 3) = substr (s2, 5, 3);
```

Note that the variable "i" contains the value 15 after the operation.

# SECTION 4

# PASCAL IN THE MULTICS ENVIRONMENT

Features specific to the Multics implementation of Pascal are discussed in this section.

## USING PASCAL FILES IN THE MULTICS ENVIRONMENT

The intent of this section is not to explain Pascal input and output (I/O) procedures; a standard Pascal text can be used for this purpose. The intent is to describe how to direct I/O to and from Multics files.

### Declaring Pascal Files

Multics Pascal has three predefined files named *input, output*, and *error*. These files must, when they are used, be named in the parameter list of the program header. They should not be declared in the *var* section of the main program. They have the following default attach descriptions:

Table 4-1. Default Attach Descriptions of Pascal Files

| Pascal File Name | I/O Switch Name | Attachment |
|---|---|---|
| input | pascal_input_ | syn_ user_input |
| output | pascal_output_ | syn_ user_output |
| error | pascal_error_ | syn_ error_output |

These files are, by default, open at the beginning of the program *(reset (input), rewrite (output), rewrite (error))*.

All files other than the predefined files *input, output* and *error* correspond to I/O switches of the same name. The following three types of user-declared files are supported:

- permanent files
- static files
- local files

Permanent files are named in the parameter list of the program header and are declared in the *var* section of the main program. Permanent files have no default attach description and are not opened by default.

Static files are not named in the parameter list of the program header but are declared in the *var* section of the main procedure. These files have a default attach description that refers to a temporary segment allocated in the process directory at the first invocation of the program, and which is preserved until the end of the process (or "termination" of the program). Static files are not opened by default.

Local files are declared in the internal procedures of the program. They have a default attach description that consists of a temporary segment allocated when the procedure begins and freed when the procedure exits (return, release, nonlocal goto). Local files are not opened by default.

If you are using a file that does not have an attach description by default, or you do not want to use the default attachment, you must attach the file before opening it (see "Connecting Files from Command Level").

### Use of The Standard File: Error (Extension)

The standard file *error* is declared in the program header, as are the standard files *input* and *output*. It is attached by default to the user terminal for the output of error messages. For instance, your program can output results (*output* file) to a Multics segment and continue to output error messages (*error* file) to the terminal.

The *error* file is an extension to the Pascal standard. To use it, your program must be compiled with the "-full" (default) control argument to the pascal command.

### Use of the Standard Files: Input and Output

Most interactive application programs accept data from the terminal and send results to the terminal. The standard Pascal files *input* and *output* (declared in the program header) are connected to the terminal by default and are automatically opened at the beginning of the program. Novice Pascal users should read this section carefully before trying to execute interactive programs.

## INPUT AND OUTPUT AT THE TERMINAL

Terminal I/O is usually performed over the three predefined file variables *input*, *output*, and *error*. Other file variables can be attached to the terminal by means of the io_call command or the nonstandard *fconnect* statement (see Connecting Files Within a Pascal Program).

To perform input and output on the terminal, use the standard Pascal I/O statements. Examples of their use with the predefined file variables *input*, *output*, and *error* are given here:

`get (input);`
read a character into the file variable *input*^.

`read (var_name1 {, ..., var_nameN})`
read variable values from *input*.

`readln (var_name1 {, ..., var_nameN})`
read variable values from *input* and position to a new line.

`write (expression1 {, ..., expressionN})`
write expression values into a buffer to be printed on *output* with the next *writeln* to *output*.

`write (error, expression1 {, ..., expressionN})`
write expression values into a buffer to be printed on *error* with the next *writeln* to *error*.

`writeln (expression1 {, ..., expressionN})`
print on "output" any output buffered for *output* followed by the expression values followed by a newline character.

`writeln (error, expression1 {, ..., expressionN})`
print on "error" any output buffered for *error* followed by the expression values followed by a newline character.

The following added statement is a Multics extension:

`flush (file_name);`
prints the contents of the output buffer for *file_name* (for example, for *output* or *error*), which contains the results of any previous *write* operations, without printing a newline. This statement is useful for interactive applications, as in:

```
write ('Enter a number: ');
flush (output);
readln (number); (* reads on the same line as the question *)
```

The standard Pascal input procedures *read* and *readln* accept information by default from a Pascal text file called *input*. Pascal output procedures *write* and *writeln* output by default to a Pascal text file called *output*. If you require the use of these files, declare them in the program heading as shown below and do not declare them in the *var* section.

**Example**

```
program test(input, output);
          .
          .
          .
```

For Multics, *input* and *output* are connected to the switches user_input and user_output which are by default normally connected to the user terminal. For example, if the following program is compiled and run with no other action, an integer is read from and then written back to the terminal.

**Example**

```
program test (input, output);
   var
     i:integer
   begin
     read (i);
     write(i)
  end.
```

When a Pascal program requires terminal input, it outputs a prompt to the terminal to notify the user that input is required. The prompt is a question mark (?) by default, but it can be changed with the pascal_set_prompt command (see Appendix C).

Note that you must compile any programs using the standard file *input* (as well as any program performing input from the terminal via any other text file) with the -interactive control argument to the pascal command.

Before using Pascal files in an I/O statement, they must be initialized (see Declaring and Initializing Pascal Files); *input* and *output* are initialized automatically at the start of program execution and should not be explicitly initialized by the program.

Items in *write* procedure lists are not output to the terminal until you issue a subsequent *writeln* or the non-standard *flush* procedure or until the end of the program is reached.

## Example

The following example demonstrates the effects that *write* and *writeln* have on output:

```
program testwrite(input, output);
   var
      i:integer
   begin
      write ('Enter integer: ');
      read (i);
    writeln ('Integer =', i)
   end.
```

Compile the program with the -interactive control argument to the pascal command as follows:

```
pascal testwrite -interactive
```

Execute the program as follows:

```
testwrite
?6
Enter integer: Integer=     6
```

Note that the input prompt "Enter integer" appeared *after* the question mark. If the *write* statement is changed to *writeln* and you recompile the program, the following result is obtained:

```
testwrite
Enter integer:
?6
Integer=    6
```

The *write* statement can be used with the *flush* statement as follows:

```
program testwrite(input, output);
   var
      i:integer
   begin
      write ('Enter integer: ');
      flush;
      read (i);
      writeln ('Integer =', i)
   end.
```

Note that in the above instance, the program must be compiled with -full (the default) and -interactive.

## INTERACTIVE MODE

If you do not compile with the -interactive argument, when you open an input file (by reset) that is attached to the terminal, Pascal asks for the first line; it needs the first character to set the correct values for the file variables *eoln, eof and f^* (file buffer) which are supposed to be valid after this operation. The program prompts immediately at the terminal, asking for the first line, before any other execution takes place.

These automatic prompts are undesirable, however, in the case of a program that prompts explicitly:

```
program square_root(input, output);

  var number integer;

  begin         (* implicit reset(input), rewrite(output) *)
    write ('Give me an integer: ');
    flush (output);
    readln (number);
    writeln ('The square root is ', sqrt (number));
  end.
```

When compiled in the default manner, this program produces the following scenario:

```
square_root
?       (first character requested by reset)
Give me an integer: ?2
?       (first character of new line requested by readln)
The square root is: 1.41423562373095O5E+00
```

Interactive mode provides a way of suppressing the extra prompts. When a program is compiled with the -interactive control argument, prompts on a file are deferred until the first actual reference to the file (*get, read, readln,* or reference to the file window (file_name), *eoln, or eof*). Therefore when compiled with -interactive, the same program operates as follows:

```
square_root
Give me an integer: ?2
The square root is 1.41423562373095O5E+00
```

## STANDARD INPUT AND OUTPUT TO MULTICS FILES

To divert input from or output to the terminal to a Multics segment, use the io_call (io) command (see the Commands and Active Functions manual). For example, to allow a program to accept input from a segment "test_data", you must issue the following io command before you run the program:

```
io attach pascal_input_ vfile_ test_data
```

where vfile_ is the name of a standard Multics I/O_module that controls file storage. This establishes a connection between the Pascal file *input* and the segment "test_data".

When you no longer require the data in *test_data*, break the connection between *input* and the segment with the io command as follows:

```
io detach pascal_input_
```

When you run a program that takes data from the terminal, *input* is connected automatically to the terminal but is not disconnected at the end of the program run. You must explicitly disconnect it using the above io command before *input* can be connected to a segment.

To divert output from the terminal to a segment, you must follow a similiar procedure to that outlined above. For example, if you want the output from a program to be diverted to a segment called "data_output", issue the following command line before running the program:

```
io attach pascal_output_ vfile_ data_output
```

When finished, detach the segment with:

```
io detach pascal_output_
```

## INITIALIZING PASCAL FILES

Before any I/O procedures or functions can be used with a file, declared as either *text* or *file of ...*, you have to initialize it. Initialization consists of:

- Connecting or attaching the file to a physical resource (a Multics segment, the terminal, etc.)
- Opening the file in input mode via the Pascal *reset* statement, or in output mode via the *rewrite* statement

The default files *input* and *output* do not have to be initialized with *reset* or *rewrite* and are automatically initialized.

## CONNECTING FILES FROM COMMAND LEVEL

You can connect files to segments or devices before running a program by using the io_call command to attach an I/O switch. The switch must have the same name as the Pascal file and be specified in lowercase.

**Example**

The program below reads a set of integers from a Pascal file called *filein*, writes their sum to the default file *output*, and terminates on a negative number. Before the program can be run, you must attach the input file *filein* with the io command also shown below.

```
program readandsum(output, filein);
   var
      filein : text
      number, sum : integer;
   begin
      sum := 0;
      number := 0;
      reset (filein);
      repeat
         sum := sum + number;
         read (filein, number)
      until number < 0;
      writeln ('Sum = ', sum)
   end.
```

The following command line causes this program to take data from a segment called "indata":

```
io attach filein vfile_ indata
```

Before the program can be run again, the switches *filein* and *output* must be detached.

## CONNECTING FILES FROM WITHIN A PASCAL PROGRAM (EXTENSION)

Pascal files can also be connected to Multics segments by using the nonstandard procedure *fconnect* which is a Multics extension to standard Pascal. Programs using fconnect must be compiled with "-full" (the default).

The syntax of the *fconnect* procedure call is:

```
<fconnect_procedure_statement> =
   fconnect (<file_variable_access>, <attach_string>)
```

where:

file_variable_access
: is a reference to the file to be attached.

attach_string
: is a character string containing a Multics attach description string. This string specifies an I/O_module and any arguments required to define a Multics segment or device. This string can contain references to active functions.

**Examples**

```
fconnect (ttyin, 'syn_ user_input');
fconnect (output, 'vfile_ output_file');
fconnect (filef, 'vfile_ [pd]>foo.output');
```

The *fconnect* statement does the following:

- Closes the file if it was open.
- Detaches the file if it was attached.
- Attaches the file using the given attach description.

## OPENING A PASCAL FILE

Once a file is attached, you must open it using the *reset* statement (for input) or the *rewrite* statement (for output). If a file is not attached when either of these statements is executed, the default attach description (if any; for example, syn_ user_input for *input*) is used to attach the file. If there is no default attach description, an error occurs. If the file is already opened, it is closed and re-opened with the same attachment.

## CLOSING A PASCAL FILE (EXTENSION)

Pascal files are closed by default when the procedure where they are declared becomes inactive (normal *end*, nonlocal *goto* into an outer procedure, stack release). The nonstandard predefined procedure *fclose* lets you close a file before the end of the program. The fclose statement is a Multics extension to standard Pascal and must be compiled with -full.

The syntax of the *fclose* procedure call is:

```
<fclose_procedure_statement> =
   fclose (<file_variable_access>)
```

where file_variable_access is a reference to the file to be closed.

**Example**

```
fclose (data_output)
```

## RANDOM ACCESS I/O: FUPDATE, FPUT, FGET

Multics Pascal lets you access random or direct access files with the following nonstandard procedures:

- *fupdate* - opens a switch for direct update
- *fput* - transfers an item into the file buffer (analogous to the standard procedure *put*)
- *fget* - transfers an item from the file (analogous to the standard procedure *get*)

The syntax for the *fupdate* procedure call is:

```
<fupdate_procedure_statement> =
   fupdate (<file_variable_access>)
```

where file_variable_access is a reference to the file to be opened in direct update mode.

**Example**

```
fupdate (student_file)
```

The syntax for the *fput* procedure call is:

```
<fput_procedure_statement> =
   fput (<file_variable_access>, <integer_expression>)
```

where:

file_variable_access
: is a reference to a file previously opened in direct update mode.

integer_expression
: specifies the number of records to be written.

**Example**

```
fput (student_file, student_nbr)
```

This statement causes the item in the file buffer to be output. Items are stored in records and each record has a key that is the character string representation of the integer record number.

The syntax of the *fget* procedure call is:

```
<fget_procedure_statement> =
   fget (<file_variable_access>, <integer_expression>)
```

where:

file_variable_access
: is a reference to a file previously opened in direct update mode.

integer_expression
: specifies the number of records to be read.

**Example**

```
fget (student_file, student_nbr)
```

The above *fget* statement inputs an item to the file buffer.

If you use these procedures, the files must be connected with the Pascal *fconnect* procedure or io command as described earlier.

**Example**

The program below writes 100 records to a direct access file. Each record contains an integer whose key value is the square of the key. The program reads the record whose key is 50 and prints out its value.

```
program randomaccess(output, rfile);
   var
      rfile : file of integer;
      i : integer;
   begin
      fupdate(rfile);
      for i := 1 to 100 do
         begin
            rfile^ := i*i;
            fput (rfile)
         end;
      fget(rfile, 50);
      writeln(rfile^)
   end.
```

## COMMUNICATION WITH PL/1 PROGRAMS

Specifications given here for PL/1 can be extended to other languages such as FORTRAN, wherever these languages are compatible with PL/1.

## Calling a Pascal Main Program

The following example illustrates how to call a Pascal main program.

### Example (PL/1)

```
dcl mainpascal entry options (variable);
call mainpascal;
```

to call a Pascal program declared as:

```
program mainpascal ( ... );
```

A parameter list can be transmitted, but this parameter list must have descriptors; arguments can only be character strings. The last argument can be a fixed bin(35) return code (refer to the *argc, argv* and *stop* extensions).

### Example

At command level:

```
pascal_program arg1 arg2 name other string
```

In a PL/1 program:

```
dcl pascal_program entry options(variable);
dcl code fixed bin (35);
dcl name char (32);
dcl string char (168);

call pascal_program (arg1, arg2, name, string);
```

or:

```
call pascal_program (arg1, arg2, code);
if code ^= 0 then
.....
```

## Calling a Pascal Exported Procedure or Function

The following examples illustrate how to call a Pascal exported procedure or function.

**Example 1 (PL/1)**

```
dcl proc1$entry1 entry (fixed bin(35), float bin(63));
```

referring to:

```
program proc1;
...
$export entry1 $
procedure entry1 (var i : integer, a : real);
```

**Example 2**

```
dcl func2$entry2 entry (char(3)) returns (ptr);
```

for:

```
program func2 (...);
...
$export entry2, ... $
function entry2 (cs : packed array [1..3] of char) : ptrtype;
```

**Access to a Pascal Exported Variable**

The following example illustrates how to access Pascal exported variables.

**Example**

```
dcl pascal_program$num fixed bin(35) external static;
dcl pascal_program$string_ptr ext;
dcl string char(32) based(pascal_program$string_ptr);
```

referring to:

```
program pascal_program;
...
$export num, string_ptr$
var
   num : integer;
   string_ptr: ^packed array[1..32] of char;
   string: packed array[1..32] of char;
```

If the Pascal program above is compiled with the -private_storage or -ps control argument, the variable is allocated in a segment named pascal_program instead of the user free area. This method is used because the Pascal exported variable is not necessarily the same as the PL/1 external variable with the same name.

**Access to PL/1 from Pascal**

The following examples illustrate how to call PL/1 programs from Pascal.

**Example 1 (Pascal)**

```
$import 'pllproc (pl1)' : pllproc $
procedure pllproc (var a, b : integer); external;
```

to call the PL/1 program:

```
pllproc : procedure (a, b);
   dcl (a, b) fixed bin(35);
```

**Example 2 (Pascal)**

```
$import 'pllfonc (pl1)' : pllfunc $
function pllfunc : real; pl1;
```

to call the PL/1 program:

```
procedure pllfunc returns (float bin(63));
```

## CALLING THE MULTICS COMMAND PROCESSOR FROM PASCAL

The following example illustrates how to call the Multics command processor from a Pascal program.

```
program multics_comm(input, error);

 $import
    'cu_$cp (pl1)' : comm_processor $

 const
     max_len = 100;
  type
    line = packed array [1..max_len] of char;
    comm_ptr = ^line;
    line_len = 0..max_len;
 var
    comm_addr : comm_ptr;
    comm_len : line_len;
    error_code : integer;
    procedure comm_processor
    (p1 : comm_ptr; p2 : line_len; var p3 : integer); external;
 begin
   new (comm_addr);
    comm_len := 0;
    while (not eoln) and (comm_len < max_len) do
       begin
          comm_len := comm_len + 1;
          read (comm_addr^ [comm_len]);
       end;
```

```
    if not eoln then
       begin
          writeln  (error, 'command line too long (100 chars max).');
          while not eoln do get (input);
       end
    else
       comm_processor (comm_addr, comm_len, error_code);
    dispose (comm_addr);
end.
```

Compile the above program as follows:

```
pascal multics_comm
```

Execute the program as follows:

```
multics_comm

?ls [wd]>multics_comm.**

Segments = 2, Lengths = 2.

re      1  multics_comm
rw      1  multics_comm.pascal

r 15:28 0.671 76
```

## PARAMETER LISTS

In general, Pascal procedures do not accept parameter lists including descriptors and do not generate descriptors in procedure calls. For this reason, it is not possible to call most of the PL/1 procedures declared with (*) descriptors or declared options (variable). However, there is one exception: Pascal conformant arrays of type integer or real can be passed to PL/1 procedures whose parameters are (*) arrays of equivalent types.

Table 4-2. Variable Equivalence

| Pascal | Fortran | PL/1 |
|---|---|---|
| integer | integer | fixed bin(35) |
| real | double precision | float bin(63) |
| packed array[1..N] of char | character*N | char(N) |
| boolean true | integer = 1 | fixed bin(35)=1 |
| boolean false | integer = 0 | fixed bin(35)=0 |
| integer -1 | logical true | |
| integer 0 | logical false | |
| pointer(nil) | Not available | pointer(null) |
| char | Not available | pascal_char<br>2 dum bit(27) unal,<br>2 ch char(1) unal, |
| packed array<br>[a..b : integer] of char | Not available | (*) char (1) unal |
| array [a..b : integer;<br>c..d : integer] of real | | (*,*) float bin (63) |
| packed array<br>[a..b : integer] of<br>integer | Not available | (*) fixed bin (35) unal |
| record<br>length : 0...n;<br>string : packed array<br>[1...n] of char<br>end; | Not available | char (n) varying; |

When calling a PL/1 program, a Pascal program should pass only the string array portion (record.string), not the entire record.

## INITIALIZATION OF VARIABLES

In standard Pascal, uninitialized variables have an undefined value. Use of an uninitialized variable results in an error. Multics Pascal does not flag variables to show that they are unintitialized. Therefore, variables must be initialized explicitly.

For instance, use of an uninitialized local pointer can cause a fatal process error. The pointer, initialized in the Multics stack, has usually been assigned a value by another procedure. To avoid this problem, compile your programs with the -debug control argument (the default). Debug mode initializes locals, internal globals and allocated blocks to blanks (octal \040) (all bytes will have this value). External (exported) globals are set to an initial value of zero (octal \000) by default (allocation by Multics dynamic linker), but they will have an initial value of \040 if you compile the program with the -ps arg (allocation in private data segment). Therefore, an attempt to use an uninitialized pointer causes a nonfatal error. (Standard Multics error message: ascii data where pointer expected).

If you compile your program with the -no_debug control argument which slightly shortens the program's execution time, locals will have no special initial value (current stack value), globals and allocated blocks will have an initial value of zero.

## PASCAL AREA MANAGEMENT

A Pascal area consists of one or more temporary segments where blocks are allocated and deallocated by the Pascal *new, dispose,* and *reset* procedures. The default size for a Pascal area is one segment (255 records). You can increase, decrease, or reset the size using the pascal_reset_area command (see Appendix C.)

The Pascal procedure *new* (P) sets the pointer P to null if there is no more room in the area for the requested allocation.

The Pascal procedure *dispose* (P) sets the pointer P to a null value.

## PROGRAM HEADER

In Multics Pascal, as in most Pascal implementations, the program header contains, and only contains, the names of files used by the program (see Declaring Pascal Files above).

**Example**

```
program foo(input, file1, file2);

...

var file1, file2 : file of real;
```

## DEBUGGING A PASCAL PROGRAM USING PROBE

The Multics probe command provides symbolic, interactive debugging facilities for programs written in Pascal and other Multics supported programming languages. Its features let you interrupt a running program at a particular statement, examine and modify program variables in their initial state or during execution, examine the stack of block invocations, and list portions of the source program. You can find a full description of the probe interactive debugging facility in the *Commands and Active Functions* manual. This subsection is not meant to teach you the use of probe; it describes the information required to use probe in a Pascal environment.

Invoked on a Pascal program, probe understands all of the Pascal data types, including enumerated types, typed pointers, sets, records, and user-defined types. Probe understands the Pascal builtin functions chr, eof, eoln, false, nil, ord, and true.

Array indices are enclosed in brackets, for example a[i,j]. Cross-section ranges are written with .., as in b[first..last]. Probe uses the asterisk (*) to refer to a complete cross-section row as in a[*,3] or b[*]. References to record fields must specify all levels; implicit level names are not allowed. For example, a.b.c.d cannot be abbreviated as a.d as can sometimes be done with PL/1 structure elements.

Pointer values are written with circumflex (^) as the up-arrow, for example p^ to indicate the value that p points to. String constants are enclosed in single quotes: 'This is a string'. The two boolean values are true and false.

# APPENDIX A

# MULTICS DEVIATIONS FROM STANDARD PASCAL

Although Multics Pascal does adhere to the standard ISO Pascal, there are several areas where Multics deviates from the standard. The following is a list of Multics deviations from standard Pascal. The parenthetical reference numbers refer to the ISO Pascal standard.

- The concept of an undefined variable is not implemented (i.e., the compiler does not "flag" undefined variables); their use is not detected as an error. The following undefined variables are not implemented:
  - uninitialized variables (6.2.3.5, 6.6.5.4.)
  - the control variable of a *for* statement after the *for* statement (6.8.3.9)
  - the field of an inactive variant or an uninitialized field (6.4.3.3)
  - the buffer variable after a *put* statement
  - an undefined pointer after a call to *dispose* (has a null value)(6.6.5.3)
- In a *dispose*, it is not an error if the pointer points to a variable that is the actual variable of an active procedure or used with an active *with* statement (6.6.5.3).
- In a *dispose* of the long form, it is generally not an error if the parameters passed do not have the same value or are not of the same number as in the corresponding *new* statement (only on the length is checked).
- It is not an error to use in an expression, assign in a statement, or pass as an actual parameter, a variable created by a *new* statement of the long form (6.6.5.3).
- It is not an error to modify a selector of a variant when it has been defined in a *new* statement of the long form (6.6.5.3).

- Each occurrence of an identifier is associated with the last declaration in the current (or including) block until it is modified by another declaration in the current block (6.2.2, 6.3, 6.4.1, 6.6.1).

- A component of an array cannot be a *file* (6.4.3.2).

- A record cannot have a component of file type (6.4.3.3).

- Using a component of a record that has not been initialized is not detected as an error. Using a component of a record when the case selector has an incorrect value is not detected as an error (6.4.3.3).

- The maximum size of a set is 288 elements. For that reason, it is impossible to define set_type = *set of* ordinal_type if the ordinal type has more than 288 elements (6.4.3.4).

- The compiler checks that there is at least one occurrence of the assignment of the function in a function procedure, but it does not check at execution time whether you return from the function without giving it a value (6.6.2).

- *pack* and *unpack* are not allowed on conformant arrays (6.6.3.7).

- No error is detected if the current file position of a file *f* is altered while the file's buffer variable *f^* is an actual variable parameter, or an element of the record variable list of a *with* statement, or both (6.5.5).

- (* .... *) is a comment, as well as { .... }. But (* and {, *) and } are not synonyms. A comment that begins with (* must end with *) and a comment that begins with { must end with } (6.1.9).

- Set overlapping is not always detected (6.4.5, 6.4.6, 6.7.2.4).

- Modification of the selector field of a record when this record has been allocated by a *new* statement of the long form (using the value of this field) is not detected as an error.

- No check is performed when a label is used (it must only have been declared in the current block or in a containing block). For instance, *goto* to a branch of an *if* statement or a *case* statement from outside this statement or from another branch of this statement is not detected (6.8.1).

- A selector field can be passed to a procedure (6.6.3.3).

- An error is not detected if the control variable of a *for* statement is modified in a procedure contained in the block (6.8.3.9).

- An array of packed type can be passed as an actual parameter to a variable conformant array. (6.6.3.7.3).

- Two string constants of the same length can be passed as actual parameters to variable conformant arrays of same schema (6.6.3.8).

## IMPLEMENTATION RESTRICTIONS OF MULTICS PASCAL VARIABLES AND IDENTIFIERS

The following table lists the range boundaries that apply to Multics Pascal variables and identifiers:

Table A-1. Range Boundaries of Variables and Procedures

| |
|---|
| maximum positive integer (maxint) = 34359738367 |
| maximum positive real (maxreal) = 1.70141183460469232e+38 |
| minimum positive real (minreal) = 1.46936793852785938e-39 |
| maximum set range = 288<br>(For "set of x..y", x must be >=0 and y must be < 288.<br>For a set of enumerated type, the enumerated type cannot have more than 288 elements.) |
| identifiers can have up to 32 chars |
| global internal variables (declared at main level and not imported or exported) cannot occupy more than 16k words. |
| local variables (internal to procedures) cannot occupy more than 16k words. |

# APPENDIX B

# FRENCH TRANSLATION OF SYMBOLS

French translation of predeclared or reserved symbols is as follows:

| English | French | English | French |
|---|---|---|---|
| $export | $exporte | get | prendre |
| $import | $importe | goto | allera |
| $include | $include | if | si |
| $options | $options | in | dans |
| $value | $valeur | input | entree |
| abs | abs | integer | entier |
| and | et | label | etiquette |
| arctan | arctan | maxint | entmax |
| argc | nbarg | maxreal | reelmax |
| argv | arg | minreal | precision |
| array | tableau | mod | mod |
| begin | debut | new | creer |
| boolean | booleen | nil | nil |
| case | cas | not | non |
| char | car | odd | impair |
| chr | carac | of | de |
| const | const | or | ou |
| cos | cos | ord | ord |
| dispose | liberer | otherwise | autrement |
| div | div | output | sortie |
| do | faire | pack | tasser |
| downto | bas | packed | paquet |
| else | sinon | page | page |
| end | fin | pred | pred |
| eof | fdf | procedure | procedure |
| eoln | fdln | program | programme |
| error | erreur | put | mettre |
| exp | exp | read | lire |
| external | externe | readln | lireln |
| false | faux | real | reel |
| fappend | allonger | record | article |
| fclose | fermer | repeat | repeter |
| fconnect | connecter | reset | relire |
| fget | fprendre | rewrite | recrire |
| file | fichier | round | arrondi |

| English | French | English | French |
|---|---|---|---|
| flush | vider | set | ensemble |
| for | pour | setmax | ensmax |
| forward | plusloin | sin | sin |
| fput | fmettre | sqr | carre |
| function | fonction | sqrt | rac2 |
| fupdate | fupdate | sread | lirech |
| stop | stop | type | type |
| succ | succ | unpack | detasser |
| swrite | ecrirech | until | jusque |
| text | texte | var | var |
| then | alors | while | tantque |
| to | haut | with | avec |
| true | vrai | write | ecrire |
| trunc | tronc | writeln | ecrireln |

# APPENDIX C

# PASCAL COMMANDS

This appendix contains all of the available Pascal commands, namely:

- pascal
- pascal_area_status
- pascal_create_area
- pascal_delete_area
- pascal_file_status
- pascal_indent
- pascal_reset_area
- pascal_set_prompt

The online help facility also provides full documentation of the commands. To use it, simply type "help" followed by the command name.

---

**Name: pascal**

*SYNTAX AS A COMMAND*

```
pascal path {-control_args}
```

*FUNCTION*

invokes the Pascal compiler, which compiles a source program written in Pascal and produces a Multics executable object segment. If compilation errors are encountered, error messages are printed on user_output.

*ARGUMENTS*

path
is the pathname of the source segment. The ".pascal" suffix is assumed.

*CONTROL ARGUMENTS*

-add_exportable_names, -aen
adds names of exported variables and procedures to the object segment.

-brief_map, -bfm
produces a compilation listing containing source, error messages, and a statement map.

-brief_table, -bftb
generates a partial symbol table consisting of only a statement table that gives the correspondence between source line numbers and object locations for use by symbolic debuggers. The table appears in the symbol section of the object segment. This control argument does not significantly increase the size of the object segment.

-conditional_execution VAR_NAME true/false, -cond VAR_NAME true/false
forces the value of the conditional compilation variable VAR_NAME to either "true" or "false". This control argument overrides any assignments of VAR_NAME in the text of the program. See Section 3 for a description of conditional compilation.

-debug, -db
generates code to check for references outside of array bounds, invalid assignments, values that are out of range, and a variety of other potential errors. Also initializes program storage to blanks (\040) so that a reference through an uninitialized pointer will cause a fault_tag_1 condition. (Default)

-english
assumes that Pascal reserved words are in English as opposed to French. (Default)

-error_messages, -em
prints error messages on user_output as well as including them in the listing segment. (Default)

-french
accepts Pascal reserved words in French. Type "help pascal_french_keywords" for the correspondence between French and English reserved words.

-full_extensions, -full
allows use of all nonstandard extensions defined for Multics Pascal. (Default)

-interactive, -int
allows text files to operate in interactive mode. On *reset* or *readln*, *get* of next character is deferred until the next reference to the file or to one of the variables attached to the file, such as *eof*, *eoln* and *file^*. See Section 2 for a description of interactive mode.

-io_warnings, -iow
allows warnings to be printed by I/O procedures called by the compiled program. (Default)

-list
produces a compilation listing including source, error messages, map and cross-reference of symbols, statement map, and generated code in symbolic ALM.

-long_profile, -lpf
generates additional code that records the virtual CPU time and number of page faults for each source statement. It is incompatible with the -profile control argument. The profile command can handle both regular and long profiles. Use of this feature adds considerable CPU overhead to heavily executed code. The extra CPU time is subtracted out so that it does not appear in the report printed by the profile command.

-map
produces a compilation listing including source, error messages, map and cross-reference of symbols, and statement map.

-no_debug
does not generate code to test for references outside of array bounds, values out of range, or other errors, nor does it initialize storage to blanks.

-no_error_messages, -nem
does not print error messages on user_output. They are still included in the listing segment.

-no_interactive, -nint
does not allow text files to operate in interactive mode. (Default)

-no_io_warnings, -niow
does not print I/O warnings if a nonfatal error occurs in I/O procedures called by this program.

-no_list
does not produce a compilation listing. (Default)

-no_long_profile, -nlpf
does not generate additional code to record the virtual CPU time and number of page faults for each source segment. (Default)

-no_private_storage, -nps
causes exported variables to be allocated dynamically in external static. (Default)

-no_profile, -npf
does not generate code to meter the execution of source statements. (Default)

-no_relocatable, -nonrelocatable, -nrlc
generates an object segment that cannot be bound, and saves 10%-20% compilation time.

-no_table, -ntb
does not generate a symbol table in the object segment.

-page_length N, -pl N
specifies a page length for the listing segment. The default is 59 lines.

pascal

-private_storage, -ps
allocates all exported variables in a segment in the process directory named progname.defs, where progname is the entryname of the path argument, without the .pascal suffix. This segment is created if it does not exist.

-profile, -pf
generates additional code to meter the execution of individual statements. Each statement in the object program contains an additional instruction to increment an internal counter associated with that statement. After a program has been executed, the profile command can be used to print the execution counts.

-relocatable, -rlc
generates an object segment that can be bound. (Default)

-sol_extensions, -sol
allows only French SOL extensions to be used. Type "help pascal_extensions" for a list of SOL extensions.

-standard
allows only (ISO) standard Pascal to be used. The default is -full_extensions.

-table, -tb
generates a full symbol table for use by symbolic debuggers. The symbol table is part of the symbol section of the object segment and consists of two parts: a statement table that gives the correspondence between source line numbers and object locations, and an identifier table containing information about every identifier actually referenced by the source program. This control argument usually causes the object segment to be significantly longer. (Default)

*NOTES*

If incompatible control arguments are specified, the rightmost one is used.

Multics Pascal is case-insensitive. All identifier names are mapped to lowercase in the program and in the program's symbol table. As a result, the Pascal program header:

```
program: Foo;
```

produces a segment entry point with the name "foo".

*NOTES ON LISTING*

The Pascal compilation listing contains the following sections in the order shown:

1. Header: gives the full pathname of the source segment, the Multics site indentification, date and time of compilation, and the compiler indentification.

2. Source: with lines numbered sequentially. In include files, file number precedes the line number.

3. Error messages (if any).

4. Storage requirements for the object segment.

5. List of source files used.

6. Complete map and cross-reference for symbols declared and used, symbols declared and never used, and symbols declared by default.

7. Displacement for fields given in octal (bytes), locations for variables given in octal (words), and sizes given in octal (bytes).

8. "DEF:" followed by the number of the line where the symbol is defined. "REF:" followed by the number of the line(s) where the symbol is referenced. A star (*) is printed for each reference where the variable or field is set or passed by reference ("var" parameter) to a subroutine.

9. Statement map: gives the octal location of the first instruction of each statement of the source program.

---

**Name: pascal_area_status**

*SYNTAX AS A COMMAND*

```
pascal_area_status {names} {-control_args}
```

*FUNCTION*

Displays and sets attributes of specified Pascal areas. These areas are temporary segments. Allocation is performed by the Pascal *new* statement, deallocation by the the *dispose* and *reset* statements.

*ARGUMENTS*

names
are relative pathnames of Pascal object segments that have their own private areas. (See the pascal_create_area command.)

*CONTROL ARGUMENTS*

-all, -a
operates on all private Pascal areas as well as on the default Pascal area.

-brief, -bf
does not print a dump of each allocated block. (Default)

pascal_area_status

-default
specifies the default area used by Pascal to allocate storage.

-dump
prints a comprehensive, unformatted dump of the area(s). This control argument is intended for use by the maintainers of the Pascal compiler and related software.

-long, -lg
prints a dump of each allocated block.

-no_dump
does not print a comprehensive dump as printed by -dump. (Default)

-no_status, -nst
does not print status information.

-no_trace
does not print the address and length of each block. (Default)

-status, -st
prints the maximum size, the size of the allocated blocks, and the maximum number of blocks.

-trace
prints the address and length of each block.

*NOTES*

Names and control arguments can be present in any order.

If no areas are specified, -default is assumed. If no actions are specified, -status is assumed.

If more than one of -list, -dump or -long_dump is specified, only the last one is performed. In addition, if more than one action is specified, the operations are performed in the following order:

```
-status -trace -long -dump
```

pascal_create_area

**Name: pascal_create_area**

*SYNTAX AS A COMMAND*

```
pascal_create_area names {-control_args}
```

*FUNCTION*

creates temporary, private areas in the process directory for the specified Pascal object segments. All *new* operations executed by these object segments will use the associated private areas.

*ARGUMENTS*

names
: are the relative pathnames of Pascal object segments which are to have their own private areas. An error occurs for each object segment for which a private area has already been created.

*CONTROL ARGUMENTS*

-brief, -bf
: suppresses the error message that is printed when the private area for a specified program already exists.

-long, -lg
: allows the error message that is printed when the private area for a specified program already exists. (Default)

-size N
: sets the maximum size of each area to N pages. The default size is 225 records.

*NOTES*

By default, the Pascal *new* operation uses the default Pascal area in the process directory. This area, and any that are created, can be examined using the pascal_area_status command.

**Name: pascal_delete_area**

*SYNTAX AS A COMMAND*

```
pascal_delete_area names {-control_args}
```

*FUNCTION*

deletes the private areas associated with the specified Pascal object segments.

*ARGUMENTS*

names
are the relative pathnames of Pascal object segments whose private areas are to be deleted.

*CONTROL ARGUMENTS*

-brief, -bf
suppresses the message that is printed when a specified program is active on the Multics stack.

-long, -lg
allows the message that is printed when a specified program is active on the stack. (Default)

---

**Name: pascal_file_status**

*SYNTAX AS A COMMAND*

```
pascal_file_status
```

*FUNCTION*

displays information on the status of all standard Pascal files currently in use and all files of active Pascal procedures in the Multics stack.

**Name: pascal_indent**

*SYNTAX AS A COMMAND*

```
pascal_indent old_path {new_path} {-control_args}
```

*FUNCTION*

indents a Pascal source program according to a standard set of conventions described below.

*ARGUMENTS*

old_path
: is the pathname of the source segment to be indented. The .pascal suffix is assumed.

new_path
: is the optional pathname of the indented result. The .pascal suffix is assumed. If this argument is omitted, the indented copy replaces the original segment. However, if errors are detected in the source, a temporary indented copy is created instead and its pathname is printed in an error message.

*CONTROL ARGUMENTS*

-brief, -bf
: suppresses warning messages for invalid or non-Pascal characters found outside a string or comment. Errors corresponding to suppressed messages do not prevent the original source segment from being replaced.

-comment N, -cmt N
: indents comments at column number N. Comments are lined up at this column unless they occur at the beginning of a line and are preceded by a blank line. The default column for comments is 61.

-french
: assumes that the source program is written in French. See Appendix B for a list of French keywords.

-highlight, -hl
: translates reserved symbols of the Pascal language to lowercase if -up is specified, uppercase otherwise so that they stand out from the rest of the text.

-indent N, -in N
: indents each level an additional N spaces. The default number of spaces is 5.

-lmargin N, -lm N
: sets the left margin for top-level program statements after the Nth column. The default for N is 10.

-long, -lg
    allows warning messages for invalid or non-Pascal characters.

-lower_case, -lc
    translates all uppercase letters outside of strings and comments to lowercase.

-no_highlight, -nhl
    does not translate Pascal reserved symbols to the opposite case (upper or lower) from the rest of the text.

-upper_case, -uc
    translates all lowercase letters outside of strings and comments to uppercase.

*NOTES ON INDENTING STYLE*

Multiple spaces are replaced by single spaces, except inside strings and for non-leading spaces and tabs in comments. Trailing spaces and tabs are removed from all lines before indenting. Spaces are inserted before left parentheses, brackets and braces and removed after them. Spaces are inserted after right parentheses, brackets and braces and removed before them. Spaces are inserted around the constructs =, ^=, <>, <=, >=, :=, ;, :, and operators in expressions.

Parentheses, brackets and braces must balance. Also, *begin*, *case*, and *repeat* keywords must balance with their corresponding *end* statements. The same is true of *repeat* and *until* constructs.

---

**Name: pascal_reset_area**

*SYNTAX AS A COMMAND*

```
pascal_reset_area {names} {-control_args}
```

*FUNCTION*

frees all blocks in the specified areas.

*ARGUMENTS*

names
    are relative pathnames of Pascal object segments that have their own private areas. (See the pascal_create_area command.) If no names are specified, the default Pascal area is reset.

*CONTROL ARGUMENTS*

-size N
    sets the maximum size of each specified area to N records after resetting the area. The default size is 255 records.

pascal_set_prompt

**Name: pascal_set_prompt**

*SYNTAX AS A COMMAND*

```
pascal_set_prompt {string} {-control_args}
```

*FUNCTION*

sets the prompt string used by Pascal programs in interactive mode. Type "help pascal_terminal_io" for a description of interactive mode.

*ARGUMENTS*

string
    specifies the prompt string.

*CONTROL ARGUMENTS*

-no_prompt, -npmt
    causes there to be nothing printed for a prompt.

*NOTES*

If no arguments are specified, the default prompt "?" is restored.

# INDEX

W

CUT ALONG LINE

**HONEYWELL INFORMATION SYSTEMS**
**Technical Publications Remarks Form**

**TITLE** MULTICS PASCAL USER'S GUIDE

**ORDER NO.** GB62-00

**DATED** MARCH 1984

**ERRORS IN PUBLICATION**

**SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION**

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

**FROM: NAME** _____ **DATE** _____

**TITLE** _____

**COMPANY** _____

**ADDRESS** _____

_____

CUT ALONG LINE

FOLD ALONG LINE

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

**HONEYWELL INFORMATION SYSTEMS**
**200 SMITH STREET**
**WALTHAM, MA 02154**

**ATTN: PUBLICATIONS, MS486**

FOLD ALONG LINE

**Honeywell**

Together, we can find the answers.

# Honeywell

**Honeywell Information Systems**
**U.S.A.**: 200 Smith St., MS 486, Waltham, MA 02154
**Canada**: 155 Gordon Baker Rd., Willowdale, ON M2H 3N7
**U.K.**: Great West Rd., Brentford, Middlesex TW8 9DH **Italy**: 32 Via Pirelli, 20124 Milano
**Mexico**: Avenida Nuevo Leon 250, Mexico 11, D.F. **Japan**: 2-2 Kanda Jimbo-cho, Chiyoda-ku, Tokyo
**Australia**: 124 Walker St., North Sydney, N.S.W. 2060 **S.E. Asia**: Mandarin Plaza, Tsimshatsui East, H.K.

40215, 584, Printed in U.S.A.

GB62-00