MUNIX PASCAL

Version 3.5

Documentation-No.: G0930.051-0188

Best.-Nr.: G0930.051-0188

Trademarks:

MUNIX, CADMUS	for PCS
DEC, PDP	for DEC
UNIX	for AT&T

MUNIX - PASCAL

Version 3.5
Release Notes

Documentation-No.: G0930.051-0188

Best.-Nr.: G0930.051-0188
DF: rel.h relV.2-32 relV.2
Author's initials: DK, LY, RG

Trademarks:

MUNIX, CADMUS	for PCS
DEC, PDP	for DEC
UNIX	for AT&T

Table of Contents

* * * * *

This document was printed by PCS using the laser beam printer LBP 68000

1. PASCAL for MUNIX V.2/32

1.1. Update from Version 3.4-32 to Version 3.5-32

The address space restriction of 32 KBytes for automatic variables in main programs or subroutines has been abolished.

Warning

- The operating system allows a maximum address space of 3 MBytes for automatic variables in one routine.

Fixed bug

- The sizes of predefined data type variables are always the same as the equivalent user defined data type variables.

The new pc command option -Y *count* facilitates the increase of the maximum value of symbol table entries handled by the code generator to *count*.

Further information about changes and releases may be found in the following sections. (For fixed bugs, see "PASCAL for MUNIX V.2)

1.2. Update from Version 3.3-32 to Version 3.4-32

- The Version 3.4-32 uses the new code generator nc1 from the C compiler
- The MUNIX Pascal Compiler has been adapted to the symbolic debugger *sdb*(1). Compiling sources with -g option symbol table and line number information is included in the produced object file which facilitates debugging MUNIX Pascal programs at source language level. Please note that it is impossible to debug symbolically optimized programs.

For the new version the documentation has been updated. Particularly a chapter for using SDB with MUNIX Pascal-32 has been added (see User's Guide).

Further information about changes and releases may be found in the following sections.

1.3. Adaptation of MUNIX Pascal to the 32-bit System

The MUNIX Pascal compiler version 3.3-32 has been adapted to the 99xx/32 32-bit system, based on the MC68020 microprocessor. This package has the version number 3.3-32.

The Pascal part of the compiler has undergone no changes, it was just recompiled. Version 3.3-32 uses the phases c1, c2 from the C compiler producing COFF format.

Some options in the *pc* command have been changed. The version 3.3-32 uses the floating point arithmetic of the M68881 coprocessor and only the four byte libraries (libp.a, libpdisp.a, libpmark.a, libpm.a).

2. PASCAL for MUNIX V.2

2.1. Update from Version 3.4 to Version 3.5

Fixed bugs

- Set operations with set reference parameter function correctly.
- When using constants in programs, the constants had been destroyed in some cases. This works correctly now.
- Calls of the standard procedures pack and unpack don't cause an internal compiler error.
- Value parameter passing of character arrays with a length of 1, 2 or 3 function correctly.
- In the case of repeated reset or rewrite operations on the same file, the file is closed each time before opening again .
- Actual parameters of formal string parameters are checked now.
- Conformant arrays :
  Lists of conformant array parameters don't cause runtime errors.
  Type character for bound identifiers is possible.

Known bug

- EOF in comments crashes the compiler.

2.2. Update from Version 3.3 to Version 3.4

Debugging programs with *adb*(1) parameters and variables may be referenced symbolically.

Fixed bugs

- When procedure calls have conformant array parameters as the last parameter, further parameters are not expected.
- Wrong recursive record type definition is detected during analysis of type declaration.
- For-statements with final value expressions of the following kind don't cause an error :

```
...
var controlvar : (e1,e2);
...
for controlvar := e1 to pred(e2) do
    ...
```

Warning

- After dispose(ptr) ptr becomes undefined. The programmer should make sure that no accesses to undefined pointers occur.

2.3. Update from Version 3.2 to Version 3.3

Lazy-Input is now implemented.
Interactive programs may be easier to implement using this concept. The following program does what the user expects:

```
program p (input, output);
var i: integer;
 begin
  write('enter number');
  read(i);
 end.
```

Fixed bugs

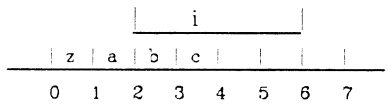
- Assignment of arrays, sets and parameters are now properly handled
- succ, trunc, put, LN and ARCTAN don't cause ld error
- output of *short*-variables
- wrong formal parameters will be flaged as *syntax error* and don't crashes the compiler

Warning

- The C-Code generator handles a limited set of symbols and will stop with the message *symbol table overflow* when this limit is reached.
- Files should be declared and exported in the main program (i.e. **program** head and **export** declaration) if they will be used across different modules. Inside a module a file may be declared only locally to a procedure.
- Word boundaries may sometimes produce unexpected results. Specifically the customer should take care about variants in records as in the following example:

```
record
   case z: boolean of
   true:  (i: integer);
   false: (a,b,c: char);
end;
```

The figure beneath shows how this variable is stored:



2.4. Update from Version 3.1 to Version 3.2

The Version 3.2 now supports four types of floating point processing:

(1) Motorola Fast Floating Point Package (FPP)

(2) Motorola IEEE Floating Point Package (MOT341)

(3) The floating point board with the National Semiconducter NSC floating point processor (NSC)

(4) The floating point coprocessor M68881 for the M68020 CPU (MOT881).

Two options have been added for the floating point coprocessor M68881:

-fH and **-fT**.

2.5. Update from Version 2.6 to Version 3.1

- The compiler structure has been heavily modified. Specifically, the code generator of the C-Compiler is used. The old pass1 is replaced by p0 and the old pass2 is replaced by c1.
- /usr/include/pc/init.h has minor changes.
- String parameters must be declared as VAR parameter.
- Local variables of a procedure and global variables of a program are restricted to 32 Kb. If you need more you may use EXPORT/IMPORT variables or the heap.
- The full range of floating point arithmetics of the C-Compiler is now supported, i.e. single precision, double precision, floating point processor, etc. However, a mixture of them is not allowed.
- 2 Byte and 4 Byte integers are supported (option -2 and -4).
- The pc command has some more options.
- The documentation has been updated.

MUNIX - PASCAL
User's Guide
Version 3.5

Documentation-No.: G0930.051-0188

The Pascal User's Guide is intended for people developing new Pascal programs, or compiling and executing existing Pascal programs on MUNIX systems. This manual gives also some insight into the Pascal System structure its components and its behaviour.

MUNIX Pascal is an extended implementation of the Pascal language. Specifically Pascal complies almost completely with the requirements of the ISO standard proposal for Pascal.

This manual is designed for programmers who have a working knowledge of Pascal. Detailed knowledge of MUNIX is helpful but not essential.

Best.-Nr.: G0930.051-0188
DF: 0 1.t 2 3 4 5 6.t 61 62 63 64 7.t 8.t 9.t 92.t 93.t 94 95.t
Author's initials: DK,LY,RG

Table of Contents

* * * * *

1. Introduction

The Pascal User's Guide is intended for people developing new Pascal programs or compiling and executing existing Pascal programs on MUNIX systems. This manual also gives some insight into the Pascal System structure its components and its behaviour.

This manual is designed for programmers who have a working knowledge of Pascal. Detailed knowledge of MUNIX is helpful but not essential. It is advisable to get familiar with the MUNIX documentation and MUNIX standards.

Pascal was designed by Professor N. Wirth as a language for teaching structured programming techniques and as such is used widely in educational institutions. It has also gained popularity as a general-purpose language because it contains a set of language features that make it suitable for many different programming applications. Pascal has furthermore strongly influenced the development of several languages (e.g. ADA). The Pascal language includes a variety of control statements, data types, and predefined procedures and functions.

☛ Throughout this document the following notation is used:

Keywords and **predefined** identifiers are printed in bold face.

Syntactic variables as well as *UNIX*† *components* are printed in italic font.

Metasymbols "{,}" and "[,]" are used for optional parts (0..∞ and 0..1), "(,)" bracket syntactical units and "/" separates alternatives. Three dots (...) mean a repetition of the preceeding item. Terminal symbols are printed in roman face; to distinguish metasymbols from terminal symbols apostrophes ' are used if necessary.

†UNIX is a Trademark of Bell Laboratories.

1.1. Installation Guide for PASCAL Version 3.5

The PASCAL Software is on a streamer or magnetic tape. The streamer or the tape has been created in cpio-format relative to the root directory.

You will need at least 1080 free blocks on the /usr disk.

To install it, proceed as follows:

login as bin
cd /

for streamer :
cpio -ivmdS < /dev/xxx xxx : device name (streamer)

for magnetic tape :
cpio -ivmdB < /dev/xxx xxx : device name (magnetic tape)

2. Supported Language

Pascal is an extended implementation of the Pascal language [1]. MUNIX Pascal adheres to the Pascal language as described in the suggested ISO Standard. This "draft Standard" is a cleaned up version of the original Pascal and has been submitted to ISO for acceptance.

2.1. Deviations

Pascal deviates from the standard proposal in the following items:

(1) Only the first 16 characters of an identifier are significant. The truncation of an identifier to 16 characters alters the meaning of a conforming program.

(2) Standard procedures and functions are not allowed as parameters, as in previous standard proposals. Identical results with minor loss in performance can be obtained by declaring user procedures. For example:

```
function userodd(i: integer) : boolean;
    begin
    userodd := odd(i)
    end;
```

(3) Procedures that are to be used as parameters must be declared at main level.

(4) **files** are not allowed in structured data.

(5) Assignment to **for** control variables is done after evaluation of the initial expression.

(6) The reserved word **nil** may be redefined.

(7) A **goto** between branches of a statement is permitted.

(8) For **text**files, no final end-of-line is supplied unless requested explicitely by the user.

(9) A null string is accepted by the compiler.

2.2. Extensions

2.2.1. Separate Compilation

Pascal is able to compile socalled **modules**, a collection of declarations, procedures and functions. The result of the compilation (an a.out object module) can be handled by the usual MUNIX

components, i.e. they can be stored in libraries, bound(loaded) with other a.out modules, etc.

The separate compilation feature is further supported by enabling import and export of variables, procedures and functions. Modules implemented in Pascal, C or assembler can be linked to Pascal modules. Procedures and functions are imported by using a directive in the declarations part: **extern** for Pascal- and assembler- and **externc** for C-procedures; they are exported implicitely by being defined on the outermost level within a compilation unit. The user is responsible for parameter and result compatibility.

Variables are imported or exported by using the newly introduced keywords **import** or **export** instead of **var**. The predefined variables **input** and **output** are (per default) exported from a mainprogram, if they are listed in the program heading. If used in a module, they have to be imported and must be mentioned in read/write statements. It is not possible to import or export labels!

The syntax for a compilation unit is:

compilation_unit =
program *name* '(' *files* ')' ; *block* .
/ **module** *name* ; {*declaration*} .

block =
{*declaration*} *compound_statement*

2.2.2. Additional Standard Procedures

The following additional standard procedures are available:

addr
This function returns the address of the parameter. The returned address is compatible with all pointer types.

convert
convert (variable, typename) returns its first argument as having **type** typename.

☛ No run-time widening is done; e.g.
convert (apointer, anotherpointertype) works, but
convert ('A', **integer**) won't work!

mark, release
mark and release allow to use the heap as a stack. mark(p) stores the current value of the heap pointer in p. release(p) restores the heap pointer to p. Within one program either **dispose** or **mark** and **release** can be used, but not both simultaneously.

☛ New & dispose use *malloc(3)* and *free(3)* which implement a rather straight-forward memory management. Extensive use of heap will therefor result in remarkable run time penalties. To avoid this problem, use new with mark & release which have a simple and fast memory allocation scheme (see *pc (1)*).

pcclose, pseek
They supply a Pascal interface to the MUNIX system calls *close* and *lseek*.

errorexit
Exit a program and force a core dump.

message
Write a string to the MUNIX file *stderr*.

itoa, atoi
Convert **integer** to ascii and vice versa.

date, ptime
Get date and time in ascii representation.

clock
Get cpu time used by the current process.

2.2.3. Strings

String variables are unique to Pascal. Essentially, they are of type **packed array of char** with a dynamic 'length' attribute. The actual length of a **string** is determined by a final zero-byte. **string** variables are not compatible with variables of type **packed array [...] of char**.

☛ No range checking is done for **string** operations.

The default maximum length of a **string** variable is 80 characters. This value can be overridden in the declaration by appending the desired length enclosed by [] :

```
var s  : string;      { 81 bytes will be allocated }
var s1: string [17]; { 18 bytes will be allocated }
```

A **string** variable has a maximum length of 255 characters.

Assignment to a **string** variable can be performed using the assignment statement, the **read** standard procedure or some other routine (e.g. a **C** standard function). **strings** can be compared irregardless of their current lengths. Furthermore, it is possible to access the components of a **string** variable; the first one has index 0.

When a **string** variable is used as parameter to **read** or **readln**, all characters up to, but not including, the end-of-line character in

the input file will be assigned to it.

When a **string** is written without specifying the field width, the actual number of characters written is equal to the dynamic **string** length. If the field width is longer than the dynamic length, leading blanks are inserted. If the field width is smaller, the **string** is truncated on the right.

Constant strings ('hugo') are compatible with type **packed array** [1..n] **of char** (where n is equal to the string length) and with type **string**. They are also terminated by a zero byte that is not included in the length computation and are limited to a length of 255 characters.

☛ Remember that constant 'a' is a character and not a string. Strings of length one must be supplied by other means (e.g. a **C** routine).

2.2.4. Generic Pointers

Generic pointers provide a tool for generalized pointer handling. Variables of type **address** can be used in the same manner as any other pointer variable with the following exceptions:

- generic pointers cannot be deferenced since there is no type associated with them.
- generic pointers cannot be used as an argument to **new**.
- any pointer can be assigned to a generic pointer. Use **convert** for assigning a typed pointer to a generic pointer.

Example:

```
$ cat example.p
        program pt;
        type
         pc = ^char;
         pi = ^integer;
        var
         a:address;
         vi:pi;
         vc:pc;
        begin
         new(vi);
         a:=convert(vi,pc);
         vc:=a;
        end
        .
```

2.2.5. Attribute packed

As an extension, the attribute **packed** can be applied also to simple types:

```
type byte = packed 0 .. 255;
{ variables of type byte will be allocated one byte }
```

Packed subranges that fit in the ranges -2^7 .. 2^7-1 or 0 .. 2^8-1 are represented in one byte; those fitting in the ranges -2^{15} .. $2^{15}-1$ or 0 .. $2^{16}-1$ are implemented in one word (two bytes).

This feature is supported by one-byte, two-byte and four-byte signed and unsigned arithmetic. But the user should keep in mind, that in most cases the packed data have to be extended to a full **integer** entity. See appendix for details.

Types **short** and **cardinal** are defined as

```
type
  short    = packed -32768 .. 32767;
  cardinal = packed 0 .. 65535;
```

2.2.6. Default case

In a case statement a default case can be defined. Either **otherwise** or **else:** will be accepted.

2.2.7. Declaration

The order of declaration for labels, constants, types, variables, functions and procedures has been relaxed. Any order and any number of declaration sections may be used. Furthermore, **import** and **export** variable declaration are implemented to support the separate compilation feature.

An identifier must be declared before it is used. Two exceptions exist to this rule:

(1) Pointer types may be forward referenced as long as the declaration occurs within the same type-definition-part

(2) Functions and procedures may be predeclared with a forward declaration.

The syntax for a block is:

```
block =      {declaration} compound_statement
declaration =
        import ( names : type ) ; ... ;
        /export ( names : type ) ; ... ;
        /var ( names : type ) ; ... ;
        /label label , ... ;
        /const ( name '=' constant ) ; ... ;
        /type ( name '=' type ) ; ... ;
        /function_declaration ;
        /procedure_declaration ;
```

2.2.8. Underscore as letter

The character '_' is significant and can be used in forming identifiers.

2.2.9. Alternate Symbols

There are two representations for comment symbols ('(*' , '*)' and '{' , '}') and for bracket symbols ('(.' , '.)' and '[' , ']').

2.2.10. Exponent

A lower case **e** may be used to indicate real numbers.

2.2.11. Hexadecimal Constants

Hexadecimal integers are indicated by a preceding "#". The syntax for a hexadecimal integer is:

```
unsigned_number    = digit {digit}  / # hexadigit {hexadigit}
digit                                        = 0/1/2/3/4/5/6/7/8/9
hexadigit                                    = digit /A/B/C/D/E/F/a
```

2.2.12. Character Constants

Certain non-printable characters may be represented according to the following table of escape sequences:

\\	backslash
\b	backspace
\f	form feed
\n	newline
\r	carriage return
\t	horizontal tabulator
\ *ddd*	

The escape sequence \ *ddd* consists of a backslash followed by 1,2, or 3 octal digits which are taken to specify the value of the desired character. If the character following the backslash is not one of those specified, the backslash is ignored.

2.2.13. Additional Predefined Identifiers

See also /usr/include/pc/init.h

```
const
        minint          = -maxint - 1;
        minshort         = -32768;
        maxshort         = 32767;
        mincard          = 0;
        maxcard          = 65535;
        minchar          = '\000';
        maxchar          = '\377';
type
        alfa            = packed array [1..16] of char;
        short            = packed minshort .. maxshort;
        cardinal         = packed mincard .. maxcard;
        {address is predefined too}
import
        argc            : integer;
        argv            : ^ array [1..100] of ^ string;
        environ          : ^ array [1..100] of ^ string;

function   pcclose (var f: file_of_any_type): integer; externc;
function   pseek (var f: file_of_any_type; offset: integer;
                  whence: short): integer; externc;
procedure message (var s: string); externc;
procedure itoa (i: integer; var s: string); externc;
function   atoi (var s: string): integer; externc;
procedure date (var datevar: string); extern;
procedure ptime (var timevar: string); extern;
function   clock : integer; extern;
```

2.3. Implementation Definitions

(1) **maxint** is 2 147 483 647.

(2) **set** bounds are 0 and 255.

(3) A variable is selected before the expression is evaluated in an assignment statement.

(4) Default field width specification:
12 for **integers**, 12 for **reals** and 5 for **booleans**.

(5) **reset** on the standard input file resp. **rewrite** on the standard output file is allowed.

2.4. Error Handling

(1) Uninitialized or undefined variables are not detected.

(2) A missing **reset** or **rewrite** statement is not detected (see 'Limitations of Pascal').

(3) No runtime checks are performed on the tag field of variant records.

(4) No bounds checking is performed on overlapping set operands.

(5) A missing assignment to the function value variable is not detected.

(6) No checks are inserted to check pointers after they have been assigned a value using the variant form of new.

(7) No bound checks are inserted for the **succ**, **pred** or **chr** functions.

(8) The **for** control variable is not invalid after the execution of the **for** statement.

(9) Two nested **for** statements with the same control variable are permitted, but may run into an infinite loop.

(10) Type declarations of the kind

```
. . .
type t = type1;
. . .

procedure p;
type
      pt = ^ t;
      t  = type2;
. . .
```

are not correctly analysed.

3. Pascal under MUNIX

3.1. Creating and Executing a Program

Programs are created and executed with MUNIX commands. With each command, you include information that further defines what you want the system to do. Of prime importance is the file specification, which indicates the file to be processed. You can also specify qualifiers that modify the processing performed by the system (**$** is the system prompting symbol, all your entries are terminated with <carriage return>).

A program is **entered** or corrected by any editor of the user's taste. The file name of Pascal source programs must have the suffix '.p'.

```
$ edit <name>.p
```

The *pc (1)* command **compiles** and **links** the Pascal program. The resulting object module is left in <name>.

```
$ pc -o <name> <name>.p
```

The program is **loaded** and **run** by:

```
$ name
```

The only program parameters supported by the Pascal language are files. There are three ways to associate an (external) MUNIX file specification with an (internal) Pascal file specification. The standard Pascal files **input** and **output** are always associated with the logical MUNIX files *stdin* and *stdout*. Their comfortable and flexible use is described in [2]. All other Pascal files can be associated with any MUNIX file either by assignment within a commandline:

```
$ name pascalfile_1=MUNIX_file_specification    \
          pascalfile_2=MUNIX_file_specification [1] \
          . . .
```

or - if an assignment is missing - interactively:

```
$ name pascalfile_2=MUNIX_file_specification
pascalfile_1 ? MUNIX_file_specification
  . . .
```

[1] As these assignments are considered as one argument each, there must be no blanks before or after the '=' sign.

Example:

```
$ cat example.p
        program example (eingabe,ausgabe);
         var eingabe,ausgabe:text;
        begin
         reset(eingabe);
         rewrite(ausgabe);
        end
        .

$ pc -o example example.p
$ example eingabe=/dev/tty
ausgabe ? example.aus
```

It is advisable to define a shell procedure for a more convenient file assignment especially if some of the files are "fixed" or work files.

As an extension of Pascal, the external filename can be stated in the corresponding **reset** or **rewrite** statement. In this case, a file assignment at run time will be meaningless.

For efficiency, output is generally buffered; the buffer is flushed only when it is full, or - in case of **text** files - if a **writeln** is stated. To facilitate interactive I/O, output to a **file of char** will be unbuffered, if the file is connected to a terminal (*/dev/tty*). Input from a terminal is line-oriented unless the terminal is in raw mode (see *stty(1)*, *ioctl(2)*).

3.2. Support of MUNIX Facilities

A Pascal program has full access to all MUNIX system calls as well as files. The system calls can be accessed like C procedures (see below). The objects are found in the standard library. The predefined procedure **halt** provides a return of an "exit code" to the system. Furthermore the "system variables" argc, argv and environ are provided for access to the command arguments and the process environment. argc indicates the number of arguments, whereas argv references an array of argument strings. The actual length "i" of environ is determined by environ^[i] = **nil**. See 2.2.13 for a declaration of these variables.

The file specifications are command arguments too, i.e. argv^[1]^[0] is the first character of the program name. If you access argv^ or environ^, it is essential to switch off the pointer test.

Example:

The following statement will print out the environment:

```
{$t- }
{switch off pointer test, see section compiler options}
begin
i := 1;
while environ^ [i] <> nil do
   begin
   writeln(output, environ^[i]^);
   i := i+1
   end
end;
```

The C preprocessor *cpp* [3] may be used without limitations.

C and assembler modules can be loaded with Pascal. When connecting Pascal modules with C modules you should take care of parameter compatibility (see 8.). Moreover, you must be sure that the C modules do not use the *sbrk/brk* system call which interferes with the Pascal heap.

3.3. Debugging MUNIX Pascal-32 with SDB

For all modules that will be debugged the option -g should be specified for compiling these modules. SDB also works on modules compiled without this option, but its capabilities are limited to machine level.

Detailed information about the use of SDB may be found in sdb [4].

You will notice the following limitations and features while using SDB for MUNIX Pascal :

- Symbolic constants cannot be referenced.
- Subrange types are not distinguished from the according scalar types.
- Indices of array variables are relative to the lower bound 0 independent of their declarations in programs. SDB knows only the number of elements in every dimension. Enumeration and character type indices are numerated up from value 0 also.
- Variables of set types are treated as arrays of bytes. To display these variables more meaningfully the hexadecimal format may be used.
- The main program is treated as procedure named 'main'.

The following example is an introductory demonstration of how to debug MUNIX Pascal. The debugged program is designed only for this purpose and has no other use. The example is in no way complete.

Compile program modules with -g option

```
$ pc -c -fH -g demo.p demo1.p
$ pc -g -fH demo.o demo1.o
```

Execute file a.out using SDB with no core file. The source files of a.out may be found in the directory ./src . Please note "$" is the operating system prompt and "*" is the SDB prompt.

```
$ sdb a.out - ./src
Source path: ./src
No core image
```

Print name of the current procedure.

```
*e
main() in "./src/demo.p"
```

Print ten lines of the current file starting at line 1.

```
*1z
1: program demo ;
2:
3: type rec  = record
4:           i : integer ;
5:           r : real    ;
6:         end ;
7:    feld = array[1..6] of integer ;
8:
9: var  recp    : ^rec ;
10:     rec1    : rec   ;
```

Print another ten lines starting at current line.

```
*z
10:     rec1    : rec   ;
11:    vektor   : feld ;
12:    i,limit  : integer ;
13:
14: procedure proc(var vektor1 : feld ; li : integer) ; extern ;
15: function sum ( a,b : integer ) : integer ;        extern ;
16:
17: begin
18:  limit := 6 ;
19:  for i := 1 to limit do
```

Print another ten lines starting at current line.

```
*z
19: for i := 1 to limit do
20:   vektor[i] := i ;
21: proc(vektor,4) ;
22: new(recp) ;
23: recp^.i := 3 ;
24: recp^.r := 1.2 ;
25: rec1 := recp^ ;
26: i := sum(3,i) ;
27: end .
```

Change to source file demo1.p.

```
*e demo1.p
"./src/demo1.p"
```

Print ten lines of the current file starting at line 1.

```
*1z
1: module demo1 ;
2:
3: type  feld = array[1..6] of integer ;
4:
5: procedure proc( var vektor1 : feld ; li : integer) ;
6: var i : integer ;
7: begin
8:  for i := 1 to li do
9:    vektor1[i] := vektor1[i] + 1 ;
10: end ;
```

Print another ten lines starting at current line.

```
10: end ;
11:
12: function sum ( a,b : integer ) : integer ;
13: begin
14:   sum := a + b
15: end ;
16: .
```

Set a breakpoint at line 21 in main program.

```
*main:21b
main:21 b
```

Set a breakpoint at the entrance of procedure proc.

```
*proc:b
proc:7 b
```

Run a.out without arguments.

```
*r
a.out
Breakpoint at
main:21:  proc(vektor,4) ;
```

Print out variable 'vektor' of main program.

```
*main:vektor/
vektor[0]/ 1
vektor[1]/ 2
vektor[2]/ 3
vektor[3]/ 4
vektor[4]/ 5
vektor[5]/ 6
```

Note that indices are relative to lower bound 0 !

Print out one element of 'vektor'.

```
*vektor[3]/
vektor[3]/ 4
```

Continue execution.

```
*c
Breakpoint at
0x204 in proc:7: begin
```

Single step.

```
*s
proc:8:  for i := 1 to li do
```

Print out parameter vektor1. Since it is a reference parameter its address is printed.

```
*proc:vektor1 /
0x3f7fee44
```

Print out parameter li. Since it is a value parameter its value is printed.

```
*proc:li/
4
```

Print a 10 line window around current line.

```
*w
3: type  feld = array[1..6] of integer ;
4:
5: procedure proc( var vektor1 : feld ; li : integer) ;
6: var i : integer ;
7: begin
8:  for i := 1 to li do
9:    vektor1[i] := vektor1[i] + 1 ;
10: end ;
11: begin
12:   sum := a + b
```

Set a temporary breakpoint at line 10 and continue.

```
*10c
Breakpoint at
proc:10: end ;
```

Dereference the reference parameter 'vektor1' using the c convention of [0].

```
*proc:vektor1[0]/
vektor1[0]0]/ 2
vektor1[0]1]/ 3
vektor1[0]2]/ 4
vektor1[0]3]/ 5
vektor1[0]4]/ 6
vektor1[0]5]/ 7
```

Change one element of 'vektor1' and print it out.

```
*vektor1[0][5]!100
*vektor1[0][5]/
100
```

Set a temporary breakpoint at line 26 in main and continue.

```
*main:26c
Breakpoint at
main:26:  i := sum(3,i) ;
```

Print a 10 line window around current line.

```
*w
21: proc(vektor,4) ;
22: new(recp) ;
23: recp^.i := 3 ;
24: recp^.r := 1.2 ;
25: rec1 := recp^ ;
26: i := sum(3,i) ;
27: end .
```

Print out record component 'i' pointed to by 'recp'.

```
*recp->i
3
```

Print out components of record variable 'rec1'.

```
*rec1/
rec1.i/ 3
rec1.r/ 1.2
```

Single step without leaving current procedure.

```
*S
main:27: end ;
```

Call function 'sum' with other arguments and print out result value.

```
*sum(4,rec1.i)/
7
```

Quit the debugger.

```
*q
```

3.4. Limitations of Pascal

- Because of the separate compilation feature, a missing **reset** or **rewrite** cannot be detected by the compiler and will most probably cause the program to crash with an address error at the first attempt to access the corresponding file-variable.
- In general, it is possible to **reset input** or **rewrite output** with the effect, that *stdin* or *stdout* is repositioned to the beginning of the associated file. If the file is connected to a pipe this will have no effect at all.

- Because of a very straight-forward register allocation mechanism, there may be very deeply nested expressions that do not compile.

4. Compiler Options

Compiler options inside a Pascal program are "{$<option>}". Each **option** consists of a lower or upper case letter followed by "+" or "-". Options are separated by commas. There must be no blanks between "{" and "$" or between a comma and the succeeding option. The following options are supported:

B/b +/-	b+ accept C-string notation (i.e. with backslash)
D/d +/-	d+ produces code for pointer, subrange and arithmetic overflow check and the tracing of line numbers
E/e +/-	e- will suppress extension warnings
T/t +/-	t+ produces code for pointer check
V/v +/-	v+ produces code for arithmetic overflow check
W/w +/-	w- will suppress warning messages

Defaults: {$b+,d+,e-,w+}
d+ implies v+ and t+.
d - implies v - and t -.

Options appearing before the **program** or **module** symbol are overwritten by options in the *pc (1)* command.

5. Error Handling

Errors are detected and reported by several components of the Pascal system:

- preprocessor *cpp*, see *pc (1)*
- compiler
- loader *ld*, see *pc(1)*
- run-time system

5.1. Compile Time Detection of Source Errors

p0 detects syntactical and some semantical errors and (optionally) deviations from the standard language definition. Errors that are not detected are listed in 2.4. *p0* does not produce error messages or a listing but compiles a condensed version of the diagnostics that will be printed in a readable form by an extra pass *perror*. If only warnings are issued compilation proceeds otherwise it will be terminated.

The option "-L" (see *pc (1)*) produces a full listing with error messages. The default is a listing containing the offending line, its predecessor and the error message. Sometimes an error causes several messages to be printed in which case all but the first one can be ignored.

Furthur passes detect no source errors, but might report a violation of a compiler limitation (see 3.3) or a compiler error (though, of course, it should not). Please let us know if you get an compiler error or an unknown error message.

5.2. Other Errors Detected at Compile Time

During compilation, MUNIX resources can be exhausted, e.g. file system, process table, or memory overflow, etc. Furthermore MUNIX can deny access to files. Extensive treatment of these errors is beyond the scope of this manual. They are described in the MUNIX documentation.

5.3. Runtime Errors

Errors occurring at run time are always fatal, i.e. the program will report an error message, dump the core file and then abort. With the help of the MUNIX debugger *adb* [5] or *sdb* the user can generate a (partially) symbolic post mortem dump which indicates the location of the fatal error, the number of the corresponding source line (if the debug option was on), the dynamic calling sequence, etc.

The error message should comprise a sufficient diagnosis of the error detected. As far as file access is concerned, the errors are mostly reported by the MUNIX system and must be investigated using the documentation. Other messages indicate programming errors such as divide by zero, integer overflow, etc.

6. Pascal System Components

Figure 6.1 gives an overview of the Pascal system components. The preprocessor is the same as the C-compiler's, and does macro preprocessing and including of source files.

The first pass *p0* performs the lexical, the syntactical and the semantical analysis. It constructs a parse tree for each block and outputs it.

The extra pass *optim* is optional and is called by the *-O* option in the pc command (*see pc(1)*).

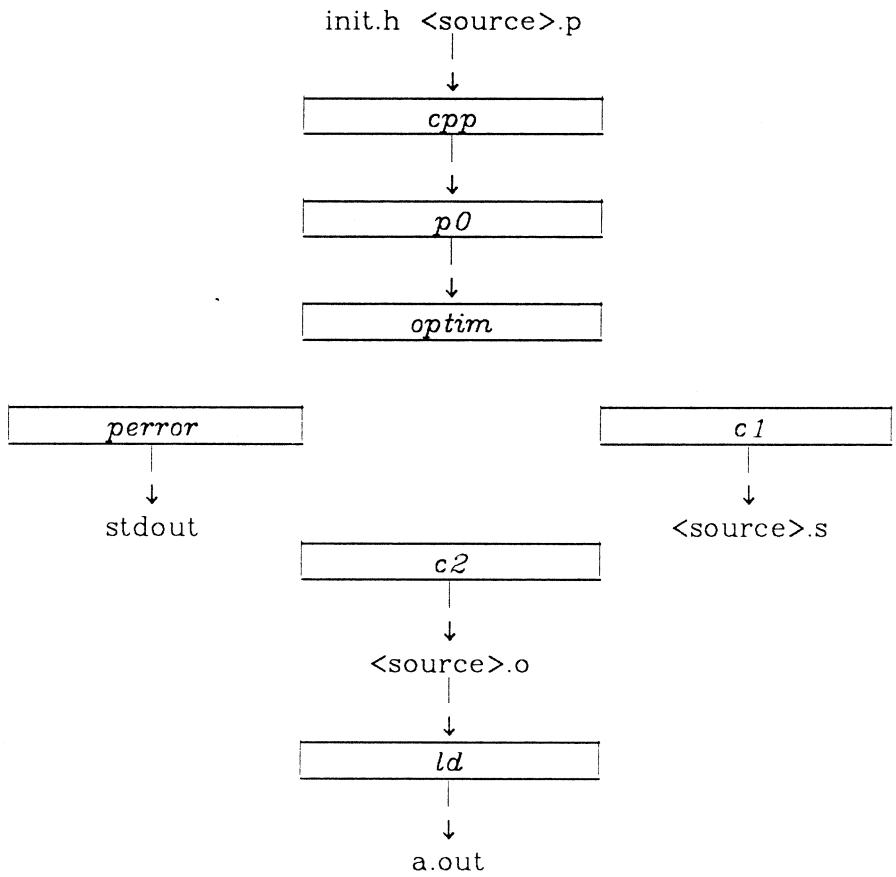


Figure 6.1: Pascal System Components*

The extra pass *perror* produces a source listing if requsted, and prints error messages based on the diagnostics compiled by *p0*.

The second pass *c1* is the same as the C-Compiler's and generates code in several files. The generated code is output in several files.

The third pass *c2* is the same as the C-Compiler's and collects the code distributed in various files. It produces the file <name> containing the object code in *a.out* format [6].

* For MUNIX Pascal-32 the pass nc1 takes the place of c1 and c2. It produces object code in COF format.

6.1. Hardware and Software Environment

Pascal runs on CADMUS Systems under MUNIX Version V.2.

Features of these systems are:

- large memory (≥ 1MB)
- hard disk (≥ 50MB)
- memory management unit

MUNIX is a time-sharing system and provides all features needed by Pascal. Of great importance are the file system, the command interpreter (shell), and the object module management (*ar (1)*, *ld (1)* [7]). Of equal, if not greater, weight are all those MUNIX components which make life easier: *ed*, *med*, *make*, etc.

6.2. P0

The first pass performs lexical analysis, parsing, declaration handling, tree building, and some optimization. This pass is largely machine independent.

The *lexical analysis* is a conceptually simple procedure that reads the input and returns the tokens of the Pascal language as it encounters them: identifiers, constants, operators and keywords. Comments are skipped, decimal and hexadecimal constants, characters and strings are properly dealed with.

The first pass *parses*, as the original Pascal-P4 compiler, the tokens in a top down, left right, recursive descendent fashion. During the processing of a declaration part a symbol table is built up, addresses are allocated to variables and procedures, and the semantic of declarations is checked.

During the processing of the statement part a parse tree is built. The proper use of operators and operands is checked. Some complex syntactical structures are broken down into simpler ones.

6.3. Optimization Pass

The *optim* pass is optional and is invoked by the *-O* option in the *pc* command. Optimization will result in smaller object code and faster run time of your programms.

This pass implements several optimizations: common subexpression elimination, subscript expression optimization, constant folding and an increased usage of registers for variables.

The optimization pass will increase the compile time of a program, we suggest you to use the -O Option only for the final version of your already debugged programs or modules.

6.4. Cross Reference

The cross reference program *xref* produces the usual cross-reference list with identifiers and line numbers.

xref is implemented as an independent component. The reasons are:

- The compiler is not bothered with cross referencing.
- The multiprogramming (parallel processing) of the system can be exploited.
- Many programming errors can be found with the help of a cross-reference listing; i.e. it is not necessary to start the big, resource consuming, compiler.

xref reads from standard input. The following options are interpreted by xref:

-c C-comments will be skipped.

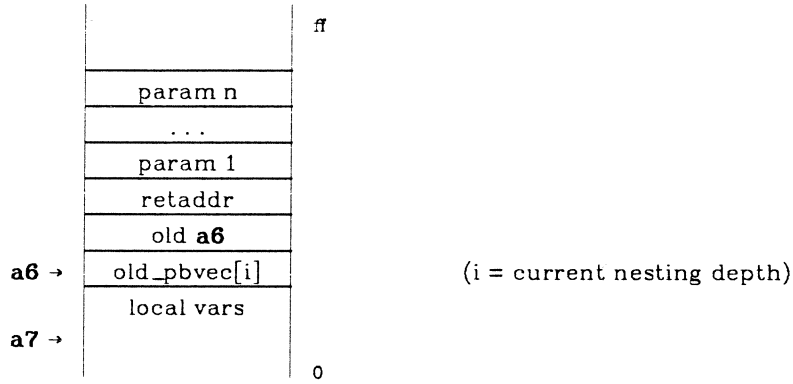
-p Pascal-comments will be skipped.

-P Packed files will be handled.

7. Calling Conventions

Procedure calls are implemented by a commonly used mechanism defining stackframes, which are allocated or deallocated as a procedure is activated or deactivated. In our implementation four registers and a vector of pseudo registers are used: **a7** is used as stackpointer, **a6** references the current stackframe base. **d0** returns a function result and the system variable _pbvec[0..maxdepth] stores the base addresses of all currently accessible stackframes. The layout of a stackframe is shown below.

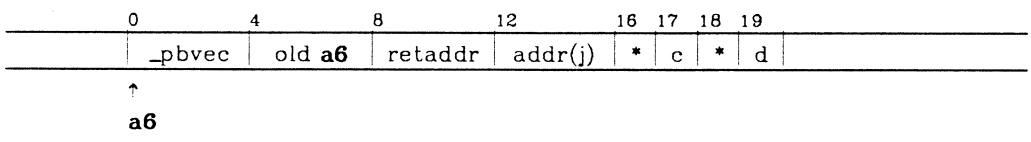
stackframe



A parameter is passed by-reference or by-value depending on whether it was declared as **var** parameter or not. The representation of parameters in memory is the same as for other variables with the exception that they are always word aligned, i.e. a parameter occupying an odd number of bytes in memory will always be followed by a byte of undefined storage.

Example:

stackframe for
procedure p(**var** j:**integer**; c,d:**char**);
after procedure entry



* undefined

8. Data Representation and Allocation

In a.out modules program data fall into three segments: the **text** segment, the **data** segment and the **bss** segment. Pascal uses only two of them: the text segment contains program code and constants whereas static data (variables declared at the outmost level in a module) and **export**ed variables are stored in the bss segment. Automatic and dynamic data are allocated at runtime on the stack and "free memory" is managed by the *brk/sbrk* system calls.

To cope with alignment, one general rule can be stated: any data allocated more than one byte is aligned on a word (2 bytes) boundary.

Variables of scalar and pointer types allocate storage space as summarized in table 8.1. Variables of subrange types are allocated as variables of the associated scalar types. For example, a type 1..10 is considered a subrange of **integer** and therefor allocates a longword. Variables of **packed** subrange types are implemented in one byte (-128 .. 127; 0 .. 255), two bytes (-32768 .. 32767; 0 .. 65535) or four bytes (all others). The structured types are stored as described below.

Type	Storage	Allocation
character	8 bits (1 byte)	Byte
boolean	8 bits (1 byte)	Byte
short	16 bits (1 word)	Word
integer	32 bits (1 longword)	Word
real*	32 bits (1 longword)	Word
enumerated	8 bits (1 byte) if type contains 256 elements or less; 16 bits (1 word) otherwise	Byte if type contains 256 elements or less; Word otherwise
pointer	32 bits (1 longword)	Word

Table 8.1: Storage of Scalar and Pointer Types

* MUNIX Pascal-32 always allocates 64 bits for 'real' variables. On the 16-bit System, you can choose 32- or 64-bit format.

A **set** allocates storage depending on the ordinal value of its largest element: the number of bytes it occupies is equal to the ordinal value rounded up to the nearest byte boundary. Since the size of a **set** is limited to 256 elements, with ordinal values from 0 to 255, a **set** occupies at most 32 bytes.

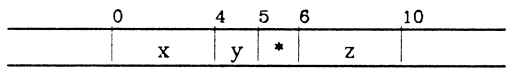
An **array** is stored and aligned according to the type of its components. For example, each element of a character **array** is stored in a byte and aligned on a byte boundary; if the **array** has more than one component then the total **array** is aligned a one word boundary. Similarly, each element of an **array of set of** 3..21 occupies three bytes and is aligned on a word boundary.

strings and constant strings are internally terminated by a binary 0; i.e. they adhere to the C convention.

Records are stored and aligned field by field according to the type of the field. For example, a variable of type

```
record
  x: integer;
  y: boolean;
  z: 1..10
end;
```

is aligned on a word boundary because it occupies more than one byte of storage. The figure beneath shows how this variable is stored:

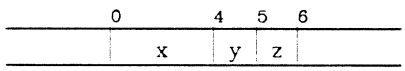


* undefined

The attribute **packed** does not affect the allocation of data structures; it is only interpreted with subranges as stated earlier. To yield a more compact representation than in the example above, you had to define the following structure:

```
record
  x: integer;
  y: boolean;
  z: packed 1..10
end;
```

which would result in the following storage allocation:



9. Appendix

9.1. Examples

9.1.1. Sample program

```
 1  program ackermann(input,output);
 2   var x, y: integer;
 3   function ack(i,j: integer): integer;
 4    begin
 5    if     i = 0 then  ack:=j+1
 6    else  if j = 0 then  ack:=ack(i-1,1)
 7          else ack:=ack(i-1,ack(i,j-1))
 8    end;
 9
10   begin
11    repeat
12     writeln(output,'Enter two integers. Terminate with zero.');
13     read(input,x,y);
14     writeln(output,'acker(',x:0,',',y:0,') = ',ack(x,y):0)
15     until x=0
16   end.
```

9.1.2. Cross reference

```
ack                3p    5    6    6    7    7    7   14
ackermann          1
i                  3f    5    6    7    7
input              1    13
integer            2     3    3
j                  3f    5    6    7
output             1    12   14
read              13
x                  2v   13   14   14   15
y                  2v   13   14   14
```

9.1.3. Separate Compilation

```
 1 program ackermann(input,output);
 2 var x, y: integer;
 3
 4 import counter: integer;
 5
 6 function ack(i,j: integer): integer;  extern;
 7
 8 begin
 9  repeat
10    writeln(output,'Enter two integers. Terminate with zero.');
11    read(input,x,y);
12    counter:=0;
13    writeln(output,'acker(',x:0,',',y:0,') = ',ack(x,y):0);
14    writeln(output,'number of calls=',counter:5);
15  until x=0
16 end.
```

```
 1 module ack;
 2
 3 export counter: integer;
 4
 5 function ack(i,j: integer): integer;
 6  begin
 7  counter:=counter+1;
 8  if       i = 0 then  ack:=j+1
 9  else if       j = 0 then  ack:=ack(i-1,1)
10         else   ack:=ack(i-1,ack(i,j-1))
11  end;
12 .
```

9.2. Coercions

The following gives an overview of the coercions implemented by Pascal to provide consistent use of one-, two- and four-byte signed and unsigned operands. The rules are valid for all arithmetic and relational operations.

Index expressions are always extended to a full **integer** value. Expressions representing set elements are considered unsigned one-byte.

	f	i	i1	i2	u	u1	u2
f	f	f	f	f	f	f	f
i	f	i	i	i	i	i	i
i1	f	i	i1	i	i	i	i
i2	f	i	i	i2	i	i	i
u	f	i	i	i	u	u	u
u1	f	i	i	i	u	u1	u
u2	f	i	i	i	u	u	u2

Notation

f: floating point
i: (signed) integer
u: unsigned (0 .. 2^{31}-1)
i1: one-byte integer
i1: two-byte integer
u1: one-byte unsigned
u2: two-byte unsigned

9.3. Standard Procedures and Functions

For a detailed description see also [1].

Procedure	*Parameter*	*Result*	*Function*
abs(x)	**integer** or **real**	same as x	Computes the absolute value of x.
addr(x)	any type	**address**	Type address is compatible with all pointer types.
arctan(x)	**integer** or **real**	**real**	Computes the arcus tangens of x.
atoi(s)	**string**	**integer**	Convert ascii string to **integer**.
chr(x)	**integer**	**char**	Returns the character whose ordinal number is x.
clock		**integer**	Returns the cpu time (in milliseconds) used by the current process.
convert(a,t)	any type	t	Returns value of a with type t.
cos(x)	**integer** or **real**	**real**	Computes the cosinus of x.
date(x)	**string**		Return date in ascii representation: dd-mon-yyyy
dispose(p)	any pointer		Deallocates heap memory pointed to by p. After dispos p becomes undefined.
eof(f)	file	**boolean**	End of file encountered. **true** only when the file position is after the last element in the file.

eoln(f)	file	**boolean**	End of line encountered. **true** only when the file position is after the last character in a line. The value of f^ is a space.
errorexit			Exit program and force a core dump.
exp(x)	**integer** or **real**	**real**	Computes the base of the natural logarithmus raised to the power of x.
get(f)	file		Moves the current file position to the next element
halt(x)	**integer**		Terminates the execution of the program and returns the value of x. See also the system call **exit**(2).
halt			same as **halt**(0).
itoa(i,s)	**integer,string**		Convert **integer** to ascii; the result is delivered in variable s.
ln(x)	**integer** or **real**	**real**	Computes the natural logarithmus of x.
mark(x)	any pointer		Stores the current value of the heap pointer into x.
message(x)	**string**		Write string to *stderr*.
new(p)	any pointer		Allocates heap memory and returns the address in p.
new(p,t1,..tn)			Allocates heap memory to pointer. p^ must be a record with variants.
odd(x)	**integer**	**boolean**	Returns **true** if the integer x is odd; **false** otherwise.
ord(x)	any scalar type except **real**	**integer**	Returns the ordinal (integer) value corresponding to the value of x.

pack(a,i,z)			z:=a[i..n]; where i..n: index range of z.
page(f)	file		skip to the top of a new page before printing the next line of the textfile f. The default for f is **output**.
pcclose(f)	file		See system call close (2).
pred(x)	any scalar type except **real**	same as x	Returns the predecessor value of x.
pseek(f,o,w)	file, **integer** **short**		See system call lseek (2).
ptime(x)	**string**		Return time in ascii representation: hh-mm-ss.0 .
put(f)	file		Appends f^ to the file f. The default for f is **output**.
read(f,x)	file type of x depends on the filetype		Reads the value of x from the file f. The default for f is **input**.
readln(f)	**text**		Skips to the beginning of the next line. The default for f is **input**.
read(f,x1,..,xn)			same as **read**(f,x1); **read**(f,x2,..xn)
readln(f,x1,..,xn)			same as **read**(f,x1,..,xn); **readln**(f)
release(x)	any pointer		Loads the heap pointer with the value of x.
reset(f[,s])	file,**string**		Resets file for reading. The optional second parameter designates a MUNIX pathname.

rewrite(f[,s])	file,**string**		Resets file for writing. The optional second parameter designates a MUNIX pathname.
round(x)	**real**	**integer**	trunc(x + 0.5) if x >= 0 trunc(x - 0.5) if x < 0
sin(x)	**integer** or **real**	**real**	Compute sinus of x.
sizeof(x)	any type	**integer**	Returns size used to represent variables of same type as x.
sqr(x)	**integer** or **real**	same as x	Computes the square of x.
sqrt(x)	**integer** or **real**	**real**	Compute square root of x.
succ(x)	any scalar type except **real**	same as x	Returns the successor value of x.
trunc(x)	**real**	**integer**	Truncate x to a **integer** value.
unpack(z,a,i)			a[i..n]:=z; where i..n: index range of z.
write(f,x)	**text** type of x depends on the filetype		Writes the value of x to the file f. The default for f is **output**.
writeln(f)	file		Starts a new line. The default for f is **output**.
write(f,x1,..,xn)			same as **write**(f,x1); **write**(f,x2,..xn)
writeln(f,x1,..,xn)			same as **write**(f,x1,..,xn); **writeln**(f)

9.4. Syntax Equations

```
letter = a/b/c/d/e/f/g/h/i/j/k/l/m/n/o/p/q/r/s/t/u/v/w/x/y/z
         /A/B/C/D/E/F/G/H/I/J/K/L/M/N/O/P/Q/R/S/T
         /U/V/W/X/Y/Z/_

digit = 0/1/2/3/4/5/6/7/8/9

name =                letter { letter / digit }
constant_name =       name
type_name =           name
variable_name =       name
...
names =               name , ...
label =               unsigned_number

compilation_unit =
   program name '(' names ')' ; block .
   / module name ; {declaration} .

block = {declaration} compound_statement

declaration =
   / import ( names : type ) ; ... ;
   / export ( names : type ) ; ... ;
   / var    ( names : type ) ; ... ;
   / label  label , ... ;
   / const  ( name '=' constant ) ; ... ;
   / type   ( name '=' type ) ; ... ;
   / function_declaration ;
   / procedure_declaration ;

constant =
   [ sign ] ( unsigned_number / constant_name )
   / character_string

sign = + / -

unsigned_constant =
    unsigned_number / character_string
   / constant_name / nil

unsigned_number =
   digit {digit}  / # hexadigit {hexadigit}

hexadigit =           digit /A/B/C/D/E/F/a/b/c/d/e/f
character_string = / *characters enclose by ' ' */
```

---------- t y p e ----------

type = *type_name* / *new_type*
new_type = *simple_type* / *structured_type* / ^ *type_name*

simple_type = *ordinal_type* / **real**

ordinal_type =
'(' *names* ')'
/ *constant* .. *constant*
/ *ordinal_type_name*

structured_type =
[**packed**] *unpacked_structured_type*
/ *structured_type_name*

unpacked_structured_type =
array '[' *ordinal_type* , ... ']' **of** *type*
/ **file of** *type*
/ **record** [*field_list* [;]] **end**
/ **set of** *ordinal_type*

field_list =
fixed_part [; *variant_part*]
/ *variant_part*

fixed_part = (*names* : *type*) ; ...
variant_part = **case** [*tag_field_name* :] *tag_type* **of** *variant* ; ...
variant = *case_constant_list* : '(' [*field_list* [;]] ')'

case_constant_list = *case_constant* , ...
case_constant = *constant* [.. *constant*]

---------- e x p r e s s i o n ----------

variable = (*variable_name* / *field_name*)
{ '[' *expression* , ... ']'
/ . *field_name*
/ ^ }

factor =
variable
/ *unsigned_constant*
/ *function_name* ['(' *actual_parameter* , ... ')']
/ *set*
/ '(' *expression* ')'
/ **not** *factor*

```
actual_parameter =
   expression
   / procedure_name / function_name

set    =        '[' [ member , ... ] ']'
member =        expression [ .. expression ]

term = factor
   { ( * / '/' / div / mod / and ) factor }

simple_expression = [ sign ] term
   { ( + / - / or ) term }

expression = simple_expression
   [ ( <> / = / < / > / <= / <= ) simple_expression ]


---------- procedure ----------

procedure_declaration =
   procedure_heading ;
   ( block / directive )

function_declaration =
   function_heading ;
   ( block / directive )

directive =                forward / extern / externc
procedure_heading =        procedure name [ '(' parameter ; ... ')' ]

function_heading  =        function name ['(' parameter ; ... ')']
                 : result_type

parameter =
    function_heading / procedure_heading
   / names : type
   / var names : ( type_name / array_type )

array_type =
   array '[' index_type ; ... ']' of (type_name / array_type )

index_type =
   name .. name : ordinal_type_name


---------- statement ----------
```

statement = [*label* :]
compound_statement
/ **case** *expression* **of** *case_element* ; ... [;] **end**
/ **for** *name* := *expression*
(**to** / **downto**) *expression*
do *statement*
/ **goto** *label*
/ **if** *expression* **then** *statement*
[**else** *statement*]
/ **repeat** *statement* ; ... **until** *expression*
/ **while** *expression* **do** *statement*
/ **with** *variable* , ... **do** *statement*
/ (*variable* / *function_name*) := *expression*
/ *procedure_name* ['(' *actual_parameter* , ... ')']

compound_statement =
begin *statement* ; ... **end**

case_element =
((*case_constant* / **else**) : / **otherwise**)
statement

9.5. Reserved Identifiers

The following identifiers are predefined by Pascal.

abs	environ	mincard	readln
addr	eof	minchar	real
address	eoln	minint	release
alfa	errorexit	minshort	rewrite
arctan	exp	new	round
argc	false	nil	short
argv	get	odd	sin
atoi	halt	ord	sqr
boolean	integer	pack	sqrt
cardinal	itoa	page	succ
char	ln	pcclose	text
chr	mark	pread	true
clock	maxcard	pseek	trunc
convert	maxchar	ptime	unpack
cos	maxint	put	write
date	maxshort	read	writeln
dispose	message		

Furthermore, there are identifiers of data and subroutines used by the Pascal run time system that are referenced at linking time. Among them are system calls (e.g. open, close) and some C standard subroutines; all others are listed below.

Warning: if a programmer uses these identifiers to define his own procedures or types and the linker ld will reference the users' entity and report no error.

_copen	_pcmpa	_pjerr	_pwrc
_error	_pcmpg	_pmulm	_pwrc1
_ferror	_pcmpl	_popen	_pwrf
_flushbuffer	_pcmpm	_pperr	_pwri
_paddm	_pconf	_prdi	_pwrr
_pblank	_pdot2	_prdr	_pwrs
_pbvec	_pelem	_prds	_syserror
_pcase	_pemptyset	_pserr	atan
_pcerr	_pemptystr	_pset	closefiles
_pchki	_pgoto	_psqr	final
_pchkp	_pierr	_psubm	fsqr
_pchku	_pin	_puerr	ln
_pcidx	_pinit	_pwrb	main

10. References

[1] Jensen K.; Wirth N.: *Pascal User Manual and Report*

Second Corrected Reprint of the Second Edition 1978, Springer Verlag

[2] *MUNIX Volume 2*

[3] Kernighan B.; Ritchie D.: *The C Programming Language*

Prentice-Hall Software Series; Prentice Hall, Inc.; New Jersey

[4] *MUNIX Commands; Volume Ia; sdb (1)* and

MUNIX Programming and Administration; Volume II; sdb

[5] *MUNIX Commands; Volume Ia; adb (1)* and

MUNIX Basics, Options; Volume III; adb (5)

[6] *MUNIX Programming; Volume 1b; a.out (5)*

NAME

pc – Pascal compiler

SYNOPSIS

pc [option] ... file ...

DESCRIPTION

pc is the MUNIX Pascal compiler. It accepts several types of arguments:

Arguments whose names end with '.p' are taken to be Pascal source programs; they are compiled, and each object program is left on a file with suffix '.o'. The '.o' file is normally deleted, however, if a single Pascal program is compiled and loaded all at one go. Errors detected by the compiler are listed on *stdout*. This Pascal compiler has the *cpp*-identifier **m68000** predefined, i.e. "#ifdef m68000" is true.

Filenames are built from the basename of the source program and a certain suffix.

A call "pc test.p" is the same as "pc -c test.p && ld -t -X /lib/crt0.o test.o -lpdisp -lpffp -lp -lffp -lc && rm test.o". The call "pc *.o" is the same as "ld -t -X /lib/crt0.o *.o -lpdisp -lpffp -lp -lffp -lc". If your directory contains the file xyz.p, a call "make xyz" will result in "pc -o xyz xyz.p".

The following options are interpreted by *pc*. See *ld*(1) for load options.

–c Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.

–d Switch on the debug mode.

–e Display extension warning messages to *stdout*.

–f[FNSHT]

Floating point options. One of these options must be given, if the program includes floating point operations.

–f The option -f alone is default and will compile code for the Motorola Fast Floating Point Package, which is fast but does not have double precision arithmetic. The libraries **libpffp.a** and **libffp.a** will be searched before libc.a. Implicit define options: -DFFP.

–fF

The option -fF will compile code for the Motorola IEEE Floating Point Package. This package offers single and double precision according to the IEEE specifications. The execution speed is rather slow, as the generated code is emulated in Software. The libraries **libpmot.a** and **libmot.a** will be searched before libc.a. Implicit define options: -DIEEE and -DMOT_IEEE.

–fN

The option -fN will compile code for the CADMUS FPP board with a NSC 16081 floating point processor. The numbers are also in IEEE format, but the generated code is much different from the one produced with option -fF. The libraries **libpnsc.a** and **libnsc.a** will be searched before libc.a. Implicit define options: -DIEEE and -DNSC_IEEE.

-fS

The option -fS will compile code with Single precision for the CADMUS FPP board with a NSC 16081 floating point processor. The libraries **libpnSc.a** and **libnSc.a** will be searched before libc.a. Implicit define options: -DNSS_IEEE.

-fH

The option -fH will compile code for the Motorola 68881 floating point coprocessor. This code will only run if the CPU is 68020. The option -M is set implicitly. The libraries **libp881.a** and **lib881.a** will be searched before libc.a. Implicit define options: -DIEEE and -DMOT_IEEE. See also *mot881*(3)

-fT

The option -fT also compiles code for the Motorola 68881 floating point coprocessor. In addition, calls to functions sin, cos, exp etc. are converted into floating point instructions fsin, fcos, fexp etc. and executed by the 68881. The option -M is set implicitly. The libraries **libp881.a** and **lib881.a** will be searched before libc.a. Implicit define options: -DIEEE and -DMOT_IEEE. See also *mot881*(3)

-g Adds entries to the symbol table of the produced .o module which indicate line numbers and corresponding code location. To set a breakpoint at line 75 you can tell adb "L75:b". Make sure that only one module of a program is compiled with this option.

-m Load **new, mark & release** instead of **new & dispose**.

-n Suppress execution ("dry run") of *pc* commands.

-o *output*

Name the final output file *output*. If this option is used the file 'a.out' will be left undisturbed. This option is passed on to the linker.

-p Arrange for the compiler to produce code which counts the number of times each routine is called; also, if loading takes place, replace the standard startup routine by one which automatically calls *monitor*(3C) at the start and arranges to write out a *mon.out* file at normal termination of execution of the object program. An execution profile can then be generated by use of *prof*(1).

-t0 Find only the designated compiler pass0 in the file whose name is constructed by a **-B** option. In the absence of a **-B** option, the *string* is taken to be '/usr/lib/o'.

-v Trace and print *pc* commands. Temporary files are not deleted.

-w Display warning messages to *stdout*.

-4 Changes the integer size from 2 (default) to 4. This makes programs which neglect the differences between int and pointer or int and long much more portable. On the other hand, the produced code is less efficient. The linker transforms the option -llib to -lLlib, e.g. -lc to -lLc or -lcurses to -lLcurses. These libraries are themselves compiled with option -4.

–B*string*
Find substitute compiler pass0 in the file named *string* with the suffix p0. If *string* is empty, use a standard backup version.

–D*name=def*

–D*name*
Define the *name* to the preprocessor, as if by '#define'. If no definition is given, the name is defined as 1.

–E Run only the macro preprocessor and send the result to the standard output. The output is intended for compiler debugging; it is unacceptable as input to *pc*.

–I*dir* Files of '#include' type whose names do not begin with '/' are always sought first in the directory of the *file* argument, then in directories named in **–I** options, then in directories on a standard list.

–L Arrange for the compiler to produce code which puts the current linenumber into the stackframe during execution. The C-stacktrace of *adb*(1) (command $c or $C) will then give for each active procedure the current linenumber.

–M Produce code for the Motorola 68020 cpu. Some new instructions of the 68020 like extb.l, or 32 bit multiply/divide, or bit field instructions, or scaling of index variables by 1,2,4 or 8 are generated with this option. Code thus generated will normally not run on the 68000/68010.

–O Invokes optimisation phase.

–P Run only the macro preprocessor and place the result for each '.p' file in a corresponding '.i' file and has no '#' lines in it.

–S Compile the named Pascal programs and leave the assembler-language output on corresponding files suffixed '.s'.

–T With this option initialized data is put into the text- rather than the data segment. Thus for programs like the shell, whose text segments are shared between many users, memory space can be saved.

–U*name*
Remove any initial definition of *name*.

Other arguments are taken to be either loader(ld) option arguments, or Pascal-compatible object programs, typically produced by an earlier *pc* run, or perhaps libraries of Pascal-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name **a.out.**

FILES

file.i	preprocessor output file
file.l	error and listing file
file.o	object file
file.p	input file
file.s	assembler listing file
a.out	loaded (linked) output

/tmp/ptm*	temporary files for *pc*
/lib/cpp	preprocessor
/usr/lib/p0	compiler pass0 for *pc*
/lib/c[12]	compiler passes for *pc*
/lib/crt0.o	runtime initialization
/lib/mcrt0.o	runtime initialization for profiling
/lib/lcrt0.o	runtime initialization for option -4
/lib/mlcrt0.o	runtime initialization for profiling and option -4
/usr/lib/perror	prints errors and listing
/usr/lib/libpmark.a	Pascal runtime-support (new, mark & release)
/usr/lib/libpdisp.a	Pascal runtime-support (new & dispose)
/usr/lib/libp.a	Pascal runtime-support
/usr/lib/libffp.a	Floating point library FFP
/usr/lib/libmot.a	Floating point library Motorola IEEE
/usr/lib/lib881.a	Floating point library M68881 IEEE
/usr/lib/libnsc.a	Floating point library NCS IEEE
/usr/lib/libnSc.a	Floating point library NCS
/usr/lib/libpffp.a	Pascal Floating point library FFP
/usr/lib/libpmot.a	Pascal Floating point library Motorola IEEE
/usr/lib/libp881.a	Pascal Floating point library M68881 IEEE
/usr/lib/libpnsc.a	Pascal Floating point library NCS IEEE
/usr/lib/libpnSc.a	Pascal Floating point library NCS
/lib/libc.a	standard library, see *intro* (3)
/usr/include/pc/init.h	initialisation of Pascal compilation
/usr/lib/op0	backup compiler pass0 for *pc*

SEE ALSO

K. Jensen and N. Wirth *Pascal User Manual and Report* Springer Verlag 1978

monitor(3C), prof(1), adb(1), ld(1), fp(3), mot341(3), mot881(3)

DIAGNOSTICS

The diagnostics produced by *pc* itself are intended to be self-explanatory. Occasional messages may be produced by the loader(ld). Any messages from the other passes should be send to PCS.

NAME

pc – Pascal compiler

SYNOPSIS

pc [option] ... file ...

DESCRIPTION

Pc is the MUNIX Pascal compiler. It accepts several types of arguments:

Arguments whose names end with '.p' are taken to be Pascal source programs; they are compiled, and each object program is left on a file with suffix '.o'. The '.o' file is normally deleted, however, if a single Pascal program is compiled and loaded all at one go. Errors detected by the compiler are listed on *stdout*. This Pascal compiler has the *cpp*-identifier **m68000** predefined, i.e. "#ifdef m68000" is true.

Filenames are built from the basename of the source program and a certain suffix.

A call "pc test.p" is the same as "pc -c test.p && ld /lib/crt0.o test.o -lpdisp -lp -lc && rm test.o". The call "pc *.o" is the same as "ld -t /lib/crt0.o *.o -lpdisp -lp -lc". If your directory contains the file xyz.p, a call "make xyz" will result in "pc -o xyz xyz.p".

The following options are interpreted by *pc*. See *ld*(1) for load options.

–c Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.

–d Switch on the debug mode.

–e Display extension warning messages to *stdout*.

–fH The option -fH will compile code for the *Motorola M68881 floating point coprocessor*. The libraries **libpm.a** and **libm.a** will be searched before **libc.a**. Implicit define options: -DIEEE and -DMOT_IEEE. See also *mot881*(3M)

–fT The option -fT also compiles code for the *Motorola M68881 floating point coprocessor*. In addition, calls to functions sin, cos, exp etc. are converted into floating point instructions fsin, fcos, fexp etc. and executed by the M68881. The libraries **libpm.a** and **libm.a** will be searched before **libc.a**. Implicit define options: -DIEEE and -DMOT_IEEE. See also *mot881*(3M)

–g Adds line number and symbol table information to the produced .o file necessary for symbolic debugging using *sdb*(1). Cannot be used with -O option since optimization may rearrange code and cause line number information to be incorrect. Must be given for compiling and loading as well. During loading, the library libg.a will be included.

–m Load **new, mark & release** instead of **new & dispose**.

–n Suppress execution ("dry run") of *pc* commands.

–o *output*

Name the final output file *output*. If this option is used the file 'a.out' will be left undisturbed. This option is passed on to the linker.

-p Arrange for the compiler to produce code which counts the number of times each routine is called; also, if loading takes place, replace the standard startup routine by one which automatically calls *monitor*(3C) at the start and arranges to write out a *mon.out* file at normal termination of execution of the object program. An execution profile can then be generated by use of *prof*(1).

-t0 Find only the designated compiler pass0 in the file whose name is constructed by a **-B** option. In the absence of a **-B** option, the *string* is taken to be '/usr/lib/o'.

-v Trace and print *pc* commands. Temporary files are not deleted.

-w Display warning messages to *stdout*.

-B*string*
Find substitute compiler pass0 in the file named *string* with the suffix p0. If *string* is empty, use a standard backup version.

-D*name=def*

-D*name*
Define the *name* to the preprocessor, as if by '#define'. If no definition is given, the name is defined as 1.

-E Run only the macro preprocessor and send the result to the standard output. The output is intended for compiler debugging; it is unacceptable as input to *pc*.

-I*dir* Files of '#include' type whose names do not begin with '/' are always sought first in the directory of the *file* argument, then in directories named in **-I** options, then in directories on a standard list.

-L Arrange for the compiler to produce code which puts the current linenumber into the stackframe during execution. The C-stacktrace of *adb*(1) (command $c or $C) will then give for each active procedure the current linenumber.

-O Invokes optimisation phase.

-P Run only the macro preprocessor and place the result for each '.p' file in a corresponding '.i' file and has no '#' lines in it.

-S Compile the named Pascal programs and leave the assembler-language output on corresponding files suffixed '.s'.

-T With this option initialized data is put into the text- rather than the data segment. Thus for programs like the shell, whose text segments are shared between many users, memory space can be saved.

-U*name*
Remove any initial definition of *name*.

-Y *count*
Set the maximum value of symbol table entries handled by the code generator to *count*.

Other arguments are taken to be either loader(ld) option arguments, or Pascal-compatible object programs, typically produced by an earlier *pc* run, or perhaps libraries of Pascal-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name **a.out.**

FILES

file.i	preprocessor output file
file.l	error and listing file
file.o	object file
file.p	input file
file.s	assembler listing file
a.out	loaded (linked) output
/tmp/ptm*	temporary files for *pc*
/lib/cpp	preprocessor
/usr/lib/p0	compiler pass0 for *pc*
/lib/nc1	compiler pass1 for *pc*
/lib/crt0.o	runtime initialization
/lib/mcrt0.o	runtime initialization for profiling
/usr/lib/perror	prints errors and listing
/usr/lib/libpmark.a	Pascal runtime-support (new, mark & release)
/usr/lib/libpdisp.a	Pascal runtime-support (new & dispose)
/usr/lib/libp.a	Pascal runtime-support
/usr/lib/libpm.a	Pascal Floating point library M68881 IEEE
/lib/libc.a	standard library, see *intro* (3)
/lib/libm.a	Floating point library M68881 IEEE
/usr/include/pc/init.h	initialisation of Pascal compilation
/usr/lib/op0	backup compiler pass0 for *pc*

SEE ALSO

K. Jensen and N. Wirth *Pascal User Manual and Report* Springer Verlag 1978

adb(1), sdb(1), ld(1), prof(1), monitor(3C), fp(3M), mot881(3M)

DIAGNOSTICS

The diagnostics produced by *pc* itself are intended to be self-explanatory. Occasional messages may be produced by the loader(ld). Any messages from the other passes should be send to PCS.