

***NDP Pascal-***



***Micro***  
***Way***

# NDP Pascal-386/486 v. 4.0D for UNIX V.3 Release Notes

## 1.0 Overview

The NDP compilers, available in four languages (C, C/C++, Fortran, and Pascal) allow you to compile, link, and execute 80486 and 80386 32-bit protected mode code.

## 1.1 Bug Fixes

Weitek libraries and libraries for use with the standard UNIX profiler have been updated. Additionally, the `-g` switch (debugger information) and the `-n2` switch have been updated.

Problems with the `cabs` function have been corrected.

## 1.2 Changes in Previous Releases

Before version 4.0b, loop unrolling was either on or off. The `-ur` switch was added in version 4.0b which allows the user to control how a loop is unrolled. The syntax of the switch is:

```
-ur={number}
```

where `{number}` is 2, 4, 8, 16, 32 or 64. Which switch to use for the best performance depends on the code being optimized.

## 2.0 Technical Support

If you encounter difficulty in the installation or operation of the NDP compilers, please notify Microway Technical Support. You should have the language, platform, operating system, peripherals, and your user number available.

Microway, Inc.  
Box 79 Research Park  
Kingston, MA 02364  
UNITED STATES

Phone: +508/746-7341  
Fax: +508/746-4678

40

40

40

**NDP 386|486**

**UNIX**

**USER'S MANUAL**

***Microway***<sup>®</sup>

Research Park  
Box 79

Kingston • Massachusetts 02364 • USA



NDP C|C++-386, NDP C|C++-486, NDP Fortran-386, NDP Fortran-486, NDP Pascal-386, NDP Pascal-486, and Microway are trademarks of Microway, Inc.

UNIX is a registered trademark of AT&T.

Cyrix and EMC87 are trademarks of Cyrix Corporation.

Intel, SX, 287, 386, 387, 486, i486, and i860 are trademarks of Intel Corporation.

Microsoft, Microsoft Fortran, and MS-DOS are registered trademarks of Microsoft Corporation. OS/2 is a registered trademark of International Business Machines, Inc.

OS/386 is a trademark of Ergo Computing, Inc.

Phar Lap, 386|DOS-Extender, 386ASM, 386LINK, and 386|VMM are trademarks of Phar Lap Software, Inc.

Weitek is a trademark of Weitek Corporation.

# Contents

<b>Set Up</b>	<b>1</b>
1.1 System Requirements	1
1.2 Disk Contents	1
1.3 Installation	1
1.3.1 Environment Variables	2
1.4 Testing the Compiler	2
1.5 Troubleshooting	2
<b>Using The Compiler</b>	<b>5</b>
2.1 The Compiler Driver	5
2.2 Compiler Driver Syntax	5
2.2.1 Example	6
2.3 Compiler Options and Switches	6
<b>Optimizations</b>	<b>11</b>
3.0 Overview	11
3.1 Memory Allocation	12
3.2 Register Allocation by Coloring	12
3.3 Static Address Elimination	13
3.4 Register Coalescing	14
3.5 Prolog and Epilog Code Optimization	15
3.6 Peephole Optimizations	15
3.7 Speed Optimizations	15
3.8 Loop Rotation	16
3.9 Loop Invariant Analysis	16
3.10 Strength Reduction	17
3.11 Dead Code Elimination	19
3.12 Inline Multiplication and Division	19
3.13 Constant Propagation	20
3.14 Constant Expression Folding	20
3.15 Common Subexpression Elimination (CSE)	21
3.16 Live/Dead Analysis	21
3.17 Cross Jumping (i.e., Tail Merging / Code Hoisting)	21
3.18 Loop Unrolling	22
3.19 Inliner	23
3.19.1 Size vs. Frequency of Use	23
3.19.2 Recursion	23
3.19.3 Definition of Function is Exported/Imported/Static	24
3.19.4 Address of Function Taken	24
3.19.5 Nested Functions	24
<b>Runtime Organization and Numerics</b>	<b>25</b>
4.1 Lower Level Characteristics	25
4.2 Integer Data Type	25
4.3 Single Precision Real	25

4.4 Double Precision Real	26
4.5 Single and Double Real Encodings	27
4.6 Language Data Types	28
4.7 Internal Registers	29
4.7.1 General Purpose Registers	29
4.7.2 Segment Registers	30
4.7.3 The 80386/80486 Flags Register	31
4.7.4 Systems Control Registers	32
4.8 The 80387 Register Set	33
4.8.1 80387 Data Registers	33
4.8.2 The Status Word Register	33
4.8.3 The Control Word Register	36
4.9 Weitek Architecture	38
4.9.1 Weitek Data Registers	38
4.9.2 The Weitek Process Context Register	38
4.10 Numeric Exceptions	39
4.10.1 NDP Compilers' Handling of Numeric Exceptions	41
4.11 An Introduction to the IEEE Number System	42
4.11.1 IEEE Representation of Real Numbers	42
4.11.2 Precision and Denormals	45
4.11.3 Infinities and NaNs	46
<b>Mixing Languages</b>	<b>49</b>
5.1 General Rules	49
5.1.1 Linking Restrictions	49
5.1.2 Data Type Differences	50
5.1.3 Naming Conventions	51
5.1.4 Parameter Passing	52
5.1.5 Output Buffers	53
5.2 Calling Between NDP Fortran and NDP C C++	53
5.3 Calling between NDP Fortran and NDP Pascal	55
5.4 Calling between NDP C C++ and NDP Pascal	56
5.5 Interfacing Assembly Language	57
5.5.1 Reasons for Writing Assembly	57
5.5.2 Using the Intelligent Assembler to Optimize Code	58
5.5.3 General Rules	61
<b>Porting Programs</b>	<b>63</b>
6.1 Compatibility with other Compilers	63
6.2 Word-Size Problems	63
6.3 Byte-Order Problems	63
6.4 Alignment Requirements	64
6.5 Floating-Point Range and Accuracy	64
6.6 Assembly Language Interfaces	64
6.7 Expression Evaluation Order	64
6.8 Illegal Assumptions About Optimizations	65
6.8.1 Implied Register Usage	66
6.8.2 Memory Allocation Assumptions	66
6.8.3 -OM and -OLM Considerations	66
6.9 Problems with Source-Level Debuggers	66
6.10 Problems with Compiler Memory Size	66

**ASCII Character Set**

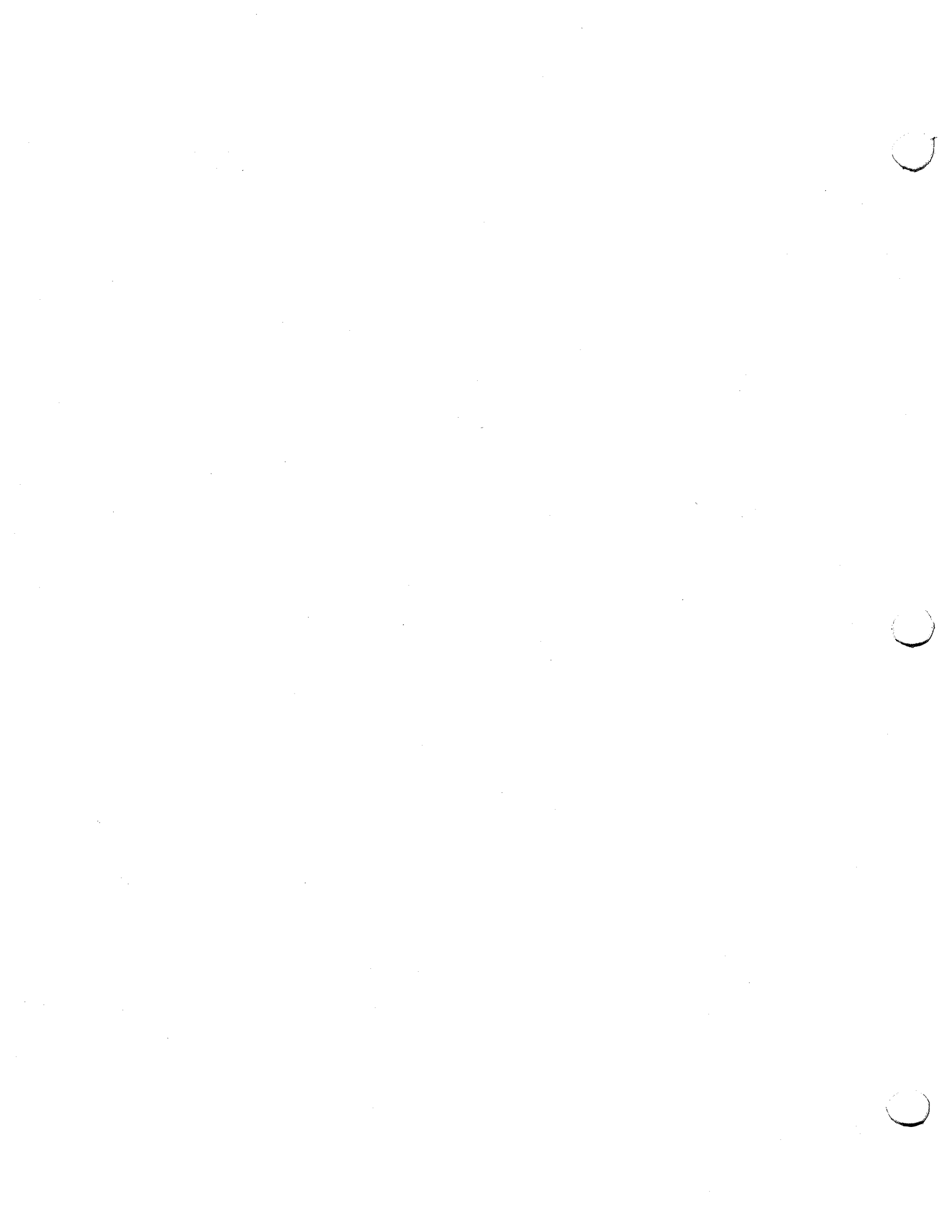
**69**

**Index**

**71**



00  
00



# 1

## Set Up

The NDP-386/486 compiler family is completely hosted on UNIX System V.4. The NDP compiler family uses all features of the native tool chain as supplied by AT&T.

Each compiler is shipped with a compiler driver and necessary runtime libraries. All native UNIX system calls are supported. The NDP compiler supports GUI packages only in UNIX System V Release 4, not Release 3.x.

The compilation process is a matter of compiling, assembling, linking, and loading/running a program.

Should you encounter any problems, please contact Technical Support:

Microway, Inc.  
Box 79  
Kingston, MA 02364  
(508) 746-7341 Voice  
(508) 746-4678 FAX

### 1.1 System Requirements

The NDP 386/486 compilers require the following hardware and software:

- A 386 or 486
- High density 5.25" diskette drive (3.5" disks available on request)
- A hard disk with at least 3 MB of free space.
- Development version of UNIX System V Release 3 or Release 4, specified at purchase time. (NDP Pascal is available for UNIX V.3 only at present.)

### 1.2 Disk Contents

This release comes on two or three diskettes, depending on language. The following files should be available on the diskettes:

1. All compilers: libc.a, libcp.a, libc1167.a, libc1167p.a, libm.a, libmp.a, libm1167p.a
2. Fortran only: mf486, ndpf486, libf.a, libfp.a, libf1167.a, lif1167p.a, hi.f
3. C and C++: assert.h, ctype.h, errno.h, exterr.h, float.h, limits.h, locale.h, math.h, setjmp.h, signal.h, stdarg.h, stddef.h, stdio.h, stdlib.h, string.h, strings.h, time.h, varargs.h, stat.h, times.h, types.h, vm.h
4. C only: mc486, ndpc486, hi.c
5. Pascal only: mp486, ndpp486, libp.a, libpp.a, libp1167.a, libp1167p.a, hi.p

### 1.3 Installation

To install the NDP compiler, follow these steps:

1. Log in as root or make sure you have write permissions to /usr.
2. If the directory /usr/microway doesn't exist, create it:

```
mkdir /usr/microway
```

3. Change to that directory:

```
cd /usr/microway
```

4. Insert the compiler diskette #1 into the floppy drive, then type:

```
cpio -icBduvm</dev/rdsk/f{drive}q{size}dt
```

where *{drive}* is the diskette drive you are using (0 or 1) and *{size}* is the capacity of the diskette (15 for 5.25 inch disks and 18 for 3.5 inch disks). If the message End of Medium on Input appears, insert the next diskette and type:

```
/dev/rdsk/f{drive}q{size}dt
```

5. This step is needed for the 5.25 inch versions only. Move the software to the appropriate directories using the following command(s):

```
Fortran: mv mf486 /usr/bin/mf486
C/C++:  mv mx486 /usr/bin/mx486
Pascal: mv mp486 /usr/bin/mp486
          mv hi.p src
```

### 1.3.1 Environment Variables

Before running the compiler, you should make sure that `/usr/bin` is in your path. This is where the compiler driver lives.

If you prefer to break out the components of the NDP compiler, tools, and libraries, the following environment variables are valid.

```
setenv inc path
setenv lib path
setenv src path
setenv tools path
setenv npd path
```

## 1.4 Testing the Compiler

Change to the `/usr/microway/src` directory and test the compiler using one of the following commands:

```
mf486 hi.f      (Fortran)
mx486 hi.c      (C)
mx486 hi.cxx    (C++)
mp486 hi.p      (Pascal)
```

This produces an executable file named `a.out`. To run this file under UNIX, type:

```
./a.out
```

The message "Hi!" should be displayed.

## 1.5 Troubleshooting

If you cannot get the compiler to produce the executable `a.out`, the trouble is probably in the configuration of the system. That is, the driver cannot find the files it is looking for.

Problems of this type may be pinpointed using the switches `-v` and `-rt2`. Switch `-v` causes the driver to display the commands of each subprocess. It is a good way to uncover problems which relate to the search path used by the compiler and driver. Switch `-rt2` displays the name of each file opened by the compiler.

See Appendix A of the *Language Reference Manual* for a list of errors with their solutions.





# 2

## Using The Compiler

This chapter explains how to compile a program using the compiler driver and its switches. The switches allow you to select compiler optimizations, vary code generation to suit the environment, and specify alternative locations of the library and include files, among other options.

### 2.1 The Compiler Driver

Microway's compiler driver simplifies program development by automatically compiling, optional assembling, linking and loading/running an NDP program with a single command.

The compiler driver executes the NDP-386/486 compiler, assembler, and linker to produce an executable binary file. The driver runs the compiler, which creates an assembly language file from the source file. Next, the driver runs the assembler, which creates an object file from the assembly file. The driver then invokes the linker, passing the appropriate object files, default libraries, linker switches, and any additional information supplied on the compiler driver command line by the user. The linker creates the executable binary file from these modules. Finally, the compiler driver deletes the assembly and object files created; they may be retained using the `-keep` switch. (See *Section 2.3* for related toggles `-c`, `-keepobj`, `-keeps`, and `-s`).

### 2.2 Compiler Driver Syntax

**Syntax:** `driver_name [switches] file(s)`

*driver\_name* is one of the following:

mf486  
mx486  
mp486

*switches* is a list of optional switches separated by spaces. Each switch begins with a minus sign (-). The following sections discuss the switches.

*file(s)* is a list of one or more file names. File names must be separated by spaces; the wildcard characters, question mark (?) and asterisk (\*), may be used.

The compiler driver will accept Fortran, C, C++, Pascal and assembly source code files, object module files, and library files as input. The file extension designates the input file type using the following conventions:

- Filenames with the extension `.f` or `.for` are assumed to be Fortran source programs. Files with the extension `.r` are assumed to be Fortran source files that require the Ratfor preprocessor. Files with the extension `.c`, `.cpp` or `.cxx` are assumed to be C/C++ source programs. Files with the extension `.p` or `.pas` are assumed to be Pascal programs. The driver compiles them into assembly files using the appropriate compiler and leaves them as `.s` files in the current directory.
- Filenames with the extension `.s` are assumed to be assembly source programs. `/bin/as` assembles them into object files with the extension `.o`.
- Filenames with the extension `.o` are assumed to be object programs or object program libraries compatible with the NDP compilers. These filenames are passed to the linker.

- File names with the extension `.a` are assumed to be library files compatible with the NDP compilers. `/bin/ar` creates and manipulates them. These file names are passed to the linker.
- Filenames with no extension, or with an extension other than mentioned above are passed to the linker untouched.

By default, the compiler driver will convert the input file into an executable binary program. The `-S` switch stops the process after the assembly file is created, while the `-c` switch halts the process after the object file is created. For example, the following commands all produce the executable file `a.out`:

```
mf486 test1.f
mx486 test1.cxx
mp486 test1.p
mf486 test1.s
mx486 test1.o
```

### 2.2.1 Example

As an example, assume we have two files, `main.p` (containing the main program) and `sub1.p` (containing some subroutines). We want to compile these programs and produce the executable file `test.out`. The command is:

```
mp486 main.p sub1.p -o test.out
```

The compiler compiles the two source files into assembly files, which are assembled into object files. The driver invokes the linker, passing it the names of the two object files just created, the `-o` switch and its argument, and the names of the default object files and libraries. The linker produces an output file named `test.out`. The output file can be run by typing in:

```
./test.out
```

## 2.3 Compiler Options and Switches

The following table describes the switches used by the compiler driver:

Switch	Description
<code>-2.1</code>	Recognize AT&T 2.1 rules.
<code>-ansi</code>	Assume input program conforms to ANSI standard. (C and Pascal, not C++)
<code>-ansiconform</code>	Create ANSI code but refrain from inlining math functions (C only).
<code>-c</code>	Cause the compiler driver to compile each source file to the object file level only. The driver does not call the linker and produces no executable code.
<code>-cg1</code>	Turn on runtime checking of subranges and array bounds. The code will be much slower under this option.
<code>-cg2</code>	Allocate all variables to memory.
<code>-cg3</code>	Allocate code temporaries to memory.
<code>-cg4</code>	Prepend all variables with an underscore.
<code>-cg5</code>	Output an assembly file with the extension <code>.asm</code> . The default is <code>.S</code> .
<code>-cg6</code>	Do not put an underscore in front of global variables and procedures.
<code>-cg7</code>	Avoid jumps with inline code.

- clink Link libraries appropriate for C source code; used when the context is ambiguous such as `mx486 hi.o`.
- cpplink Link libraries appropriate for C++ source code; see `-clink` above.
- Dname Define the symbol *name* (C|C++ only)
- Dname=text Define *name* to have the value of *text* (C|C++ only)
- f1 Accept characters as unsigned. (Fortran only)
- f2 Turn off the compile-time checking of FORMAT statements. Use this option if your runtime supports FORMAT statement features that NDP Fortran does not recognize. (Fortran only)
- f3 Pad Hollerith constants on right with blanks. The default is that only the first byte of the Hollerith constant is significant and the constant is zero padded on the left. (Fortran only)
- f4 Compile lines starting with x, X, d, or D. The default is to treat them as comments. (Fortran only)
- f5 Do not accept dollar signs (\$) in names. The default allows dollar signs for VMS compatibility. (Fortran only)
- f6 Enable backslash editing, e.g., `\n` represents new line. (Fortran only)
- f7 Local variables are automatic (on stack) by default. (Fortran only)
- fdiv Reduces the amount of time required to divide, at the expense of two bits of precision. It precision is critical, do not use this switch.
- flink Link libraries appropriate for Fortran source code; see `-clink` above.
- g Generate executable code compatible with source level debugger.
- ga Generate frame pointer for stack traces.
- hasm Intermix assembly and source code in the assembly file. You should use `-S` or `-keeps` with this switch to retain the assembly file.
- i2 Make default integer size 2-byte; this results in code that runs more slowly (Fortran only).
- i4 Make default integer size 4-byte; this is the default (Fortran only).
- ident1 Accept but do not output `#identifier` (C|C++).
- ident2 Accept and output `#identifier` (C|C++).
- Idir Search for include file names in the directory *dir* before searching the standard directories.
- lname Causes the compiler driver to direct the linker to search the library named *lname.a*. For example, `-lfft` will add `libfft.a` to the search path.
- list Output a `.LST` file showing all source code including include files with line numbers.
- LIST Like `-list` but omit path names and line numbers.
- minit Allow multiple initializations (Fortran only).
- n0 Produce 80287 code with library calls, not inline transcendentals.
- n1 Produce 80287 code with inline transcendentals, not library calls.
- n2 Produce 80387 code with inline transcendentals, not library calls.
- n3 Advanced Intel coprocessor stack utilization.

- n4           Generate code for the Weitek coprocessor family.
- n5           Generate Microway style Weitek macro instructions in the assembly file. This will not produce executable code.
- n6           Promote no float.
- n7           Use Weitek 3167 multiply accumulate instruction. Requires -n4.
- n8           Use Weitek 3167 square root instruction. Requires -n4.
- nof77        Use Fortran 66 conventions. Also turns on `-onetrip` (Fortran only).
- O            Do all operations.
- o *name*      (Lower-case o.) Cause the compiler driver to place the executable output file into the file *name*. *name* must be preceded by a space and can include an extension. By default, the output file will have the name `a.out`.
- off          Same as `-offfp`, `-offa`, `-offh`, `-offs`, and `-offn`.
- offa         Do not move frequently used procedure and data addresses into registers.
- offcse       Do not do common subexpression elimination (local/global CSE).
- offh         Turn off cross jumping optimization.
- offn         Keep invariant floating point expressions in loops.
- offp         Disable peephole optimizer.
- offr         Force variables to be stored in memory.
- offs         Turn off dead code elimination.
- OL           Optimize the program to be as fast as possible even if it is necessary to make the program bigger. In particular, most of the available resources are allocated to optimizations of the innermost loops. The `-OL` compile time option will perform optimizations that may make the program faster but larger. It is counterproductive to specify `-OL` on code that contains no loops or that is rarely executed as it will make the whole program larger but no faster. You can experiment with a program to discover which modules benefit from `-OL` and which ones do not. The `-x482` option may be used with `-OL` to enable various loop optimizations without turning on loop unrolling. In addition, `-OL` inlines all scalar multiplies, and replaces larger block moves with sequential moves instead of an inline loop.
- OLM          This option is equivalent to `-OL` and `-OM`.
- OM           Add memory optimizations to `-O`.
- on           Same as `-on2cse`, `-onlr`, `-onrc`, `-onrepeep`, `-OLM`.
- on2cse       Do common subexpression elimination (CSE) twice.
- onetrip      Execute at least one iteration of every DO loop. The `-onetrip` switch may be required for successful execution of certain old Fortran programs. (The use of the `-onetrip` option makes the compiler incompatible with the ANSI Fortran 77 standard default of executing no iterations of the DO loop when the lower bound is greater than the upper bound.)
- onlr         Do loop unrolling.
- onrc         Do register caching.

-onrepeep	Repeat peephole optimizations until no further improvement is achieved.
-onw	Emit a warning when dead code is eliminated.
-p	Generate code for profiling.
<del>-plink</del>	<del>Link libraries appropriate for Pascal source code; see -clink above.</del>
-rt1	Output file names are created by appending the appropriate extension to the source file name; requires -s.
-rt2	Display the names of files as they are opened.
-rt3	Continue to compile after a code generator abort or Internal Compiler Error (ICE).
-rt4	Recognize all 80386 library calls instead of inlining routines (e.g., memcmp).
-s	Do not produce object files or executable files, produce only assembly language files. For each source language file specified, compile the source language file into assembly language output. Put the assembly language output into a file with the extension .s.
-u	Make "undefined" the default data type for undeclared variables, as if "implicit undefined (A-Z)" were placed at the top of each routine (Fortran only).
-U	Do not convert upper case user-supplied names in Fortran to lower case. By default, Fortran is not case sensitive and all Fortran names that are externally visible are in the object file in lower case. If one wishes to gain access to names defined in C as upper case, this option can be used. However, use of this option makes the compiler incompatible with the ANSI Fortran 77 standard.
-uname	Undefine the definition of the symbol <i>name</i> (C/C++ only).
-ur=#	where # is 2, 4, 8, 16, 32, or 64. Unrolls a loop # times. This switch MUST be used with -OLM. For example: mp486 -OLM -ur=32 liver.p -o liver
-v	Causes the compiler driver to display the program name and command line arguments of each subprocess it invokes.
-vms	Accept VAX VMS Fortran compatibility over Fortran 77 interpretation. (Fortran only)
-vmsi	Enable VAX extensions for intrinsic functions only (Fortran only).
-W	Make all warnings fatal.
-w	Suppress warning messages.
-Wa, toggle	Pass the specified toggle(s) to the assembler.
-Wl, toggle	Pass the specified toggle(s) to the linker.

-p4

increased compiler set size from 32 to 256



# 3

## Optimizations

### 3.0 Overview

The NDP compilers do many optimizations, several of which are not available in other compilers. These optimizations can reduce the size of a program by 30 percent and increase its speed by a factor of up to four. The compiler does typical optimizations such as constant folding, strength reduction in simple operations and loops, code hoisting between blocks, movement of invariant expressions out of loops, and conversion of multiplications and divides into shifts and additions when advantageous. The compiler also eliminates: redundant jumps, unreachable code, and never-referenced variables and expressions. The key to the effectiveness of the optimizations is a global data-flow analysis that maximizes the use of CPU registers and numeric coprocessor registers for the storage of variables. The compiler does the following optimizations:

- Memory Allocation
- Register Allocation
- Static Address Elimination
- Register Coalescing
- Prolog and Epilog Code Optimizations
- Passing Parameters in Registers
- Various Peephole Optimizations
- Speed Optimizations
- Loop Rotation
- Loop Invariant Analysis
- Strength Reduction
- Dead Code Elimination
- In-line Multiplication and Division
- Constant Propagation
- Constant Expression Folding
- Local and Global Common Subexpression Elimination (CSE)
- Live/Dead Analysis
- Cross Jumping
- Loop Unrolling
- Inliner

The NDP compilation process is made up of three phases: the front end, the intermediate optimizer, and the back end.

The front end phase translates each procedure into an idealized internal representation, including a directed flow graph (DFG), for each of the semantic operations allowed by the language. This first phase, which includes lexical analysis and parsing into DFG's, also does several optimizations such as strength reduction.

The DFG is then passed to the intermediate optimizer where optimizations are done on each DFG before code is generated. Each node of the DFG represents a block of code that has one entry and one exit point.

The code generator (back end phase) takes optimized DFG as input and maps it onto the operations of an ideal 32 bit processor that has an infinite number of virtual registers. The register allocation is done by a register coloring algorithm that coalesces and maps the virtual registers into the 80386 or 80486 and Intel or Weitek coprocessor registers. Optimizations such as instruction scheduling and peephole optimizations occur during the code generation phase.



Most optimizations can be turned on by supplying the compiler with the `-on` option.

```
mf486 -on dwhet.f
```

The rest of the chapter provides a brief description of each optimization during the compilation process.

### 3.1 Memory Allocation

The NDP compilers allocate variables based on their size, frequency of use, and other attributes. Variables that are never used usually are not allocated. Variables normally are sorted to allocate the smaller and more often used variables first, and the larger and less often used variables later.

### 3.2 Register Allocation by Coloring

Register allocation is used to keep the most commonly used values always in registers. The entire function or subroutine is examined to determine which local variables and parameters are used most often. The most commonly used variables and parameters are allocated to machine registers. No memory is allocated for them. This optimization provides the most significant savings in execution speed and the largest reduction in program size. Referencing a variable in a register takes about one third of the space and time required for referencing a variable in memory.

All local variables of the main program, or any function, are candidates for allocation to a register unless they are passed to a function. The register allocator uses the global data-flow analysis to find the lifetime of each variable. This information makes it possible to increase the number of variables that can be stored in registers by using the same register for variables that do not overlap in the same procedure. Two variables may be allocated to the same register if there is no place in the program in which both variables hold a value that will be used later. Usually, all local variables stay in registers if possible. When register space is exceeded, the compiler creates procedures which set up and use the stack for local storage.

**Programmer Hint:** Procedures that are coded so that they do not require storage on the stack have reduced calling overhead and run faster. Also, variables that are passed as parameters to procedures have to stay in memory. If you have to pass a "hot" variable to a procedure, then (1) assign it to a dummy and pass the dummy or (2) use a dummy in the innermost loop of the calculation and assign it outside of the loop. I/O statements behave like procedures; it is better to print duplicates of important variables than the variables themselves. See the example in *Section 3.18, Loop Unrolling*, page 22.

Register allocation may be turned off with the `-cg2` compiler switch. This may be necessary for specialized programs that do not want variables stored in registers, but rather require that variables reference physical memory.

All variables that are candidates for register storage will be allocated to the available registers to give either the fastest or densest code possible (as controlled by the `-OL` compile time option). Most compilers allocate all local variables in memory. The NDP compilers will allocate as many local variables to registers as it can. The quality of the code generated with this allocation scheme is exceptional.

In the following example, The NDP compiler allocates I and J to the same register because their lifetimes do not overlap. The translation is one-to-one from source code into assembly language.

**NDP Fortran Code**

```

SUBROUTINE PROC
INTEGER I,J
I = 1
1 CALL F
I = I + 1
IF (I.LT.10) GOTO 1
J = 1
2 CALL G
J = J + 1
IF (J.LT.10) GOTO 2
END

```

**NDP C/C++ Code**

```

proc() {
  int i,j;
  i = 1;
  do {
    f();
    ++i;
  } while (i < 10);
  j = 1;
  do {
    g();
    ++j;
  } while (j < 10);
}

```

**NDP Pascal Code**

```

procedure f; external;
procedure g; external;
procedure proc;
var
  i, j: integer;
begin
  i := 1;
  repeat
    f;
    i := i+1;
  until (i>=10);
  repeat
    g;
    j := j + 1;
  until (j>=10)
end;

```

**386 Assembly Language**

```

mov ebx, 1 ;i=1, uses ebx
align 4
L7:
call _f_
inc ebx ;i=i+1
cmp ebx, 10 ;i.lt.10
jge L6 short
jmp L7 short
align 4
L6:
mov ebx, 1 ;j=1, also uses ebx
align 4
L5:
call _g_
inc ebx ;j=j+1
cmp ebx, 10 ;j=lt.10
jge L4 short
jmp L5 short
align 4

```

**3.3 Static Address Elimination**

A valuable optimization the NDP compilers do is to store frequently used static addresses in registers. Since the static addresses are 4 bytes long, if a static address is used just twice in a function, it is faster and smaller to load the address into a register at the beginning of the procedure or function and always use register indirect addressing to access it. In this way, most static references shrink to one third of the space and use less execution time. For example:

**NDP Fortran**

```

SUBROUTINE P
COMMON /X/X
INTEGER X
DO 10 I=1,9
10 X = X + I
RETURN
END

```

**NDP C/C++**

```

p() {
  static int x;
  int i;
  for (i=1;i<10;i++)
    x=x+i;
}

```

**NDP Pascal**

```

procedure p;
var
  i, j: integer;
begin
  for i := 1 to 10 do
    x := x+1
end;

```

### 386 Assembly Language

```

mov eax, 9                ;i=1,9
lea ecx,dword ptr ds:_X_ ;static address elimination,
align 4                   ;address of common |x| to register
L23:
inc dword ptr [ecx]       ;x=x+1
dec eax                   ;decrement number of loops left
jne L23 short
; .ef:
;

```

The improvements by the NDP optimizer can be summarized as:

- Static Address Elimination:                   2 instructions per iteration
- No frame pointer:                            3 instructions
- Instruction Scheduling:                    2 instructions and 1 gap per iteration

Static address elimination also plays an important role in loop unrolling of array subscripts (see *Section 3.18, Loop Unrolling*, page 22). In this case, the addresses of sequential elements in an unrolled loop are computed by using indexed addressing along with a statically stored base address that changes once per loop.

## 3.4 Register Coalescing

Register coalescing organizes the computation of expressions to ensure that values end up in the registers where they will be needed. This eliminates shuffling the values in registers to set them up as needed. Most microprocessor compilers will copy the arguments of a computation into scratch registers, do the computation in the scratch registers, and then copy the result to the destination. The NDP compilers use the destination register in the computation to save unnecessary copies of the source registers into scratch registers.

For example:

NDP Fortran	NDP C/C++	NDP Pascal
<pre> SUBROUTINE p INTEGER i, j, k i = 1 j = 1 DO k = 1,10   i = i*4+j END DO END </pre>	<pre> p() { int i,j,k;   i=1;   j=1;   for (k=1;k&lt;= 10;k++)     i=i*4+j; } </pre>	<pre> procedure p; var i,j,k: integer; begin   i := 1;   j := 1;   for k := 1 to 10 do     i := i * 4 + j   end; end; </pre>

### 386 Assembly Language:

```

mov eax, 1                ;i=1
mov ebx, 1                ;j=1
mov ecx, 10               ;number of loops=10
align 4
L21:
lea eax,[ebx][eax*4];i=i*4+j
dec ecx                   ;decrement number of loops

```

The instruction `lea eax, [ebx][eax*4]` does the computation and stores its result directly in the correct register, representing `i`, rather than in a scratch register, demonstrating register coalescing. The most interesting optimization in the code is the way `i=i*4+j` is coded. `eax` represents `i`; `ebx` represents `j`. There is an instruction that multiplies two numbers and adds

a third, but this can be done with addresses, commonly when arrays are involved. In the instruction

```
lea eax, [ebx][eax*4]
```

`eax` and `ebx` are treated as if they were addresses but those "addresses" are just the values of `i` and `j` is `i*4`, `[ebx][eax*4]` is `j+i*4`. The result is stored in `eax`, which is `i`.

### 3.5 Prolog and Epilog Code Optimization

Most compilers use a frame pointer register in each function. The frame pointer is used to access local variables, to point up the call stack to allow stack traces to be printed during debugging, and to unwind the stack for an exception mechanism. The frame pointer is valuable but it is usually not necessary. The NDP compilers generate a frame pointer only if adding the frame pointer will not expand the code. Otherwise, they do not set up a frame pointer in each function. Instead of creating a frame pointer, the NDP compiler accesses all local variables by using the stack pointer.

If it is necessary to have a frame pointer in every function, the `-ga` compile time option can be specified on the compiler driver command line. This compile time option guarantees that there will always be a frame pointer, but it increases the size of the program.

If a function is very short, the entry and exit code may take a large fraction of the space and execution time of the procedure or function. If, as a result of global optimization, the parameters and local variables of a function are allocated in registers (usually true for a procedure or function of 20 lines or less), the compiler can often eliminate the procedure entry and exit code entirely. The bottom line here is that the code is even better than handwritten code because it is often not practical for an assembly language programmer to keep track of all register contents.

Default Code	With <code>-ga</code> option
<code>_P_ proc near</code>	<code>_P_ proc near</code>
<code>  push dword ptr ds:L8</code>	<code>  push ebp ;extra code</code>
<code>  call _f_</code>	<code>  push dword ptr offset ds:L8</code>
<code>  pop ecx</code>	<code>  call _f_</code>
<code>  ret</code>	<code>  pop ecx</code>
	<code>leave       ;extra code</code>
	<code>  ret</code>
	<code>align 4</code>

### 3.6 Peephole Optimizations

Peephole optimizations are local improvements to the code that are certain to be correct without further analysis of the surrounding code. An example would be a move from one register to another, followed by a move in the reverse direction. In this case, one of the move instructions is unnecessary and may be removed. All the peephole optimizations that have been implemented are safe for the target environments. If there is any reason to suppress these optimizations, use the `-offp` compile time option. In some situations, code can be improved by repeating the peephole optimization phase. The `-onrepeep` switch repeats peephole optimization until no further improvements can be made. It is not on by default because it causes the flow graph to be traversed a second time, which is expensive in terms of compilation speed.

### 3.7 Speed Optimizations

The `-OL` compile time option selects the speed optimizations. This improves the speed of the program, but usually at the cost of making the program larger.

To increase the speed of a program, it is necessary to identify which instructions are executed most often and concentrate the optimizations in these areas. Computer languages have two main constructs for repeating the execution of instructions: loops and procedures. By making specific optimizations for each of these constructs, it is possible to improve the performance of most programs significantly.

The `-OL` compile-time option should be used only on modules in which most processing occurs in loops. If `-O` is specified, the compiler does everything it can to reduce program size. The `-OL` compile-time option will sacrifice program size to increase the performance of loops. It will allocate nearly all the registers to the variables and temporaries used in the innermost loop of a procedure or function. This will prohibit them from being used for variables that are used elsewhere. If `-OL` is specified on a main program, the compiler could do much work to optimize many loops that are rarely executed. This would result in a program getting larger, but not very much faster. The `-OL` switch also invokes loop unrolling (see *Section 3.18*, page 22).

### 3.8 Loop Rotation

Many compilers generate a termination test at the top of the loop and an unconditional branch from the bottom of the loop to the top of the loop. The loop will execute two branch instructions on each iteration of the loop.

A better way to generate code for loops is to place the test at the bottom of the loop. This is called "loop rotation." If it can be determined at compile time that the loop will always execute at least once, then the loop is entered from the top. If it cannot be determined that the loop will be executed at least once, then an unconditional branch to the termination test is placed before the loop entry. With the test at the bottom, only one branch is executed on each iteration of the loop.

For example:

<b>NDP Fortran</b>	<b>NDP C/C++</b>	<b>NDP Pascal</b>
<pre> SUBROUTINE P INTEGER X,I DO 10 I=1,9   X = X + 1 10 CONTINUE RETURN END </pre>	<pre> p() {   int x,i;   for (i=1;i&lt;10;i++)     x = x+1; } </pre>	<pre> procedure p; var   x,i: integer; begin   for i := 1 to 9 do     x := x + 1   end; end; </pre>

#### 386 Assembly language

```

mov eax, 1      ;x=1
mov ecx, 9      ;number of loops=9
align 4
L23:
add eax,eax    ;x=x+x
dec ecx        ;decrement number of loops
jne L23 short  ;test and branch moved to bottom of loop

```

### 3.9 Loop Invariant Analysis

Loop invariant analysis is used to speed up loops. Each loop is examined for expressions and address calculations that do not change in the loop. These computations are moved out of the loop and the value is stored in a register. This optimization is particularly valuable for removing array subscripts from a loop when the subscripts are variables or expressions that are not modified in the loop. In a small loop, all invariant expressions will be accessed with register mode and all invariant addresses will be accessed with register indirect modes. This optimization usually eliminates all computations of invariant expressions and addresses in loops.

**NDP Fortran**

```

SUBROUTINE LOOP
INTEGER i,j,k
i=1
j=7
k=1
DO WHILE (I.LT.j-2)
  i=i+1
  k=k+k
END DO
END

```

**NDP C/C++**

```

loop () {
  int i,j,k;
  i = 1;
  j = 7;
  k = 1;
  while (i<j-2) {
    i=i+1;
    k=k+k;
  }
}

```

**NDP Pascal**

```

procedure loop;
var i,j,k: integer;
begin
  i := 1;
  j := 7;
  k := 1;
  while (i<j-2) do begin
    i := i+1;
    k := k+k
  end
end;

```

**386 Assembly Language**

```

mov eax, 1 ;i=1
mov ecx, 1 ;k=1
mov ebx, 5 ;The invariant j-2 --> constant 5
cmp eax,ebx
jge L19 short

```

In the source example, *j* is not changed in the loop, but *j-2* would need to be recomputed for each loop iteration. This loop invariant is optimized by doing the computation once outside the loop, and using the result in the loop at each iteration.

**3.10 Strength Reduction**

A reduction in strength occurs when a less expensive operation (in terms of execution size or speed) replaces a more expensive one, as happens when a multiplication replaces an exponentiation (e.g., becomes  $x \cdot x$ ), or an addition replaces a multiplication (e.g., becomes  $x+x$ ).

Most compilers do simple strength reductions such as the conversion of multiplies and divides into shifts. However, only the most advanced compilers do strength reductions on loop indices. The NDP compilers do a strength reduction on loops that have an index variable that is incremented by a constant on each iteration of the loop (such as a FOR loop). When a loop index variable is used as the subscript for an array, most compilers will multiply the loop index by the size of the array elements and add this offset to the base of the array. Each such reference requires at least three instructions. In the NDP compilers, a register outside of the loop is loaded with the address of the array element to be accessed on the first iteration of the loop. The array access is then done using the indirect register addressing mode. On each iteration, the element size is added to the register so that it contains the address of the element to be accessed on the next iteration of the loop. The reduction in strength involves substituting an addition of a constant to a register for a multiplication of the loop index by a loop invariant value. This optimization results in a four to ten fold increase in speed.

Strength reduction and loop invariant analysis involving array subscripts are particularly important to Fortran programmers, for whom repetitive array indexing in DO loops is common. NDP Fortran does the strength reductions and loop invariant analyses that many mainframe programmers have come to expect.

**NDP Fortran**

```

SUBROUTINE MATMUL(A,B,C)
REAL A(100,100), B(100,100), C(100,100)
DO 10 I = 1, 100
DO 10 J = 1, 100
  A(I,J) = 0
DO 10 K = 1, 100
  10 A(I,J) = A(I,J) + B(I,K) * C(K,J)

```

```
RETURN
END
```

**NDP C/C++**

```
matmul(float a[100][100], float b[100][100], float c[100][100] {
    int i,j,k;.
    for (i=0; i<100; i++) {
        for (j=0; j<100; j++) {
            a[i][j] = 0;
            for (k=0; k<100; k++)
                a[i][j]=a[i][j]+b[i][k]*c[j][k];
        }
    }
}
```

**NDP Pascal**

```
type
    matrix=array[1..100, 1..100] of float;
procedure matmul(a,b,c:matrix);
var
    i,j,k: integer;
begin
    for i := 1 to 100 do
        for j := 1 to 100 do begin
            a[i,j] := 0;
            for k := 1 to 100 do
                a[i,j] := a[i,j]+b[i,k]*c[k,j]
            end
        end
    end;
end;
```

**Matrix Multiply inner loop comparison in 386 Assembly Language:**

```
L68:
    mov esi,[ecx]+(-4)      ;Load C(K,J)
    imul esi,[eax]+(-400)  ;B(I,K)*C(K,J)
    add [ebx],esi          ;A(I+J)=A(I+J)+B(I+K)+C(K+J)
    mov esi,[ecx]          ;Repeat for K+1
    imul esi,[eax]
    add [ebx],esi
    mov esi,[ecx]+4        ;Repeat for K+2
    imul esi,[eax]+400
    add [ebx],esi
    mov esi,[ecx]+8
    imul esi,[eax]+800    ;Repeat for K+3
    add [ebx],esi
    add eax,1600           ;Increment B to next address
    add ecx,16             ;Increment C to next address
    dec edi
    jne L68 short
```

The strength reduction shows up in the assembly code, in the instructions:

```
add eax, 1600
add ecx, 16
```

Instead of multiplying an index by the size of the array at each step, an addition to the current location is made.

Also note the loop unrolling optimization (See *Section 3.18 Loop Unrolling*, page 22). With a loop unrolling factor of 4, 25 iterations of 4 merged loops takes the place of 100 iterations, increasing performance by eliminating the number of times the loop control code is executed.

### 3.11 Dead Code Elimination

The NDP compilers will eliminate any block of code that has no predecessor block in the DFG (i.e., there exists no path to the block) or any sequence of code that is not reachable (i.e., code following a return statement).

Other optimizations may expose potential Dead Code Eliminations that would otherwise be inane. Consider that constant propagation may convert a conditional jump into an unconditional jump. This would eliminate a path out of the associated block creating dead code. Consider the following example:

NDP Fortran	NDP C/C++	NDP Pascal
<pre> SUBROUTINE P INTEGER X,Y X = 1 Y = 1 IF ( 0 ) THEN   X = X + 1   Y = Y + 1 ENDIF Y = Y + 1 RETURN X = X + 1 END </pre>	<pre> p() {   int x,y;   x = y = 1;   if (0) {     x=x+1;     y=y+1;   }   y=y+1;   return ();   x=x+1; } </pre>	<pre> procedure p; var   x, y: integer; begin   x := 1;   y := 1;   if (0=1) then begin     x := x+1;     y := y+1   end;   y := y+1 end; </pre>

There are two opportunities here for dead code elimination. The first is the IF statement and its enclosed body; the second is the code ( $x=x+1$ ) following the RETURN statement. These pieces of code will never be executed, and disappear from the generated code.

### 3.12 Inline Multiplication and Division

Since the instructions to do an integer multiplication potentially take nine cycles to execute, it is often faster to do constant multiplies by a series of shifts and adds (or subtracts). For instance, a multiply by four is a shift left by two, multiplying by five is a shift left by two followed by an add, and multiplying by seven is a shift left by three followed by a subtract. For an example of this, see the code under *Section 3.4, Register Coalescing*, page 14.

Integer division is much worse; it takes about 60 cycles to do a divide. When dividing a constant the compiler can calculate a (floating-point) reciprocal at compile time and convert the divide into a floating-point multiply which only takes about 15 cycles. In certain rare cases when using 16-bit integers, the compiler can do a divide using an integer multiply and a shift.

Floating-point division can often be accelerated by calculating a reciprocal either in the compiler (if division is by a constant), or at the head of a loop if the divisor is a loop invariant.

Consider the following example:

NDP Fortran	NDP C/C++	NDP Pascal
<pre> SUBROUTINE SHIFTS INTEGER i,j i=i*4 j=j*8 END </pre>	<pre> shifts() {   int i,j;   i=i*4;   j=j*8; } </pre>	<pre> procedure shifts; var i,j: integer; begin   i := i*4;   j := j*8 end; </pre>



The compiler coded the two multiplications in the source file,  $i*4$  and  $j*8$ , as shifts in the assembly file:

```
sal dword ptr [eax], byte ptr 2
sal dword ptr [ecx], byte ptr 4
```

### 3.13 Constant Propagation

The NDP compilers will back-substitute any variable  $V$  with the constant  $C$ , if  $C$  was the last value assigned to  $V$ . More simply, the compiler analyzes variable assignments and determines if they can be propagated to constant assignments.

Consider the following code in which 1 is back-substituted in the IF statement:

NDP Fortran	NDP C/C++	NDP Pascal
INTEGER DEBUG	int debug;	debug: integer;
...	...	...
DEBUG = 1	debug = 1;	debug := 1;
...	...	...
IF (DEBUG.EQ.1) THEN	if (debug==1) then {	if (debug == 1) then begin
...	...	...
ENDIF	}	end;

Also consider:

NDP Fortran	NDP C/C++	NDP Pascal
FLAG = 1	flag = 1;	flag := 1;
SAVE_FLAG = FLAG	save_flag = flag;	save_flag := flag;

Instead of assigning FLAG as SAVE\_FLAG, the constant 1 can be assigned to SAVE\_FLAG.

### 3.14 Constant Expression Folding

#### NDP Fortran

```
SUBROUTINE CONSTANTS (A)
REAL*4 A,PI,RADIUS
PARAMETER(PI=3.14,RADIUS=4.2)
A = PI * RADIUS**2
END
```

[NDP C/C++ doesn't have "constants" per se]

#### NDP Pascal

```
procedure constants (a: float);
const
pi = 3.14;
radius = 4.2;
var
a: float;
begin
a := pi*radius*radius;
end;
```

#### 386 Assembly Language

```
; .bf:
mov eax,[esp]+4
mov dword ptr [eax],1113427699 ;pi*radius**2 computed at compile time
; .ef: ;
ret
```

### 3.15 Common Subexpression Elimination (CSE)

NDP compilers will eliminate common subexpressions across the DFG. In addition, Global CSE also keeps track of copy propagation such that, in the following example, expressions A+B and C+D would be recognized as redundant:

NDP Fortran	NDP C++	NDP Pascal
INTEGER A, B, C, D, E, F, G, H	int a, b, c, d, e, f, g, h	a, b, c, d, e, f, g, h: integer;
...	...	...
E = A+B+G	e = a+b+g;	e := a+b+g;
D = A	d = a;	d := a;
C = B	c = b;	e := b;
B = G	b = g;	b := g;
F = C+D+H	f = c+d+h;	f := c+d+h;

Since A+B+G and C+D+H are redundant, C+D+H would not have to be calculated. Only the value of E would have to be assigned to F.

### 3.16 Live/Dead Analysis

Computations whose results are never used are eliminated. The NDP compilers also eliminate dead stores. This is an extension of common subexpression elimination. Consider:

NDP Fortran	NDP C++	NDP Pascal
INTEGER X, Y	int x, y;	y, y: integer;
...	...	...
X = 1	x=1;	x := 1;
Y = 2	y=2;	y := 2;
X = Y + Z	x=y+z;	x := y+z;

Since X is never referenced between the two assignments, the first assignment of X may be eliminated.

### 3.17 Cross Jumping (i.e., Tail Merging / Code Hoisting)

Two or more nodes in the DFG that end in the same sequence of code that have a common successor node are reorganized to eliminate the redundant code.

Cross jumping is also commonly called tail merging because tails of nodes are merged. Less commonly, it is called code hoisting, for code is hoisted from one node to another.

Consider:

NDP Fortran	NDP C++	NDP Pascal
IF (AB .EQ. 1) THEN	if (ab==1) {	if (ab = 1) then begin
A = A+1	a = a+1;	a := a+1;
B = B+1	b = b+1;	b := b+1;
CALL FOO	foo();	foo;
ELSE	} else {	end else begin
B = B+1	b = b+1;	b := b+1;
CALL FOO	foo();	foo
ENDIF	}	end;

is reorganized to:

NDP Fortran	NDP C++	NDP Pascal
IF (AB.EQ.1) THEN	if (ab == 1) a=a+1;	if (ab=1) then a:=a+1;
a=a+1	b = b+1;	b := b+1;
end if	foo();	foo;
B = B+1		
CALL FOO		

In some instances, entire nodes may be eliminated. Consider:

<b>NDP Fortran</b>	<b>NDP C++</b>	<b>NDP Pascal</b>
<pre>IF (AB .EQ. 1) THEN   A = A+1   B = B+1   CALL FOO ELSE   A = A+1   B = B+1   CALL FOO ENDIF</pre>	<pre>if (ab==1) {   a = a+1;   b = b+1;   foo(); } else {   a = a+1;   b = b+1;   foo(); }</pre>	<pre>if (ab=1) then begin   a := a+1;   b := b+1;   foo; end else begin   a := a+1;   b := b+1;   foo; end;</pre>

is reorganized to:

<b>NDP Fortran</b>	<b>NDP C++</b>	<b>NDP Pascal</b>
<pre>A = A+1 B = B+1 CALL FOO</pre>	<pre>a = a+1; b = b+1; foo();</pre>	<pre>a := a+1; b := b+1; foo;</pre>

### 3.18 Loop Unrolling

If loop invariants (indices) can be determined at compile time, it may be advantageous to duplicate (unroll) the body of a loop  $N$  times rather than making  $N$  iterations through the loop. Execution speed increases because the looping mechanism overhead is eliminated, but at the cost of increased code size. NDP compilers can unroll loops up to 8 times. However, unrolling large loops may adversely affect register coloring because more contention is added to the DFG. Also, loop unrolling can generate difficult code to debug.

Loop unrolling is effective because it eliminates the need to execute certain pieces of code with each iteration. A loop contains two kinds of code, the body that does the work and the loop control code that determines whether another iteration of the loop is needed. By determining the number of iterations at the beginning and by unrolling the loop, it is not necessary to execute the loop control code with each iteration.

Loop unrolling is done by merging several loop bodies into a new body that does the work of the merged bodies. This new larger loop requires a single piece of loop control code. The number of loop bodies that are combined inside the merged body of the unrolled loop is the unrolling factor. The NDP compilers use an unrolling factor of four. When a loop is unrolled by a factor of 4, four bodies and four pieces of loop control code merge into four bodies with a single piece of loop control code.

If a loop contains more than 4 iterations, a new loop is created that iterates once for each merged loop. If a loop contains 100 iterations, a new loop of 25 iterations will process the merged loop that has unrolled 4 loops. If a loop contains 103 iterations, the first 100 will be processed as above, and the last 3 iterations will be processed separately. This last group of 3 iterations is often called the cleanup code.

There are several expenses related to loop unrolling. The size of the code can increase by as much as a factor of five in size for unrolled loops. Another expense is the data dependency analysis the compiler must do to avoid aliasing of cached variables. This is particularly costly for nested loops, and may slow compilation time by as much as a factor of ten.

Because of the costs associated with loop unrolling, the compiler invokes heuristics, which only unroll loops for which the payoff is significant. If we assume that the body takes as long to execute as the end, loop unrolling can give you an improvement of:

$$2T_{\text{end}} * U / (T_{\text{end}}(U+1)) = 2U / (U+1)$$

where  $T_{\text{end}}$  is the time to execute the end code, or loop control code, and  $U$  is the unrolling factor. The maximum improvement available approaches 2 as  $U$  increases. With  $U$  set to 4, as in the NDP compilers, the improvement is 1.6, or 80% of the maximum.

As the size of the loop body increases in relation to the size of the loop control code, the improvement available by loop unrolling decreases. The NDP compilers use a heuristic that prevents loop unrolling when the size of the loop body is more than three times the size of the loop control code.

Unrolling is less effective when the body takes significantly longer than the loop control code to execute. It is also less effective when the body contains floating point instructions, particularly if they are towards the end. The Intel coprocessors may take up to 5 times as long to execute a floating point instruction as the CPU does to execute an integer instruction. The floating point instruction execution can overlap the execution of integer instructions. If the floating point instruction is the last instruction in a loop body, the loop control end code can execute while the floating point instruction is still executing. Here, eliminating the end code will not result in a performance improvement. With a Weitek coprocessor, this overlap does not occur, and loop unrolling can lead to significant improvements. See *Section 3.10*, page 17 for an example of loop unrolling.

### 3.19 Inliner

Microway's Inliner is a general purpose utility that replaces a call to a function with the logic contained inside that function. This avoids the overhead of prolog and epilog code inside the function, register saving and restoring, with the potential cost of increased code size. Inlining of functions can increase the potential for other optimizations also, such as loop optimizations (if the call is in a loop), copy propagation, constant folding, and instruction scheduling. Inlining is considered an optimization because generally its correct use increases execution speed of the resultant module.

Microway's Inliner is available for all NDP languages. Syntactically, the inliner works differently for each of the NDP languages. In C++, the type qualifier `inline` designates that a function is to be inlined. In C and Pascal, the type qualifier `-Inline` designates that a function is to be inlined. In Fortran, command line options are used. Command line options also may be used as well as the respective type qualifiers for C, C++, and Pascal. (See *Section 2.3*, page 6 for switches for using the inliner.)

There are various issues that must be addressed when considering the inlinability of a function:

- Size vs. Frequency of Use
- Recursiveness
- Definition of Function is Exported/Imported/Static
- Address of Function Taken
- Is it a Nested Function? (in Pascal)

#### 3.19.1 Size vs. Frequency of Use

Small frequently used functions are the best candidates for inlining. The cost of register saving and restoring, and prolog and epilog code execution, must be weighed against the time spent in the body of the function. It is the programmer's responsibility to choose wisely which functions are to be inlined. A good profiler may aid in this decision process.

#### 3.19.2 Recursion

Functions that are recursive are not inlined.

### 3.19.3 Definition of Function is Exported/Imported/Static

Functions that have their definition exported may be inlined, but must still be defined. Functions that have their definition imported may be inlined. Functions that are static may be inlined and the definition may be discarded.

A general problem with inlining is that a function definition is not available at compile time. Consider this problem in C/C++:

```
#include <string.h>
...
foo() {
    memcpy(s1,s2,size);
}
```

`memcpy()` is declared as external in `string.h` and its definition is not seen until link time. Any library routine or routine whose definition is imported cannot be traditionally inlined. Seeing this limitation, Microway has provided a general solution.

A library archive file is created when inlining is activated with `-inline`. Routines whose definitions are imported can be inlined because the inlining database stores each routine in a library (`.inl`) file and retrieves it as required as compilation progresses.

### 3.19.4 Address of Function Taken

Functions whose addresses are taken may be inlined. The definition may not be discarded, so an address is still available.

### 3.19.5 Nested Functions

Only Pascal may contain nested procedures. As a rule, nested procedures that are contained within procedures that are inlined, are inlined as well. In the instance that the nested routine must be defined, it is cloned, and multiple copies of the cloned procedure may exist.

# 4

## Runtime Organization and Numerics

This chapter details runtime organization and numerics as they relate to the NDP 386/486 compilers.

### 4.1 Lower Level Characteristics

The Intel 80386/80486 memory is byte addressed with 32-bit addresses. Bytes are ordered with the least significant byte of a multiple byte value stored at the lowest address (little endian), the opposite of the IBM 370. Bits are numbered with bit zero as the least significant bit.

Floating point values are stored in IEEE 854 format (32- and 64-bits), with the least significant byte at the lowest address. Character encodings are ASCII. The use of IEEE 854 and IEEE 754 are interchangeable in this discussion.

### 4.2 Integer Data Type

An 80x86 integer is a 32-bit signed value in two's complement form. *Figure 4-1* lists two character and six integer types and their values.

Type	Range
signed character	-128 to 127
unsigned character	0 to 255
short integer	-32,768 to 32,767
unsigned short integer	0 to 65,535
integer	-2,147,483,648 to 2,147,438,647
unsigned integer	0 to 4,294,967,296
long integer	-2,147,483,648 to 2,147,428,647
unsigned long integer	0 to 4,294,967,296

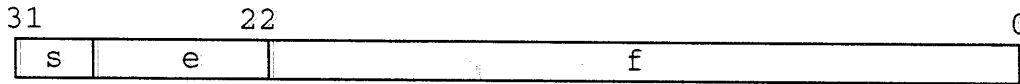
Figure 4-1. Character and Integer Types

### 4.3 Single Precision Real

A single precision real data type is a 32-bit binary floating-point number. Bit 31 is the sign bit (*s* in the example); bits 30..23 are the exponent (*e*); and bits 22..0 are the fractional part (*f*). The values for single precision real are IEEE-854-Std conformant; they obey the following rules:

1. If  $e = 0$  and  $f \neq 0$ , or  $e = 255$ , then a floating-point source-exception is generated.
2. If  $0 < e < 255$ , then the value is  $-1s * 1.f * 2^{e-127}$ . (The exponent adjustment 127 is called the bias.)
3. If  $e = 0$  and  $f = 0$ , then the value is signed zero.

See *Figure 4-2* for the special values of NaN's, INF's, and other anomalies.



- s = Sign of Fraction (1 bit)
- e = Biased Exponent (8 bits)
- f = Fraction (23 bits)

#### Single Precision Real

### 4.4 Double Precision Real

A double precision real data type is a 64-bit binary floating-point number. Bit 63 is the sign bit; bits 62..52 are the exponent; and bits 51..0 are the fraction. A double precision value occupies an even/odd pair of floating-point registers. Bits 31..0 are stored in the even-numbered floating-point register; bits 63..32 are stored in the next higher odd-numbered floating-point register. The values for double precision real are IEEE-854-Std conformant:

1. If  $e = 0$  and  $f \neq 0$  or  $e = 2047$ , then a floating-point source-exception is generated.
2. If  $0 < e < 2047$ , then the value is  $-1s * 1.f * 2^{e-1023}$ . (The exponent adjustment 1023 is called the bias.)
3. If  $e = 0$  and  $f = 0$ , then the value is signed zero.



- s = Sign of Fraction (1 bit)
- e = Biased Exponent (11 bits)
- f = Fraction (52 bits)

#### Double Precision Real

## 4.5 Single and Double Real Encodings

Figure 4-2 shows both single and double real encodings.

		Class	Sign	Biased Exponent	Significand ff-ff*	
P O S I T I V E	N a	Quiet	0	11..11	11..11	
			0	11::11	11::11	
	N	Signalling	0	11..11	01..11	
			0	11::11	00::01	
			Infinity	0	11..11	00..00
	R E A L	N o r m a l s	Normals	0	11..10	11..11
				0	00::01	00::00
		D e n o r m a l s	Denormals	0	00..00	11..11
				0	00::00	00::01
				Zero	0	00..00
		Zero	1	00..00	00..00	
N E G A T I V E	D e n o r m a l s	Denormals	1	00..00	00..01	
			1	00::00	11::11	
	N o r m a l s	Normals	1	00..01	00..00	
			1	11::10	11::11	
			Infinity	1	11..11	00..00
	S i g n a l l i n g	S i g n a l l i n g	Signalling	1	11..11	00..01
				1	11::11	01::11
		Q u i e t	Quiet	1	11..11	10..00
				1	11::11	11::11

The integer bit of the significand is implied and not stored. For single precision, the biased exponent is 8 bits and the significand is 23 bits; for double precision, the biased exponent is 11 bits and the significand is 52 bits.

Figure 4-2. Real Encodings



## 4.6 Language Data Types

Figure 4-3a shows the encodings for Fortran's data types.

Data Type	Size (Bits)	Alignment
BYTE	8	8
CHARACTER*1	8	8
CHARACTER*n	8*n	8*n
LOGICAL	32	32
LOGICAL*1	8	8
LOGICAL*2	16	16
LOGICAL*4	32	32
INTEGER (default)	32	32
INTEGER (switch -i2)	16	16
INTEGER*1	8	8
INTEGER*2	16	16
INTEGER*4	32	32
REAL	32	32
REAL*4	32	32
REAL*8	64	64
DOUBLE PRECISION	64	64
COMPLEX	64	32
COMPLEX*8	64	32
COMPLEX*16	128	64
DOUBLE COMPLEX	128	64

Figure 4-3a. Fortran Data Types

Figure 4-3b shows the encodings for C/C++'s data types.

Data Type	Size (Bits)	Alignment
char	8	8
char[n]	8*n	8*n
short	16	16
long	32	32
int	32	32
float	32	32
double	64	64

Figure 4-3b. C/C++ Data Types

Figure 4-3c shows the encodings for Pascal's data types.

Data Type	Size (Bits)	Alignment
char	8	8
boolean	8	8
integer	32	32
float	32	32
real (switch -P3)	32	32
real (default)	64	64
double	64	64

Figure 4-3c. Pascal Data Types

## 4.7 Internal Registers

The registers described below are those of the 80386 SX, 80386 DX, 80486 SX, and 80486, whose register set also includes the registers of the 80487SX.

The register set of the 80386/80486 appears in *Figure 4-4*.

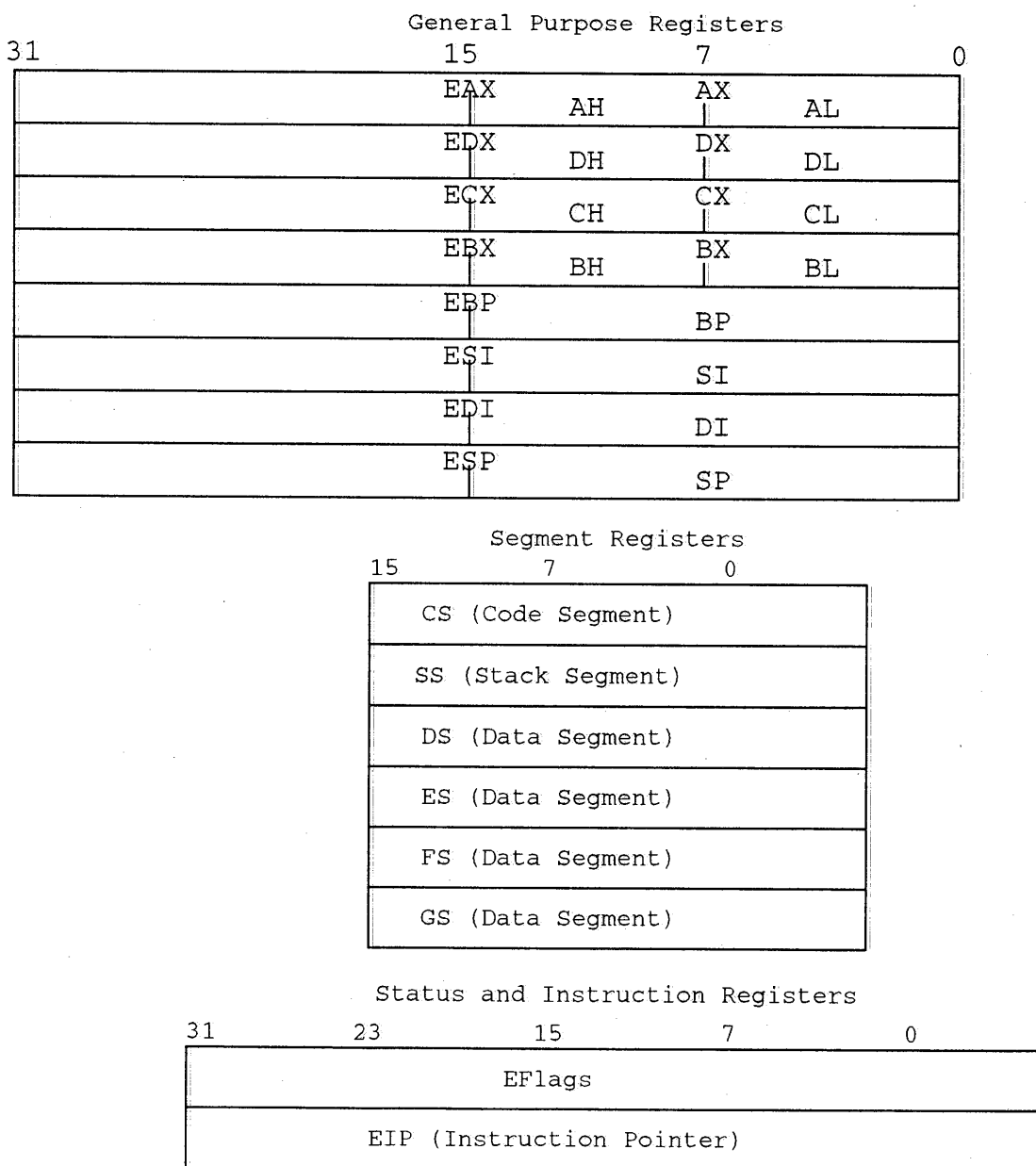


Figure 4-4. Applications Register Set

### 4.7.1 General Purpose Registers

As *Figure 4-4* shows, there are eight general-purpose registers. Each of these includes another, smaller register located in its lower word. In the cases of EAX, EBX, ECX and EDX, the lower word register is further divided into two registers of a single byte each.

All eight general-purpose registers can be used with logical, most integer math, and most 32-bit addressing instructions. Many also have special purposes assigned to them:

EAX	Extended (dword) Accumulator register. Certain instructions, such as MUL and DIV require the use of EAX as a source or destination register. Other instructions, including MOV, treat EAX or AX specially, having a special, shortened, opcode when it is a destination or source.
AX	(word) Accumulator register. Usage of AX is similar to that of EAX, but applies in 16-bit code. Additionally, it is used for ASCII adjust instructions.
AL	(the Accumulator's Lower byte). Along with AX and EAX, AL is used as destination and source for IN and OUT instructions for reading from and writing to ports. AL is used for decimal adjust instructions.
EBX/BX	Extended Base register and base register. BX is used as a base in 16-bit addressing (not supported by the NDP compilers, which do all their work with 32-bit addressing).
ECX/CX	Extended Counter register and Counter registers. ECX and CX are used with LOOP instructions and string instructions for that purpose.
EDX/DX	Extended Data Register and Data Register. EDX is used for the high dword of the DX dividend for DIV instructions. DX is the lower word of EDX. It is used as for the high word of the dividend for word-sized DIV instructions.
ESP/EBP	Extended Stack Pointer and Extended Base Pointer. ESP and EBP are used to point at the stack. ESP points to the last value placed on the stack. EBP is often used to point at a stack frame. The stack grows downward. On entry into a routine, ESP points at the return address and parameters are above ESP. If there are no local automatic variables, parameters may be accessed through ESP where there are local variables or a stack frame may be created by setting EBP to a value relative to ESP's on entry (ESP's entry value is usually 8) and subtracting the size of the frame from ESP. All references to local automatic variables and parameters can then be made through EBP.
SP/BP	(word) Stack Pointer and Base Pointer. SP and BP are natural to 16-bit code, which requires a 16-bit stack. They must not be used in 32-bit code, which requires a 32-bit stack.
ESI/EDI	Extended Source Index register and Extended Destination Index register. ESI and EDI are used to point at the sources and destinations of a variety of instructions, such as REP MOVSD.
SI/DI	Source Index (word) register and Data Index register. SI and DI use 16-bit addressing, and are not particularly useful in most 32-bit code.
EIP	Extended Instruction Pointer. Instructions execute one by one as the EIP points to them. EIP is not accessed directly but is updated as instructions execute depending on their nature and size.

### 4.7.2 Segment Registers

CS, DS, ES, FS, GS and SS hold segment selectors for protected mode segments. In real mode programs, they hold segment addresses, which are distinctly different in character from segment selectors. A segment selector is not an address; it is a handle, or, perhaps more accurately, it indexes into a lookup table. In addition, protected mode segments are not contiguous in physical RAM, but are paged. Address decoding can, for practical purposes, be regarded as instantaneous.

CS	Holds the descriptor of the Code Segment. Memory accessed through CS cannot be written to, but can be executed as instructions.
----	---

- DS Data Segment. Most addresses held in general purpose registers act through it, unless there is an address segment override associated with the instruction.
- ES Extra Segment. It is used with EDI or DI for string MOV instructions.
- FS, GS New segments in the 80386. They are used by placing address segment override bytes on the instructions that access memory through them.
- SS Stack Segment

NDP compilers expect called subroutines to respect the values in EBX, ESI, EDI, EBP, ESP and all segment registers. This means that a subroutine must restore all values it finds in the segment registers it uses.

### 4.7.3 The 80386/80486 Flags Register

Figure 4-5 shows the 12-bit 80386/80486 flags register, called EFLAGS.

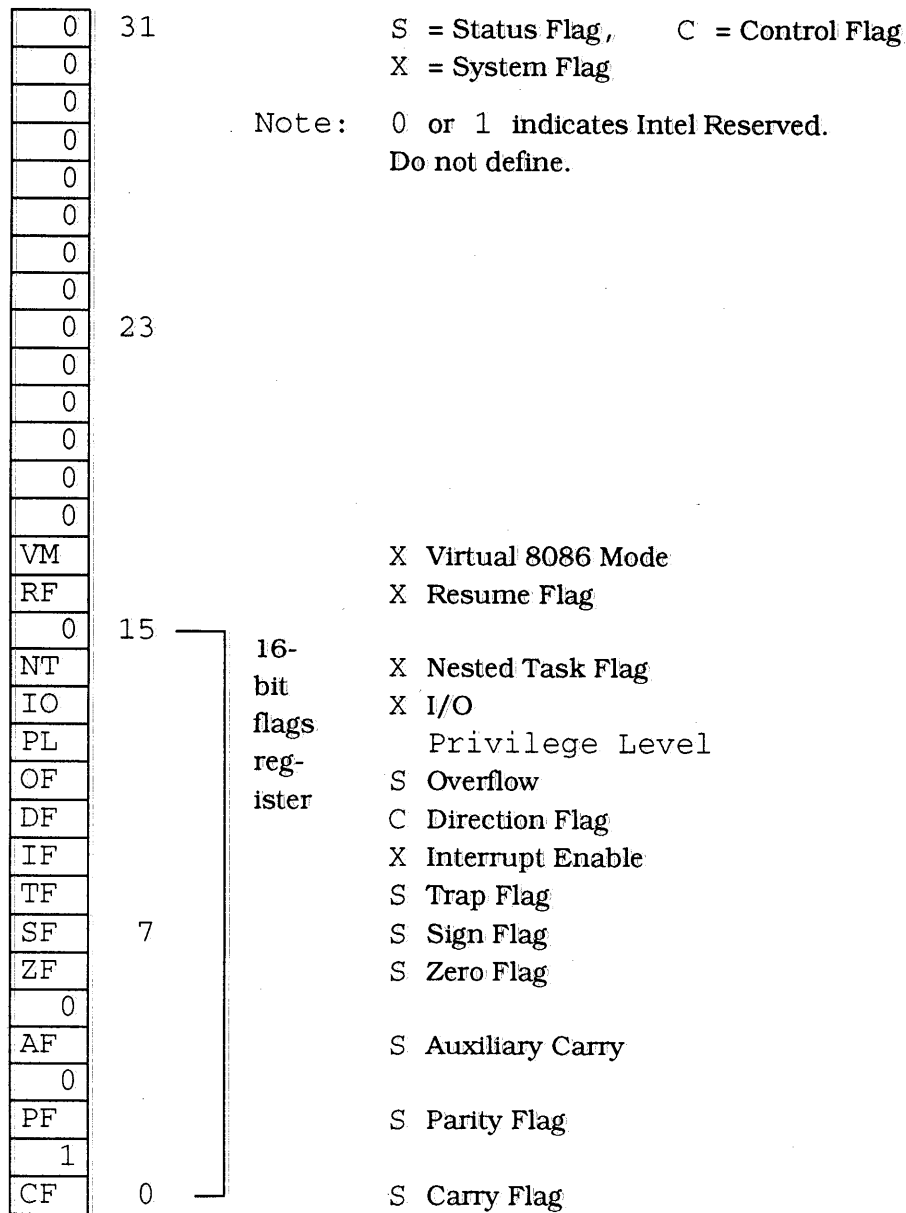


Figure 4-5. EFLAGS Register

VM Virtual Mode flag. VM is not usefully accessed in protected-mode code.

RF	Resume Flag. RF temporarily disables debug exceptions so an instruction can be restarted successfully. It is useful to programmers building debuggers.
NT	Nested Task flag. NT is useful for controlling interrupt returns in nested tasks. This flag is not normally useful to applications programmers.
IOPL	I/O Privilege Level. An application that finds it useful to adjust this 2-bit field probably will not be able to do so. IOPL defines the application's rights to do I/O through IN, INS, OUT and OUTS instructions and to change the IF flag with CLI and STI instructions.
IF	Interrupt-enable Flag. The operating system or DOS extender determines whether the application has access to IF. IF's purpose is to disallow or allow interrupts.
DF	Direction Flag. DF is the only flag on the process of the control flag type. Its value determines whether the index registers, DI and SI, auto-increment or auto-decrement with string instructions.
TF	Trap Flag. Variously included by Intel as a status flag or not, TF is used to set single stepping for debuggers.
CF	Carry Flag. CF is set on high-order bit carry or borrow. Otherwise arithmetic operations clear it.
PF	Parity Flag. PF is set or cleared depending on whether the low-order eight bits of a result contain an even or odd number of set bits.
AF	Adjust Flag. AF is set or cleared on a bit carry or borrow involving the high-order nibble in the AL register.
ZF	Zero Flag. ZF is set or cleared depending upon whether the result of an operation is 0.
SF	Sign Flag. SF is set or cleared depending on the sign of the result of an operation.
OF	Overflow Flag. OF is cleared or set depending on whether the result of an operation is within bounds of representation.

#### 4.7.4 Systems Control Registers

Applications programs do not normally reference the systems control registers, CR0, CR1, CR2 and CR3. Whether they are accessible depends on the operating system or DOS extender.

CR0 contains six one-byte fields called system control flags. These are as follows:

PE	Protection Enable flag. Setting this flag puts the system into protected mode.
MP	Math Present flag. MP controls the function of the WAIT instruction.
EM	EMulation flag. EM indicates whether math coprocessor functions are to be emulated.
TS	Task Switched flag. The system sets TS as tasks are switched. It is used in connection with coprocessor functions.
ET	Extension Type, is used to indicate the type of math coprocessor (80287 or 80387).
PG	PaGing flag, is used to indicate whether the processor uses page tables to translate linear addresses into physical addresses.

The remainder of CR0 is reserved.

CR1 is reserved.

CR2 holds the page fault linear address. It is used for handling page faults.

CR3 is only used when PG is set. It holds the page directory base register, and is used for locating the page table for the current task.

## 4.8 The 80387 Register Set

The 80387 register set, shown in *Figure 4-6*, includes eight 80-bit data registers; three 16-bit registers: the control register, the status register, and the tag word register. The Intel 80486 has a built-in 80387-compatible FPU.

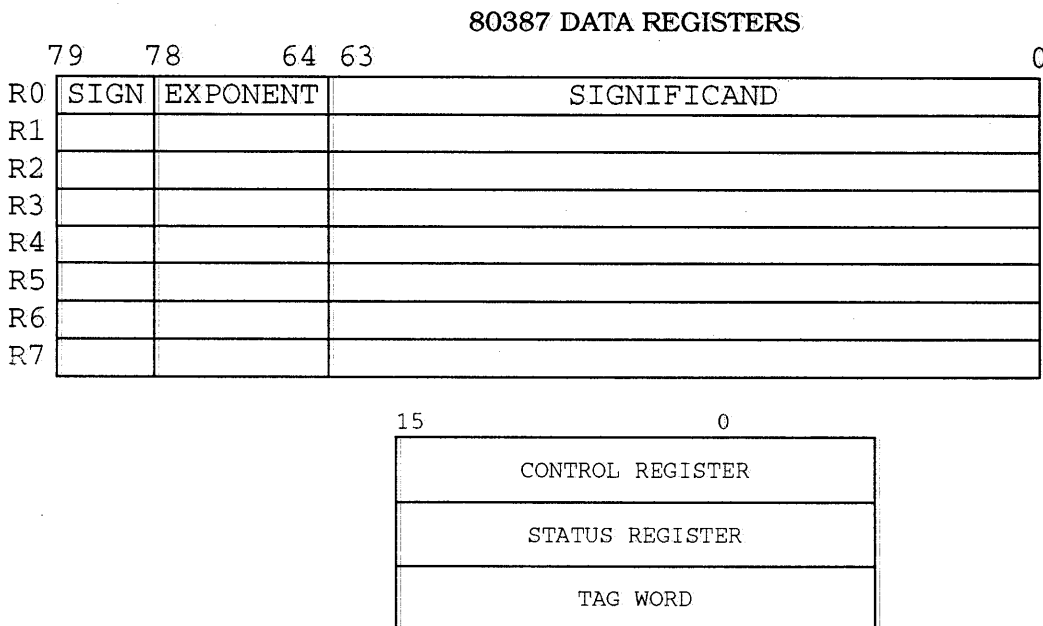


Figure 4-6. 80387 Register Set

### 4.8.1 80387 Data Registers

The 80-bit data registers are referred to in two different ways. As actual hardware they are R0 through R7 (or, historically, ST0 through ST7 -- note lack of parentheses). Software sees them as a stack through which data moves as it is pushed on or popped off ST(0) through ST(7). The most commonly used forms of instructions operate on ST(0), which is the top of stack.

The first datum loaded normally goes into ST(0). When a second datum is loaded, the first is normally "pushed" into ST(1) and the second in ST(0). A third will "push" the first into ST(2), and the second into ST(1). What is happening is that a pointer to the top of the stack is decrementing.

NDP compilers make special use of registers in the floating point unit. Specifically, functions returning real values, when they are compiled to run on the Intel floating point unit, always leave their return values in ST(0). Special optimizations also can cause special use of registers. See *Chapter 5* for further discussion of optimization.

### 4.8.2 The Status Word Register

The Status Word Register, shown in *Figure 4-7*, contains several flags indicating the status of the coprocessor chip. Also, four bits of the Status Word form a field called the Condition Code, and three form the Top of Stack Pointer. The Top of Stack Pointer points at one of the 80-bit data registers, R0 through R7, which is associated with ST(0). The Condition Code, in connection with the FXAM instruction, can be used to discover what kind of number is in ST(0).

B	15	Busy	
C3		Condition Code	
T			
O		Top of Stack Pointer	
P			
C2		Condition Code	
C1		Condition Code	
C0		Condition Code	
ES	7	Error Summary Status	
SF		Stack Fault	
PE		] Excep- tion flags	Precision
UE			Underflow
OE			Overflow
ZE			Zero Divide
DE			Denormalized Operand
IE	0		Invalid Operation

Figure 4-7. 80387 Status Word Register

Use the Condition Code, shown in *Figure 4-8*, to learn about the results of an operation.

C3	C2	C1	C0	Value at TOP
0	0	0	0	+Unsupported
0	0	0	1	+NaN
0	0	1	0	-Unsupported
0	0	1	1	-NaN
0	1	0	0	+Normal
0	1	0	1	+Infinity
0	1	1	0	-Normal
0	1	1	1	-Infinity
1	0	0	0	+0
1	0	0	1	+Empty
1	0	1	0	-0
1	0	1	1	-Empty
1	1	0	0	+Denormal
1	1	0	1	+Unsupported
1	1	1	0	-Denormal
1	1	1	1	-Unsupported

Figure 4-8. Condition Code Defining Operand Class

When used in connection with the FXAM instruction, the Condition Code helps to reveal what kind of number is in  $ST(0)$ . Consider Figure 4-9.

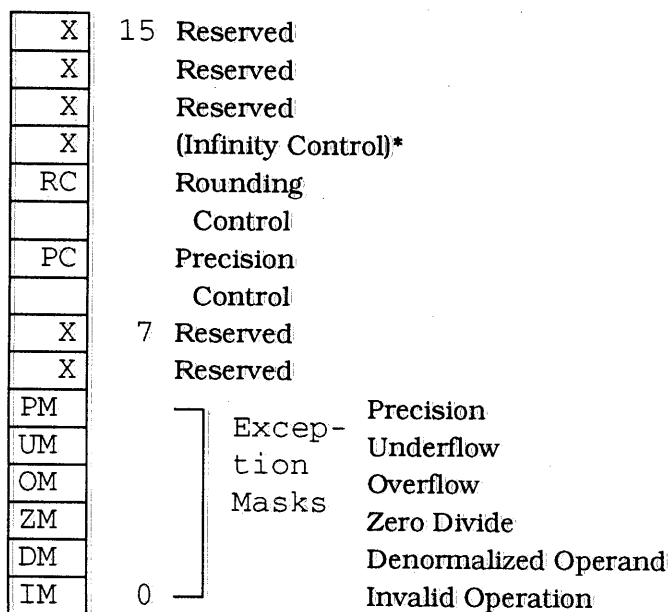


Instruction	CO(S)	C3(Z)	C1(A)	C2(C)
FPREM, FPREM1	Three least significant bits of quotient Q2                      Q0		Q1 or O/U#	Reduction 0 = complete 1 = incomplete
FCOM, FCOMP, FCOMPP, FTST, FUCOM, FUCOMP, FUCOMPP, FICOM  FICOMP, FXAM  FCHS, FAB, FXCH, FINCTOP, FDECTOP, Constant loads FSTRACT, FLD, FILD, FBLD, FSTP (ext real)  FIST, FBSTP, FST FRNDINT, FSTP, PADD, FMUL, FDIV, FDIVR, FSUB, FSUBR, FSCALE, FSQRT, FPATAN F2XM1, FYL2X, FYL2XP1  FPTAN, FSIN, FCOS, FSINCOS	Result of Comparison  Operand Class  UNDEFINED  UNDEFINED  UNDEFINED	Zero or O/U#  Sign or O/U#  Zero or O/U#  Roundup or O/U#  Roundup or O/U# undefined if C2 = 1	Operand is not comparable.  Operand Class  UNDEFINED  Reduction 0 = complete 1 = incomplete	
FLDENV, FRSTOR	Each bit loaded from memory			
FLDCW, FSTENV, FSTCW, FSTSW, FCLEX, FINIT, FSAVE	UNDEFINED			

Figure 4-9. Condition Code Interpretation

### 4.8.3 The Control Word Register

The Control Word register, illustrated in *Figure 4-10*, is used to regulate the actions of the coprocessor. There are six bits in the control word called exception masks. When an exception is masked, the corresponding sticky bit in the status word is set; no other action is done. Each mask bit corresponds to an exception flag in the status word, described above.



**Precision Control**  
 00 - 24 bits (single precision)  
 01 - (reserved)  
 10 - 53 bits (double precision)  
 11 - 64 bits (extended precision)

**Rounding Control**  
 00 - Round to nearest or even  
 01 - Round down (toward  $-\infty$ )  
 10 - Round up (toward  $+\infty$ )  
 11 - Chop (truncate toward 0)

\*This "infinity control" bit is not meaningful to the 80387. To maintain compatibility with the 80287, this bit can be programmed; however, regardless of its value, the 80387 treats infinity in the affine sense ( $-\infty$ ,  $+\infty$ ).

Figure 4-10. 80387 Control Word Format

PM Precision Mask.  
 UN Underflow Mask.  
 OM Overflow Mask.  
 ZM Zero divide Mask.  
 DM Denormalized operand Mask.  
 IM Invalid operation Mask.

Applications produced by NDP compilers run, by default, with all exceptions masked except invalid operations.

Two bits of the control word regulate precision control. These are present to satisfy IEEE definitions of compliance. They are set to highest precision by default. There is no benefit to setting them to reduced precision; the only result of setting them to reduced precision is a loss of precision.

Two bits of the control word control rounding. If both bits are set to 0, rounding is toward nearest or, in the event there is no nearest, toward even. If they are 01, rounding is toward  $-\infty$ ; if they are 10, rounding is toward  $+\infty$ ; if they are 11, rounding is toward 0. Default is round nearest.

The infinity control bit is obsolete and ignored.

The Tag Word describes the types of values in the 80-bit data registers. It splits into eight 2-bit fields, starting at the least significant, numbered TAG(0) through TAG(7). These refer to the registers as addressed as hardware, i.e., TAG(0) refers to R0, TAG(1) to R1, etc. If a tag field value is 00, then the value in its register is called a "valid," that is, it is a normal or denormal,

but not zero. If the value is 01, then the number in the register is zero. If the value is 10, then the number in the register is a NaN or an infinity.

## 4.9 Weitek Architecture

The Weitek coprocessors (1167, 3167, 4167) have a register set consisting of thirty-two 32-bit registers, which can be treated in pairs as 64-bit registers, and a 32-bit Process Context Register. Figure 4-11 illustrates these below.

WD0 -->	WS0 (Restricted)	WS1
WD2 -->	WS2	WS3
WD4 -->	WS4	WS5
⋮	⋮	⋮
WD30 -->	WS30	WS31

Figure 4-11. WTL Register File

### 4.9.1 Weitek Data Registers

The thirty-two single precision registers are named WS0 through WS31. Sixteen double precision registers are all even-numbered, and are named WD0 through WD30. Each double precision register is actually an even/odd numbered pair of single precision registers. WS0 and, hence, WD0 are restricted use registers, having special purposes for some instructions.

The NDP compilers use WS2 and WD2 for the return values of single precision and double precision functions respectively. Special usage of other registers depends upon optimizations.

### 4.9.2 The Weitek Process Context Register

The Process Context Register consists of five fields, MDSEL, MD, AE, EM and CC.

#### MDSEL

MDSEL, the MoDe SElect field, (bits 28-31 of the Process Context Register), determines which fields in the Process Context Register are to be changed on reinitialization. On reinitialization, the EM, CC and AE fields are always updated. If MDSEL is set to 0000, the MD field is also updated, but if MDSEL is set to 1100, it is not. Weitek does not define other values.

#### MD

MD, the MoDe field, bits 25-27 of the Process Context Register, determines what method of rounding to use. It has two fields, RND, bits 26-27, and IRND, bit 25. If RND is 00, rounding is toward nearest; if 01, rounding is toward zero; if 10, rounding is toward infinity; if 11, rounding is toward -infinity. If IRND is 0, integers are rounded in the same direction as reals; if IRND is 1, integers are rounded toward zero.

#### AE

AE, the Accumulated Exception field, consists of eight 1-bit flags indicating exceptions that have occurred while executing code. Many of these are practically identical to 80387 exceptions that are similarly named; refer to Section 4.8.2, page 33 for descriptions of these exceptions.

They consist of the following:

DE            Data chain Exception flag. This is obsolete and unused.

UOE	Undefined Opcode Exception flag. Indicates an undefined operation was attempted. This is always a problem, indication something wrong with the code in the executing program at the time of the exception. Causes could range from bad code generation by a compiler or assembler to runaway pointers in the application.
PE	Precision Exception flag. Similar to that of the 80387.
UE	Underflow Exception flag. Similar to that of the 80387.
OE	Overflow Exception flag. Similar to that of the 80387.
ZE	Zero divide Exception flag. Similar to that of the 80387.
EE	Exception Enable Flag. Used to enable or disable exception handling.
IE	Invalid Operation Exception flag. Similar to that of the 80387.

**EM**

EM	Exception Mask field. Consists of seven 1-bit flags indicating masking of related exceptions. These include the following:
DM	Data chain Flag. This is obsolete and unused.
UOM	Undefined Opcode Flag. Masks UOE's, described above.
PM	Precision Flag. Masks precision exceptions, described above.
UE	Underflow Flag. Masks underflow exceptions, described above.
OE	Overflow Flag. Masks overflow exceptions, described above.
ZE	Zero divide Flag. Masks zero divide exceptions, described above.
IE	Invalid Operation Flag. Masks invalid operation exceptions, described above.

**CC**

CC	Condition Code field. Consists of three bits, numbered 8 (called C0), 10 (called C1) and 14 (called Z), of the Process Context register. It is used for comparing or testing numbers. Its meaning is as follows:
----	--

If C0=0, C2=0, and Z=0 the result is "Greater than."

If C0=0, C2=0, and Z=1 the result is "Equal."

If C0=1, C2=0, and Z=0 the result is "Less than."

If C0=1, C2=1, and Z=1 the result is "Unordered."

## 4.10 Numeric Exceptions

For our purposes, a numeric exception is a numeric error occurring during operations that take place in the body of the program (i.e., inline) rather than within a library routine. In this limited sense, numeric exceptions apply only to operations a single coprocessor instruction does and the coprocessor's direct and immediate response to an error condition. Algebraic operations (addition, subtraction, multiplication, division and square roots) are done inline whereas trigonometric or transcendental operations are not. Our definition includes division by zero and taking the square root of a negative number but excludes taking the log of a non-positive number.

For errors that happen in a library routine, software in the library handles errors during trigonometrics and transcendentals and the coprocessor handles errors during algebraic operations. Most of the library routines have built-in safeguards that prevent out-of-range arguments from being passed to the coprocessor. In the NDP compilers, these move an error value into an internal compiler variable whose value can be obtained by calling the function `geterrno` (in Fortran), or checking the variable `errno` in C/C++. The return values pertinent

to math functions are 33 (error in domain) and 34 (error in range). You will have to monitor the value returned at key points by `geterrno` to discover whether your program has suffered any errors. You should be aware, however, that these rules are not absolute. For more information on the NDP compilers' response to exceptions within a library routine, look under the routine's name in the appropriate *NDP Library Reference Manual*.

The Intel and Weitek numeric coprocessors respond in very similar ways -- for the most part identically -- to numeric exceptions. Both types have a default response that takes place entirely within the chip and a customizable response that takes place in software, outside the chip. The programmer can choose which response the chip will take by masking exceptions or unmasking them. For convenience, we call the former passive error handling and the latter active error handling. If exceptions are unmasked, an error condition will cause the coprocessor to generate an interrupt request. The exact route the request takes varies among machines, and usually need not concern the applications programmer.

Below is the list of exceptions recognized by the Intel 80387 and 80860:

1. Invalid Operation
2. Denormal Operand
3. Zero Divide
4. Overflow
5. Underflow
6. Precision (inexact result)

The sections on coprocessor architecture (*Sections 4.8, 4.9*) expand on the nature and meaning of each of these errors.

Computational operations in the mathematics realm differ fundamentally from computational operations in the cybernetic realm (i.e., within computing machines). For example, numbers are continuous and unlimited but the operands of computing machines are discrete and bounded. What constitutes a numeric error and how to handle it properly turns out to be a very complex question for the NDP environment. A division by zero is unquestionably a mathematical error but in the cybernetic world can be important -- see the example in *Section 4.11*, page 42, of a parallel network of resistors that takes advantage of both zero and infinity in duplicating the real world behavior of the network. Perhaps that is why computer jargon uses the term "exceptions" instead of "errors." The many "provinces" of the cybernetic realm can differ greatly among themselves, as many of our customers discover while porting applications from mainframes to the NDP environment. Because the IEEE number system is unlike any other they have encountered on a computer before, an error in an NDP environment is far from an error on an old-fashioned mainframe. For example, operations like division by zero and division by infinity are well defined. The results generated by each can even be signed, making it possible to compare infinities.

Not only does the user have a large say in determining what an error is, but virtually everyone disagrees on what should and should not be treated as an error. To accommodate differing views of what constitutes an error, we made it possible for the user either to change the response characteristics of the default handler, or to write his/her own. The default exception handler that is part of the NDP runtime environment employs a combination of active and passive responses. It unmaskes the exceptions considered most severe, allowing the coprocessor's default response for the others. We chose this because the passive response of the coprocessor is the most reasonable for most exceptions. Many exceptions even produce results that are "self-healing" when used with the reciprocal operations. If a masked error occurs at any time during the program, a Microway NDP language routine will print out an error message when the program ends.

You may want to modify the compiler's default exception-handling response without going so far as to write a custom exception handler. Consider the invalid operation exception such as trying to take the square root of a negative number, which has no mathematical or cybernetic meaning in the world of real numbers. The compiler normally unmaskes this exception; if such an error occurs, the program prints an error message and halts. In your particular application, you may want the program to proceed to the end of a certain series of calculations -- perhaps

in a curve-fitting program -- even if such an exception occurs. You can mask the invalid operation bit, thus enabling the NDP's default response to an invalid operation, which is to return a number known as an indefinite real, one of the "quiet NaNs," and continue (for more on these, see *Section 4.11*, on IEEE numbers, page 42). If an indefinite real enters a calculation stream, it propagates to the end, allowing you to test the final result. If it is an indefinite real, you can adjust and repeat.

#### 4.10.1 NDP Compilers' Handling of Numeric Exceptions

The coprocessor's default response to an exception is to emit a result that is appropriate for the error that has just occurred, to set the appropriate bit in the status register, and to continue processing. For example, division by zero will produce infinities while invalid operations such as the square root of a negative number will produce a NaN (Not-a-Number). Both infinities and NaNs have special binary representations in the IEEE format (see *Section 4.11*, page 42). Any binary numeric operation that has a NaN as an argument produces a NaN as a result, i.e., once a NaN enters a calculation stream it propagates to the end. For the most part, the chip's default responses result in acceptable -- even innocuous -- consequences for all operations except invalid operations and denormals.

To find errors, the NDP compilers have a filter in place at all locations, including the PRINT and WRITE routines, at which binary reals are converted to strings. When the filter finds a NaN or infinity, it outputs "NaN" or "Infinity" rather than a string of digits that might be mistaken for a legitimate value. At the end of the program, the exiting routine checks to see if any errors occurred. If so, a brief message to that effect is sent to the standard output device before the program returns to the operating system. However, for invalid operations (and, with a Weitek, undefined opcode and data chain exceptions), the NDP compilers' response is active exception handling, i.e., an interrupt is generated immediately. Information about the state of the coprocessor goes to the standard output device, and the program returns to the operating system. The programmer can change the default behavior by leaving the default handler in place and masking or unmasking exception response bits or by substituting his or her own exception handler (See *Section 2.6.1, Exception Handling*).

The default active exception handling response of an NDP program is roughly the same as that found in many runtime environments -- dump some information about the state of the machine where the exception occurred and exit the program. This minimal response to exceptions may be perfectly adequate for any given program. On the other hand, it may be too harsh, as even invalid operations do not necessarily indicate a serious program flaw. In many real world problems that involve iterative solutions, especially in simulations of physical systems, the systems being simulated tend to wander into regions that generate invalid numerical operations. In these situations, it is often possible to come up with a reasonable response to the exception that will deflect the simulation back into a "stable" region. For example, one Microway developer recalls a case in which he was taking the square root of the difference of two large numbers that were almost identical. The simulation in question would converge only when the absolute value divided by 2 replaced negative differences. For those users who require more than the minimum, we have provided high-level functions with which the users can write their own exception handlers.

Before field errors occur, the user can place checkpoints in the program. For example, one can test a divisor against zero before dividing and pay a time penalty for doing so if this makes sense in a particular program. As a practical matter, this may often be perfectly adequate. On the other hand, an advantage of active exception handling is that it is never invoked unless an error occurs. A further advantage is that the user can craft his own exception handler from high-level routines provided in this product by Microway.

The programmer is in the best position to decide whether to use active handling, passive handling, or placing checkpoints. The many issues that might be encountered in a custom-written exception handler are beyond the scope of this manual. We touch upon them only insofar as they may help in the understanding of the error handling functions provided with

the compiler. Successful use of these functions presupposes some knowledge of the coprocessors' underlying architecture. Using the functions does not necessarily require a full understanding of the often esoteric issues involved in exception handling. However, those who want a fuller understanding should read the next section.

## 4.11 An Introduction to the IEEE Number System

This section is for users who want to write their own exception handlers. Users planning to use the Microway default exception handler also may find the following discussion useful for understanding and responding to error messages such as "invalid operation." Invalid operation, under IEEE specifications, does not include overflow, underflow, divide by zero, or divide by infinity.

Understanding NDP exceptions is inseparable from understanding IEEE-754 floating-point numbers, in particular, the set of special values that the IEEE system recognizes, handles and produces. These values simplify and improve the handling of errors but complicate the idea of what an error is. For example, the following formula algebraically determines the resistance of a network of parallel resistors:

$$R_t = 1 / (1/R_1 + 1/R_2 + 1/R_3)$$

It is well determined where any of the resistances in the network are zero (i.e., the total resistance of the network is zero when any of its components is zero). Normal floating-point number systems, however, do not handle this case because it requires that division by zero generate an infinity and that division by infinity generate a 0. In a normal numeric system there are zeros but no infinity, and dividing by zero is always an invalid operation. The numbers plus and minus zero are special (i.e., they have their own special representation) along with the numbers plus and minus infinity. The existence of special numbers at both extremes of the number system (instead of at zero only) are what set the IEEE number system apart from the older systems used by micros and mainframes. Besides signed zeros and infinities, the special numbers include a group of numbers called denormals that have reduced precision, a group of numbers called NaNs that are not numbers at all (i.e., they are combinations of bits that fall outside the domain of the valid numbers and are intended for special purposes), and a special NaN called an indefinite real.

### 4.11.1 IEEE Representation of Real Numbers

IEEE floating-point numbers are binary: they include a binary fraction that is multiplied by 2 raised to a binary power and the number -1 raised to the power 0 or 1 (which controls the sign). A binary fraction is easy to interpret: the digits to the left of the binary point are an ordinary unsigned integer while the digits to the right show which of the negative powers of 2, (1/2, 1/4, 1/8, . . .) are to be included. Consider the binary floating-point number, 101.101.

$$\begin{array}{rcccccc} \text{power} = & 4 & 2 & 1 & 1/2 & 1/4 & 1/8 \\ & 1 & 0 & 1 & 1 & 0 & 1 \end{array}$$

$$101.101 = 4 + 0 + 1 + 1/2 + 0 + 1/8 = 5.625$$

The binary floating-point numbers used by the IEEE convention cannot be used to represent any arbitrary number (i.e., they do not have infinite precision) and they have an associated "granularity." Numbers near zero are usually capable of being more precisely expressed than very large numbers. In other words, if we examine the spectrum of IEEE real numbers, we find that the number of expressible real numbers between 0 and 1 is almost equal to the number of expressible real numbers between 1 and infinity (more on this below).

The number system used to represent IEEE reals combines three binary numbers: a sign bit, a binary exponent consisting of 8, 11 or 15 bits and a binary significand (mantissa) consisting of 24, 53 or 64 bits. For example, the decimal floating-point number +4.0 can be expressed as the product of -1 raised to the power of the sign bit times a significand (mantissa) of 1.000000

times 2 raised to the 2 power. To represent any IEEE floating-point number we use the following formula:

$$\text{real\_number} = (-1)^s * (\text{significand}) * (2)^{\text{exp}}$$

For the decimal number +4.0 the three components have the following values:

```
s = 0 (i.e., the number is positive)
significand = 1.000000000000000000000000 (i.e., 1.0)
exp = 00000010 (i.e., 2)
```

Before going on, please note that the significand is in the form 1.00..0. This format was not an accident -- IEEE significands are always expressed in normalized format, which means having a single 1 to the left of the binary point. The process of getting reals into this format is called normalization and involves shifting the binary point to the left or right, as appropriate, until a 1 appears ahead of the binary point (properly adjusting the exponent while this is being done, of course). This saves one bit when the number is stored in memory.

We will now examine how a binary real number is encoded in a computing machine and how the number system is extended to introduce the special numbers that play a very important role in IEEE exception handling. We will treat all numbers as if they were non-negative, because the negative values are just the mirror image of the positive values.

If we count the bits used above to represent the three parts of the number 4.0, we find that we used 33 bits, 1 more than the 32 bits we normally associate with single real numbers. The "extra" bit is the leading 1 in the significand. Since the standard states that valid reals always have a single 1 to the left of the binary point, this bit never changes and therefore does not have to be explicitly stored in the computer representation of the number. This reduces the number of bits to 32, and effectively increases the precision of 32-bit numbers by a single bit.

Since both positive and negative exponents are represented in the computer, the binary value used to represent them must be some form of signed integer. The exact scheme chosen plays an important role in determining how the special numbers with which this section deals are encoded. The goal is to take by eminent domain from the binary floating-point number space, a representation for the extra numbers we need to make the number system well-behaved around zero and between the largest numbers and infinity. We do this by expropriating a binary value at the bottom and top of the exponent's range, and using these two values to encode the special numbers near zero and infinity.

As an example, let's examine what happens with the exponent of the single real. Single real (32-bit) floating-point numbers have an 8-bit exponent that can take on 256 values. The most obvious tactic is to map one of these 256 values to an exponent of zero, leaving 255 for positive and negative exponents, but this leaves nothing for special-purpose numbers. Therefore we reserve the exponent values 0 and 255 for the extremes of the range, leaving the integers 1..254 to represent all "ordinary" exponents: negative, zero, and positive. We use one of this 254 member set to represent an exponent of zero, leaving only 253 members for the positive and negative exponents, which creates an asymmetry in the exponent set. The IEEE committee chose to have 126 negative exponents and 127 positive exponents. We represent the exponent zero using the value 127 (7FH) and the values between 1 and 126 will map to negative exponents while the numbers between 128 and 254 map to positive exponents. The value 127 is known as the bias, and is added to the true exponent to obtain the binary representation in the computer. A binary exponent representation consisting of all zeros is, as we said above, a special case. If the significand is also all zeros, then the value is a true zero; otherwise, it is a denormal. A binary representation of 255 represents infinity or a NaN, depending on the value of the significand. The relationship between the true exponent and its representation in the computer -- the biased exponent -- including a description of what each biased exponent follows:

Biased exponent	True exponent	Comment
0	not defined	zero, denormals
1	-126	smallest exponent



2	-125	
125	-2	
126	-1	
53	126	
254	127	largest exponent
255	not defined	infinity, NaNs

Consider what happens to the number 4.0 above. IEEE represents 4.0 as 1 times 2 raised to the power of 2, which means the number 4.0 will have a true exponent of 2. Looking in our table we see that a true exponent of 2 combines with the bias 127 to form a biased exponent of 129.

The three binary components of an IEEE floating-point number real are arranged in the order: sign bit, biased exponent and normalized significand. This is not necessarily the way the numbers are stored in memory, which is machine dependent. Intel processors, for example, store values "backwards," i.e., low byte to high byte. When the processor moves them into a register, it reverses their "backwards" orientation so they are placed high byte to low byte. That is, the highest byte is the most significant byte, the next highest is the next most significant, and so on. This is the way numbers in any number system (binary, decimal, octal, or what have you) are ordinarily arranged. The IEEE floating-point specification is designed for generality, and leaves details of implementation to the manufacturer. The NDP compilers assume a normal arrangement, but the actual arrangement of a particular value in an Intel machine depends on whether it is in memory or a register.

Let's now examine what happens to the binary components of +4.0 as we build the IEEE representation. As mentioned, for the case of single reals, the exponent is biased with 127, which when added to an exponent of 2 yields an exponent of 129. The official IEEE binary representation of the number 4.0 becomes:

```

sign bit      = 0 (i.e., it's positive)
exponent      = 10000001 (i.e., biased exponent = 129)
significand   = 000000000000000000000000
                (fraction = 1.0 and the leading 1 is now implied)

```

Taking these bits and lining them up in a row we get the 32-bit IEEE single real number in binary (with its hexadecimal representation below):

```

binary      0100 0000 1000 0000 0000 0000 0000 0000
hexadecimal 4   0   8   0   0   0   0   0

```

We print the above notation stretched out to show how each 4-bit nibble maps to a hexadecimal digit. For ease of readability, the hexadecimal number is:

```
40 80 00 00
```

This is its normal byte sequence, and the way a debugger would display the value in a 32-bit Intel processor register. If the debugger displays the same number in memory, however, it will show the following:

```
00 00 80 40
```

Let's now examine the rules used to build the IEEE floating-point number types: single real (32 bits) and double real (64 bits). The rules are fundamentally the same as those used to build the single real numbers above, except that the sizes of the exponents, biases and significands are a function of the type being used. The parameters all three types use are summarized in the table below, along with the maximum and minimum size of the exponent for each.

#### Representation of Exponents

Real Type	Single	Double
Total binary bits	32	64
Significand width	23	52

Exponent width	8	11
Exponent bias	127(7FH)	1023(3FFH)
Maximum exponent	127	1023
Minimum exponent	-126	-1022

To see these rules in action, we will now build the number 4.0 in the double real format:

#### Double Real format of 4.0

```

sign = 0
exponent = 2 + 3FFH = 1000 0000 001
significand = 0..0 (52 bits wide)
binary      0100 0000 0001 0000 0000 0000 0000 0000
hexadecimal 4 0 1 0 0 0 0 0

```

The IEEE representation:

```
40 10 00 00 00 00 00 00
```

is stored in memory by the Intel CPU as:

```
00 00 00 00 00 00 10 40
```

### 4.11.2 Precision and Denormals

To understand what happens to floating-point precision as we scan over the real numbers, let us examine the case of single reals. We shall focus on the non-negative values, remembering the negative side of the IEEE number line is the mirror image of the positive. The mantissa here is 24 bits, an implicit leading bit that is always 1 and 23 explicit bits that can be either 0 or 1. Therefore the interval between any two powers of 2 contains just 223 unique significands, e.g., the number space between 1 and 2 can be divided into 8,388,608 unique real numbers and so can the interval between 1/2 and 1. The number of reals stays constant in any interval, so that there are approximately 8.38 million real numbers between 1 and 2, between 2 and 4, between 4 and 8, etc. Likewise, there are approximately 8.38 million real numbers between 1 and 1/2, between 1/2 and 1/4, between 1/4 and 1/8, etc.

Thus, the density per interval grows between successively smaller negative powers of 2, and diminishes between cardinals in each interval between successively greater powers of 2. Thus we have around 8.38 million values between 1 and 2, but only around 4.18 million reals between 2 and 3, because the same number of significands must be stretched out over a greater range. In the interval from 4 to 8 there are around 2.09 million reals between cardinals, and from 8 to 16, around 1.04 million. By the same logic, we see that in the interval between 8,388,608 and 16,777,216, we will have just one floating-point number per cardinal, which is the same granularity as an ordinary integer representation. Above 16,777,216, 32-bit integers do a more exact job of representing numbers; they suffer from having a limited range and do a much worse job with numbers smaller than 8,388,608. On the average, however, floating-point numbers are a better choice than integers, if you need precision near zero or a large dynamic range.

To understand the denormal, we must look at what happens as the numbers get smaller, which is equivalent to asking what happens as the IEEE representation approaches zero. We begin by taking the smallest valid single real ( $1.0 \cdot 2^{-126}$ ), and start dividing it by 2. This smallest number has a biased exponent of 1, which drops to 0 when it is divided by 2. An exponent of all zeros is used as a flag to indicate that the exponent has now passed its smallest value (-126) and that the system is now reducing the significand by factors of 2. The process begins by moving the first (implicit) significant binary digit to the right of the binary point. The special numbers between the smallest number and zero no longer have an implied 1 at the head of their significand. As the values are progressively halved, the significand bits continue being shifted right, until the number system runs out of room. When the process finally shifts the last significant digit out of the fraction, it arrives at the IEEE representation of zero, which is a zero biased exponent and a zero significand. In effect, the number system trades precision

for range by turning off normalization for the sequence of numbers between the smallest number and zero.

number	biased exp	implied bit	significand
2-126	1	1	000000000000000000000000
2-127	0	0	100000000000000000000000
2-128	0	0	010000000000000000000000
2-129	0	0	001000000000000000000000
2-130	0	0	000100000000000000000000
...			
2-149	0	0	000000000000000000000001
zero	0	0	000000000000000000000000

There are 8,388,607 ( $2^{23}-1$ ) of these positive single real denormals, providing a "cushion" between the smallest valid number and zero. Since you can't halve your value and keep it too, the system must give up something with the introduction of denormals, and that something is precision. Denormals extend the low end range of single reals, from  $2^{-126}$  to  $2^{-149}$ , but for every power of two that they drop, they lose a bit of precision, till at the low end the precision of the significand is reduced to a single bit, while double precision is 15 digits. If you could determine the distance of the moon with this much precision, you could determine the distance within two miles. With double precision, your range would be within .0012 inches. Weitek coprocessors do not support denormals. Any number that would become denormal becomes 0.

### 4.11.3 Infinities and NaNs

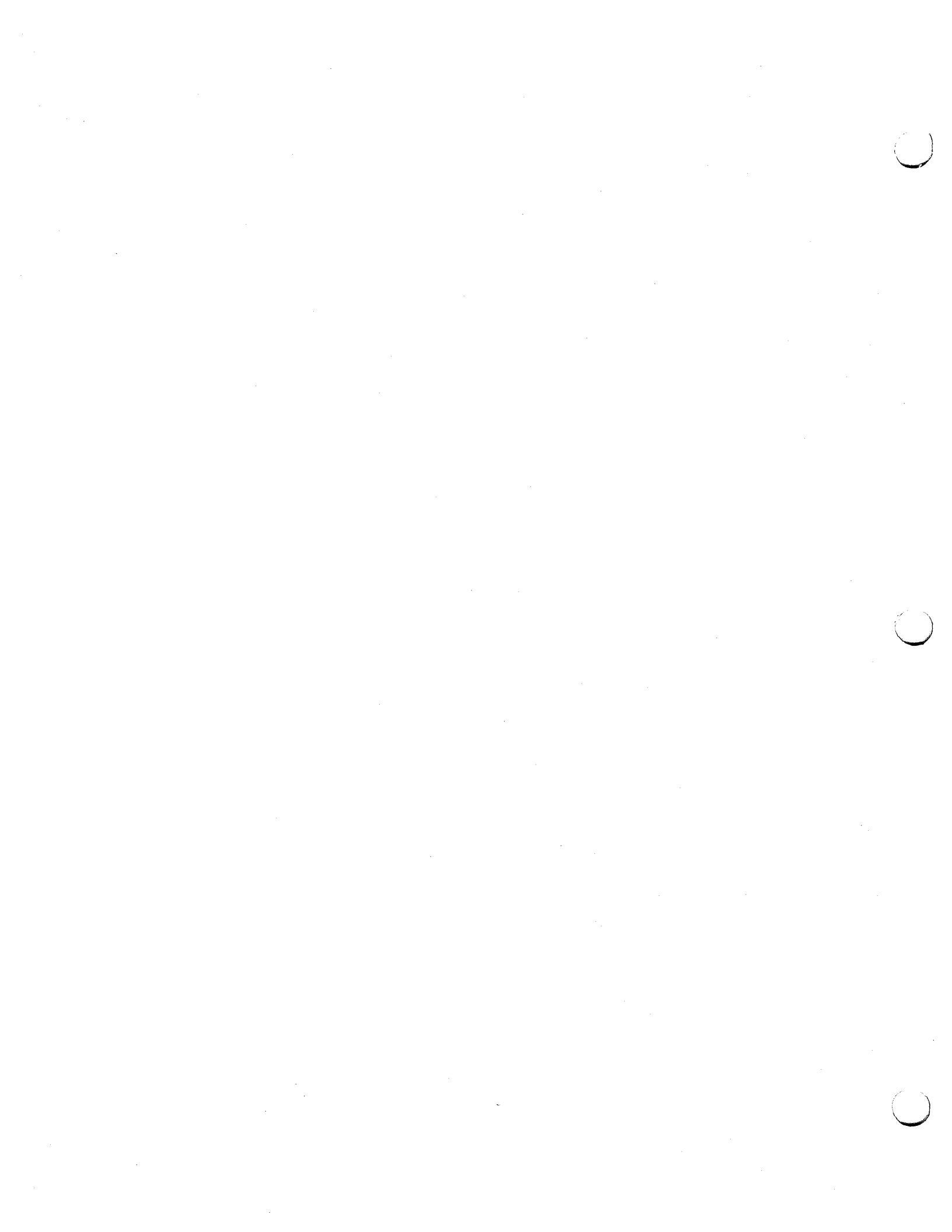
The encodings for all the types of numbers that can be used in the IEEE representation appear in *Figure 4-2*, page 27. The largest possible biased exponent is always reserved for special numbers, (infinities and NaNs) just as the smallest biased exponent is reserved for zero and denormals. Normals refer to numbers that can be normalized, i.e., valid numbers.

As *Figure 4-2* shows, the biased exponents of infinities and NaNs have all bits set to one. They can be signed positive or negative but as we did with denormals, we shall focus on the non-negative values, remembering the negative side of the IEEE number line is the mirror image of the positive. Infinities and NaNs are distinguished by the bit pattern of their mantissas. There is only one pattern for the significand of an infinity: the leading bit is one, all the rest are zeros. Infinities are the masked response to an overflow or a division by zero. (The rounding mode in effect determines whether these errors produce infinity or the largest representable number.) The use of infinities as operands has well-defined results. Some of these are legal and produce sensible results, for example, division of a finite number by infinity, which produces zero as a result. Legal arithmetical operations on infinities are always exact. Illegal operations on infinities result in the invalid operation exception. Because the NDP default exception handler is set up to detect invalid operations (i.e., they are unmasked), the handler will get invoked any time infinities are used illegally and generate an error message saying "invalid operation."

*Figure 4-12* summarizes how infinities are used. It should be noted that the exceptions (overflow or division by zero) can be unmasked so that, rather than return an infinity, they will instead trigger the exception handler. The NDP languages' default, however, is that they be masked.

Infinity Operands and Results		
Operation	Operands	Result
Addition	$+\infty$ plus $+\infty$	$+\infty$
	$-\infty$ plus $-\infty$	$-\infty$
	$+\infty$ plus $-\infty$	Invalid operation
	$-\infty$ plus $+\infty$	Invalid operation
	$\pm\infty$ plus $\pm X$	$\pm\infty$
	$\pm X$ plus $\pm\infty$	$\pm\infty$
Subtraction	$+\infty$ minus $+\infty$	$+\infty$
	$-\infty$ minus $-\infty$	$-\infty$
	$+\infty$ minus $-\infty$	Invalid operation
	$-\infty$ minus $+\infty$	Invalid operation
	$\pm\infty$ minus $\pm X$	$\pm\infty$
	$\pm X$ minus $\pm\infty$	$-\pm\infty$
Multiplication	$\pm\infty * \pm\infty$	$**\infty$
	$\pm\infty * \pm Y, \pm Y * \pm\infty$	$**\infty$
	$\pm 0 * \pm\infty, \pm\infty * \pm 0$	Invalid operation
Division	$\pm\infty / \pm\infty$	Invalid operation
	$\pm\infty / \pm X$	$**\infty$
	$\pm X / \pm\infty$	$**0$
	$\pm\infty / \pm 0$	$**\infty$
Compare	$+\infty : +\infty$	$+\infty = +\infty$
	$-\infty : -\infty$	$-\infty = -\infty$
	$+\infty : -\infty$	$+\infty > -\infty$
	$-\infty : +\infty$	$-\infty < +\infty$
	$+\infty : \pm X$	$+\infty > \pm X$
	$-\infty : \pm X$	$-\infty < \pm X$
	$\pm X : +\infty$	$\pm X < +\infty$
	$\pm X : -\infty$	$\pm X > -\infty$
Key: X Zero or nonzero positive operand Y nonzero positive operand. * Sign of original $\infty$ operand. -* Complement of sign of original $\infty$ operand. ** Exclusive OR of signs of operands.		

Figure 4-12. Infinity Operands and Results



# 5

## Mixing Languages

The NDP compilers (NDP C|C++, NDP Fortran, and NDP Pascal) provide great power and flexibility by making it possible to write programs that use modules written with any Microway language or with assembly language. The ability to mix modules written in different languages allows the programmer to code each task in the most appropriate and natural language for the task.

Although the NDP compilers have many extensions that should make it unnecessary to use assembly, the ability to write assembly routines and link them with code generated by the NDP compilers gives the programmer greater control. Generally, programmers use assembly routines to fine-tune programs by optimizing for speed or size, and to access the PC hardware or system services in ways not normally available through high-level languages.

### 5.1 General Rules

The NDP compilers have a high degree of compatibility. Regardless of the high-level language, each compiler translates source code into the same intermediate language that is eventually output as object code and is linked with common runtime routines. All the compilers use the same memory model, a flat model. Moreover, the NDP compilers all push parameters onto the stack in right-to-left order. That is, the last named is pushed first, then the next to the last, and so on. Thus the arguments end up on the stack in the order in which they are named in the parameter list. (The exception to this rule is the lengths of strings passed by Fortran, see below.) The runtime system for all three compilers is an NDP C|C++ application and includes the complete NDP C runtime library, so that an NDP C|C++ module called by an NDP Fortran or NDP Pascal program will have all the runtime support it expects.

There are several unavoidable differences among the NDP compilers that arise directly from the definitions of the languages themselves. There are also differences that occur to maintain compliance with industry standards. The following sections explain these differences.

#### 5.1.1 Linking Restrictions

Modules written in NDP C|C++, NDP Fortran, NDP Pascal, and assembly can be linked freely among themselves, with the following restrictions:

1. If a program has any NDP Fortran modules that use Fortran I/O, (a) the main module must be written in Fortran and it must be linked with the library LIBF, or (b) the main program must initialize Fortran I/O properly. For instance, let us assume the following Fortran subroutine is to be linked into a C|C++ program:

```
subroutine fortran_subroutine (i)
integer i,j
character*64 string
if (iargc().gt.0) then
  do j=1,iargc()
    call getarg (j,string)
    write (6,210) j,': ',string
210  format(i5,a,a)
  enddo
else
  write(6,*) 'No command line arguments were given'
endif
write(6,220) 'Input argument: ',1
```

```

220 format(1x, a, i5)
    return
end

```

Because this Fortran routine does I/O, the Fortran I/O system must be initialized. If the main program is a Fortran program, the initialization occurs automatically. If the main program is a C/C++ program, the initialization must be done explicitly:

```

#include <stdio.h>
int xargc; /* used if Fortran calls IARGC */
char **xargv; /* used if Fortran calls GETARG*/
int zero = 0; /* used by parts of LIBF */
int one = 1; /* used by parts of LIBF */
void rec_init();
void (*initrec)() = rec_init; /* used by parts of LIBF */
void (*uninitrec)() = rec_uninit; /* used by parts of LIBF */
char *tempfiles[100] = {NULL, }; /* used by parts of LIBF */
main(int argc, char *argv[]) {
    int i;
    xargc = argc;
    xargv = argv;
    fmt_init(); /*initialize Fortran formatting*/
    rec_init(); /*initialize Fortran units */
    /*program body */
    i = 5;
    fortran_subroutine_ (&i);
    rec_uninit(); /*close flush Fortran units */
}

```

The data declarations for zero, one, initrec, uninitrec, are needed to avoid symbol conflicts during the link stage. Omitting any of these declarations causes the standard version of main() to be linked, and duplicate symbols result. The link file must include a reference to the Fortran library. If, for instance, the executable program is built using the C/C++ driver, e.g., mx386, then the Fortran library must be included on the command line:

```
mx386 main.cxx forsub.o -lf3
```

The -lf4 switch will cause libf4.a to be linked in.

2. Programs that contain any modules written in NDP Pascal and that perform Pascal I/O must have the NDP Pascal library, lp4.a, linked in.

### 5.1.2 Data Type Differences

Usually, the NDP compilers share the same major data types, all having the same range of values and the same rules for operations. There are, however, some important differences (and ways around them).

#### Integers

NDP Pascal has no predefined 16-bit integers, as are found in NDP C/C++ (short int) and NDP Fortran (INTEGER\*2). This difficulty can be overcome by creating two new types:

```

type short -32768..32767 { 16-bit integer }
type ushort 0..65535 { 16-bit unsigned }

```

This allows you to pass these items between NDP Pascal and the other NDP languages.

## Char

NDP Pascal's predefined 8-bit data type, `char`, is restricted to values between 0..127, unlike NDP C/C++ (`char`, `unsigned char`) and NDP Fortran (`CHARACTER*1`, `INTEGER*1`). This difficulty can be overcome by creating two new types:

```
type schar -128..127 { signed char }
type uchar 0..255   { unsigned char }
```

This allows you to pass these items between NDP Pascal and the other NDP languages. Remember that in Fortran all integers are signed.

## Floating Point

NDP C/C++ and NDP Pascal by default promote all floats to double, but NDP Fortran does not. This problem affects only those functions that return floating point values and use the Weitek coprocessor, or that pass between Fortran and C/C++ or NDP Pascal, using NDP Fortran's `%VAL` operator. Passing by reference is not affected. There are three ways around this problem:

1. Declare all floating point parameters as doubles.
2. Include in your C/C++ programs function prototyping to coerce floats to remain floats; or use the `-p3` switch to compile NDP Pascal programs, causing `REAL` values to be interpreted as 4-byte values.
3. Pass by reference.

## 5.1.3 Naming Conventions

The differences in naming conventions among the compilers are related to case sensitivity and the use of leading/trailing underscores.

NDP C/C++ compiler is case sensitive; it outputs identifier names exactly as they are entered. It appends an underscore to the beginning of all identifiers but not to the end.

NDP Fortran is not case sensitive. It transforms all identifier names into lower case and outputs them in that form. It also appends an underscore to the beginning and end of all identifiers. Fortran can be made case sensitive by using the `-U` (upper-case U) compiler switch. Identifiers will then be output exactly as they are entered, with a leading and a trailing underscore.

NDP Pascal is by default not case sensitive; it normally outputs identifier names in lower case, like NDP Fortran. Like C/C++, it appends an underscore to the beginning of all identifiers but not to the end. NDP Pascal can be made case sensitive by using the `-p1` compiler switch. Identifiers will then be output exactly as they are entered, with a leading underscore.

Consider the following identifiers:

Language	Identifier	Output to assembly file
NDP C/C++	Flag	<code>_Flag</code>
NDP Fortran	Flag	<code>_flag_</code>
NDP Fortran (-U)	Flag	<code>_Flag_</code>
NDP Pascal	Flag	<code>_flag</code>
NDP Pascal (-p1)	Flag	<code>_Flag</code>

To make an identifier global between NDP C/C++, NDP Fortran, and NDP Pascal, use all lower case letters and postpend an underscore to the C/C++ and Pascal names, as in the following example:

Language	Identifier	Output to assembly file
NDP C/C++	<code>flag_</code>	<code>_flag_</code>
NDP Fortran	<code>flag</code>	<code>_flag_</code>
NDP Pascal	<code>flag_</code>	<code>_flag_</code>



### 5.1.4 Parameter Passing

All the NDP compilers extend character and integer values to four-byte size when passing them on the stack.

#### Passing Values

Except for strings (see below), NDP Fortran passes only by reference, i.e., it passes the address of the data item. The default in C|C++ is to pass by value. The default in Pascal is to pass by value, unless the var keyword is used in the formal parameter list.

When being called from NDP Fortran, the C|C++ function must declare its formal parameters as pointers. When calling NDP Fortran, an NDP C|C++ function must pass each actual argument as a pointer.

When passing parameters to or from NDP Fortran, the NDP Pascal routine must declare its formal parameters with the var keyword. Alternatively, the Fortran program can pass by value using the keyword %VAL in the actual argument list. It is also possible to change a pass-by-reference of a variable to an effective pass-by-value by assigning that variable to a local variable and using only the local variable.

Note that by default, NDP Fortran variables are REAL\*4, but NDP Pascal variables declared as "real" default to DOUBLE (REAL\*8), unless the Pascal module is compiled with the -p3 switch.

#### Passing Strings

When NDP Fortran passes a string, it passes both the address of the string and its length, but the length is passed by value, not by reference. Further, Fortran pushes the lengths of every string in the parameter list onto the stack, and then the other data items, including the string addresses. For example:

```
INTEGER i
CHARACTER*10 S1
CHARACTER*20 S2
CALL F(S1,i,S2)
```

The stack will look like this:

```
20 (length of S2) Bottom
10 (length of S1)
(address of S2)
(address of I)
(address of S1) Top
```

When passing strings from NDP C|C++ to NDP Fortran, the actual argument list in the NDP C|C++ program must declare a char pointer and an int for each string passed so that the interface between the two routines will be set up the way the NDP Fortran compiler expects. Assuming the example above, the C|C++ version of the code would look like this:

```
char *ps1;
int *pi;
char *ps2;
int l1,l2;
f(ps1,pi,ps2,l1,l2);
```

When passing strings from NDP Fortran to NDP C|C++, there are two possible approaches. First, the formal parameter list in the NDP C|C++ program may declare a char pointer and an int for each string passed if it is necessary for the NDP C|C++ function to know the length of the string. For example:

```
void f(char *ps1, int *pi, char *ps2, int l1, int l2)
{
    ...
}
```

Second, in C|C++, a string has no intrinsic length associated with it. Its end is marked with a null byte (value 0, not character '0'). In Fortran, every string has a fixed length and characters within it do not necessarily have special meaning. If the NDP Fortran calling program (or the NDP C|C++ function) places a null byte following the last significant character in the string, the NDP C|C+ string handling functions will be able to handle it in the normal way.

When passing strings from NDP Pascal to NDP Fortran, the formal parameter list in Pascal must declare strings with the var keyword and an integer for each string passed so that the interface between the two routines will be set up the way the NDP Fortran compiler expects. Assuming the example above, the Pascal code would look like this:

```

type
  a10 = packed array [1..10] of char;
  a20 = packed array [1..20] of char;
var
  s1 : a10;
  s2 : a20;
  i, l1,l2 : integer;
procedure p_(var s1:a10; var s2:a20; var i:integer;
  l1,l2:integer); external;
begin
  p_(s1,i,s2,l1,l2);
end.

```

When passing strings from NDP Fortran to NDP Pascal, there are two possible approaches. First, the formal parameter list in the Pascal program may declare strings with the var keyword and an integer for each string passed, as in the above example.

Second, a Pascal procedure can ignore length parameters completely, since the length of a Pascal datum is part of its type. This causes no conflict, since the lengths are passed highest on the stack. A cautious program, however, might access the lengths of the strings passed and check them against the declared lengths, to ensure there is no discrepancy.

### 5.1.5 Output Buffers

One area of difficulty exists in interfacing NDP C|C++ and NDP Fortran. If both the C|C++ functions and Fortran routines do screen I/O, it may be necessary to flush the C|C++ output buffer after using a function such as printf(); otherwise, the output may print to the screen out of order. With NDP Fortran, the runtime environment flushes the buffer every time an end of line occurs, while NDP C|C++ does not. The C|C++ output buffer can be flushed with the fflush() function, or by outputting a newline ('\n') character.

## 5.2 Calling Between NDP Fortran and NDP C|C++

The following example demonstrates passing parameters, both string and numeric, between NDP Fortran and NDP C|C++. For passing strings, it uses the first of the two methods mentioned above. Note in the code below that a trailing underscore has been added to the lower case function names, to match the underscore added by the Fortran compiler. The compilers also add an underscore to the beginning of the name.

### Listing 5-1: Fortran Main Program

```

c This program calls the external routines SCALAR, STRING,
c and STRUC. Compile this program and link it with the object
c file created by Listing 5-2 or Listing 5-3.
c
CHARACTER*50 STRING
STRUCTURE /STRUC/
INTEGER I

```

```

      INTEGER CH
      REAL*8 R
ENDSTRUCTURE
RECORD /STRUC/ REC
ISCALAR = ISCAL1(1)
PRINT 100, 'Value returned = ', ISCALAR
100 FORMAT (1X,A,I3)
PRINT *
      STRING = 'This string was passed to an external module.'
      CALL STR1 (STRING)
c If you are linking this program with C|C++ routines, you
c should null terminate string here, or in the C|C++ module. A null
c replaces the first non-significant character.
c As in C|C++, the length of the string has to be at least one
c character longer than the number of significant
c characters, so that there is a character that can be
c overwritten with a null. If you are linking with Pascal, it
c serves no purpose to null terminate the string. If the
c last character in the string is a significant character,
c i.e., not a space, it still has to be overwritten with a
c null.
      i = len (string)
      IF (STRING ( i:i ) .ne. char(32)) THEN
        STRING ( i:i ) = char(0)
      ELSE
c find the last significant character in string
        DO WHILE (STRING (i:i) .eq. char(32))
          i = i-1
        END DO
c overwrite first non-significant character
        STRING (i+1:i+1) = char(0)
      END IF
c The C|C++ function str1 changed the value in string. It wrote
c a new string and null character. Yet, Fortran will still
c write 50 characters, including any characters after the
c null that str1 inserted.
      PRINT *, STRING
      CALL STRUC1 (REC,%VAL(3))
      PRINT *, 'Structure: REC.I ', REC.I
      PRINT *, '          REC.CH', REC.CH
      PRINT *, '          REC.R ', REC.R
      END

```

### Listing 5-2: C|C++ Routines Called by Fortran and Pascal

```

/* These functions are called by a main program. Compile
   this program using the -c switch, then link
   with the main program. Note that the function names have
   an underscore appended, to match the one Fortran
   appends to its names.
*/
#include <stdio.h>
/* receive scalar from Fortran and return another */
int iscall_ (int *i)
{ printf("\nValue passed = %d\n", *i);
  return(*i*10);
}
/* receive string from Fortran and modify it */

```

```

void str1_ (char *str, int len)
{   printf("%s", str);
    sprintf(str, "This string was modified by NDP C|C++.");
}
/* receive structure from Fortran and modify it */
struct struc
{   int i;
    char *ch;
    double dbl;
};
void struc1_ (struct struc*struc2, int i)
{   struc2->i = i;
    struc2->ch = (char *) malloc (i);
    struc2->dbl = i * 1.1;
    return;}
}

```

### 5.3 Calling between NDP Fortran and NDP Pascal

The following example demonstrates passing parameters between NDP Fortran and NDP Pascal. It uses the Fortran code in *Listing 5-1*. When passing strings, it uses the first of the two methods mentioned. Note that a trailing underscore has been added to the lower case function names, to match the underscore added by the Fortran compiler. The compilers also add an underscore to the beginning of the name.

#### Listing 5-3: Pascal Routines Called by Fortran Main Program

```

{ These functions are called by a main program.  Compile
  this program using the -c switch, then link with the
  main program and Pascal library.}
{ Receive scalar from Fortran and return another }
function iscall_ (var i : integer) : integer;
begin
    writeln;
    writeln('Value passed = ', i : 2);
    iscall_ := i*10;
end; {iscall_}
{ Receive string from Fortran and modify it }
type CHARSTR = packed array [1..81] of char;
procedure str1_ (var str: CHARSTR; len: integer);
begin
    write(str:len);
    str := 'This string was modified by NDP Pascal.'
end; {str1_}
{ Receive a structure from Fortran and modify it }
type struct =
    record
        i : integer;
        ch : ^CHARSTR;
        dbl : double;
    end; {struct}
function malloc (i: integer): ^CHARSTR; external;
function struc1_ (var str: struct; i: integer): integer;
begin
    str.i := i;
    str.ch := malloc (i);
    str.dbl := i*1.1;

```

```

struc1_ := i;
end; {struc_1}
char string[81], *str;

```

## 5.4 Calling between NDP C|C++ and NDP Pascal

Because of the similarities between C|C++ and Pascal, it is a simple matter to mix these languages. The key point is case sensitivity. You should keep identifiers in lower case or use the `-p1` switch to turn on case sensitivity when compiling your Pascal modules. Note that in these examples, the functions were defined with a trailing underscore to support Fortran. If Fortran support is not needed, the trailing underscore can be dropped from the definition.

### Listing 5-4: Pascal Main Program

```

{ This program calls the C|C++ functions in Listing 5-2.
  Compile this program and link it with the object file
  created by Listing 5-2. }
program main (input, output);
type CHARSTR = packed array [1..81] of char;
type struct =
  record
    i : integer;
    ch : ^CHARSTR;
    dbl : double;
  end; {struct}
var i : integer;
var string : CHARSTR;
var len : integer;
var struc2 : struct;
type foo = record
  { This trick convinces Pascal to accept the equality
    of pointers and integers by using a variant record. }
  case boolean of
    true: (ptr: ^CHARSTR);
    false: (int: integer);
  end;
var bar: foo; static;
{ C|C++ functions must be declared external.}
function iscall_ (var i : integer) : integer; external;
procedure str1_ (var str: CHARSTR; len: integer); external;
function struc1_ (var struc: struct; i: integer): integer; external;
begin
  i := 1;
  i := iscall_(i);
  writeln('Value returned: ', i);
  writeln;
  string := 'This string was passed to an external module.' ;
  str1_(string, 50);
  writeln(string:50);
  i := 3;
  struc1_(struc2, i);
  bar.ptr := struc2.ch;
  writeln('Structure: struc2.i = ', struc2.i);
  writeln('          struc2.ch = ', bar.int);
  writeln('          struc2.dbl = ', struc2.dbl);
end.

```

**Listing 5-5 C/C++ Main Program**

```

/* This program calls the Pascal functions in Listing 5-3.
   Compile this program and it link with the Pascal Library
   and the object file created by Listing 5-3.
*/
#include <stdio.h>
main ()
{
    int i, len;
    char string[81], *str;
    struct struc
    {
        int i;
        char *ch;
        double dbl;
    } struc2;
    i = 1;
    i = iscall_(&i);      /* pass the address of i */
    printf("Value returned: %d\n", i);
    str = string;
    strcpy(str, "This string was passed to an external module.");
    strl_(str, strlen(str));
    printf("\n%s", str);
    i = 3;
    struc1_(&struc2, i);
    printf("Structure:\tstruc2.i=%d \n\t\tstruc2.ch=%ld\n\t\tstruc2.dbl=%f",
          struc2.i, struc2.ch, struc2.dbl);
}

```

**5.5 Interfacing Assembly Language**

NDP C/C++, Fortran, and Pascal each translates source code into the same well-defined intermediate language that ultimately becomes object code. The resulting object module is linked with other object modules and the runtime libraries. The core of these libraries is the same for all three compilers.

**5.5.1 Reasons for Writing Assembly**

Four common reasons for writing assembly language routines are:

1. To optimize for executable speed or size.
2. To access system services, i.e., DOS and ROM BIOS.
3. For direct access to the PC's hardware.
4. To build an interface between otherwise incompatible code.

**Optimizing**

Optimizing for executable speed or size sometimes requires handcrafting the code in assembly language. It is theoretically possible to create a compiler that produces "human grade" code, but it is impractical because the compilation process would be too time-consuming.

One technique used to start writing an assembly language module is to let the compiler generate a "bare bones" program skeleton (i.e., write a procedure in the target language that uses each of the variables in the module in a trivial manner, such as an assignment. This takes care of properly passing variables and allocating local storage). Then, flesh out this skeleton by hand in assembly language.

A second approach is to write the procedure or program in a high-level language, let the compiler turn it into assembly language and then clean up the code using optimizations that for time or safety reasons are not available to the compiler.

A third alternative is to use Microway's Intelligent Assembler. The Intelligent Assembler, an assembly language parser built in to the compiler, can be used to build and maintain assembly language modules under certain situations. See *Section 5.5.2*, page 58.

### Accessing System Services

Accessing system services, both DOS and ROM BIOS, is done using software interrupts. Calling these services used to mean using assembly language. But these days interface routines are often provided by languages for interfacing interrupts, and the NDP family is no exception. The ease of writing in a high-level language is offset by the drawback that inefficiencies inevitably creep in. Again, the programmer may optimize for speed or size by turning to assembly language.

### Accessing PC Hardware

Using assembly language to access directly a PC's hardware, involves is one of the most powerful tools available to the programmer. For example, it is not uncommon to find a 100-to-1 speed difference between routines that write directly into video RAM and those that write to the screen through the ROM BIOS, making direct access very attractive. Even writes directly to the screen, thereby violating a basic principle of operating systems: device drivers should be the only routines allowed to manipulate the hardware directly. We have provided routines callable from high-level source code for block moving characters to the screen; they are `blk_bm` and `blk_mb`. We also should point out that two facilities now available in our languages, the `mapdev` function (which maps a device into the address space of your program) or, if writing in NDP C/C++, using register aliased variables, may be a better alternative in many situations.

Another method of directly accessing the PC's hardware is by reading from and writing to ports. Again, we have provided high-level routines to do this but we have also provided assembly language examples to do the same thing. The examples are provided to help the programmer understand key ideas and issues involved in directly controlling the PC's hardware using assembly language code.

## 5.5.2 Using the Intelligent Assembler to Optimize Code

Microway's Intelligent Assembler is an assembly language parser that is built into the compiler. It allows assembly code to be included in modules written in high-level languages. At parse time, the compiler ignores the assembly section except to determine whether the line is a pragma, a label, or a comment. At code generation, the compiler parses the lines and compares the contents to a list of reserved words and variable names.

Using the NDP Intelligent Assembler has several advantages. The compiler handles the details of writing NDP-compatible code, such as generating header and segment information. The compiler also keeps track of variables, which can be referenced symbolically. Waiting until the code generation phase makes it possible to expand variable names and labels into memory locations because, at that point, the locations are already known. The Intelligent Assembler also understands Weitek code and expands Weitek macros in the same way as the compiler.

The assembler code included in a module with the Intelligent Assembler is subject to the optimizations the compiler makes. This is beneficial for 386 code, but is especially useful for 486 code, where the processor makes intelligent decisions about the 80486 code alignments necessary for optimal performance. The Intelligent Assembler derives its name from this ability.

The Intelligent Assembler has limitations. It does not flag illegal syntax or addressing and is best used on code that is known to be correct. It does not duplicate the functions of a complete assembler. It is often quite instructive to use a complete assembler on code intended for the Intelligent Assembler, to ferret out errors. The Intelligent Assembler is best used to

simplify the inclusion of assembly code -- by letting the compiler generate header and segment information -- and to include assembly code that will undergo optimizations, particularly regarding 80486 code alignments.

### Pragmas to Activate Assembly Language Parsing

Two pragmas activate and deactivate assembly language parsing:

```
#pragma asm on
```

and

```
#pragma asm off
```

Code between the two pragmas is parsed as assembly language by the code generator. The following example illustrates how these work:

```
char str[100]="Oh Annie please be kind, $";

main()
{
    char newstr[100];
    strcpy(newstr,"and kiss me once or twice!\n\r$");
    #pragma asm on;
    lea edx, _str
    mov ah, 9
    int 0x21
    lea edx, _newstr
    mov ah, 9
    int 0x21
    #pragma asm off;
}
```

This program uses DOS function calls to print out two strings. It illustrates the use of the pragmas. It also brings up a point about addressing.

Dereferencing usually requires more than one line of assembly language, so it does not make sense to support it with an assembler.

Unfortunately, many references that look straightforward are not. For example, the `lea` instructions above are quite simple, but if they were `mov` instructions, such as the following:

```
mov edx, dword ptr offset _str
```

and

```
mov edx, dword ptr offset _newstr
```

the second one would not work. The first will work because `_str` is stored in a known memory address; under a debug program, the code produced might look like this:

```
mov edx, 4780
```

`_newstr`, however, is on the stack, and the assembly language expansion of its reference would have to look like this:

```
mov edx, ebp-64
```

This instruction is not legal, and the Intelligent Assembler does the best it can, and produces the following line

```
mov edx, [ebp-64]
```

which is legal, but not what is desired.

Only simple references are allowed; where you want to load an address, consider whether the `lea` instruction or the `mov` instruction is appropriate.



## Assembler Directives

At present the assembler will accept three directives, "\$radix," "\$\$frame" and "\$\$noret." The radix directive is output to a generated assembly file as `.radix x` where `x` is either 16 or 10. Since the value given remains for the rest of the assembly file, and NDP Fortran outputs decimal values, the programmer must remember to leave the radix in decimal for the C/C++ compiler to continue output after the assembly language area is completed. Numbers may be given in hexadecimal by beginning them with a digit in the range of 0-9 and ending with an "h" or, alternately, may be given in standard C/C++ format, as in "0x21," which goes to the assembly language output file as "021h."

The second directive, "\$\$frame," tells the compiler to generate a stack frame for local variables even if it sees that it is unnecessary to do so for the purposes of the C/C++ language. The directive is like the `-ga` command line switch except that it only applies to the function in which it appears rather than globally to a module. It is important to use `$$frame` wherever parameters are referenced and the stack is being manipulated. Without the "\$\$frame" directive, references to parameters are made through `esp`. In an assembly language output file, the directive becomes `;$frame` and is a comment to the Assembler.

The `$$frame` directive is used in the following code to prevent problems involving use of `esp`:

```
dosprint(str);
char *str;
{
    #pragma asm on
    $$frame
    $radix 16
    push  edx
    push  eax
    mov   edx, _str
    mov  ah, 9
    int  21
    pop  eax
    pop  edx
    $radix 10
    #pragma asm off
}
```

Because the `$$frame` directive has been used, the compiler provides `dosprint` with a prologue that builds a stack frame and points `ebp` at it. The reference to `_str` is made through `ebp` in the output code, and the function does as expected. Without the `$$frame` directive, the compiler has no way to know that the stack is manipulated and no stack frame is built. Its code then references `_str` as "[`esp+4`]," which would be correct if `esp` is never changed, but here is wrong, because there are push instructions.

The third directive, "\$\$noret," tells the compiler to omit any "leave" and "ret" instructions that otherwise would be inserted automatically at the end of the function. The purpose of this is to allow processing to fall through from one function to the next, a means for providing alternative entry points or, perhaps more importantly, alternative names for the same procedure.

A single library module can usually be used transparently by both NDP C/C++ and NDP Pascal. NDP Fortran, however, has different enough code that it often needs alternative entry points, and it accesses these by having postpended underscores on its symbolic names. In the following code, the first function has no body and it does not even have a return instruction. This means that in the `.o` file, it has the same address as the second function and occupies the same space. Calls to either function will go to the same place. In the following `print_it_` is the Fortran entry point; `print_it` an entry point for C/C++ or Pascal.

```
void print_it();
print_it_(str)
```

```

char *str;
{
    #pragma asm on;
    $$noret
    #pragma asm off;
}
print_it(str)
char *str;
{
    ...
}

```

### 5.5.3 General Rules

While it is beyond the scope of this manual to teach the reader assembly language, we intend to show the interface between programs written in each of the three NDP languages and assembly language routines.

In writing 386/486 assembly code, keep the following conventions in mind:

1. The code segment is:

```
codeseg segment dword er use32 public 'code'
```

2. The data segment is:

```
dataseg segment dword rw use32 public 'data'
```

3. Parameters are pushed onto the stack in right-to-left order, i.e., the last named is pushed first, then the next to the last, and so on. Thus the arguments end up on the stack in the order in which they are named in the parameter list. (The exception to this rule is the lengths of strings passed by Fortran; see below.)
4. Function values are returned in:

Data type	Register
integer or character	EAX, AX, or AL
pointers	EAX
single precision	ST(0) (80x87)
single precision	FP2 (mW1167)
double precision	ST(0) (80x87)
double precision	FP2, FP3 (mW1167)

It is possible to write assembly language code so that the same routine could be called from any of the NDP languages. To do this, all identifier names should be in lower case, having an underscore at the beginning and end. Parameters, except lengths of strings passed in Fortran, will be passed by reference and the assembly code must be written accordingly. Strings passed from NDP Fortran and NDP Pascal should deliberately have a null byte inserted to terminate the sequence of characters or, alternately, NDP C|C++ programs could explicitly pass the lengths of strings as well as their addresses. Consider the following example:

#### Listing 5-6: 386 Assembly

```

; This code, when called, returns an integer value equal to
; the number of CPU clock ticks since midnight. The routine
; can be called from NDP C|C++, NDP FORTRAN, and NDP Pascal.
; Its function is completely redundant, since it
; operates by calling another function, sec_100, which is
; identical in function, but it does illustrate calling
; conventions.
; The return value will be placed in eax by sec_100, and need
; not be referenced here. FORTRAN requires that this routine
; be declared integer, or it will look for the return value in

```

```

; st(0) or ws(2) depending on coprocessor option.
name tick.s
assume cs:codeseg
assume ds:nothing
codeseg segment para er use32 public 'code'
_ticks_ proc near
    call _sec_100
    ret
    align 16
_ticks_ endp
extrn _sec_100:near
public _ticks_
codeseg ends
end

```

The following statements will call the above routine from the appropriate language.

#### NDP Fortran

```

integer itime, ticks
real*4 seconds
seconds=
& ticks(itime)/100.

```

#### NDP C/C++

```

int itime = 0;
float seconds;
seconds=
    ticks_(itime)/100;

```

#### NDP Pascal

```

var itime :integer;
var seconds :float;
seconds :=
    ticks_(itime)/100;

```

# 6

## Porting Programs

Programs that compile and operate correctly when compiled with one compiler may not operate correctly when ported to another vendor's compiler, such as the NDP line, because of the leeway allowed by the language specifications in implementing certain features of the language. The problem is that many programmers, when porting a program from an IBM mainframe or VAX minicomputer to the PC, make illegal assumptions about the underlying machine architecture and how the compiler interacts with it. The following discussion on porting programs to the NDP compilers describes requirements and tells how to avoid common problems.

### 6.1 Compatibility with other Compilers

The NDP compilers use the same calling conventions for all subroutines, routines, procedures, and functions. Therefore, code from all NDP languages can freely call each other (see *Chapter 5*).

Implementation of the NDP compilers is virtually identical for both DOS and UNIX System V. As a result of this and of the fact that the NDP compilers were adapted from UNIX compilers, programs written with the NDP compilers should run without problem on a VAX or under UNIX System V on a PC.

### 6.2 Word-Size Problems

Some machines are byte addressable. That is, they have addresses that refer to 8-bit bytes. They have operations that operate on 8, 16, 32, 64 and 128-bit quantities. Other machines are word addressable, having addresses that refer to words of a standard size varying from 16 to 64 bits. They have operations that operate on multiples of the word size. The Intel 386 is byte addressable.

If two different machines have different word sizes, or if one is word addressable and the other is byte addressable, a program that operates on one machine may not operate on the other machine for several reasons. The word size affects the range of numbers implemented by the INTEGER data type. The word size also affects the precision and range of the REAL and DOUBLE PRECISION data types.

The most common word-size problems are (often undetected) integer overflows and floating point underflows, and loss of precision. The layout of bit-aligned data structures will vary with the word size, so overlaying structures in memory makes a program difficult to port to another compiler. Another facet of this problem occurs when using integer variables to do address calculations: these calculations are often not portable.

### 6.3 Byte-Order Problems

Since the success of the IBM 360, byte machines have been more popular than word machines. The advantage of a byte machine is its efficient processing of character data. The general acceptance of byte machines has led to easier program portability between machines.

There is, however, one major problem in porting between byte machines. The first successful byte machine, the IBM 360, placed the most significant byte of a multiple-byte integer value at the lowest address. This is known as the big endian method of data representation. Many byte machines, such as those based on the MC68000 and Z8000, have followed the IBM convention.

The second successful byte machine, the Digital Equipment PDP-11, placed the least significant byte of a multiple-byte integer value at the lowest address. This is known as the

little endian method of data representation. Descendants of the PDP-11 such as the VAX, Intel 80x86-based PC's, Clipper, and NS32000, have followed the DEC convention. These two groups seem to be so well entrenched that no agreement on byte ordering is possible. Since the NDP 386/486 compilers operate on Intel 80386/80486-based machines, it is little endian.

Porting applications between machines with different byte ordering is often unreliable if the applications being ported overlay characters and integers in memory or use character pointers to integer variables.

## 6.4 Alignment Requirements

the NDP compilers always align multiple-byte data items on appropriate address multiples so that all accesses will be legal and efficient. The maximum optimal alignment is the largest alignment required by any data type for optimal access. It is typically the word size of the external system bus, which is 32 bits for both the 80386 and the 80486. *Chapter 4* defines the exact alignment conventions for the NDP compilers. By following simple rules, the programmer can prevent illegal and inefficient references.

The compiler always aligns parameters and local variables within the stack at an optimal offset from the beginning of the frame. The compiler also rounds up the size of the frame to the maximum optimal alignment of the processor. If the initial stack pointer is aligned to the maximum optimal alignment of the processor, and if the program involves no explicit (or only correct) manipulation of the stack pointer, then all stack references will be optimal.

All variables within the global frame are allocated at an optimal offset from the base of the global frame. If the assembler or linker allocates the global frame with the maximum optimal alignment of the processor, all global data references will be optimal.

Variables within a frame are optimally packed together in memory. When a data type has an alignment requirement, the least possible unused space is left between the end of the previous item and the next item so that the next item can be optimally aligned. In satisfying different alignment requirements, complex data types may be allocated differently on different machines. This will lead to the usual problems with programs that rely on memory overlays. It also will lead to problems with programs that make implicit assumptions about the size and offset of objects.

## 6.5 Floating-Point Range and Accuracy

The representation of floating-point numbers varies between machines. The range, precision, accuracy, and base vary widely and can lead to portability problems that can be addressed only through the addition of hardware, if at all. For example, single-precision numbers for the 80386/80486 have a 23-bit significand and an 8-bit exponent. In base 10, this insures a range of 10<sup>-37</sup> to 10<sup>38</sup> with a precision of at least 6 significant digits. Double-precision floating-point numbers have a 52-bit significand and an 11-bit exponent, with a guaranteed range from 10<sup>-307</sup> to 10<sup>308</sup> with 15 significant digits.

## 6.6 Assembly Language Interfaces

Programs that use embedded assembly code, or interface to external assembly code, will require all the assembly code to be redone when the program is transported to a new processor. It is a good idea to write as much of the lower-level systems software in a language like C and then optimize, by hand if necessary, the most critical code.

## 6.7 Expression Evaluation Order

The ANSI standard allows a processor to change the order of evaluation of operands in an expression, with certain restrictions. For example, if X and Z are operands of real or double-precision data type, the processor may evaluate  $X*B/Z$  as  $X*(B/Z)$ . The specific values of the

operands may cause the value of the expression to vary in precision depending on the order of evaluation. The standard does prevent a reordering of an expression when, for example, grouping two integers and dividing would cause inappropriate integer division truncation. In the above expression, if *B* and *Z* were integers then  $X*B/Z$  could not be evaluated as  $X*(B/Z)$ . If  $X=5.2$ ,  $B=2$ , and  $Z=3$ ,  $5.2*2/3$  will be evaluated as  $(5.2*2)/3=10.4/3$ , not as  $5.2*(2/3)=5.2*0$ .

A more serious porting problem occurs when the operands of an expression are functions that modify other operands in the expression, or share parameters that they modify. The ANSI standard allows  $x+y$  to be evaluated as  $y+x$ . Assume the following program format:

```
i = double(a) + constant13(a)
```

Procedure `double(a)` sets  $a = a*2$  and returns the new value of  $a$ . If  $a$  was 5,  $a$  is set to 10, and 10 is returned. Procedure `constant13(a)` sets  $a$  to 13. If the compiler evaluates the expression as  $x+y$ , the evaluation will be:

```
i = double(a) + constant13(a)
  = double(5) + constant13(10)
  = 10 + 13
  = 23
```

Note that `double(a)` reset  $a$  to 10 before `constant13(a)` was called.

If, however, the compiler, completely following the ANSI standard, evaluated the expression as  $y+x$ , the result would be:

```
i = constant13(a) + double(a)
  = constant13(5) + double(13)
  = 13 + 26
  = 39
```

Note that `constant13(a)` set  $a$  to 13 before `double(a)` was called. When an expression can be evaluated in different orders, the NDP compiler will evaluate it in left-to-right order.

A similar situation occurs when arguments to a procedure are themselves functions that modify other arguments to the procedure, or arguments to other functions that are arguments to the procedure. For example,

```
a = 5
call sub1(double(a), constant13(a))
```

If `double(a)` is evaluated first,  $a$  is reset to 10, and 10 is bound to the first parameter of `sub1`. `constant13` is evaluated,  $a$  is set to 13, and 13 is bound to the second parameter of `sub1`. The ANSI standard allows the arguments to be evaluated in the reverse order, as well, in which case `constant13(a)` is evaluated,  $a$  is set to 13, and 13 is bound to the second parameter of `sub1`. `double` is then evaluated, but  $a$  is now 13, so it is reset to 26, and 26 is bound to the first parameter. A program that depends on the order in which arguments with side effects are evaluated is non-portable.

The NDP compilers usually evaluate arguments to a procedure in left-to-right order. If a program depends on order of evaluation, that program becomes unportable. Porting it from one compiler to another may generate different results.

## 6.8 Illegal Assumptions About Optimizations

Some programs depend on the exact code that a particular compiler generates. Such programs are particularly difficult to port to advanced optimizing compilers such as the NDP line because optimizers make major changes in the code in order to make the program smaller or faster.

The following are some of the most common illegal assumptions about code generation upon which some programs rely. *Chapter 3* describes in detail the optimizations discussed here.

### 6.8.1 Implied Register Usage

Some programs rely on the exact register allocation scheme used by the compiler. Such programs will not port without modification.

### 6.8.2 Memory Allocation Assumptions

NDP, AT&T, and other vendors have different ways of allocating memory. Because of these differences, problems can arise in porting programs that depend on the memory-allocation peculiarities of other compilers.

Some programs depend upon the compiler allocating variables in memory in the order that they are declared. The NDP compilers will not necessarily allocate variables in the order of declaration.

Some programs depend upon knowing that the compiler will allocate all variables even if they are not used. The NDP compilers may not allocate unused variables.

Some programs depend upon knowing that certain variables will be allocated in memory. The NDP compilers will allocate certain variables to registers that UNIX and other compilers would always allocate to memory. Programs compiled with the NDP compilers must not make assumptions regarding the order of allocation of variables in memory, except where the language standard specifies it.

### 6.8.3 -OM and -OLM Considerations

The `-OM` and `-OLM` compile-time switches (options) should be used only in programs in which memory cannot change except under control of the compiler. Either switch tells the compiler that memory locations cannot change asynchronously with respect to the running program. For example, if the compiler reads or writes some memory location, three instructions later it must be able to assume that the same value is still in that memory location. This would not have to be the case if the memory location were a memory-mapped peripheral.

This brings up a good use of assembly language -- writing device drivers -- and points to an area where the optimizer must be used with caution, i.e., systems software, including many parts of operating systems: device drivers, memory mapped I/O locations, shared memory environments, multiple process environments, interrupt driven routines, and UNIX-style signals. Anyone who has worked on an operating system or developed a device driver will of course be aware of the critical nature of specifying optimization levels.

## 6.9 Problems with Source-Level Debuggers

Once a variable is allocated to a register it will always stay in that register; however, since other variables may share the register, the register may not always contain the value of the variable. This may cause a source-level debugger to give incorrect results. If you ask for the value of a variable at a point at which the variable is about to be assigned into a register, the compiler may have temporarily assigned that register to another variable. Always check results after they are assigned or when the current value is going to be used later. Near the end of a subroutine or function most of the local variables will no longer be used. Thus, the chance that the register has been reallocated is much higher. Use of the `-offr` switch will alleviate this problem; this will force the compiler to keep variables stored in memory.

## 6.10 Problems with Compiler Memory Size

The NDP compilers are advanced optimizing compilers. They are much better than the current generation of "optimizing" microprocessor compilers. In accordance with their greater capability they require more memory. The NDP compilers require over one megabyte just to

load. Compiling requires at least two megabytes of memory; larger programs require more memory.

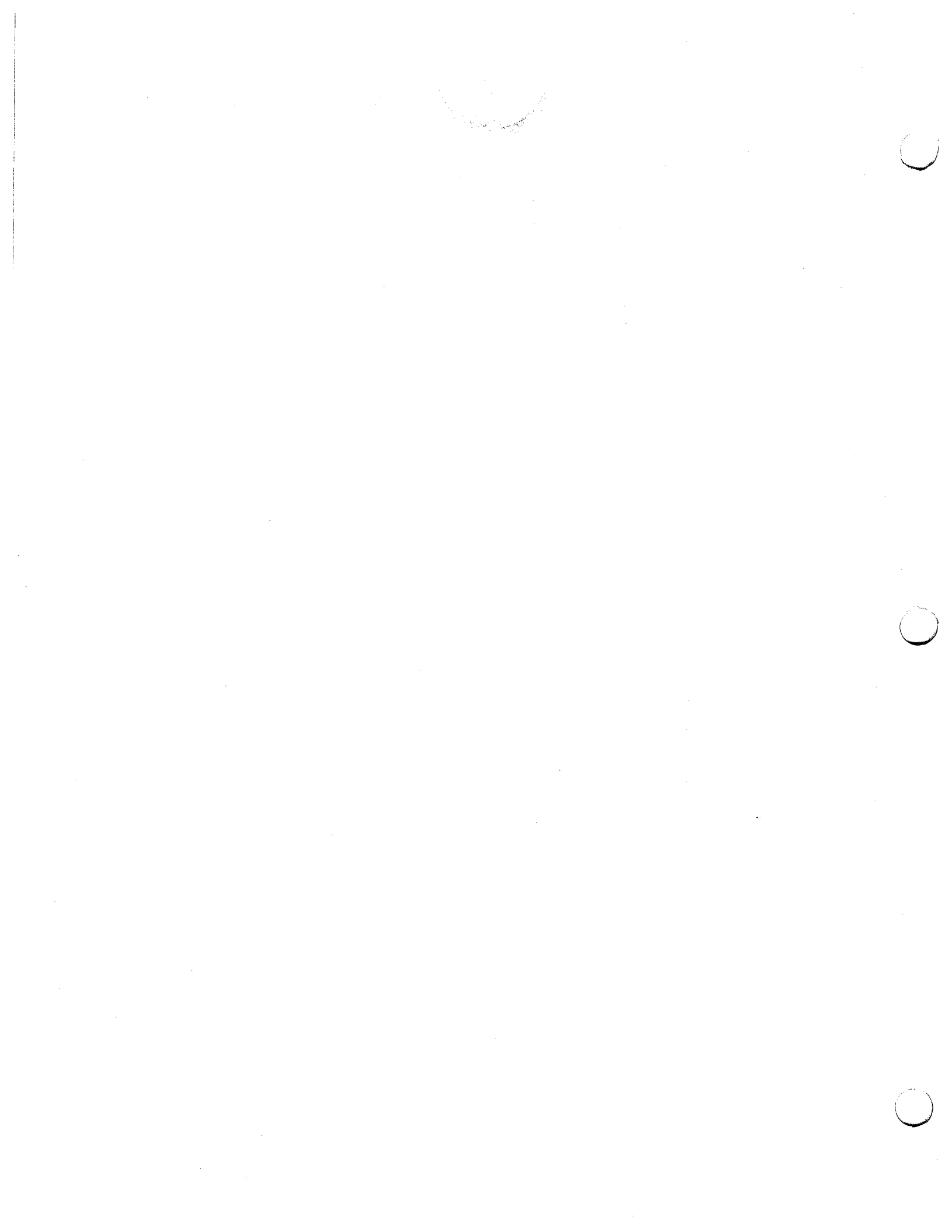
The compiler's primary uses of memory are for the program, static data structures, global declarations, parse trees, and generated machine code. Global declarations consist of the global constant, type, variable, and subroutine or function declarations. This is a major use of memory when large numbers of declarations are included in a compilation. Even unused global declarations must be stored throughout the compilation. If memory size problems exist, try to reduce the size of the include files by including just the declarations that are needed.

The NDP compilers are one-pass compilers in that they read the source program only once. Each subroutine or function is converted into a parse tree as it is read. When the end of the subroutine or function is reached, the optimizer is called with the parse tree as input. The optimizer modifies the parse tree and then passes it on to the code generator.

The code generator produces an internal representation of the machine code to be output for the subroutine or function. Another phase is then called to modify this machine code. Finally, the optimized machine code for the subroutine or function is output. After the machine code is output, the memory being used for the parse tree and machine code is reclaimed for use in compiling the next subroutine or function.

The size of the largest subroutine or function in the program determines the memory usage for parse trees and machine code. If memory size problems exist, turn off the optimizer and reduce the size of the largest subroutine or function. Simple subroutines or functions of fewer than 100 lines should not cause memory size problems. Procedures of more than 1,000 lines may require more than a megabyte of memory to compile.



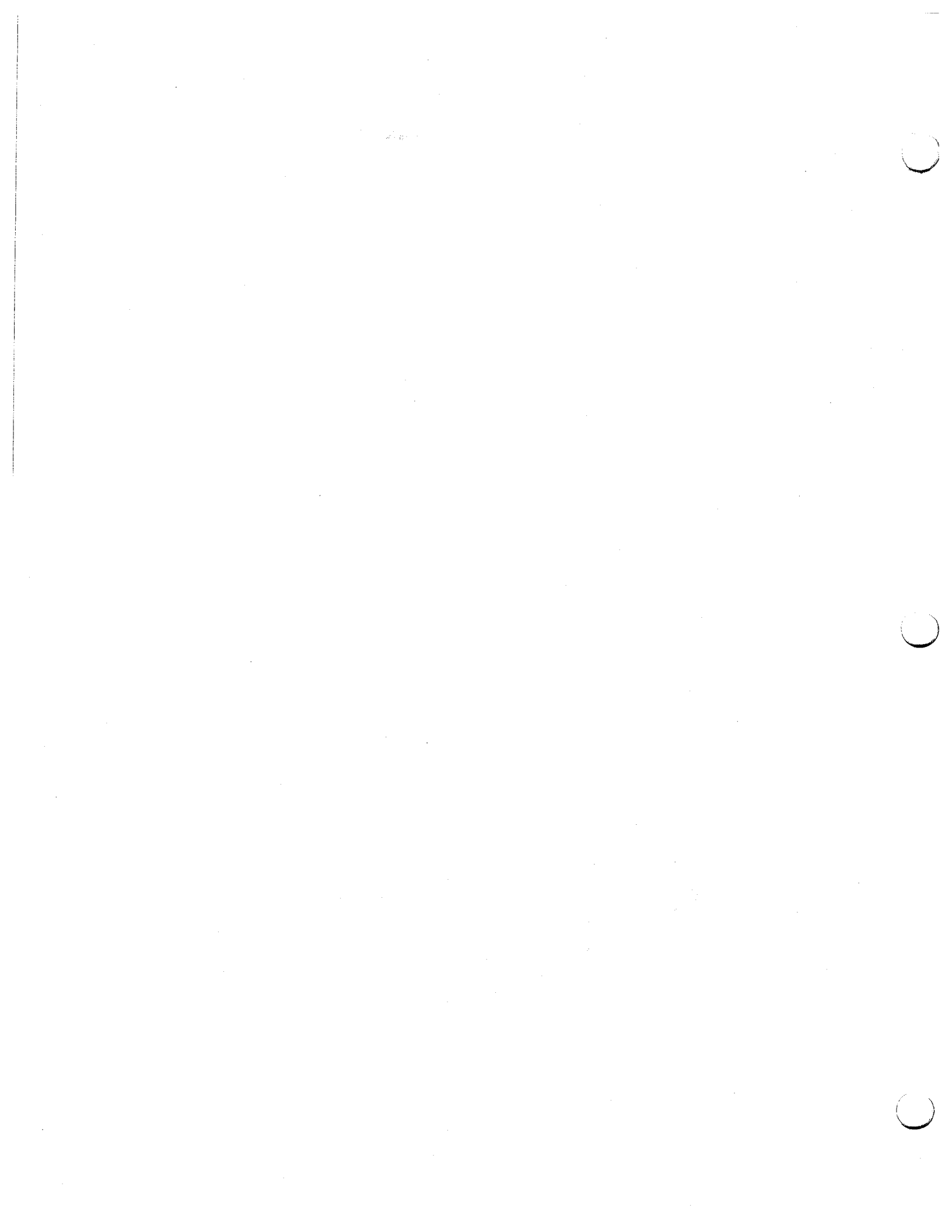


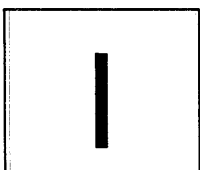
# A

# ASCII Character Set

The following is the ASCII character set, given in ascending order of precedence, with the decimal and hexadecimal equivalent values.

Char	Dec	Hex		Dec	Hex		Dec	Hex
			.	46	2E		^	94 5E
			/	47	2F		-	95 5F
NUL	0	00	0	48	30		'	96 60
SOH	1	01	1	49	31		a	97 61
STX	2	02	2	50	32		b	98 62
ETX	3	03	3	51	33		c	99 63
EOT	4	04	4	52	34		d	100 64
ENQ	5	05	5	53	35		e	101 65
ACK	6	06	6	54	36		f	102 66
BEL	7	07	7	55	37		g	103 67
BS	8	08	8	56	38		h	104 68
HT	9	09	9	57	39		i	105 69
LF	10	0A	:	58	3A		j	106 6A
VT	11	0B	;	59	3B		k	107 6B
FF	12	0C	<	60	3C		l	108 6C
CR	13	0D	=	61	3D		m	109 6D
SO	14	0E	>	62	3E		n	110 6E
SI	15	0F	?	63	3F		o	111 6F
DLE	16	10	@	64	40		p	112 70
DC1	17	11	A	65	41		q	113 71
DC2	18	12	B	66	42		r	114 72
DC3	19	13	C	67	43		s	115 73
DC4	20	14	D	68	44		t	116 74
NAK	21	15	E	69	45		u	117 75
SYN	22	16	F	70	46		v	118 76
ETB	23	17	G	71	47		w	119 77
CAN	24	18	H	72	48		x	120 78
EM	25	19	I	73	49		y	121 79
SUB	26	1A	J	74	4A		z	122 7A
ESC	27	1B	K	75	4B		{	123 7B
FS	28	1C	L	76	4C			124 7C
GS	29	1D	M	77	4D		}	125 7D
RS	30	1E	N	78	4E		~	126 7E
US	31	1F	O	79	4F		DEL	127 7F
SP	32	20	P	80	50			
!	33	21	Q	81	51			
"	34	22	R	82	52			
#	35	23	S	83	53			
\$	36	24	T	84	54			
%	37	25	U	85	55			
&	38	26	V	86	56			
'	39	27	W	87	57			
(	40	28	X	88	58			
)	41	29	Y	89	59			
*	42	2A	Z	90	5A			
+	43	2B	[	91	5B			
,	44	2C	\	92	5C			
-	45	2D	]	93	5D			





# Index

- 80387
  - register set 33
- Active error handling 40
- Alignment requirements 64
- ASCII character set 69
- Assembler
  - directives 60
    - \$\$frame (stack frame) 60
    - \$\$noret (omit return instructions) 60
    - \$radix (radix directive) 60
  - routines 49, 57
- Assembly
  - code
    - conventions for writing 61
  - language
    - common routines 61
    - interfaces 64
    - parser 58, 59
    - rationale for 57
- Biased exponent 43
- Big endian 63
- Binary
  - fraction 42
  - real number 43
- Byte order 63
  - porting problems 64
- Byte-addressable machines 63
- Calling
  - between Fortran and C 53
  - between Fortran and Pascal 55, 56
  - between NDP C/C++ and NDP Pascal 56
- Case sensitivity 9, 51, 56
- Cleanup code 22
- Code
  - generator 67
- Common Subexpression Elimination (CSE) 8
- Common subexpressions
  - elimination 21
- Compatibility
  - among NDP compilers 63
- Compiler
  - memory size 66
  - NDP 49
    - compatibility among 49
    - differences among 49
    - runtime systems 49
  - one-pass 67
  - switches 6
- Compiler driver 5
  - extensions 5
- syntax 5
- Compilers
  - optimizations 5
- Condition code 34
  - defining operand class 35
- Constant expression folding 20
- Constant propagation 20
- Cross jumping 8, 21
- Cybernetic realm 40
- Data registers
  - 80387/80487 33
  - Weitek 38
- Data types 27, 50
  - char 51
  - double-precision real 26
  - integer 50
  - single-precision real 25
- Dead code
  - elimination 19
- Dead code elimination 8
- Denormal 45
- Division
  - by infinity 40
  - by zero 40, 41
- DO loop 8
- Double real encodings 26
- Double-precision real 26
- Endian data representation 63
- Environment variables 2
- Epilog optimization 15
- Errors 40
  - handling
    - active 40, 41
    - passive 40
- Exception 40
  - flag 36
  - handler
    - creating 42
    - default 40
  - handling
    - active 41
    - passive 41
    - placing checkpoints 41
  - invalid operation 40
  - masking 40
  - masks 36
  - NDP 42
  - unmasking 40
- Exponents 44
- Expression evaluation order 64
- Extension

- .s 6
- File
  - input type 5
- Flags register 31
- Floating point 25, 51
  - accuracy and range 64
  - invariant expressions 8
  - number 25
    - 32-bit binary 25
    - 64-bit binary 26
    - binary 42
    - IEEE real 44
  - number systems 42
  - parameters 51
  - precision 45
  - underflows 63
- Frame
  - global 64
- Function
  - mapdev 58
- General purpose registers 29
- Geterrno 39
- Global frame 64
- Hardware access 58
- Identifier names 51
  - by language 51
  - C/C++ 51
  - Fortran 51
  - global 51
  - Pascal 51
- IEEE
  - 754 25
  - 854 25
  - number system 40, 42
  - reals 42
- Illegal assumptions about code generation 65
  - and use of -OM and -OLM 66
  - memory allocation 66
  - reliance on exact register allocation 66
- Implied register usage 66
- Include files 5
- Infinities 46
- Infinity 46
- Inline multiplication and division 19
- Inliner 23
- Inlining 39
- Integer
  - 16-bit 50
  - data type 25
  - overflows 63
- Integer types, 80x86
  - integer 25
  - long integer 25
  - short integer 25
  - signed char 25
  - unsigned char 25
  - unsigned integer 25
  - unsigned long integer 25
  - unsigned short integer 25
- Intel
  - numeric coprocessors 40
- Intelligent Assembler 58
- Interfacing
  - NDP languages 49
    - with assembly language 57
  - routines 58
    - with assembly language 61
- Internal
  - registers 28
- Interrupt
  - software 58
- Invalid operation 41, 42
  - exception 40
- Library routine 39
- Linking
  - restrictions 49
    - NDP Fortran modules 49
    - NDP Pascal modules 50
- Little endian 25, 63
- Live/dead analysis 21
- Long integer 25
- Loop
  - index variable 17
  - invariant analysis 16
  - optimizations 8
  - rotation 16, 17
  - strength reduction 17
  - unrolling 8, 19, 22
    - advantages 22
    - costs 22
    - disadvantages 22
- Lower level characteristics 25
- Machine code
  - optimized 67
- Mapdev function 58
- Mathematics realm 40
- Memory
  - allocation
    - assumptions 66
    - optimizations 8
    - usage 67
  - used by NDP Fortran compiler 67
- Memory allocation 12
- Mixing languages 49
  - output buffers 53
- Multiple-byte data items
  - alignment 64
- Naming conventions 51
- NaN 41, 42, 43, 46
  - indefinite real number 42
- NDP
  - compilers 49
    - compatibility 49
    - data type differences 50

- differences 49
- linking 49
- naming conventions 51
- output buffers 53
- Nested functions 24
- Normalization 43
- Normals 46
- Null byte 53
- Numeric error 40
- Numeric exception 40
  - denormal operand 40
  - handling 41
  - invalid operation 40
  - NDP Fortran 41
  - overflow 40
  - precision 40
  - underflow 40
  - within library routine 39
  - zero divide 40
- Numeric exceptions 39
- Numerics 25
- Object
  - module 57
- Optimizations 11
- Optimized machine code 67
- Optimizing 57
  - using assembly language 57
- Output
  - buffers 53
- Parameter passing 52, 53, 55
  - by reference 52
  - by value 52
- Parse tree 67
- Parser
  - assembly language 58
- Pascal 51
- Passing parameters 52, 53, 55
  - by reference 52
  - by value 52
- Passing strings 52, 53
  - by value 52
- Passing values 52
- Passive error handling 40
- PC hardware
  - accessing 58
- Peephole optimization 15
- Peephole optimizations
  - onrepeep 9
- Porting programs 63
  - aligning multiple-byte data items 63
  - memory allocation 65
  - to NDP Fortran from IBM mainframe 63
  - to the PC from VAX minicomputer 63
  - OM and -OLM switches 66
- Ports
  - reading from and writing to 58
- Pragmas 59
- Precision
  - and denormals 45
- Print\_it\_ 60
- Program
  - checkpoints 41
  - development 5
  - speed 17
- Prolog optimization 15
- Protected-mode
  - segments 30
- Push 49
- Range and accuracy
  - floating point 64
- Recursion 23
- Register
  - 80387 data registers 33
  - 80387 register set 33
  - allocation by coloring 12
  - caching 8
  - control word 36
  - flags 31
  - general purpose 29
  - internal 29
  - segment 30
  - status word 36
  - systems control 32
  - Weitek 38
- Register coalescing 14
- Runtime
  - organization and numerics 25
  - system 49
- Segment
  - addresses 30
  - registers 30
  - selectors 30
- Short integer 25
- Signed char 25
- Single and double real encodings 26
- Single real encodings 26
- Single reals 45
- Single-precision real 25
  - rules 25
- Software interrupts 58
- Source-level debuggers 66
- Speed optimizations 15
- ST(0) 33
- Stack
  - order 49, 61
- Static address elimination 13
- Status Word Register 33
- Strength reduction 17
- String handling 53
- Strings
  - passing
    - between Fortran and C 52
    - Fortran to Pascal 53
    - Pascal to Fortran 53

- System
  - services
    - accessing DOS 58
    - accessing ROM BIOS 58
- System requirements 1
- True exponent 43
- Underscores 60
  - leading 51
  - trailing 51
- UNIX V 63
- Unsigned
  - char 25
  - integer 25
  - long integer 25
  - short integer 25
- Unused variables 66
- Variables
  - unused 66
- VAX VMS
  - Fortran compatibility 9
- Warning messages, suppress 9
- Weitek
  - architecture 38
  - coprocessors 38
  - data registers 38
  - numeric coprocessors 40
  - process context register 38
- Word-addressable machines 63
- Word-size problems 63
  - floating-point underflows 63
  - integer overflows 66
  - loss of precision 63
- WTL Register File 38
- Zero
  - division by 40
- 2.1 6
- ansi 6, 51
- ansiconform 6
- c 6
- cg1 6
- cg2 6
- cg3 6
- cg4 6
- cg5 6
- cg6 6
- cg7 6
- clink 7
- cpplink 7
- Dname 7
- Dname=text 7
- f1 7
- f2 7
- f3 7
- f4 7
- f5 7
- f6 7
- f7 7
- fdiv 7
- flink 7
- g 7
- ga 7
- hasm 7
- i2 7
- i4 7
- ident1 7
- ident2 7
- Idir 7
- lf3 50
- lf3w 50
- list 7
- lname 7
- minit 7
- n0 7
- n1 7
- n2 7
- n3 7
- n4 8
- n5 8
- n6 8
- n7 8
- n8 8
- nof77 8
- O 8
- o name 8
- off 8
- offa 8
- offcse 8
- offh 8
- offn 8
- offp 8
- offr 8
- offs 8
- OL 8
- OLM 8, 66
- OM 8, 66
- on 8
- on2cse 8
- onetrip 8
- onlr 8
- onrc 8
- onrepeep 9
- onw 9
- p 9
- p1 51, 56
- p3 52
- plink 9
- rtl 9
- rt2 9
- rt3 9
- rt4 9
- S 9
- u 9
- uname 9
- ur=# 9

- v 9
- vms 9
- vmsi 9
- W 9
- Wa,toggle 9
- Wl,toggle 9





# **NDP Pascal Reference Manual**

***Microway***<sup>®</sup>

Research Park  
Box 79

Kingston • Massachusetts 02364 • USA

NDP Fortran, NDP Pascal, NDP-VMEM, NDP Link, NDP Run, and Microway are trademarks of Microway, Inc.

UNIX is a registered trademark of UNIX Systems Laboratories, Inc.

Intel, SX, 287, 386, 387, 486, i486, and i860 are trademarks of Intel Corporation.

Microsoft, MS-DOS, and Windows are registered trademarks of Microsoft Corporation.

OS/2 is a trademark of IBM corporation.

Phar Lap, 386|DOS-Extender, and 386|VMM are trademarks of Phar Lap Software, Inc.

Weitek is a trademark of Weitek Corporation.

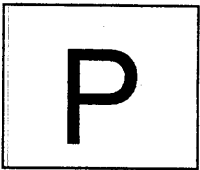
# Contents

<b>PREFACE</b>	<b>1</b>
Manual Objectives	1
Pascal Syntax Diagrams	1
A Final Request	2
<b>Base Vocabulary</b>	<b>3</b>
1.1 Identifiers	3
1.2 Reserved Words	4
1.3 Keywords	4
1.4 Special Symbols	4
1.5 Comments	5
1.6 Predefined Constants	6
1.7 Predefined Types	6
1.8 Predefined Variables	6
1.9 Predefined Functions	6
1.10 Predefined Procedures	6
1.11 Preprocessor Commands	7
1.12 Constants (unsigned integer, unsigned number, unsigned constant)	7
<b>Program Structure</b>	<b>11</b>
2.1 The Structure of Programs	11
2.2 The Lexical Scope of Identifiers	12
2.3 Declaration Order	13
2.4 Program Modules for Separate Compilation	14
<b>Pascal Declarations</b>	<b>15</b>
3.1 Program Heading (PROGRAM)	15
3.2 Label Declaration (LABEL)	15
3.3 Constant Definition (CONST)	16
3.4 Type Definition (TYPE)	16
3.5 Variable Declaration (VAR)	17
3.6 External Directive (EXTERNAL)	19
3.7 Static Directive (STATIC)	21
<b>Type Definitions</b>	<b>23</b>
4.1 Type Definitions	23
4.2 Type Compatibility and Conversions	24
4.2.1 Identical Types	24
4.2.2 Compatible Types	24
4.2.3 Assignment Compatibility	24
4.2.4 Implicit Type Conversion	25
4.3 Enumerated Types	26
4.4 The Subrange Type	27

4.5 The SET Type	27
4.6 Predefined Scalar Types: BOOLEAN, CHAR, INTEGER, DOUBLE, REAL, FLOAT	28
4.6.1 Operations and Functions for the BOOLEAN Scalar Type	29
4.6.2 Operations and Functions on the CHAR Scalar Type	30
4.6.3 Operations and Functions on the INTEGER Scalar Type	30
4.6.4 Operations and Functions of the REAL, FLOAT, and DOUBLE Scalar Types.	32
4.7 Array Type	33
4.8 Record Types	34
4.8.1 Accessing a Field	34
4.8.2 The Fixed Part	35
4.8.3 The Variant Part	35
4.8.4 Packed Records	35
4.9 Pointer Type	36
4.9.1 Operations on Pointers	37
4.10 File Type	37
4.10.1 Predefined File Type TEXT	38
4.11 Packed and Unpacked Types	39
<b>Variables</b>	<b>41</b>
5.1 Entire variables	41
5.2 Component Variables	41
5.2.1 Indexed Variables	41
5.2.2 Field Designators	42
5.2.3 File Referencing	43
5.3 Pointer Referencing	43
<b>Expressions</b>	<b>45</b>
6.1 Operators	46
6.2 Boolean Expressions	47
6.3 Function Call	48
6.4 Set Constructor	48
<b>Statements</b>	<b>51</b>
7.1 Statement Summary	51
7.2 The ASSIGNMENT Statement	51
7.3 The CASE Statement	52
7.4 The COMPOUND Statement	53
7.5 The EMPTY Statement	54
7.6 The FOR Statement	55
7.7 The GOTO Statement	56
7.8 The IF Statement	57
7.9 The PROCEDURE Statement	58
7.10 The REPEAT Statement	59
7.11 The WHILE Statement	60
7.12 The WITH Statement	60
<b>Procedures and Functions</b>	<b>63</b>
8.1 Procedure and Function Declarations	63
8.2 Parameter Transmission	64

8.2.1 Value Parameters	65
8.2.2 Variable Parameters	65
8.2.3 Formal Routine Parameters	65
8.3 Function Results	66
8.4 The FORWARD Directive	66
<b>Input and Output</b>	<b>71</b>
9.1 Overview	71
9.2 File Declaration and Initialization	71
9.3 Input and Output Processing using GET and PUT	73
9.4 Buffer Variable Restrictions	74
9.5 Input and Output Processing with READ and WRITE	76
<b>Predefined Functions and Procedures</b>	<b>79</b>
<b>Preprocessor Commands</b>	<b>95</b>
<b>Selected Bibliography</b>	<b>99</b>
<b>Interface to C and Math Libraries</b>	<b>101</b>
Overview	101
The Standard Pascal Library	101
The Math and C Libraries	101
Contents of the Math and C libraries	101
Miscellaneous Mathematical Functions	102
<b>NDP Pascal Error Messages</b>	<b>185</b>
Overview	185
C.1. Compile Time Error Messages	185
<b>Index</b>	<b>197</b>





# PREFACE

## Manual Objectives

The purpose of this manual is to present a complete description of Microway's implementation of the Pascal language. This manual describes the syntax and semantics of NDP Pascal. This manual is a reference document and is intended for people familiar with the Pascal language.

NDP Pascal implements the ANSI/IEEE standard 770X3.97-1983, a superset of Niklaus Wirth's Pascal. It includes several extensions from Berkeley 4.2 BSD Pascal and the British Standards Institute (BSI) Level 0, a preprocessor, separate compilation of modules, and interfaces to our C library.

## Pascal Syntax Diagrams

This manual describes the syntax and semantics of NDP Pascal using explanations accompanied by syntax diagrams and programming examples. The syntax is described by using a meta-language consisting of circles, ovals, and rectangles that are connected by arrows. This pictorial representation of Pascal's syntax rules is called a syntax diagram. The purpose of the syntax diagram is to give a simple, concise, and unambiguous description of the language. An explanation follows each syntax diagram describing the meaning of any symbols used, and restrictions not shown in the diagram.

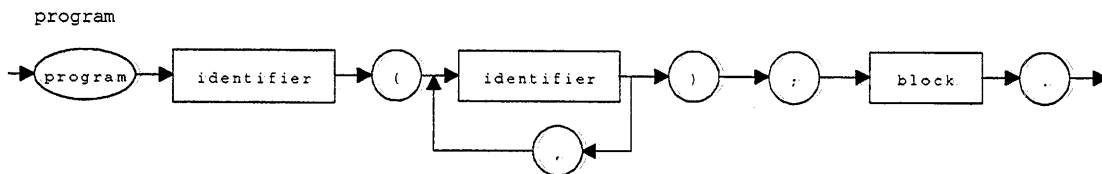
Each syntax diagram represents one or more syntax rules in NDP Pascal. The title of the diagram is the name of the syntax rule being defined. Arrows are used to show the entry and exit points of the diagram, as well as the legal paths.

A circle or oval is used to surround a symbol that is part of the Pascal language. This may be a special character, a punctuation mark, a keyword or a reserved word. A rectangle is used to enclose the name of another syntax rule that is described somewhere else in this manual. Finally, arrows are used to connect the circles and rectangles and establish the order in which these symbols must follow one another.

The title of the diagram is the name of the syntax rule being defined. A Pascal construct is formed by following the arrows around the diagram, from the beginning to the end, and concatenating the symbols encountered along this traversal.

### EXAMPLE

The following syntax diagram summarizes the definition of a Pascal program.



The items in the circles and oval represent tokens of the Pascal language, and must be entered exactly as shown. The rectangular boxes enclose the name of syntax rules that are described in another part of this manual. The arrows show that a Pascal program consists of the keyword PROGRAM, followed by an identifier, optionally followed by a parenthesized list of one or more identifiers, separated by commas. This is followed by a semicolon, a block, and finally, a period.

The purpose and meaning of the different identifiers that are possible are described in the text following the syntax diagram. In this instance, the first identifier is the program name, while the list of identifiers within the parentheses represent names used by the programmer to



indicate the program's interaction with its environment. These are traditionally the names of files accessed by the program, but NDP Pascal places no restriction on the meaning of these identifiers.

## A Final Request

This reference manual uses as many examples as possible so that each language element may be quickly and easily understood. The examples consist of programs, procedures, and functions that use a variety of Pascal constructs to solve standard problems in a natural way. We try to avoid contrived examples, so if you have a clean, crisp program that illustrates an important feature of Pascal, then we would appreciate hearing from you.

Additional copies of this reference manual may be obtained by contacting Microway as follows:

Microway, Inc.  
Research Park  
P.O. Box 79  
Kingston, MA 02364  
United States

PHONE: +508/746-7341  
FAX: +508/746-4678

# 1

## Base Vocabulary

### 1.1 Identifiers

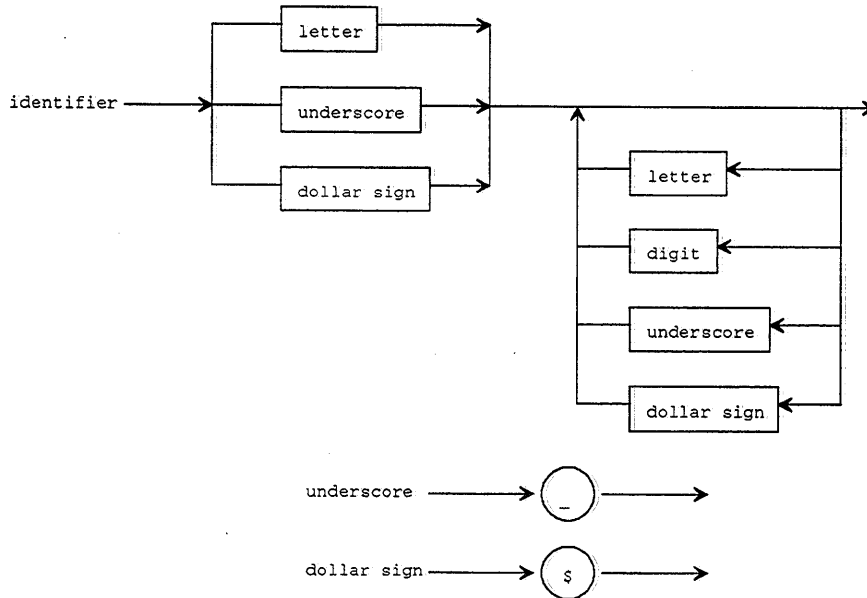


Figure 1-1 Syntax Diagrams for Letter, Digit and Hex Digit

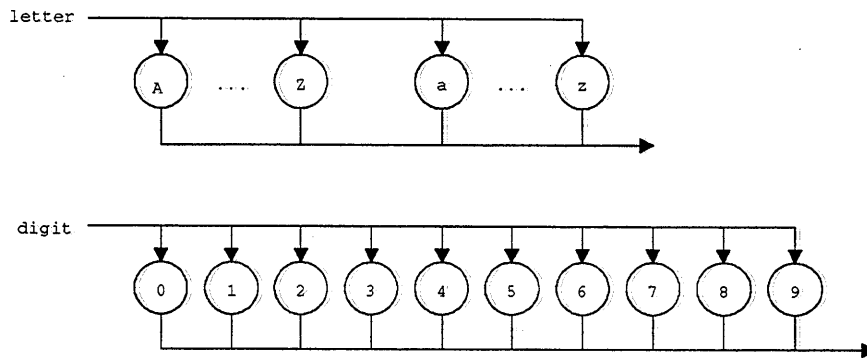


Figure 1-2 Syntax Diagram of an Identifier

An identifier is the name used for program constants, data types, variables, procedures, and functions. Identifiers may begin with a letter, a dollar sign (\$), or an underscore (\_), and no distinction is made between upper and lower case letters. (Identifiers may be made case sensitive through the use of a compiler switch. See the *NDP User's Manual* for details.) Identifiers may not begin with a digit or include either a question mark (?) or period (.). The maximum length of an identifier is 132 characters and all are significant. An identifier cannot have the same spelling as a reserved word (*Section 1.2*).

The use of long identifiers of up to 132 characters may conflict with the requirements of the assembler. Refer to your assembler reference manual for details. Also, identifiers that are to be made external must conform to the requirements of the host operating system.

The following are valid identifiers:

```
SSN
frequency
input_buf
$dollar
_u238
```

The following are invalid identifiers:

```
2k           { cannot start with a digit}
ice  cream   { embedded spaces are not allowed}
which_way?   { question mark is not allowed}
repeat       { reserved words may not be used}
```

## 1.2 Reserved Words

The following is a list of reserved words used to define the syntax of NDP Pascal. These words may not be defined as identifiers in a program. The reserved words must be separated from other language constructs by a special symbol (*Section 1.4*), a comment, or one or more spaces.

and	end	nil	repeat
array	file	not	set
begin	for	of	then
case	function	or	to
const	goto	otherwise	type
div	if	packed	until
do	in	procedure	var
downto	label	program	while
else	mod	record	with

Table 1-1 NDP Pascal -- Reserved Words

## 1.3 Keywords

The following is a list of keywords used to define the syntax of NDP Pascal. *The difference between a reserved word and a keyword is that a reserved word may not be used as an identifier, while a keyword may be used as an identifier.* Keywords must be separated from other language constructs by a special symbol, a comment, or one or more spaces.

NDP Pascal's keywords are as follows:

```
forward           external           static
```

## 1.4 Special Symbols

The table below summarizes the mathematical and notational symbols used by NDP Pascal.

Symbol	Meaning
+	addition, set union
-	subtraction, set difference
*	multiplication, set intersection
/	real division
~	Boolean not, set complement, one's complement on type INTEGER
	Boolean or, logical or on type INTEGER
&	Boolean and, logical and on type INTEGER, address of
=	equality
>	greater than

<	less than
<=	less than or equal
>=	greater than or equal
<>	not equal, Boolean exclusive or
<<	logical left shift
>>	logical right shift
:=	assignment operator
+=	"x += y" is equivalent to "x := x + y"
-=	"x -= y" is equivalent to "x := x - y"
*=	"x *= y" is equivalent to "x := x * y"
/=	"x /= y" is equivalent to "x := x / y"
=	"x  = y" is equivalent to "x := x   y"
&=	"x &= y" is equivalent to "x := x & y"
<<=	"x <<= y" is equivalent to "x := x << y"
>>=	"x >>= y" is equivalent to "x := x >> y"
..	indicates a subrange
.	period, indicates the end of a program or field specification within a record
,	comma, used to separate items in a list
:	colon, used as a separator in declarations, labels, and case statement
;	semicolon, used to separate statements and routine parameters
'	single quote, used to define character constants
^	caret, pointer symbol
[	left square bracket, array indexing operator
]	right square bracket
(	open parenthesis, function and procedure declaration and call
)	close parenthesis (same as "open parenthesis")
{	left curly bracket, open comment
}	right curly bracket, close comment
(*	open comment
*)	close comment

Table 1-2 NDP Pascal -- Special Mathematical and Notational Symbols

## 1.5 Comments

Comments are set off from the program text by using either the curly braces or the open and close comment symbols. A comment may be placed anywhere in the program text where a blank could be used. Comments may not be nested. However, a single close comment symbol will terminate one or more open comment symbols.

### EXAMPLES

```
{ This is a perfectly good comment. }
```

```
(* as is this *)
```

```
{ Comment symbols may be mixed,      *}
```

```
(* and matched. . . . }
```

```
{ Further, comments may appear on
```

```
  any number of lines. . . .
```

```
  .
```

```
  .
```

```
  }
```

```
{ this is an {illegal} nesting of comment symbols }
```

```
{ this is also (* illegal *) }
```

```
{ however, this { is OK }
```

```
function integrate{using trapezoids}(a,      {lower bound}
                                   b:real;   {upper bound}
                                   N:integer;  {# intervals}
                                   ):real;   {result}
```

## 1.6 Predefined Constants

FALSE      constant of type BOOLEAN  
 MAXINT    maximum constant of type INTEGER: 2147483647.  
 TRUE      constant of type BOOLEAN

## 1.7 Predefined Types

BOOLEAN    logical data type  
 CHAR      character data type  
 INTEGER    integer data type  
 FLOAT     floating point data type represented in 64 bits  
 REAL      floating point data type represented in 32 or 64 bits  
 DOUBLE    floating point data type represented in 64 bits  
 TEXT      file of type CHAR

## 1.8 Predefined Variables

INPUT      default input file  
 OUTPUT    default output file

## 1.9 Predefined Functions

The following is a list of the predefined functions in NDP Pascal. A detailed description of these functions is in *Chapter 10*.

ABS ( $x$ )            returns the absolute value of  $x$   
 ARCTAN ( $x$ )        returns the arctangent of  $x$   
 ARGV                returns the number of command line arguments  
 CHR ( $n$ )            returns the ASCII character whose ordinal value is  $n$   
 COS ( $x$ )            returns the cosine of  $x$   
 EOF ( $f$ )            returns TRUE if file  $f$  is at end of file  
 EOLN ( $f$ )          returns TRUE if file  $f$  is at end of line  
 EXP ( $x$ )            returns the base of the natural log (e) raised to the power  $x$   
 LN ( $x$ )             returns the natural logarithm of  $x$   
 ODD ( $n$ )            returns TRUE if the integer  $n$  is odd  
 ORD ( $x$ )            converts a scalar value  $x$  to an integer  
 PRED ( $x$ )            returns the predecessor of the scalar  $x$   
 ROUND ( $x$ )          converts a floating point  $x$  to an integer by rounding  
 SIN ( $x$ )            returns the sine of  $x$   
 SQRT ( $x$ )            returns the square root of  $x$   
 SQR ( $x$ )            returns the square of  $x$   
 SUCC ( $x$ )            returns the successor of the scalar  $x$   
 TAN ( $x$ )            returns the tangent of  $x$   
 TRUNC ( $x$ )          converts a floating point  $x$  to an integer by truncating

## 1.10 Predefined Procedures

The following is a list of the predefined procedures in NDP Pascal. A detailed description of these procedures is in *Chapter 10*.

ARGV ( $i, s$ )            copies the  $i^{\text{th}}$  command line argument into the variable  $s$

DISPOSE ( <i>p, t1, ...</i> )	deallocates a dynamic variable
GET ( <i>f</i> )	advances file pointer and assigns file component to buffer variable
NEW ( <i>p, t1, ...</i> )	allocates a dynamic variable
PACK ( <i>a, i, z</i> )	packs array <i>a</i> , beginning at index <i>i</i> , into array <i>z</i>
PAGE ( <i>f</i> )	writes an ASCII form feed to file <i>f</i>
PUT ( <i>f</i> )	copies buffer variable to the end of file <i>f</i>
READ ( <i>f, v</i> )	reads data from file <i>f</i> into variable <i>v</i>
READLN ( <i>f, v</i> )	reads data from file <i>f</i> into variable <i>v</i> , then advances to end of line on file <i>f</i>
RESET ( <i>f, s</i> )	opens a file for input
REWRITE ( <i>f, s</i> )	opens a file for output
UNPACK ( <i>z, a, i</i> )	copies packed array <i>z</i> , to array <i>a</i> , beginning at index <i>i</i>
WRITE ( <i>f, e</i> )	writes the value of <i>e</i> to file <i>f</i>
WRITELN ( <i>f, e</i> )	writes the value of <i>e</i> , and then an end of line to file <i>f</i>

## 1.11 Preprocessor Commands

The following is a list of commands interpreted by the preprocessor. A detailed description of these commands is in *Chapter 11*.

#DEFINE <i>n s</i>	replaces a name <i>n</i> with a string of characters <i>s</i>
#UNDEF <i>n</i>	cancels the previous #DEFINE on the name <i>n</i>
#INCLUDE <i>f</i>	redirects compiler input to a supplementary file <i>f</i>
#IF <i>e</i>	evaluates the text following this statement if expression <i>e</i> is nonzero or TRUE
#IFDEF <i>n</i>	evaluates the text following this statement if the name <i>n</i> is defined
#IFNDEF <i>n</i>	evaluates the text following this statement if the name <i>n</i> is not defined
#ELSE	evaluates the text following this statement if the result of the previous #IF, #IFDEF, or #IFNDEF was zero or FALSE
#ENDIF	terminates an #IF, #IFDEF, or #IFNDEF statement
#LINE <i>c f</i>	reports an error message occurring on following line as appearing in file <i>f</i> on line number <i>c</i>

## 1.12 Constants (unsigned integer, unsigned number, unsigned constant)



Figure 1-3 Syntax Diagram for unsigned integer

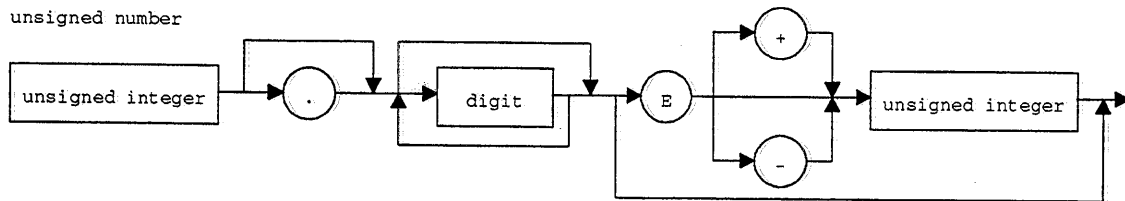


Figure 1-4 Syntax Diagram for unsigned number

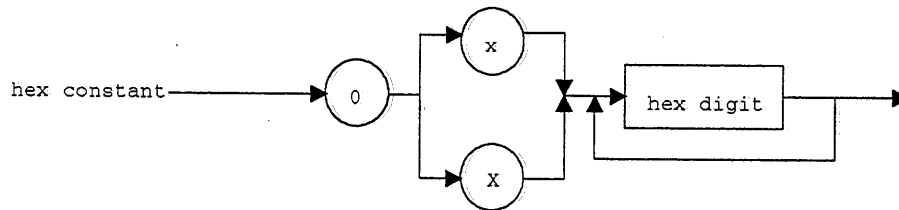


Figure 1-5 Syntax Diagram for hex constant

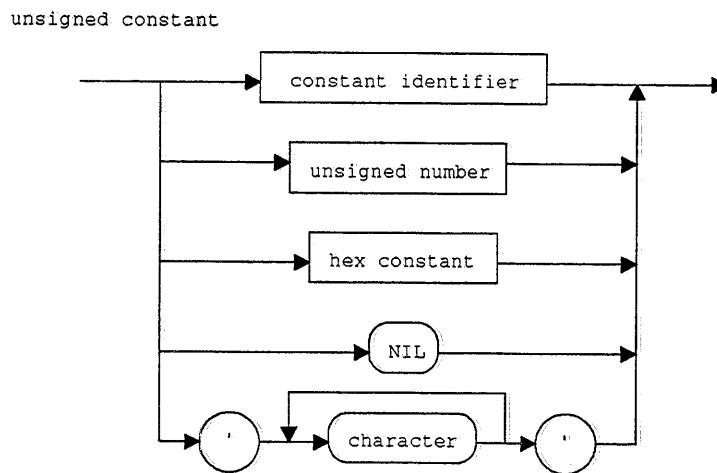


Figure 1-6 Syntax Diagram for unsigned constant

**Explanatory note on "character":**

character corresponds to the character equivalent of the decimal ASCII codes from 32 to 126 in Appendix B of the *NDP User's Manual*.

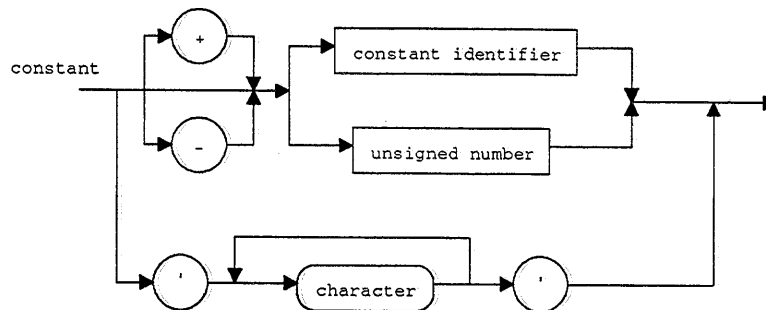


Figure 1-7 Syntax Diagram for constant

An unsigned integer is a sequence of digits. An unsigned number is an unsigned integer with either a decimal point or a scale factor, or both. An unsigned number may end with a decimal point, but if a number less than one is to be represented, it must begin with a digit. (Standard Pascal requires a digit before and after a decimal point in real numbers, a restriction that is relaxed in NDP Pascal.)

A string is a sequence of characters enclosed by quotation marks. The value of a string constant is the sequence of characters enclosed within the quotes. For the purposes of type compatibility, strings are divided into two groups: strings of length one, and strings of length greater than one. A string consisting of a single character is a string of length one and is identical to the type `char`. A string consisting of  $n$  characters is identical to the type definition:

```
packed array [1..n] of char;
```

When an apostrophe is to be used in a character string, it must be written twice. String constants are case sensitive so that upper and lower case letters must be carefully distinguished by the user. An end of line character may not appear in a character constant.

The constant `NIL` is a reserved word and represents a pointer constant that does not point to anything. `NIL` is compatible with any type definition.

The constants `TRUE` and `FALSE` are predefined Boolean scalar constants.

### EXAMPLES

The following are valid constants:

```
1024          { an unsigned integer   }
3.14159       { unsigned numbers     }
0.57721
1.2345e4
1.2345e-8

NIL           { unsigned constants   }
'four score'
SSN
input_buf

-frequency    { constants           }
+2.71828

0x80000000   { hex constants        }
0x0000FFFC
```

The following are invalid constants:

```
.12345       { leading digit is missing }
-NIL         { a plus or minus must be followed }
+'sorry'     { by a digit or identifier   }
```



C

C

C

# 2

## Program Structure

### 2.1 The Structure of Programs

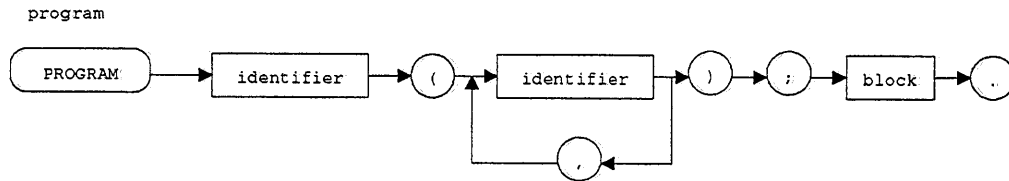


Figure 2-1 Syntax Diagram for Program

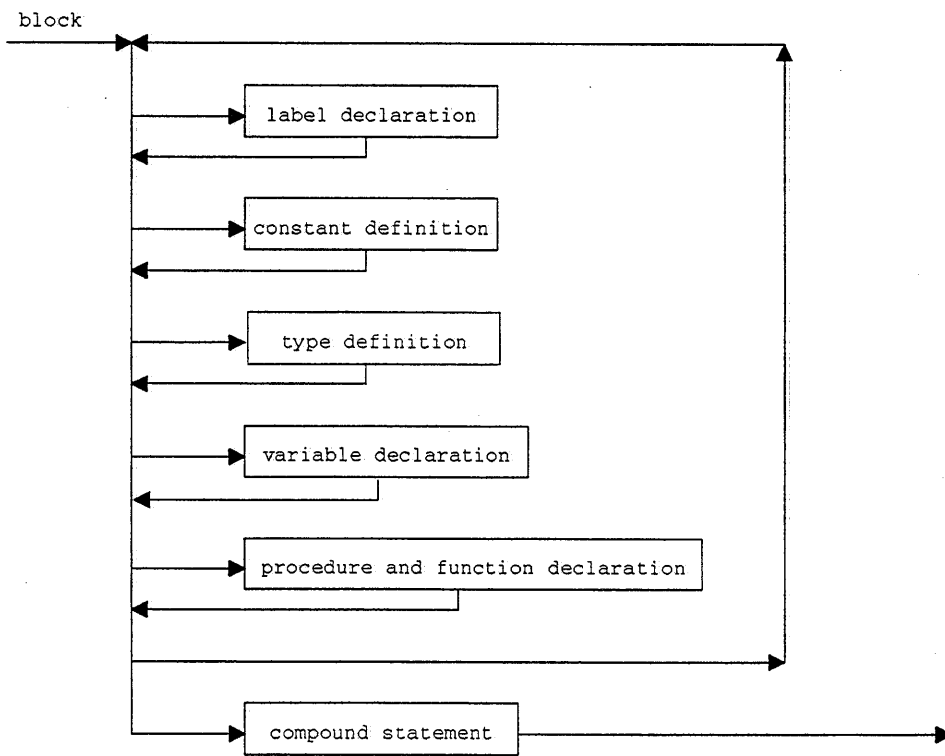


Figure 2-2 Syntax Diagram for Block

Pascal is a block structured language. This means that a Pascal program consists of a set of nested blocks. The nesting of blocks allows the definition of one block to be entirely contained within another block. At the outermost level, a program consists of a program heading, followed by a single block that defines the main program. As the syntax diagram above shows, a block is composed of the following six sections, all optional except the compound statement that constitutes the body of the block:

1. label declaration part,
2. constant definition part,
3. type definition part,
4. variable declaration part,
5. procedure and function declaration part,
6. compound statement.

The nature and exact contents of each of these components of a block are described in the following chapters.

Each block introduces a new local referencing environment. For example, a variable declared in a block *B*, say, is accessible throughout that block unless the same variable name is redefined within a sub-block of *B*. The redefinition of a variable lasts throughout the scope of the sub-block. Details regarding the lexical, or static, scope rules for identifiers are given in the following section.

Variables are allocated when a procedure or function is entered, and are deallocated when the corresponding return is made. Each invocation of a recursive routine has its own set of local variables. This is accomplished by allocating space for local variables in the same stack-like manner in which the recursive calls are nested. Thus, when a return statement is executed, the variable space corresponding to the appropriate invocation of the routine is deallocated.

## 2.2 The Lexical Scope of Identifiers

An identifier may refer to a constant, variable, label, procedure or function name, or type definition. The lexical scope of an identifier is the region of a program where the identifier may be referenced. The block structure of Pascal is used to define the lexical scope rules for identifiers. The following is a summary of the lexical scope rules for Pascal.

1. Every identifier must be defined before it is used. The two exceptions to this involve pointer variables, and procedure and function calls when there is a forward reference.
2. The scope of an identifier depends upon how the identifier was declared.
  - a) For labels, constants, types, variables, procedures and functions, the scope is the block in which the declaration occurs;
  - b) for constants denoting the values of an enumerated type, it is the most embedded block containing the type definition for the constant;
  - c) for function and procedure parameters, it is the formal parameter list and the corresponding block;
  - d) for field identifiers, it is the record definition in which they occur;
  - e) for predefined identifiers, it is an imaginary block enclosing the program.
3. An identifier may only be used within its scope of definition, and an identifier's association is unique within its scope. This means that an identifier cannot be defined twice within the same scope, either with the same or with different meanings.
4. The declarations at the beginning of each block define the local referencing environment for the block. Any reference to an identifier within the body of a block (not including any nested subblocks) is considered a reference to the local declaration for the identifier, if one exists.
5. By convention, when blocks are nested, the nesting levels are called level 0, level 1, etc., beginning with the main program. Identifiers in level *i* are in the scope of blocks declared at levels *i*+1, *i*+2, and so on.

If an identifier is referenced within the body of a block *B* and no local declaration exists, then the reference is considered to be a reference to a declaration within one of the enclosing blocks. The enclosing blocks are searched for this declaration beginning with the block immediately surrounding block *B* until the declaration is found, or the outermost block is reached. When the outermost block is reached, the predefined environment is searched for the identifier and, if not found, an error is reported.

6. If a block *B* contains a subblock *S*, then any local declarations within the subblock (or blocks that *S* may contain) are not available to the outer block *B*. Declarations within a subblock are invisible to the blocks surrounding it.

7. A declaration for the same identifier may occur in many different blocks, but a declaration in an outer block is hidden from the inner block if the inner block gives a new declaration for the same identifier. This gap in the scope of the identifier within the outer block is called a "hole in scope".

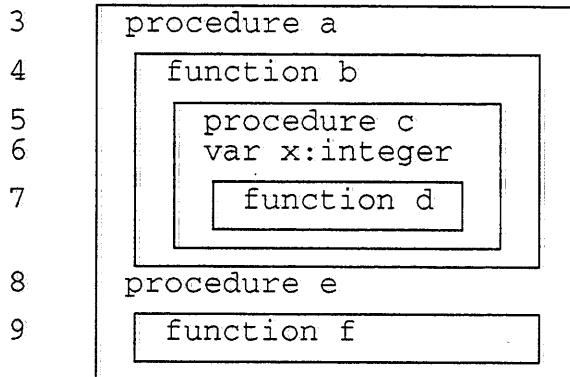
### EXAMPLE

The following example illustrates the lexical scope of identifiers in a Pascal program. The variable *x* declared on line 2 is global throughout most of the program. There is a hole in its scope in procedure *c* and function *d*, since *x* is redeclared in line 6. This means that any reference to *x* in procedure *c* and function *d* refers to the integer *x* declared on line 6, while reference to *x* in the rest of the program refers to the real *x* declared on line 2.

The chart following the program outline summarizes the availability of identifiers in different routines.

1 Program main;

2 var x:real



Variables declared in	may be referenced in
main	main, a, b, c, d, e, f
a	a, b, c, d, e, f
b	b, c, d
c	c, d
d	d
e	e, f
f	f

## 2.3 Declaration Order

Standard Pascal imposes a strict ordering of declarations that is relaxed in NDP Pascal to make it easier to use. Standard Pascal requires that all labels be declared before any constants, all constants be declared before any types, all types be declared before any variables, and all variables be declared before any procedures or functions. NDP Pascal allows the declarations to be in any order, and to appear more than once, provided that every symbol is defined before any reference to it (except as allowed by standard Pascal).

### Example

```

program order1;

function power (a,n:integer):integer;
{ Return a raised to the positive power n. }
  var ans, i : integer;
  begin
    ans := 1;

```

```

        for i := 1 to n do ans := ans * a;
        power := ans
    end;

type t1 = array [1..10] of integer;
var yy : t1;

const x = 123;

type t2 = integer;
var zz : t2;

label 99

var a : t1;
    b : t2;
    i : real;

begin
    ...
end.

```

## 2.4 Program Modules for Separate Compilation

NDP-Pascal has been extended to allow multiple module program development. In NDP-Pascal, a program consists of one or more modules, which are independently compilable units of code. There are two types of modules in NDP-Pascal: the program module and the declarations module.

The program module is the module that gains initial control when the program is executed. It contains the program declaration, the main begin-end block, and the final period. The program module may be the entire program or only part of the program. If it is only part of the program then some of the procedure, function, and variables referenced in the main program must be declared external using the `EXTERNAL` directive. These external routines and variables must be linked with the main program, and the run time library, to obtain a complete program.

A declarations module may be compiled as a unit independent of the program module. It consists of routines and variables that are to be linked with the program module, and possibly other declaration modules, in order to create a complete program. The declarations module must not contain a program statement, a main begin-end block, or a final period.

Declarations modules are useful in breaking up large programs into smaller components. Data is passed to routines through parameters and external variables. By default, the procedures, functions, and variables declared at the top level of a declarations module and at the outermost level of the program module are declared external to the linker. The `STATIC` directive can be used to prevent identifiers from being exported to other modules.

NDP-Pascal permits declarations to be given in any order. This extension allows program and declaration modules to be independent of the ordering of declarations within `INCLUDE` files.

Examples illustrating separate compilation are under the `EXTERNAL` directive, in *Section 3.6*, and under the `STATIC` directive, in *Section 3.7*.

# 3

## Pascal Declarations

### 3.1 Program Heading (PROGRAM)

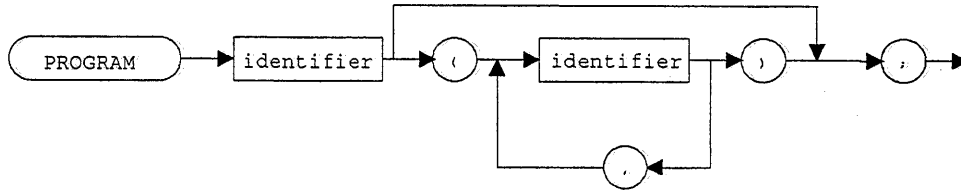


Figure 3-1 Syntax Diagram for Program Heading

The program heading is used to assign a name to a program and serves to document the file names through which the program will communicate with its environment. This is a required statement in each NDP Pascal program. The list of file names is optional, serving only for purposes of documentation and compatibility with the Pascal Standard.

The Pascal standard requires that the predefined file identifier, INPUT, be specified in the program heading if the program reads data from the file INPUT. Similarly, the predefined file identifier, OUTPUT, must be specified if the program writes to the file OUTPUT. Failure to comply causes the compiler to generate an error when the -ANSI compatibility switch is used. See the *NDP User's Manual* for details.

#### EXAMPLES

The following are valid program statements:

```
program matrix;  
program simulator (input, output);  
program regression (factors, data1_inp, data2_inp, results);
```

The following are invalid program statements:

```
program fft (data.raw, data.fft); { Periods are not allowed in }  
                                { identifier names.           }  
program abc (x,17);              { Constants are not allowed. }  
program (input);                 { The program name is missing. }
```

### 3.2 Label Declaration (LABEL)

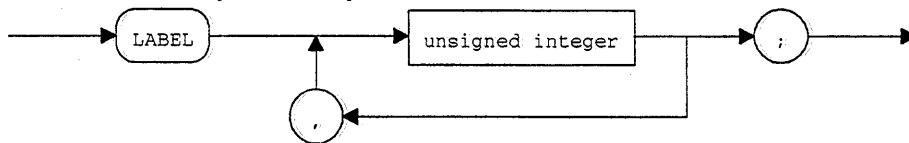


Figure 3-2 Syntax Diagram for Label Declaration

The label declaration is used to declare a label that will be used to identify a statement. Labels permit a statement to be referenced by a GOTO statement. A label is an unsigned number in the range 0 to 9999. Leading zeros in a label are not significant. Labels are separated from the statements they reference by a colon.

The scope of a label is the routine in which it is defined. Therefore, all labels accessed in a routine must be declared within that routine.

Assigning a label to a statement does not guarantee that the statement may be referenced by a GOTO statement. See the rules associated with branching under the GOTO statement in Section 7.7.

### EXAMPLES

```

label 10;

label 100, 200, 300, 301;
...

100 : getpat := (makepat (arg, 1, ENDSTR, pat) > 0;
...

301 : if (lin[i] = COMMA) or (lin[i] = SEMICOL) then begin

```

### 3.3 Constant Definition (CONST)

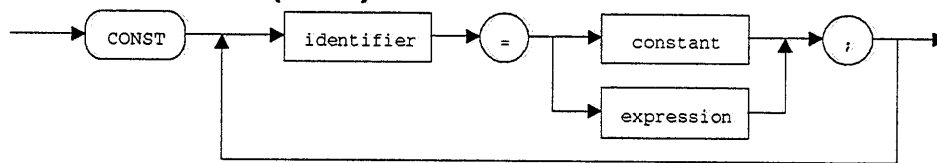


Figure 3-3 Syntax Diagram for Constant Definition

A constant definition is a name that is to be used as a synonym for a constant value. The type of a constant identifier is determined by the type in the constant expression.

NDP Pascal allows the value of a constant to be the result of an expression. The expression may contain operators, predefined functions, and the value of previously defined constants. The definition of an expression is given in Chapter 6.

NDP Pascal accepts the syntax `0x <hex digits>` or `0X <hex digits>` for hexadecimal constants.

### EXAMPLES

```

const
  ONE_K = 1024                { An integer constant }
  ZERO  = -273.15;           { A real constant }
  NA    = 6.023e23;          { A real constant }
  U     = 1.66e-27;          { A real constant }
  PI    = 3.141592653589793; { A real constant }
  COEF  = 1.0 / sqrt( 2.0 * pi); { A real constant }
  PTR   = nil;                { A pointer constant }
  VALID = true;               { A boolean constant }
  ALL_ONES = 0xffff;          { A hexadecimal constant }
  MININT  = 0X80000000;        { A hexadecimal constant }
  ANSWER  = ['Y','y','N','n']; { A set constant }
  A_PALINDROME = 'Madam, I''m Adam'; { A character string constant }

```

### 3.4 Type Definition (TYPE)

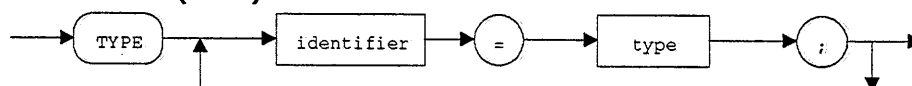


Figure 3-4 Syntax Diagram for Type Definition

A type is a set of values that a variable may assume. A type definition is used to both define a data type and assign a name to that type. There are two kinds of data types: predefined and

user defined. The predefined data types are part of the Pascal language and are described in *Chapter 4*. User-defined data types are established using the type definition.

A type definition consists of an identifier followed by an equal sign and a type clause. Type identifiers in the type clause must be already defined by a previous type definition. Recursive type definitions require pointer types.

**EXAMPLE 1**

```
Type
direction = (north, south, east, west);
row       = 1..66;
column    = 1..132;
cell = record
    barrier : boolean;
    visited : boolean
end;

maze = array [row, column] of cell;
```

**EXAMPLE 2**

The following is a recursive type definition for a linked list of integers:

```
type
listType = record
    contents : integer;
    listType : ^listType;
end;
```

**EXAMPLE 3**

The following is an illegal type definition since the type clause is recursive and does not refer to a pointer type:

```
type
matrix = array [1..n] of matrix;
```

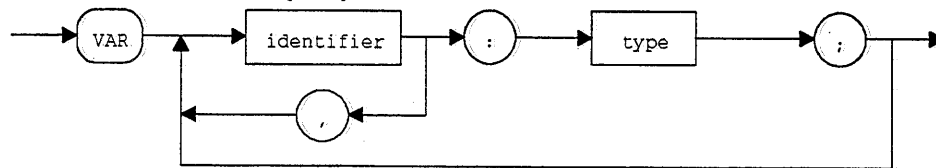
**3.5 Variable Declaration (VAR)**

Figure 3-5 Syntax Diagram for Variable Declaration

A variable declaration is used to define the type of a variable. This establishes the set of values that can be assigned to the variable.

Identifiers of the same type may be declared together by separating them with commas.

Variables are allocated when a procedure or function is entered, and are deallocated when the corresponding return is made. When a recursive call is made to a routine, space is again allocated for the variables declared in the routine. This space is allocated in a stack-like manner so that when a return statement is executed, the variable space corresponding to the appropriate invocation of the routine is deallocated.

**EXAMPLE 1**

```
var travel : direction;
var i, j, k : integer;
    x, y, z : real;
```



```
var board : maze;
```

**EXAMPLE 2**

```
type
  nameType = array [1..20] of char;
  char5     = array [1..5] of char;
  char10    = array [1..10] of char;
  char25    = array [1..25] of char;

var
  phoneBook = record
    lastName : nameType;
    firstName : nameType;
    address   : char25;
    cityState : char25;
    zip       : char5;
    phoneno   : char10;
  end;
```

**EXAMPLE 3**

```
type
  degrees = 0..360;
  percent = 0..100;

var
  weather : record
    station : array [1..20] of char;
    date    : integer;
    time    : integer;
    temp    : record
      lo : real;
      hi : real;
    end;
    humidity : percent;
    precip   : record
      rain : real;
      snow : real;
    end;
    wind     : record
      speed      : real;
      direction  : degrees;
    end;
    pressure : record
      height : real;
      direction : (up, down);
    end;
  end;
```

### 3.6 External Directive (EXTERNAL)

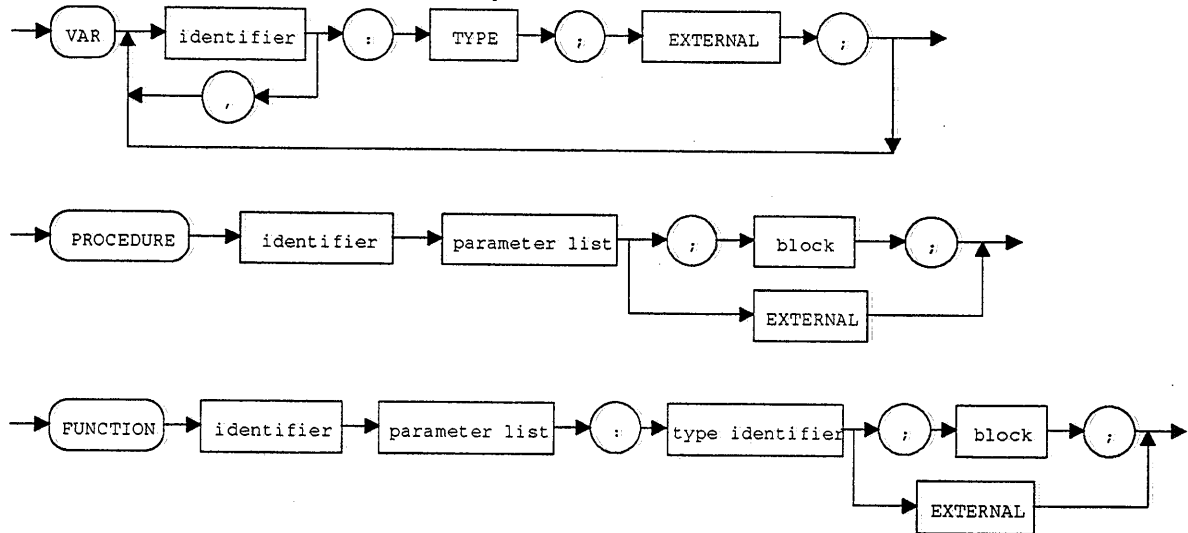


Figure 3-6 Syntax Diagram for EXTERNAL Directive

The external directive notifies the compiler that a procedure, function, or variable exists in a separately compiled module.

A procedure or function may be declared external, and then later in the same module, the procedure or function body may be given. If this is done, the procedure or function declaration must not contain a parameter list or a return type. This is to simplify use of the #INCLUDE directive, yet prevent multiple definition of symbols in the parameter list.

Declaring a variable EXTERNAL permits the sharing of data among separately compiled modules. This is done by placing the identifier EXTERNAL followed by a semicolon directly after the variable declaration.

#### EXAMPLE 1

The following example illustrates the use of the external directive with a utility function. It consists of two files: one containing the main program and one containing the function called by the main program. These files must be separately compiled and linked together before being run.

```
{===== Contents of ex001a.p =====}

program ex001a;
{ Example to illustrate the EXTERNAL directive. }
const size = 9;
type list = array [1 .. size] of real;
function binarySearch (a:list; x:real): integer; external;
var a:list;

begin
  a[1] := 0.;      a[2] := 1.;      a[3] := 2.;
  a[4] := 3.;      a[5] := 5.;      a[6] := 8.;
  a[7] := 13.;     a[8] := 21.;     a[9] := 34.;
  writeln('Index of -1 ', binarySearch(a,-1));
  writeln('Index of 0 ', binarySearch(a,0));
  writeln('Index of 7 ', binarySearch(a,7));
  writeln('Index of 34 ', binarySearch(a,34));
end. { end of ex001a }

{===== Contents of ex001b.p =====}
```

```

const size = 9;
type list = array [1..size] of real;
function binarySearch (a:list; x:real):integer;
var i, lo, hi : integer;

begin
  lo := 1;
  hi := size;
  repeat
    i := (lo + hi) div 2;
    if x < a[i] then      hi := i - 1
    else                  lo := i + 1;
    until (x = a[i]) or (lo > hi);
  if x = a[i] then
    binarySearch := i
  else
    binarySearch := 0;
  end;
end;

```

The program searches a predefined list for a series of numbers, and generates the following output:

```

Index of -1 =      0
Index of  0 =      1
Index of  7 =      0
Index of 34 =      9

```

### EXAMPLE 2

The following example illustrates the use of the EXTERNAL and INCLUDE directives. It consists of three files: a header file containing the declarations used by the other two files, a file containing the main program, and a file containing a function called by the main program. These files must be separately compiled and linked together to be run.

```

{===== Contents of ex002.ph =====}
type
  point = record
    x : real;
    y : real;
  end;

function slope (a, b: point) : real; external;

{===== Contents of ex002a.p =====}
program ex002a;
#include 'ex002.ph'
var a,b: point;

begin
  a.x := 1.0;  a.y := 2.0;
  b.x := 3.0;  b.y := 4.0;
  writeln('slope = ', slope(a,b));
end. { end of ex002a }

{===== Contents of ex002b.p =====}
#include 'ex002.ph'
function slope; { (a,b:point) : real; }
const epsilon = 1.0e-7;

begin
  if (b.x - a.x) > epsilon then

```

```

    slope := (b.y - a.y) / (b.x - a.x)
  else
    slope := maxint;
  end; { end of ex002b.p }

```

The program evaluates a function computing the slope of a line and prints the following result:

```
slope = 1.0000000000000000e+00
```

### 3.7 Static Directive (STATIC)

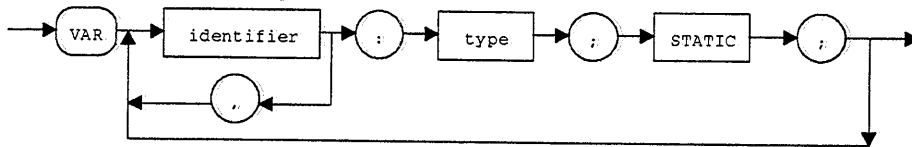


Figure 3-7 Syntax Diagram of a STATIC Directive

The static directive is used to declare static variables. Static variables are not exported to routines that are included when the #INCLUDE directive is used. Static variables can be thought of as being private to the module in which they are declared. Only those variables in the outer scope of a declaration or program module may be declared **STATIC**.

#### EXAMPLE

This example illustrates the use of the **STATIC** and **EXTERNAL** directives and consists of two files, one containing the main program and one containing the routines used by the main program. These files must be separately compiled and linked together before being run.

FILE.A contains the main program and FILE.B contains three routines.

This program makes use of three variables: *x*, *y*, and *z*. Each module has its own copy of *x*, while *y* is shared among them. This program can be understood by considering the variable space for each file in the following manner:

FILE.A		FILE.B
<i>x</i> ←	different	→ <i>x</i>
<i>y</i> ←	same	→ <i>y</i>
<i>z</i>		

After **SETUP** is executed, we have the assignments:

FILE.A	FILE.B
<i>x</i> =?	<i>x</i> =10
<i>y</i> =20	<i>y</i> =20
<i>z</i>	

Prior to **ADDUP**, we have the assignments:

FILE.A	FILE.B
<i>x</i> =1	<i>x</i> =10
<i>y</i> =2	<i>y</i> =2
<i>z</i> =100	

After execution of **ADDUP**, we have the assignments:

FILE.A	FILE.B
<i>x</i> =1	<i>x</i> =10
<i>y</i> =2	<i>y</i> =2
<i>z</i> =112	

Hence, the program will print the result: *z* = 112.

```
{===== Contents of ex003a.p =====}
```

```
program static1 (output);
procedure setup; external;
procedure addup (var q:integer); external;
var
  x : integer; static;
  y : integer; external;
  z : integer;
```

```
begin
  setup;
  x := 1;
  y := 2;
  z := 100;
  addup (z);
  writeln(' z = ', z);
end.
```

```
{===== Contents of ex003b.p =====}
```

```
procedure setup; external;
procedure addup (var q: integer); external;
var
  x : integer; static;
  y : integer;
```

```
procedure setup;
begin
  x := 10;
  y := 20;
end;
```

```
procedure addup;
begin
  q := q + x + y;
end;
```

# 4 Type Definitions

## 4.1 Type Definitions

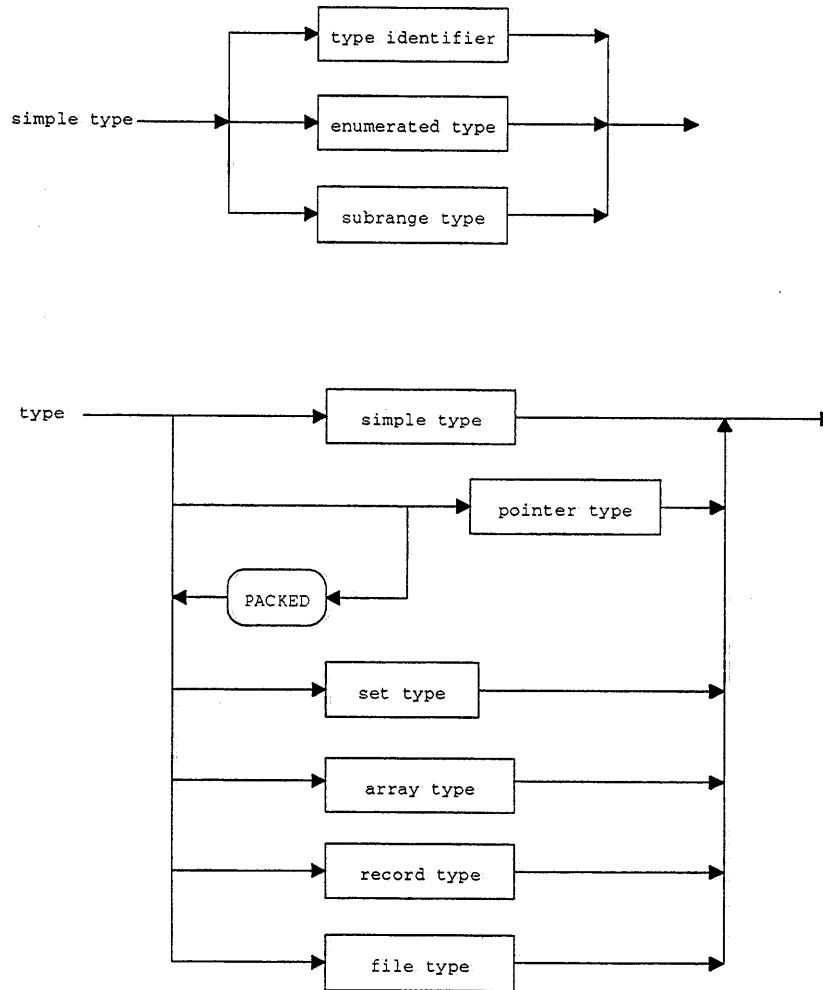


Figure 4-1 Syntax Diagrams for Simple Type and Type

The syntax diagrams above summarize the data types available in NDP Pascal. Each of these types is described in detail in a section of this chapter.

The data type determines the set of values that a variable may assume. Data types are classified as scalar, pointer, or structured. The types `BOOLEAN`, `CHAR`, `INTEGER`, `DOUBLE`, `FLOAT`, `REAL`, enumerated and subrange types are scalar data types. This means that the values may be placed on a linear scale and comparisons (less than, equal, greater than) made between them. Pointer data types are used for variables that are to contain the address of other variables, or the address of variables that will be created during program execution (dynamic variables). Structured data types consist of aggregates of other data types. The Set, Array, Record, and File type are structured data types.

A type declaration is used to assign a type identifier to a type definition. The type identifier may then be used wherever a type definition is required, for example, in a variable declaration, in a parameter list, or in another type declaration.

The data type of a variable may be declared using a type identifier, or by specifying the type definition when the variable is declared. Identifiers used as procedure or function parameters must be declared with a predefined data type or with a type identifier.

## 4.2 Type Compatibility and Conversions

The data type determines the set of values that a variable may assume. NDP Pascal supports strong typing, which means that the type of all variables must be explicitly declared. This allows the compiler to verify that each operation performed on a variable is appropriate for the type associated with the variable. Strong typing requires that rules exist in order to determine when two types are to be considered 1) identical, 2) compatible, or 3) assignment compatible. The following three sections describe these rules.

### 4.2.1 Identical Types

Two types are identical if one of the following is true:

- a. The variables refer to the same type identifier;
- b. The variables refer to two separate type identifiers that have themselves been declared equal by the following type definition:

```
type t1 = t2;
```

Type identity in Pascal is based upon the name of the type, not on the physical storage of the data in question. For example, the following are not identical types:

```
type
  r = array [1..10] of integer;
  s = array [1..10] of integer;
```

Identical types are required in the following circumstances:

1. between the actual and formal variable (VAR) parameters in a function or procedure
2. assignment between array types
3. assignment between record types

### 4.2.2 Compatible Types

Two types are compatible if one of the following is true:

- a. they are of the same type;
- b. one is a subrange of the other or they are both subranges of the same type;
- c. one type is a string literal of one character and the other is of type CHAR;
- d. they are both set types and their base types are compatible.

The empty set is compatible with any set type and the value NIL is compatible with any pointer type.

Compatible types are required in the following circumstances:

1. two values must be compatible when combined with an operator in an expression;
2. the index expression in a CASE statement must be compatible with all case constant values.

### 4.2.3 Assignment Compatibility

Assignment compatibility indicates when assignment between a variable and an expression is permitted using the assignment operator. A variable and a expression are assignment compatible if one of the following is true:

- a. the types are identical and neither is a file or a structured record type;

- b. the variable is of type FLOAT, REAL or DOUBLE and the expression is compatible with type INTEGER;
- c. the type of the variable may be a subrange of the expression if the value to be assigned is within the allowable subrange of the variable;
- d. the variable and the expression have compatible set types and all members of the expression are permissible members of the variable.

Assignment compatible types are required when an actual value parameter must be assignment compatible with the type of the corresponding formal parameter.

#### EXAMPLE

```

type
  months = (jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec);

  winter = jan..mar;
  spring = apr..jun;
  summer = jul..sep;

  column = array [1..10] of real;
  row     = column;

var
  season : set of months;
  vacation : set of summer;
  cold    : winter;
  warm    : spring;
  vector1 : column;
  vector2 : row;
  vector3 : array [1..10] of real;
  meeting, event : RECORD
      date : integer;
      time : real
  END;

```

This example illustrates the three gradations of type compatibility. Several type definitions and variable declarations are given, followed by a table summarizing the compatibility of each variable with each other. Some entries are omitted to improve readability. For example, each variable is obviously type identical with itself, so the entry "season is identical to season" has not been included. Similarly, the reflexive entries for compatible and assignment compatible types have been omitted.

variable	identical to	compatible with	assignment compatible with
season	-	vacation	vacation
vacation	-	season	season
cold	-	warm	warm
warm	-	cold	cold
vector1	vector2	vector2	vector2
vector2	vector1	vector1	vector1
vector3	-	-	-
meeting	event	event	-
event	meeting	meeting	-

#### 4.2.4 Implicit Type Conversion

NDP Pascal does type conversions on data in the following special circumstances:

- a. in a binary operation involving an integer and a float, real, or double, the integer will be converted to a float, real or double;



- b. when an integer is being assigned to a float, real or double variable, the integer will be converted to a float, real or double;
- c. an integer will be converted to a float, real or double if passed by value to a parameter requiring a float, real or double value.

The motivation behind type conversions is ease of use. The above restrictions prevent information from being lost since a data type may be converted to a data type with greater precision, but not to a type with less precision.

### 4.3 Enumerated Types

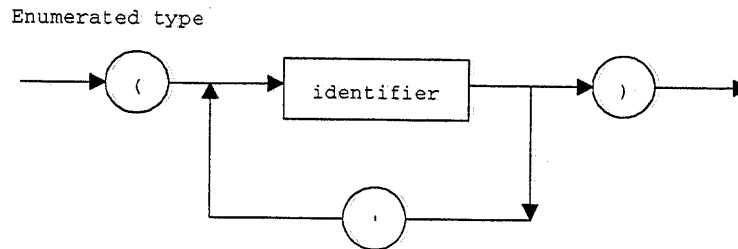


Figure 4-2 Syntax Diagram for Enumerated Type

An enumerated type is a list of names that are treated as scalar values. An enumerated type is defined by listing the values that are permitted for a variable of this type. Each value is an identifier that is treated as a constant in its own right. Enumerated types provide a mechanism that allows an identifier to be used as a constant symbol.

The names defined in the list are treated as constant values of the type being defined. The lexical scope rules, described in *Section 2.2*, specify that these names are local to the block in which the type denoter occurs. The lexical scope rules for enumerated types amount to the following:

1. A constant identifier for a type in an inner block cannot be redefined in the same block;
2. Two different enumerated types cannot have an element with the same name in the same lexical level;
3. A constant identifier may not be accessed outside the block in which it is defined. Hence it is not possible to read or write the values of constant identifiers. All enumerated constants of a single type are ordered. The first item in the list is assigned the ordinal value 0, the second item in the list is assigned the value 1, and so on. The ordinal value of an enumerated constant may be obtained using the predefined function `ORD`.

The predefined functions `PRED` and `SUCC` may be used to operate on expressions containing enumerated types. By convention there is no value less than the first enumerated constant defined in the list, and no value greater than the last constant defined in the list.

The predefined type `BOOLEAN` is an enumerated scalar with the definition:

```
type BOOLEAN = (FALSE, TRUE);
```

#### EXAMPLE

```
type
  warnings = (advisory, gale, storm, hurricane);
  occupation = (tinker, tailor, soldier, spy);
  numeral = (I, II, III, IV, V, VI, VII, VIII, IX, X);

var
  message : warning;
  roman : numeral;
  applicant : record
```

```

        name : array [1..30] of char;
        field : occupation;
    end;
piece : (pawn, knight, bishop, rook, queen, king);

```

## 4.4 The Subrange Type

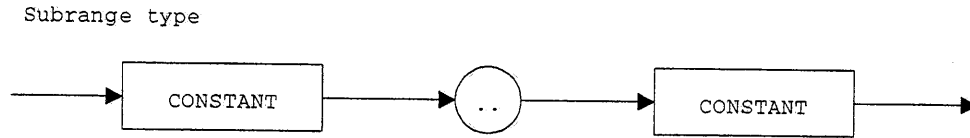


Figure 4-3 Syntax Diagram for SUBRANGE Type

A subrange type is a name given to a subset of the values of an enumerated type. The values chosen from the enumerated type must be consecutive, and the enumerated type must already be defined.

A subrange type is defined by specifying the range of values it may assume. This is done by specifying the minimum and maximum values, which may be the same, from the enumerated type that may be assigned to it. Any operation allowed on a scalar type is also allowed on any subrange of it.

### EXAMPLE

```

const
    size = 1024;

type
    vitamins = (A, D, E, C, thiamin, riboflavin, niacin, B6, B12,
               calcium, phosphorus, magnesium, iron, zinc, iodine);
    fat_soluble = A..E;
    water_soluble = thiamin..B12;
    minerals      = calcium..iodine;
    index = 0..size-1;

var
    day    : 1..31;
    month  : 1..12;
    buffer : array [index] of integer;

```

## 4.5 The SET Type

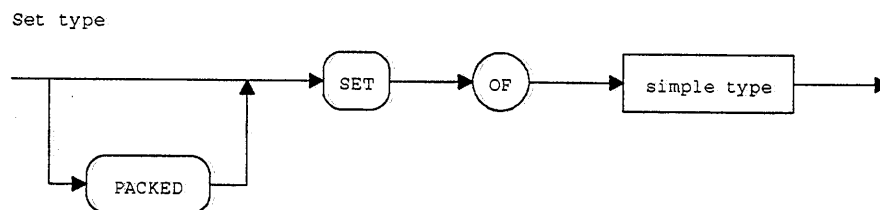


Figure 4-4 Syntax Diagram for Set Type

The SET type is any collection of values taken from a scalar type.

The following table describes the operations and functions that may be used with variables of type SET. In the following table, both  $x$  and  $y$  are type SET.

Symbol	Usage	Result Type	Description
~	~ $x$	SET	complement of set $x$
+	$x + y$	SET	set union of $x$ and $y$

-	x - y	SET	set difference of x and y
*	x * y	SET	set intersection of x and y
=	x = y	BOOLEAN	compares for x equal to y
<>	x <> y	BOOLEAN	compares for x not equal to y
<=	x <= y	BOOLEAN	tests if x is a subset of y
>=	x >= y	BOOLEAN	tests if x is a superset of y
in	x in y	BOOLEAN	tests for x in set y
:=	x := y	SET	assigns the value of y to x

Sets of CHAR, BOOLEAN, and enumerated types are implemented as sets of the base type.

Sets of integers are handled specially because of the possible large memory requirements of a set of integers. Therefore, two different sets sizes are allowed for sets of integers. The sizes are 32 and 256. By default, sets of integers are implemented as "set of 0..31". The compiler switch, -P4, causes sets of integers to be implemented as "set of 0..255". (See the *NDP User's Manual* for details.) Sets of size 32 are more efficient than sets of size 256.

The following table summarizes the storage requirements for the SET type.

type	implemented as	Compiler option
CHAR		
BOOLEAN		
enumeration		
set of integer	set of 0..31	By default
set of integer	set of 0..255	-P4 compilation switch

#### EXAMPLE 1

The following are valid set declarations:

```
type
  palette = (black, blue, green, red, white);
  color   = set of palette;
  s1 = set of char;
```

#### EXAMPLE 2

The following type declaration defines a set of 256 integers and requires the -P4 runtime option:

```
type
  s2 = set of 0..255;
```

#### EXAMPLE 3

The following code fragment requires INTEGER sets of size 256 to work correctly, hence the -P4 compilation option must be used. This is because the base type of the set is INTEGER, and the ordinal value of 'A' is 65, which requires a set of 256 elements:

```
var prefix : integer;
begin
  if prefix in [ord('K'), ord('L')] then
    ...
```

## 4.6 Predefined Scalar Types: BOOLEAN, CHAR, INTEGER, DOUBLE, REAL, FLOAT

NDP Pascal implements the following predefined scalar data types: BOOLEAN, CHAR, INTEGER, DOUBLE, REAL, and FLOAT. INTEGER, CHAR, and BOOLEAN have the type definitions given below, while DOUBLE, REAL and FLOAT implement IEEE 32 and 64 bit floating point format, least significant byte at the lowest address. Type FLOAT provides 6 to 7 decimal significant digits and type DOUBLE provide 15 to 16 decimal significant digits.

A brief summary of these types is given below, while the next five sections describe the operations allowed with each data type in detail.

```

const
  MAXINT = 2147483647;           { (2**31)-1 }

type
  INTEGER = -2147483648..MAXINT;
  CHAR = chr(0) .. chr(127);
  BOOLEAN = (FALSE, TRUE);

```

The table below summarizes the predefined data types in NDP Pascal. Space is always allocated on a 4 byte boundary. Range refers to the largest positive and negative number supported by a data type, while precision refers to the smallest positive and negative number that can be supported by a data type.

Type	Space allocated	Range	Precision	Compiler Option
BOOLEAN	4 byte (8 bits)			
CHAR	4 byte (8 bits)			
INTEGER	4 bytes (32 bits)	-2,147,483,648 to 2,147,483,647		
FLOAT	4 bytes (32 bits)	+3.39e38	+1.18e-38	
REAL	4 bytes (32 bits)	+3.39e38	+1.18e-38	-P3
REAL	8 bytes (64 bits)	+1.80e308	+2.23e-308	by default
DOUBLE	8 bytes (64 bits)	+1.80e308	+2.23e-308	

#### 4.6.1 Operations and Functions for the BOOLEAN Scalar Type

The following table describes the operations and functions that may be used with variables of type BOOLEAN. In the following table,  $x$  and  $y$  are both type BOOLEAN.

Symbol	Usage	Result Type	Description
~	~ $x$	BOOLEAN	returns complement of $x$
=	$x = y$	BOOLEAN	compares for $x$ equal to $y$
<	$x < y$	BOOLEAN	compares for $x$ less than $y$
<=	$x <= y$	BOOLEAN	compares for $x$ less than or equal to $y$
>	$x > y$	BOOLEAN	compares for $x$ greater than $y$
>=	$x >= y$	BOOLEAN	compares for $x$ greater than or equal to $y$
<>	$x <> y$	BOOLEAN	compares for $x$ not equal to $y$
	$x   y$	BOOLEAN	returns TRUE if either $x$ or $y$ are true
&	$x & y$	BOOLEAN	returns TRUE if both $x$ and $y$ are true
:=	$x := y$	BOOLEAN	assigns the value of $y$ to $x$
ORD	ORD ( $x$ )	INTEGER	returns 0 if $x$ is false, and 1 if $x$ is true.

Function values for negation:

~	result
FALSE	TRUE
TRUE	FALSE

Function values for ORD:

ORD	result
FALSE	0
TRUE	1

Function values for binary BOOLEAN operators:

	FALSE	FALSE	TRUE	TRUE	Logical Name
op	op	op	op	op	
op	FALSE	TRUE	FALSE	TRUE	equivalence
=	TRUE	FALSE	FALSE	TRUE	

<	FALSE TRUE	FALSE FALSE	
<=	TRUE TRUE	FALSE TRUE	implication
>	FALSE FALSE	TRUE FALSE	
>=	TRUE FALSE	TRUE TRUE	
<>	FALSE TRUE	TRUE FALSE	exclusive or
	FALSE TRUE	TRUE TRUE	inclusive or
&	FALSE FALSE	FALSE TRUE	and

The type `BOOLEAN` is defined as an enumerated scalar whose values are `TRUE` and `FALSE`. This is equivalent to the definition

```
type
  BOOLEAN = (FALSE, TRUE);
```

Boolean variables will occupy four bytes of memory and will be aligned on a four byte boundary.

The result of the operators `<`, `<=`, `>` and `>=` may be obtained by using the fact that `ORD(FALSE) = 0` and `ORD(TRUE) = 1`.

#### 4.6.2 Operations and Functions on the `CHAR` Scalar Type

The following table describes the operations and functions that may be used with variables of type `CHAR`. In the following table, both `x` and `y` are of type `CHAR`.

Symbol	Usage	Result Type	Description
=	<code>x = y</code>	BOOLEAN	compares for <code>x</code> equal to <code>y</code>
<	<code>x &lt; y</code>	BOOLEAN	compares for <code>x</code> less than <code>y</code>
<=	<code>x &lt;= y</code>	BOOLEAN	compares for <code>x</code> less than or equal to <code>y</code>
>	<code>x &gt; y</code>	BOOLEAN	compares for <code>x</code> greater than <code>y</code>
>=	<code>x &gt;= y</code>	BOOLEAN	compares for <code>x</code> greater than or equal to <code>y</code>
<>	<code>x &lt;&gt; y</code>	BOOLEAN	compares for <code>x</code> not equal to <code>y</code>
:=	<code>x := y</code>	CHAR	assigns the value of <code>y</code> to <code>x</code>
ORD	<code>ORD (x)</code>	INTEGER	returns the ASCII code for the symbol <code>x</code>
PRED	<code>PRED (x)</code>	CHAR	returns the character preceding <code>x</code> in the ASCII collating sequence
SUCC	<code>SUCC (x)</code>	CHAR	returns the character following <code>x</code> in the ASCII collating sequence

The type `CHAR` is a scalar type corresponding to the values in the ASCII character set.

Variables of type `CHAR` occupy one byte of memory and are allocated in four byte increments on a four byte boundary.

#### 4.6.3 Operations and Functions on the `INTEGER` Scalar Type

The following table describes the operations and functions that may be used with variables of type `INTEGER`. In the following table, both `x` and `y` are of type `INTEGER`.

Symbol	Usage	Result Type	Description
+	<code>+ x</code>	INTEGER	returns the operand
+	<code>x + y</code>	INTEGER	returns the sum of the operands
-	<code>- x</code>	INTEGER	returns the negated operand
-	<code>x - y</code>	INTEGER	returns the difference of the operands
*	<code>x * y</code>	INTEGER	returns the product of the operands
/	<code>x / y</code>	INTEGER	converts operands to <code>REAL</code> , returns real quotient.

DIV	$x \text{ DIV } y$	INTEGER	returns the integer quotient of the operands
MOD	$x \text{ MOD } y$	INTEGER	returns the integer modulus of the operands
=	$x = y$	BOOLEAN	compares for $x$ equal to $y$
<	$x < y$	BOOLEAN	compares for $x$ less than $y$
<=	$x \leq y$	BOOLEAN	compares for $x$ less than or equal to $y$
>	$x > y$	BOOLEAN	compares for $x$ greater than $y$
>=	$x \geq y$	BOOLEAN	compares for $x$ greater than or equal to $y$
<>	$x \neq y$	BOOLEAN	compares for $x$ not equal to $y$
:=	$x := y$	INTEGER	assigns the value of $y$ to $x$
&	$\& x$	INTEGER	returns the address of the operand
&	$x \& y$	INTEGER	returns the bitwise logical sum
~	$\sim x$	INTEGER	returns the one's complement of $x$
	$x   y$	INTEGER	returns the bitwise logical or
<<	$x \ll y$	INTEGER	$x$ is shifted left by $y$ bits
>>	$x \gg y$	INTEGER	$x$ is shifted right by $y$ bits
+=	$x += y$	INTEGER	equivalent to " $x := x + y$ "
-=	$x -= y$	INTEGER	equivalent to " $x := x - y$ "
*=	$x *= y$	INTEGER	equivalent to " $x := x * y$ "
/=	$x /= y$	INTEGER	equivalent to " $x := x / y$ "
=	$x  = y$	INTEGER	equivalent to " $x := x   y$ "
&=	$x \&= y$	INTEGER	equivalent to " $x := x \& y$ "
<<=	$x \ll= y$	INTEGER	equivalent to " $x := x \ll y$ "
>>=	$x \gg= y$	INTEGER	equivalent to " $x := x \gg y$ "
ODD	ODD ( $x$ )	BOOLEAN	tests for odd $x$
CHR	CHR ( $x$ )	CHAR	returns the ASCII character whose ASCII value is $x$
ABS	ABS ( $x$ )	INTEGER	returns the absolute value of $x$
SQR	SQR ( $x$ )	INTEGER	returns the square of $x$
PRED	PRED ( $x$ )	INTEGER	returns the integer $x-1$
SUCC	SUCC ( $x$ )	INTEGER	returns the integer $x+1$

The type INTEGER is a subset of the whole numbers that may be represented in 32 bits. The INTEGER type equivalent to the definition:

```

const
  MAXINT = 2147483647;      { (2**31)-1 }

type
  integer = -2147483648..MAXINT;
```

Integer variables will occupy 4 bytes of memory, and will be aligned on a 4 byte boundary.

#### EXAMPLE (ex004.p)

This example illustrates bit operations that are possible with NDP Pascal extensions. The function `getbits` in the program below is from Kernighan and Ritchie's book on C.

```

program getbit1(output);

function getbits(x, p, n: integer): integer;

begin
  getbits := (x >> (p+1-n)) & ~(~0 << n)
end;

var i,x: integer;

begin
  x := 0xf0f0;
  for i := 0 to 15 do
```

```
writeln('getbits (0xf0f0, ', i:3, ', 4) = ', getbits(x, i, 4):3)
end.
```

This program generates the following output:

```
getbits (0xf0f0, 0, 4) = 0
getbits (0xf0f0, 1, 4) = 0
getbits (0xf0f0, 2, 4) = 0
getbits (0xf0f0, 3, 4) = 0
getbits (0xf0f0, 4, 4) = 8
getbits (0xf0f0, 5, 4) = 12
getbits (0xf0f0, 6, 4) = 14
getbits (0xf0f0, 7, 4) = 15
getbits (0xf0f0, 8, 4) = 7
getbits (0xf0f0, 9, 4) = 3
getbits (0xf0f0, 10, 4) = 1
getbits (0xf0f0, 11, 4) = 0
getbits (0xf0f0, 12, 4) = 8
getbits (0xf0f0, 13, 4) = 12
getbits (0xf0f0, 14, 4) = 14
getbits (0xf0f0, 15, 4) = 15
```

#### 4.6.4 Operations and Functions of the REAL, FLOAT, and DOUBLE Scalar Types.

The following table describes the operations and functions that may be used with variables of type REAL, FLOAT and DOUBLE. In the following table, both  $x$  and  $y$  are of the same type that may be one of REAL, FLOAT, or DOUBLE. We use the abbreviation R for REAL, F for FLOAT, and D for DOUBLE.

Symbol	Usage	Result Type	Description
+	+ $x$	R, F, D	returns the operand
+	$x + y$	R, F, D	returns the sum of the operands
-	- $y$	R, F, D	returns the operand negated
-	$x - y$	R, F, D	returns the difference of the operands
*	$x * y$	R, F, D	returns the product of the operands
/	$x / y$	R, F, D	returns the quotient of the operands
=	$x = y$	BOOLEAN	compares for $x$ equal to $y$
<	$x < y$	BOOLEAN	compares for $x$ less than $y$
<=	$x <= y$	BOOLEAN	compares for $x$ less than or equal to $y$
>	$x > y$	BOOLEAN	compares for $x$ greater than $y$
>=	$x >= y$	BOOLEAN	compares for $x$ greater than or equal to $y$
<>	$x <> y$	BOOLEAN	compares for $x$ not equal to $y$
:=	$x := y$	R, F, D	assigns the value of $y$ to $x$
+=	$x += y$	R, F, D	equivalent to " $x := x + y$ "
-=	$x -= y$	R, F, D	equivalent to " $x := x - y$ "
*=	$x *= y$	R, F, D	equivalent to " $x := x * y$ "
/=	$x /= y$	R, F, D	equivalent to " $x := x / y$ "
ABS	ABS ( $x$ )	R, F, D	returns the absolute value of $x$
SQR	SQR ( $X$ )	R, F, D	returns the square of $x$
SQRT	SQRT ( $X$ )	R, F, D	returns the square root of $x$
LN	LN ( $x$ )	R, F, D	returns the natural logarithm of $x$
EXP	EXP ( $X$ )	R, F, D	returns the natural log base raised to the $x$ power
SIN	SIN ( $X$ )	R, F, D	returns the sine of $x$ (in radians)
COS	COS ( $X$ )	R, F, D	returns the cosine of ( $x$ in radians)
ARCTAN	ARCTAN ( $x$ )	R, F, D	returns (in radians) the inverse tangent of $x$
TRUNC	TRUNC ( $x$ )	INTEGER	returns the operand truncated to an integer
ROUND	ROUND ( $X$ )	INTEGER	returns the operand rounded to an integer

The type `DOUBLE`, `REAL`, and `FLOAT` are used to represent IEEE 32 and 64 bit floating point data.

## 4.7 Array Type

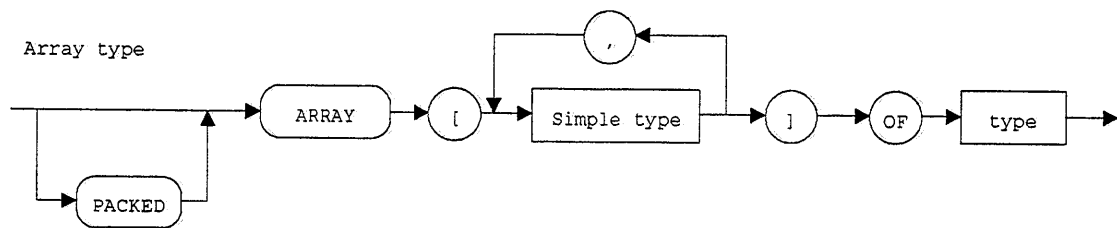


Figure 4-5 Syntax Diagram for Array Type

The array type is used to define a collection of homogeneous elements. This collection takes the form of a subscripted list where each subscript in the list corresponds to one element of the array. The index type is restricted to integer, character subranges, or enumerated types. The component type may be any simple or structured type.

The size of an array corresponds to the number of distinct values that the index may assume. This value is fixed in the type definition and cannot vary during program execution. Note that since the size of an array is part of its definition, two array types are identical only if their corresponding index types have the same cardinality.

The reserved word `PACKED` indicates that the compiler is to compress data storage to minimize the number of unused bytes between array elements. NDP Pascal always aligns each element of an array on the boundary appropriate to the component type, so the word `PACKED` has no effect. However, elements of packed arrays may not be passed as `VAR` parameters to procedures or functions.

Arrays defined with more than one index are called multi-dimensional arrays. A multi-dimensional array is equivalent to an array of arrays. For example, the array definition

```
array [r, s, t] of someType;
```

is a synonym for

```
array [r] of array [s] of array [t] of someType;
```

Array indexing is accomplished by the use of subscripts. A subscript is any expression of a type that is assignment compatible with the index type of the array, and that evaluates to one of the values of the index. The index may be any scalar except `REAL`, `FLOAT`, or `DOUBLE`. Note that while Pascal syntax allows the use of `INTEGER` as an index type, this would result in any array too large to be implemented, so this usage is flagged as a compile time error.

Arrays may be assigned to an array variable of the same type. The predefined procedures `PACK` and `UNPACK` assign elements of one array to another, while converting between packed and unpacked array types. These routines are described in *Chapter 10*.

### EXAMPLE 1

```
const
  n = 10; m = 10;

type
  decision = (yes, no, maybe);
  occurrence = 0..maxint;
  t1 = array [-127..128] of real;
  t2 = array [decision] of boolean;
  t3 = array [char] of occurrence;
```



```

t4 = array [0.. 1023, boolean] of integer;
t5 = array [integer] of char;

type
  complex = record re,im: real end;
  vector = array [1..n] of complex;
  matrix = array [1..m] of vector;
  ...

```

**EXAMPLE 2**

```

type
  prefix = (deka, hecto, kilo, mega, giga, tera, peta, exa);

var
  multiple : array [prefix] of real;
  subscript : prefix;

begin
  multiple [deka] := 10;
  multiple [hecto] := 100;
  multiple [kilo] := 1000;
  for subscript := mega to exa do
    multiple [subscript] := multiple [pred(subscript)] * 1000;
  ...

```

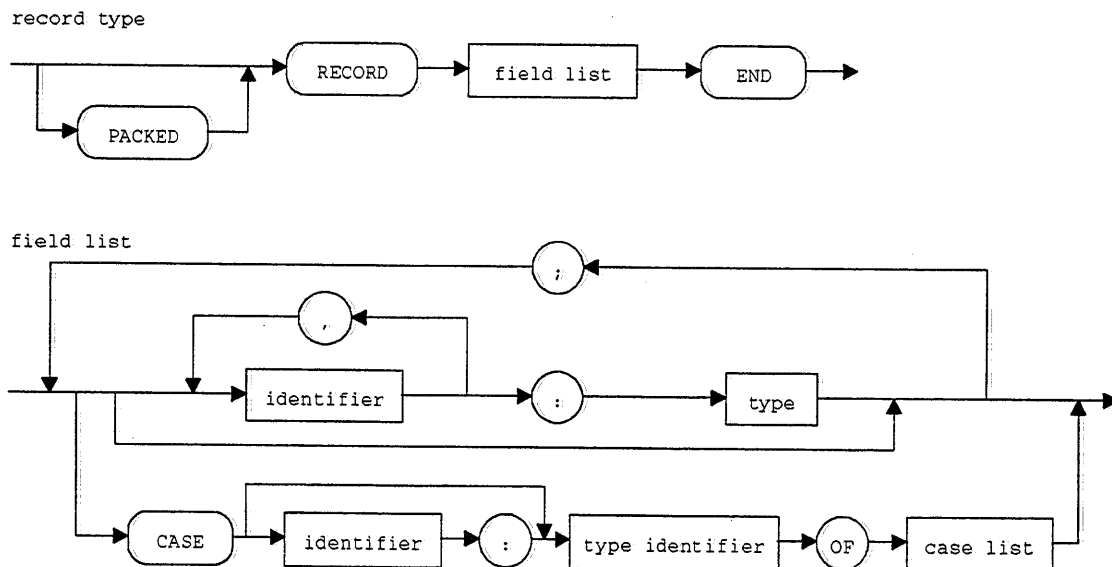
**4.8 Record Types**

Figure 4-6 Syntax Diagram for RECORD Type

The RECORD type is used to define a collection of heterogeneous components. The components, which are called records, consist of elements that may be of different types. The elements within a record are called fields.

**4.8.1 Accessing a Field**

The scope of identifiers used in a record is the RECORD type. Hence, the field names must be unique throughout the record, including the variant part if one is present. This allows any field in a record to be accessed by using the name of the field.

### 4.8.2 The Fixed Part

A RECORD type is a template for a data structure consisting of two parts: the fixed part and the variant part. The fixed part is composed of fields that will occur in every variable of the RECORD type. The variant part is composed of fields which may or may not be present in every variable of the RECORD type. The fixed part of a RECORD type, if present, must always precede the variant part.

### 4.8.3 The Variant Part

The variant part of a RECORD type allows the structure to depend upon the type of data stored in a particular variable of the record. An instance of a variant record may only assume one variant at a time. The different record variants are discriminated by using a tag field.

The tag field is a scalar value that indicates the structure of a RECORD type, i.e., it shows which variant is active. The tag field may be defined in one of three ways:

1. The tag field is an identifier within the fixed part of the RECORD type. References to the tag field have the form:

```
x : integer;
.
.
case x: of;
```

2. The tag field is an identifier defined within a case statement marking the beginning of the variant part of the RECORD type.

```
case x : integer of
```

3. The tag field is not present but is implied by the presence of a type identifier within the case statement marking the beginning of the variant part of the record. For example:

```
case t1 of
```

### 4.8.4 Packed Records

The fields in a record are assigned offsets sequentially by padding where necessary to achieve the required boundary alignment. NDP Pascal allows a packed record to be passed to a procedure or function, although this is prohibited by standard Pascal.

#### EXAMPLE 1

This example illustrates fixed RECORD types:

```
const
    team1 = 'Boston Red Sox      ';
    name1 = 'Fenway Park';

type
    grass = (artificial, natural);
    home_run = record
        left   : real;
        center : real;
        right  : real;
    end;
    stadium = record
        team      : array [1..30] of char;
        name      : array [1..30] of char;
        surface   : grass;
        capacity  : integer;
```

```

        distance : home_run;
    end;

var
    park : stadium;

begin
    park.team     := team1;
    park.name     := name1;
    park.surface  := natural;
    park.capacity := 33583;
    park.distance.lf := 315;
    park.distance.center := 420;
    park.distance.rf := 302;
...

```

### EXAMPLE 2

This example illustrates a variant record. The tag field is the type identifier, TIME:

```

type
    time = (daytime, evening)
    rating = record
        case time of
            daytime : (drama : integer;
                      quiz  : integer;
                      other : integer );
            evening : (informational : integer;
                      general_drama : integer;
                      susp_myster  : integer;
                      sitcom_comedy : integer;
                      feature_film  : integer );
        end;
...

```

## 4.9 Pointer Type

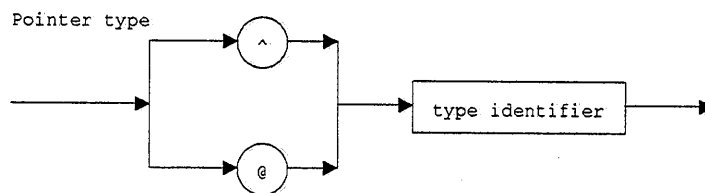


Figure 4-7 Syntax Diagram for Pointer Type

The pointer type is denoted by the caret symbol (^) or by the “commercial at” symbol (@). Either symbol may be used depending upon availability on the keyboard and programmer's preference.

A pointer is a variable that contains a memory address. Pointers are used in Pascal to reference variables that are created during program execution. Variables created in such a manner are called dynamic variables. Dynamic variables are allocated and deallocated by the predefined procedures `NEW` and `DISPOSE`, which are described in *Chapter 10*.

The pointer type is designed to point to a variable that will be created by the function `NEW`. The function `NEW` allocates space for a variable of a specified type, and returns a pointer to its memory location.

Pointers are not interchangeable and are constrained to point to the type for which they were declared. The pointer declaration indicates the type to which the pointer variable may refer. The dynamic variable created by NEW will point to the same type as its argument.

Pascal provides the named constant NIL to refer to the empty pointer. NIL is the value of a pointer that has not been assigned a value, and is compatible with every pointer type.

### 4.9.1 Operations on Pointers

Pointer types may be tested for equality or inequality as the table below shows. Listed are the operators allowed on pointer variables.

Symbol	Usage	Result type	Description
=	$x = y$	BOOLEAN	Compares for $x$ equal to $y$ . Tests if $x$ and $y$ point to the same data item.
<>	$x <> y$	BOOLEAN	Compares for $x$ not equal to $y$ . Tests if $x$ and $y$ point to different data items.

Pointers cannot be used to access individual elements of an array, string, or as an array subscript.

#### EXAMPLE

This example illustrates recursive data types using pointers:

```

type
  cell = record
    element : real;
    next_cell : ^cell;
  end;
  node = record
    element : real;
    leftchild, rightchild : ^node;
  end;
var
  dictionary : array [0..1023] of ^cell;  { A linked list }
  tree : ^node;                          { A 2-3 tree }
  ...

```

## 4.10 File Type

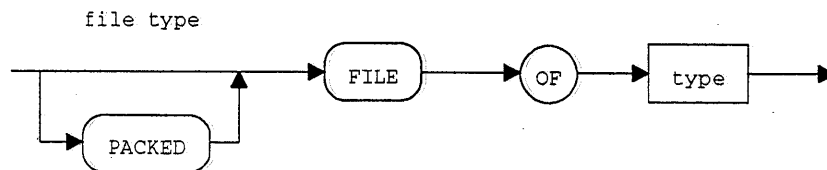


Figure 4-8. Syntax Diagram for File Type

The file type is used to provide for data persistence after a program has ended. The file type defines a collection of records where each record is of the same type. All input and output in Pascal requires use of the file type.

Variables of the file type reference records from the file with pointers called buffer variables or file pointers. The declaration of a file variable  $f$  with type  $T$  includes the implicit declaration of

a buffer variable of type  $T$ . The buffer variable is denoted  $f^{\wedge}$  and references the host operating system's input and output buffers.

Files are accessed through the following predefined functions and procedures. See *Chapter 10* for details of these routines.

<b>Function</b>	<b>Description</b>
EOF ( $f$ )	returns TRUE if file $f$ is at end of file
<b>Procedure</b>	<b>Description</b>
GET ( $f$ )	advances buffer variable $f^{\wedge}$ to the next component of input file $f$
PUT ( $f$ )	advances buffer variable $f^{\wedge}$ to the next component of output file $f$
READ ( $f, v$ )	reads data from file $f$ into variable $v$ end of line on file $f$ .
RESET ( $f, s$ )	opens a file for input
REWRITE ( $f, s$ )	opens a file for output
WRITE ( $f, e$ )	write the value of $e$ to file $f$

The following restrictions apply to the file type:

1. the file type must be passed by VAR to a procedure or function;
2. a file may not be contained within another file.

Note that while it is legal to pack a file type, this has no effect on the file's storage requirements.

#### 4.10.1 Predefined File Type TEXT

Pascal provides the predefined file type TEXT. A file of type TEXT is called a textfile and is used to store data in character format. Textfiles contain markers used to delimit the character data into lines, which improves readability if the file is viewed in printed form. Textfiles imply that the internal representation of the data will be converted to and from character format when the file is accessed. For example, when a program writes a real number to a textfile, the WRITE procedure first converts the number to its character equivalent, which is then transferred to the file. Similarly, a READ operation converts the character data into the form appropriate to the receiving variable's type.

There are two predefined textfiles, INPUT and OUTPUT, which have the following definition:

```
VAR INPUT, OUTPUT : TEXT;
```

The INPUT and OUTPUT files are used as defaults by the predefined I/O functions and procedures. When the file name is omitted from one of the predefined input or output routines, then the file INPUT or OUTPUT is assumed.

A textfile is a special case of a file. The predefined procedures and functions that operate on files operate on textfiles in the same way with the following single exception. The GET procedure returns a space (or blank) character when the end of line marker is encountered in a textfile. Additional routines are defined in Pascal to operate on textfiles and deal specifically with the end of line marker. See *Chapter 10* for details on these routines.

<b>Function</b>	<b>Description</b>
EOLN ( $f$ )	returns TRUE if textfile $f$ is at the end of line
<b>Procedure</b>	<b>Description</b>
PAGE ( $f$ )	writes an ASCII form feed to textfile $f$
READLN ( $f$ )	read data from textfile $f$ into variable $v$ , then advances to end of line on $f$
WRITELN ( $f, e$ )	writes the value of $e$ followed by an end of line marker to textfile $f$

#### EXAMPLE

```
type
  long_name = array [1..1024] of char;
  short_name = array [1..32] of char;
```

```

books = file of record
    title   : long_name;
    author  : short_name;
    publish : short_name;
    date    : integer;
    inPrint : boolean;
    ISBN    : array [1..13] of char;
    price   : real;
end;

production = file of record
    oats, peas, beans, barley : float;
end;

...

```

## 4.11 Packed and Unpacked Types

The purpose of the packed data type is to cause the compiler to store data in as compact a form as possible. The idea of packed and unpacked data types stems from the two ways in which data, particularly character data, can be stored on computers.

For example, an array of characters can be stored in consecutive words in memory. In unpacked format one character is stored at each memory address while in packed format as many characters as possible are stored at each address. This dichotomy is an issue in computers lacking byte addressing where accessing a character in a packed array requires a sequence of shifting and masking instructions. The CDC 6600, upon which Niklaus Wirth implemented an early version of Pascal, was of this type.

The existence of these two formats had a small influence in the design of the Pascal language, notably in rules regarding the passing of packed types as parameters to procedures or functions. As mentioned in the appropriate sections, these rules have remained in NDP Pascal in order to conform with the standard. Packing is a property of data in Pascal that is reflected in the data declaration or type definition. Every data or type definition may be prefixed with the keyword `PACKED`. As such, packing is considered in the rules for type compatibility. A packed type is not the same as its unpacked counterpart. Hence a packed type may not be assigned to an unpacked type, either in an assignment statement, or through parameter transmission.

The predefined procedures `PACK` and `UNPACK` are used to convert unpacked data to packed form, and vice versa. Data cannot be packed by using a type definition of the form:

```

type
    T = someType;
    packedT = PACKED T;

```

While packing is a characteristic of a type definition, knowledge of the internal details of a packed structure is considered a violation of data abstraction. For example, it should not be necessary for a program to know the internal representation of a packed array of reals in order to work correctly. This also would have a severe impact on the portability of programs. This situation arises when a procedure or function has a variable parameter.

The one situation where knowledge of the representation of a packed structure would be required in a program has been explicitly ruled out in Pascal to prevent any violation of data abstraction or independence. The rule is that a component of a packed structure cannot be passed as a variable parameter to a procedure or function. The following is an example of this:

```

program simple (output);
{ This program illustrates passing a packed actual parameter }
{ to a variable formal parameter, . . . which is illegal. }

```

```
procedure add (var i: integer);
begin
    i := i + 1;
end;

var
    a: packed record
        i,j: integer;
    end;

begin
    a.i := 5;
    add (a,i);
end.
```

Notice, however, that a component of a packed structure can be passed to a VALUE parameter. This is because data in the calling routine does not get updated, so knowledge of the internal representation of a packed structure is not necessary. Since the predefined procedures and functions in NDP Pascal are all VALUE parameters, the actual parameters to these routines may be components of packed structures.

# 5

## Variables

Identifiers denoting variables may refer to:

- 1) the entire variable,
- 2) a component of the variable, or
- 3) a variable referenced by a pointer.

In each case, the variable's type shows how it may be referenced.

### 5.1 Entire variables

When a variable's name is used, it refers to the entire variable. Array, record and set variables may be treated as units in this manner.

#### EXAMPLE

```
type
  palette = (black, blue, green, red, white);
  color   = set of palette;
  nametyp = array [1..30] of char;
  occupation = (tinker, tailor, soldier, spy);
  applicant = record
      surname : nametyp;
      field   : occupation;
      available : boolean;
  end;

var
  name1, name2 : nametyp;
  hue1, hue2   : color;
  apprentice1, apprentice2 : applicant;

begin
  . . .
  name2 := name1;
  . . .
  hue2  := hue1;
  . . .
  apprentice2 := apprentice1;
  . . .
```

### 5.2 Component Variables

A component variable is used to access an element of an array, record, or file. The variable's type indicates the syntax to be used to specify the component. An array element is accessed by an indexed variable, a record component is accessed by a field designator, and a record in a file is accessed by the file's buffer variable or file pointer.

#### 5.2.1 Indexed Variables

A component of an array is selected by specifying an index for the component. The index must appear enclosed in square brackets after the array name. The index is any expression that is assignment compatible with the index type specified in the array declaration.



Multi-dimensional arrays may be referenced in one of two ways:

1. By separating each index with a comma, and placing this list within one set of square brackets, or
2. listing each index one after another, each index enclosed in its own set of brackets.

For example,  $a[i, j, k]$  may be written in the form  $a[i][j][k]$ .

#### EXAMPLE 1

```

type
  prefix = (deka, hecto, kilo, megas, giga, tera, peta, exz);
var
  multiple : array [prefix] of real;
  subscript : prefix;
begin
  multiple [deka] := 10;
  multiple [hecto] := 100;
  multiple [kilo] := 1000;
  for subscript := megas to exa do
    multiple [subscript] := multiple [pred9subscript] * 1000;

```

#### EXAMPLE 2

```

var
  a : array [1..n] of real;
  d : real;
begin
  d := a [1,1] * a[2,2] - a[1,2] * a[2,1];
  ...

```

### 5.2.2 Field Designators

A field of a record is denoted by the record variable followed by the field name separated by a period.

#### EXAMPLE

```

const
  teamY = 'New York Yankees';
  nameY = 'Yankee Stadium';
type
  grass = (artificial, natural0;
  home_run = record
    left : real;
    center : real;
    right : real;
  end;
  s30 = packed array [1..30] of char;
  stadium = record
    team : s30;
    name : s30;
    surface : grass;
    capacity : integer;
    distance : home_run;
  end;

```

```

procedure copy30 (var d:s30; s:s30);
{ Utility routine to copy source s to destination d. }
var i:integer;
  begin for i:= 1 to 30 do d[i]:=s[i] end;

var
  park : stadium;

begin
  copy30 (park.team, teamY);
  copy30 (park.name, nameY);
  park.surface := natural;
  park.capacity := 57545;
  park.distance.left :=312.0;
  park.distance.center:=410.0;
  park.distance.right :=310.0;
  ...

```

### 5.2.3 File Referencing

There are two ways to access data within a file. One way is to use the file's buffer variable and the predefined GET and PUT procedures to access the host operating system's file buffer. The other way is to use the predefined procedures, READ, READLN, WRITE, and WRITELN. This section briefly describes how to access a file with the predefined GET and PUT procedures.

Variables of the file type reference records from the file with pointers called buffer variables or file pointers. The declaration

```
var f file of T
```

declares the file variable  $f$  with type  $T$  and includes the implicit declaration of a buffer variable of type  $T$ . The buffer variable is denoted  $f^\wedge$ .

The buffer variable points to the current record in the file, and is accessed using the notation  $f^\wedge$ . The buffer variable may be used as an ordinary Pascal variable in an assignment statement or passed as a parameter to a procedure or function. For example, if  $itemT$  is a variable of type  $T$ , then the current record of file  $f$  is accessed with the assignment statement:

```
itemT := f^;
```

The predefined procedures GET and PUT use the buffer variable to read and write to files. The PUT procedure takes data pointed to by the buffer variable and appends it to the file. The GET procedure advances the current file position to the next component and copies the value of the component to the buffer variable.

### 5.3 Pointer Referencing

The NEW procedure returns a pointer to a newly created variable. This pointer must be stored in a pointer variable. Either the pointer variable, or the dynamic variable to which it points, may be referenced. The pointer variable is accessed using its name, and the dynamic variable is accessed by appending an up arrow ( $^\wedge$ ) to the pointer variable. For example, with the following declarations:

```

type
  t = {some type definition}

var
  p, q: ^t;

```

then execution of the statement

```
NEW (p);
```

allocates a dynamic variable of type  $t$ , and assigns its address to  $p$ . The pointer variable  $p$  is bound to a dynamic variable of type  $t$ , and  $p^{\wedge}$  denotes the dynamic variable.

### EXAMPLE 1

The following two uses of pointer variables are illegal and would result in a type mismatch compilation error.

```

type t1 = integer;
var p, q : ^t1;
begin
  new (q);
  q^ := 123;      { q points to the integer 123 }
  p := q;        { p now points to what q points to,   }
                  { p and q both point to the same item }
  . . .
  p^ := 456;      { q points to the integer 456 }
  p^ := q^;      { what p points to is replaced by what q points to, }
                  { so 456 is replaced by 123.           }
  . . .

```

### EXAMPLE 2

```

type t1 = integer;
var p, q : ^t1;
begin
  new (q);
  p := q^; { type mismatch, p is a pointer type and q^ is an integer }
  p^ := q; { type mismatch, q is a pointer type and p^ is an integer }
  . . .

```

# 6 Expressions

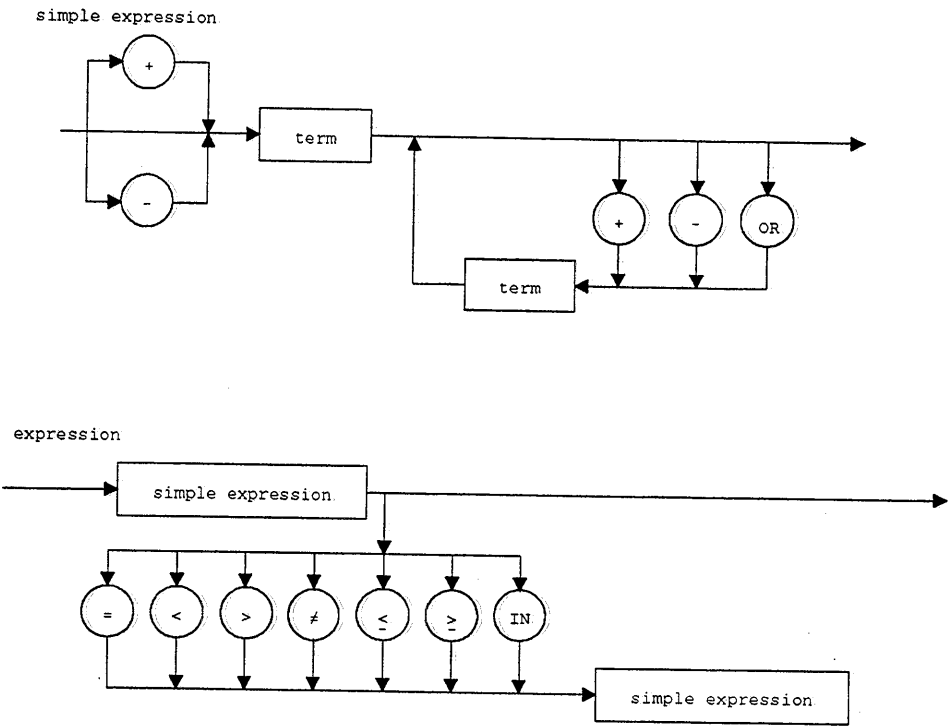


Figure 6-1 Syntax Diagrams for Factor and Term

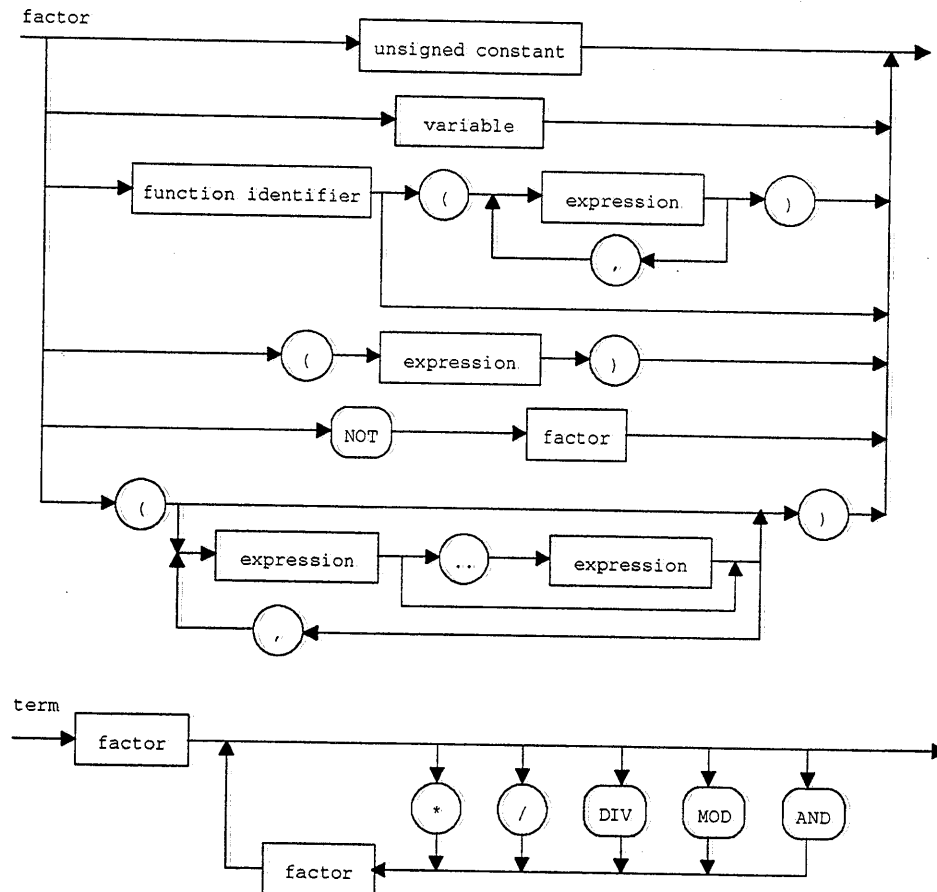


Figure 6-2 Syntax Diagrams for Simple Expression and Expression

## 6.1 Operators

Expressions permit the evaluation of mathematical formulas using constants, variables, and operators. The operators in Pascal are divided into four groups according to their evaluation precedence:

- 1) the not operator,
- 2) the multiplying operators,
- 3) the adding operators,
- 4) the relational operators,

where (1) has the highest priority or precedence and (4) the lowest. An expression is evaluated by performing the operators with the highest precedence first, then those with the next highest precedence, and so on. When operators of equal precedence occur together, they are evaluated from the left to the right. Parentheses may be used to alter the evaluation order since expressions within parentheses are evaluated first.

The Pascal standard states that no assumptions may be made regarding the order in which operands are evaluated within an expression, other than the above precedence rules. Hence programs that rely on a left to right, or right to left, evaluation order of the operands in an expression are illegal.

Note that some logical operators are at the same level of precedence as arithmetic operators. For example, boolean "and" has the same precedence as arithmetic multiplications, and boolean "or" has the same precedence with arithmetic addition. This is distinctly unlike the precedence levels defined in other programming languages.

The following tables list the four groups of operators, in decreasing order of precedence.

**The Not Operator**

- ~ boolean not
- ~ one's complement
- ~ set complement

**Multiplying Operators**

- \* multiplication
- \* set intersection
- / real division
- div integer division
- mod integer remainder
- & boolean and
- & logical and
- << logical left shift
- >> logical right shift

**Adding operators**

- + addition or unary plus
- + set union
- subtraction or unary negation
- set difference
- | boolean or
- | logical or

**The Relational operators**

- = compares equal
- <> compares not equal
- < compares less than
- <= compares < or =
- <= set subset
- > compares greater
- >= compares > or =
- >= set superset
- in set membership

## 6.2 Boolean Expressions

This section presents two points of caution regarding the evaluation of Boolean expressions. The first is that the Boolean operators have a higher precedence than the relational operators, and second, that optimizations performed during the evaluation of Boolean expressions may preclude some parts of the expression from being evaluated.

The order of evaluation of expressions involving **BOOLEAN** and **RELATIONAL** terms may not be intuitive. Since "or" has a higher precedence than "=", the following expression:

$$x = y \text{ or } u = v$$

will be evaluated as

$$(x = (y \text{ or } u)) = v$$

and not as

$$(x = y) \text{ or } (u = v)$$

The evaluation of Boolean expressions is optimized to avoid evaluating an operand if the result can be determined without doing so. For example, in the expression

$$x := y \text{ or } z$$

if  $y$  is true, there is no need to evaluate  $z$ . When an expression involves functions, that function may not be evaluated. For instance, in the expression

$$x := y \text{ or } f(z)$$

if  $y$  is true, there is no need to evaluate  $f(z)$ . It is dangerous to rely on side effects from functions in expressions since these functions may not be evaluated.

### 6.3 Function Call

A function returns a value at that point in an expression where it was invoked. The parameters in the function call must match the number and type of the parameters in the function declaration. The actual parameters must be assignment compatible with the formal parameters.

A field of a packed record or an element of a packed array cannot be passed as a VAR parameters to a function.

#### EXAMPLES

```
y := A*sin ( w*t + phase * sin (f*t) );
t := sqrt ( (sqrt(r) - 1) / (sqrt(r) + 1));
```

### 6.4 Set Constructor

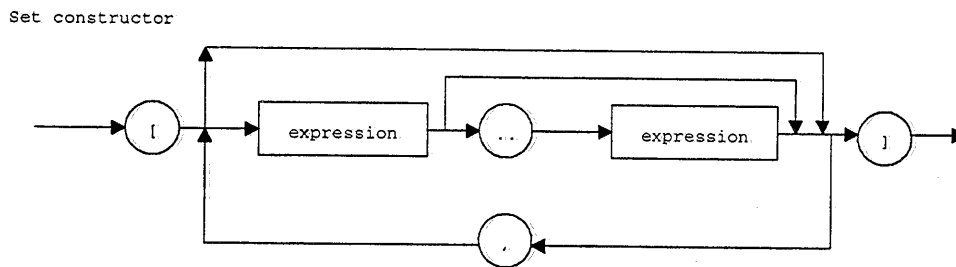


Figure 6-3 Syntax Diagram for Set Constructor

A set constructor is one or more elements of a set enclosed in square brackets.

Each element in a set constructor is either an expression, or a pair of expressions separated by two dots. All expressions must be of the same type and in the base type of the set. The pair of expressions represent the lower and upper bounds of a range of elements in the base type. An element of a set cannot be a set.

If the value of the first expression is greater than the value of the second expression, then  $[expression..expression]$  denotes the empty set.

If the base type is INTEGER, then the largest set contains 32 elements by default, or 256 elements if the appropriate compiler switch is used. For more information on NDP Pascal's compiler switches, refer to the *NDP User Manual*.

#### EXAMPLES

```
const
    etc = 'et cetera';

type
    months = set of (jan, feb, mar, apr, may, jun,
                    jul, aug, sep, oct, nov, dec);

var
    vacation : months;
```

```

p, q      : boolean;
a, b, c, s : real;

+- - - - - +
|  factors  |
+- - - - - +
365
a
etc
[jun..aug, jan]
(b*b - 4.0 * a * c)
sqrt ( s * (s-a) * (s-b) * (s-c) )
not q
not (p and q)

+- - - - - +
|  terms  |
+- - - - - +
a + b mod c
(not p) and (not q)
a*a
sqrt((b*b - 4.0*a*c)) / (a+a)
[jan..aug] * [jun..dec]

+- - - - - - - - - - +
|  simple expressions  |
+- - - - - - - - - - +
(p and q) or (not p and not q)
a*a + b*b
-b + sqrt ((b*b - 4.0*a*c)) / (a+a)

+- - - - - - - - - - +
|  expressions  |
+- - - - - - - - - - +
b*b => 4.0 * a * c
p = q
vacation in [jun..aug, jan]

```





# 7

# Statements

## 7.1 Statement Summary

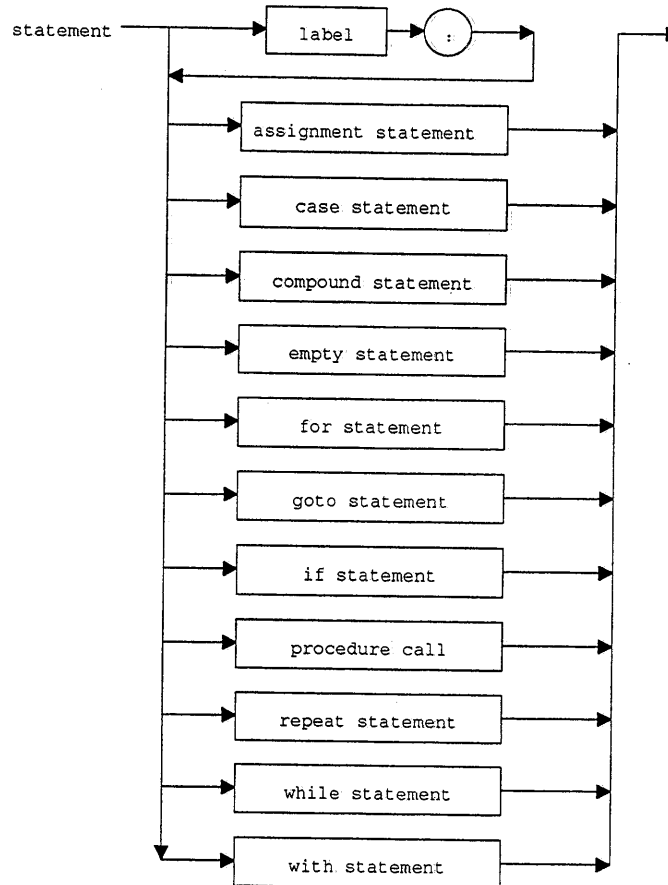


Figure 7-1 Syntax Diagram for Statement

The above syntax diagram summarizes the statements available in NDP Pascal. Each statement is described in detail in a section of this chapter.

## 7.2 The ASSIGNMENT Statement

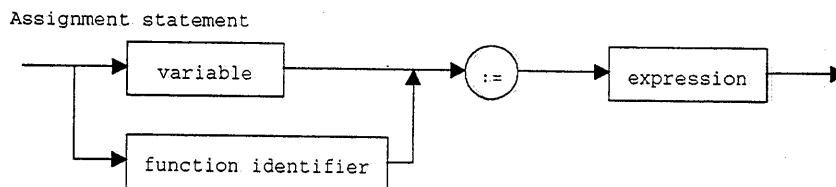


Figure 7-2 Syntax Diagram for Assignment Statement

The assignment statement assigns the value of an expression to a variable or to a function identifier. The variable or identifier, and the expression, must be assignment compatible. Type compatibility is described in *Section 4.2*.

The assignment statement permits entire arrays or records to be assigned.

## EXAMPLES

```

centigrade := (fahrenheit - 32.0) / 1.8;
E := m * c*c;
done := abs (x-y) < epsilon;

```

## 7.3 The CASE Statement

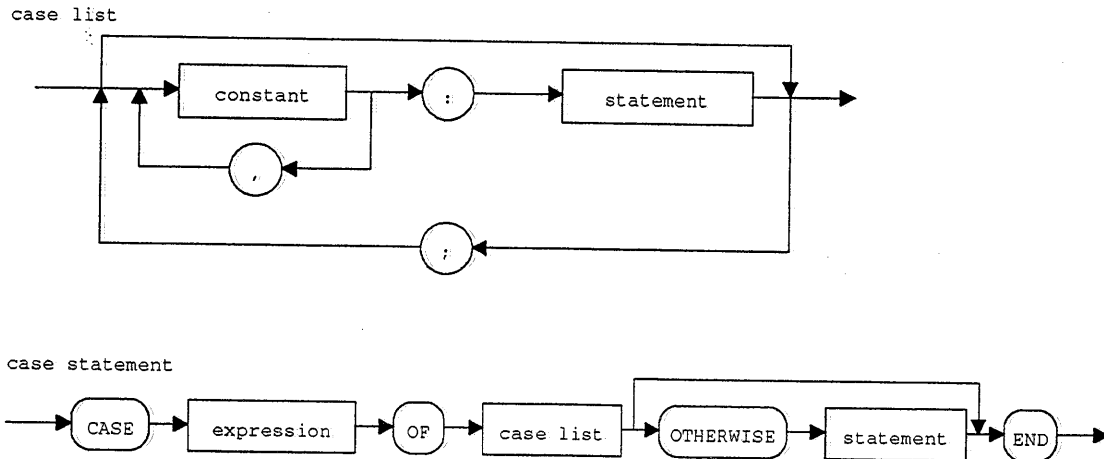


Figure 7-3 Syntax Diagrams for CASE List and CASE Statement

The case statement provides a multiple branch capability based upon the result of an expression. The case statement consists of an expression followed by a list of statements. The expression is called the selector, and must evaluate to any scalar type except REAL. The list of statements is prefixed by one or more values of the selector type, separated by commas. These values are called the case labels.

NDP Pascal evaluates the selector and then transfers control to the statement with the corresponding case label, or to the OTHERWISE clause if it is present and no case labels match the value of the selector.

The case labels may appear in any order, but they may only be listed once in a single case statement. If no case label equals the value of the expression, and the otherwise clause is not present, then the statement following the case statement is executed.

## EXAMPLE 1

```

type
  solid = (cylinder, sphere, prism, cone);

var
  shape : solid
  ...
case shape of
  cylinder : I := mass * (r*r) / 2.0;
  sphere   : I := 2.0 * mass * (r*r) / 5.0;
  prism    : I := ( a*a + b*b ) / 12.0;
  cone     : I := 3.0 * mass * (r*r) / 10.0;
end;
...

```

## EXAMPLE 2

```

var
  age : integer
  ...
case age of

```

```

1      : infant;
2      : toddler;
3,4,5  : preschool;
6..11  : elementary;
12,13,14 : juniorHigh;
15..18 : highSchool;
otherwise
      begin getUp; work; sleep end
end;
...

```

## 7.4 The COMPOUND Statement

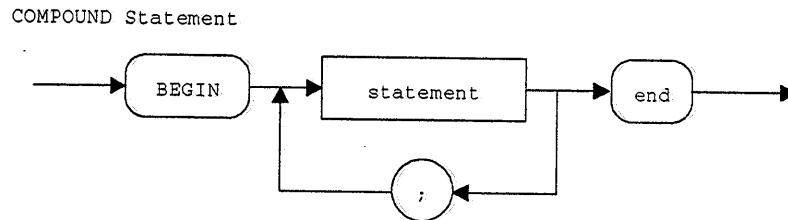


Figure 7-4 Syntax Diagram for COMPOUND Statement

The COMPOUND statement groups several statements into a single statement. The reserved words “begin” and “end” are used to bracket a series of statements that are to be executed sequentially. The statements within the compound statement are separated by semicolons.

The body of a Pascal program, procedure, or function consists of a single compound statement.

### EXAMPLE 1

```
begin t:=a[i]; a[i]:=b[j]; b[j]:=t end;
```

### EXAMPLE 2

```

const
  m = {some integer constant}
  n = {some integer constant}

var
  a : array [1..m+1] of integer;
  b : array [1..n+1] of integer;
  c : array [1..m+n] of integer;
  i, j, k : integer;

begin
  { this fragment merges the two sorted arrays a and b into c }
  i:=1; j:=1;
  a[m+1] := maxint; b[n+1] := maxint;
  for k := 1 to m+n do
  if a[i] < a[j]
    then begin c[k]:=a[i]; i:=i+1 end
    else begin c[k]:=b[j]; j:=j+1 end;
  ...

```

## 7.5 The EMPTY Statement

EMPTY statement

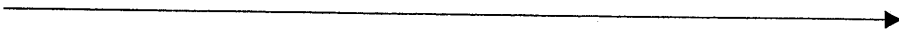


Figure 7-5 Syntax Diagram for EMPTY Statement

The EMPTY statement is a statement that does not do anything. It consists of no symbols and has no effect on the execution of a program. The EMPTY statement serves mainly as a convenience in many programming situations. The following are typical uses of the EMPTY statement:

- 1) as a place holder for a label,
- 2) to allow the existence of extra semicolons in a program,
- 3) to simplify the coding of certain IF constructs.

The blank spaces in the following examples are the statement separators. The EMPTY statement has zero length.

### EXAMPLE 1

The EMPTY statement allows control to be transferred to the end of a compound statement.

```
begin
    ...
    goto 99;
    ...
99:   {empty statement}
end;
```

### EXAMPLE 2

The EMPTY statement following the assignment "c:=3" allows the semicolon to be used where not necessary. This provides coding symmetry and eases modification.

```
begin
    a := 1;
    b := 2;
    c := 3;
end;
```

### EXAMPLE 3

The following IF statement can be rewritten in a form that avoids negative logic using the EMPTY statement. The statement:

```
if not e1 then s1
  else
    if e2 then s2;
```

may be rewritten as the following:

```
if e1 then
  if e2 then s2
  else
    { empty statement }
else
  s1;
```

## 7.6 The FOR Statement

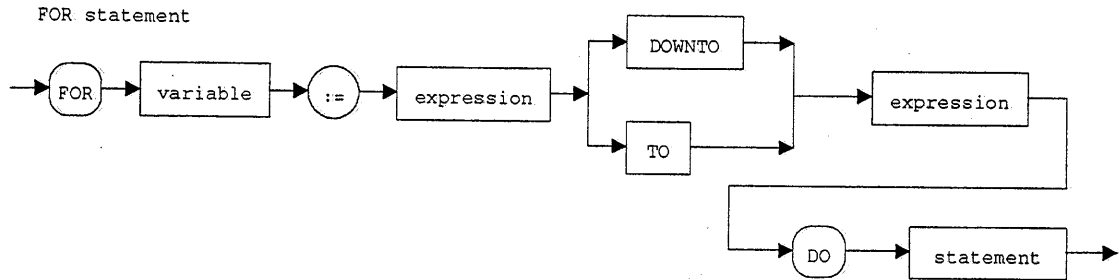


Figure 7-6 Syntax Diagram of FOR Statement

The FOR statement repeatedly executes a statement while a progression of values is assigned to a variable, called the control variable of the FOR statement.

The initial and final values of the control variable are determined once, when the FOR statement begins execution.

The FOR statement has two forms: one where the control variable increases in value and one where it decreases in value. These forms are distinguished by the reserved words TO and DOWNTO. For example, the control variable named *cv* increases in the form:

```
for cv := init to final do stmt;
```

and decreases in the form:

```
for cv := init downto final do stmt;
```

Here are the rules governing control variables in FOR statements:

1. The control variable must be a scalar type, and be assignment compatible with the initial and final expressions.
2. The control variable must be the entire variable, not an element of a structure, subscripted, field qualified or referenced through a pointer. The control variable can be a value parameter in a function or subroutine, but cannot be a variable (VAR) parameter.
3. The control variable must be within the scope of the FOR statement.
4. The control variable may not be altered in the body of the FOR statement.
5. The value of the control variable is considered undefined after the FOR statement.

Programs should not depend upon the final value of any FOR statement control variable.

The control variable serves as a counter, and is incremented with the `SUCC` function or decremented with the `PRED` function at the end of each FOR iteration. The progression of values assigned to the control variable begins with the value of the first expression, and ends with the value of the second expression.

Execution of the FOR statement proceeds by evaluating the two component expressions and then initializing the control variable. The value of the control variable is tested before the execution of the component statement. If the control variable is increasing in value, then the component statement is executed if the control variable is less than or equal to its final value. If the control variable is decreasing in value, then the component statement is executed if the control variable is greater than or equal to its final value.

control variable	FOR terminates when
increasing (TO)	control variable > final value
decreasing (DOWNTO)	control variable < final value

The component statement will not be executed at all if the initial value is greater than the final value in the ascending case, or if the initial value is less than the final value in the descending case.

**EXAMPLE 1**

This example evaluates an n-th degree polynomial contained in the array `poly` using Horner's method. The constant term is in element 0, and the coefficient of the n-th term is in element n.

```
var
  poly : array [0..n] of real;
  i : integer;

begin
  y := poly [n];
  for i := n-1 downto 0 do y := x*y + poly [i]
  ...
```

**EXAMPLE 2**

This code fragment forms the product of two n-th degree polynomials.

```
var
  p1, p2, product : array [0..n] of real;
  i, j : integer;

begin
  for i:=0 to 2*(n-1) do product [i] := 0;
  for i:=0 to n-1 do
    for j:=0 to n-1 do
      product [i+j] := product [i+j] + p1[i]*p2[j];
  ...
```

**EXAMPLE 3**

The following use of a variable parameter as a control variable in a FOR statement is illegal.

```
procedure setup (var i:integer1 ch:char);
var list : array [1..10] of char;
begin
  for i := 1 to 10 do list[i] := ch;
end;
```

**7.7 The GOTO Statement**

GOTO statement

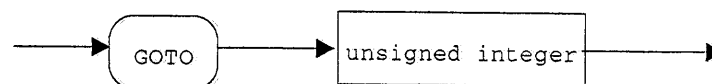


Figure 7-7 Syntax Diagram for GOTO Statement

The GOTO statement transfers control to the target label. The label must be declared within the routine that contains the GOTO. Since the scope of a label is the routine in which it is declared, it is not possible to jump into or out of a procedure or function.

There are several technical restrictions on the use of the GOTO statement that are roughly equivalent to saying that jumping into a structured statement is not allowed. Specifically, the following use of the GOTO is illegal:

1. Jumping into a compound statement from outside of the statement.
2. Jumping into a FOR, REPEAT, or WHILE loop from outside of one of these loops.
3. Jumping into a WITH statement.
4. Jumping into an IF statement, or jumping between the THEN and ELSE portions of the IF statement.

5. Jumping into a CASE statement, or between the alternatives of a CASE statement.

### EXAMPLE

This example illustrates the use of the GOTO statement and may be rewritten without the GOTO statement by using another variable as a flag.

```

const
  n = 10;

type
  t1 = array [1..n] of integer;

function common (a, b: t1) : boolean;
{ Returns true if the arrays a and b have an element in common. }
label 99;

var i, j : integer;
    result : boolean;

begin
  result := false;
  for i := 1 to n do
    for j := 1 to n do
      if a[i] = b[j] then begin
        result := true;
        goto 99
      end;
    99: common := result;
  end;

```

## 7.8 The IF Statement

IF statement

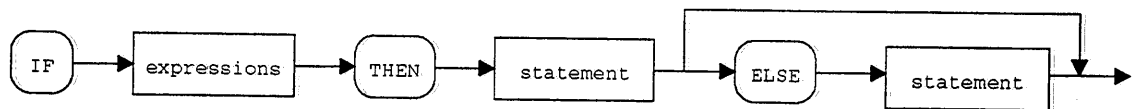


Figure 7-8 Syntax Diagram for IF Statement

The IF statement allows one of two possible statements to be executed depending upon the result of a Boolean expression. The IF statement consists of a THEN clause, optionally followed by an ELSE clause. Note that there is no semicolon between the first statement and the keyword ELSE.

If the Boolean expression evaluates to TRUE, then the statement in the THEN clause is executed, and the ELSE clause is skipped if it is present.

If the Boolean expression evaluates to FALSE, then the statement in the ELSE clause is executed and the THEN clause is skipped.

When two IF statements are nested and there is only one ELSE clause, then the ELSE clause goes to the most recent unmatched IF (scanning textually backwards from the ELSE).

For example, the statement:

```
if e1 then s1 if e2 then s2 else s3;
```

is evaluated as

```

if e1 then
  begin
    if e2 then s2
    else

```



```

                s3
    end ;
and not as
    if e1 then begin if e2 then s2 end
        else
            s3;

```

**EXAMPLE**

```

d := b*b - 4.0*a*c;

if abs(d) < epsilon then begin
    x1.r := -b / 2.0*a;
    x2.r := x1.r
end
else if d > 0 then begin
    x1.r := (-b + sqrt(d)) / 2.0*a;
    x2.r := (-b - sqrt(d)) / 2.0*a
end
else begin
    x1.r := -b / 2.0*a
    x1.i := sqrt(-d);
    x2.r := x1.r;
    x2.i := -x1.i
end;

```

...

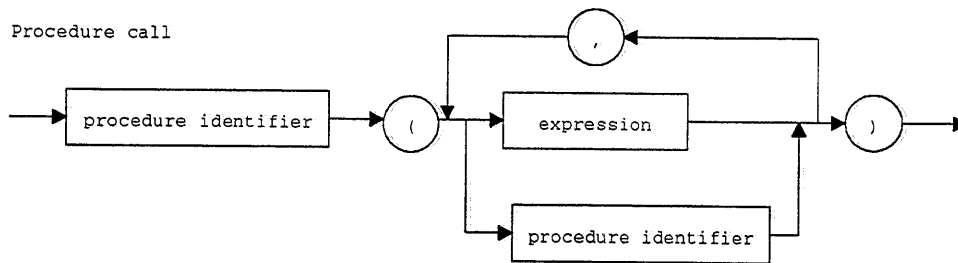
**7.9 The PROCEDURE Statement**

Figure 7-9: Syntax Diagram for Procedure Call Statement

The PROCEDURE statement causes the named routine to be executed and control returned to the statement following the call. The arguments supplied by the calling routine are called the actual parameters, while the parameters in the procedure declaration are called the formal parameters. The number of actual parameters must equal the number of formal parameters. If present, actual parameters are substituted for the formal parameters in the order in which they occur. The first actual parameter is matched with the first formal parameter, and so on. The actual parameters must be assignment compatible with the corresponding formal parameters.

The order in which the actual parameters are evaluated and associated with its corresponding formal parameter is not defined.

Formal parameters that have been declared as VAR require a variable identifier to appear in the corresponding actual. Hence, expressions or constants cannot be passed to VAR parameters.

Fields of a packed record cannot be passed to a VAR parameter.

An expression or constant can be used whenever the formal parameter is passed by value.

**EXAMPLE**

```

procedure quicksort (l, r: integer);
var i: integer;
begin
  if r > 1 then begin
    i := partition (l, r);
    quicksort (l, i-1);
    quicksort (i+1, r);
  end;
end;

```

**7.10 The REPEAT Statement**

REPEAT statement

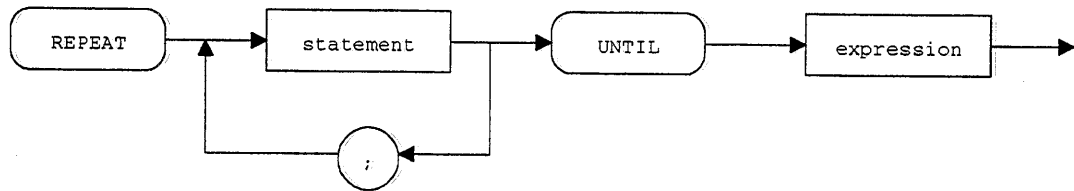


Figure 7-10 Syntax Diagram for REPEAT Statement

The REPEAT statement repeatedly executes a series of statements until a Boolean control expression becomes TRUE. The statements that constitute the loop are bracketed by the reserve words REPEAT and UNTIL. The Boolean control expression is evaluated at the end of the loop, and so the loop executes at least once.

Notice that the REPEAT statement encloses a list of statements in a manner similar to the COMPOUND statement.

**EXAMPLE**

```

const
  n = 10;

var
  a : array [1..n] of integer;

procedure selectk (k:integer);
{ Select the k-th smallest item out of an array of N items. }
{ The global array a is rearranged so that a[1],.. .,a[k] are less than }
{ or equal to a[k] and a[k+1], . . . ,a[n] are greater than or equal to a[k].}
var
  left, right, i, j, s, t : integer;

begin
  left := 1; right := n;
  while right > left do begin
    s:=a[right]; i:=left-1; j:=right;
    repeat
      repeat i:=i+1 until a[i] >= s;
      repeat j:=j-1 until a[j] <= s;
      t:=a[i]; a[i]:=a[j]; a[j]:=t;
    until j<=i;
    a[j]:=a[i]; a[i]:=a[right]; a[right]:=t;
    if i>=k then right:=i-1;
    if i<=k then left :=i+1;
  end;
end;

```

```

    end;
end; {selectk}

```

## 7.11 The WHILE Statement

WHILE statement

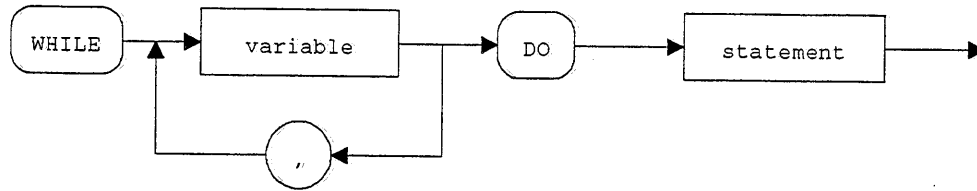


Figure 7-11 Syntax Diagram for WHILE Statement

The WHILE statement repeatedly executes a statement until a Boolean expression becomes false. Since the Boolean expression is evaluated at the beginning of each loop, the statement will be executed zero or more times.

### EXAMPLE

```

const
  n = { some integer constant }
  sentinel = 0x80000000;

var
  a : array [0..n] of integer;

procedure insertionSort;
{ Sort the elements of array a }
var i, j, s : integer;

begin
  a[0] := sentinel;
  for i := 2 to n do begin
    s := a[i];
    j := i;
    while a[j-1] > s do begin
      a[j] := a[j-1];
      j := j-1
    end;
    a[j] := s;
  end;
end;
end;

```

## 7.12 The WITH Statement

WITH statement

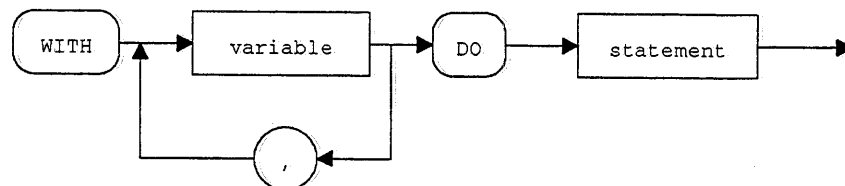


Figure 7-12 Syntax Diagram for WITH Statement

The WITH statement is used to simplify references to individual fields of a record variable. The WITH statement increases the scope of a statement so that the field names of a record variable may be used directly.

Nested WITH statements are abbreviated by separating the record variables by commas following a single WITH.

#### EXAMPLE 1

This is the example from *Section 4.8.4* redone using the WITH statement.

```
with park do begin
  team := teamY
  name := nameY;
  surface := natural;
  capacity := 57545;
  with distance do begin
    left :=312;
    center:=410;
    right :=310;
  end;
```

#### EXAMPLE 2

The following code fragment illustrates nested WITH statements.

```
with park, distance do
  begin
    left :=312;
    center:=410;
    right :=310;
  end;
```



# 8

# Procedures and Functions

## 8.1 Procedure and Function Declarations

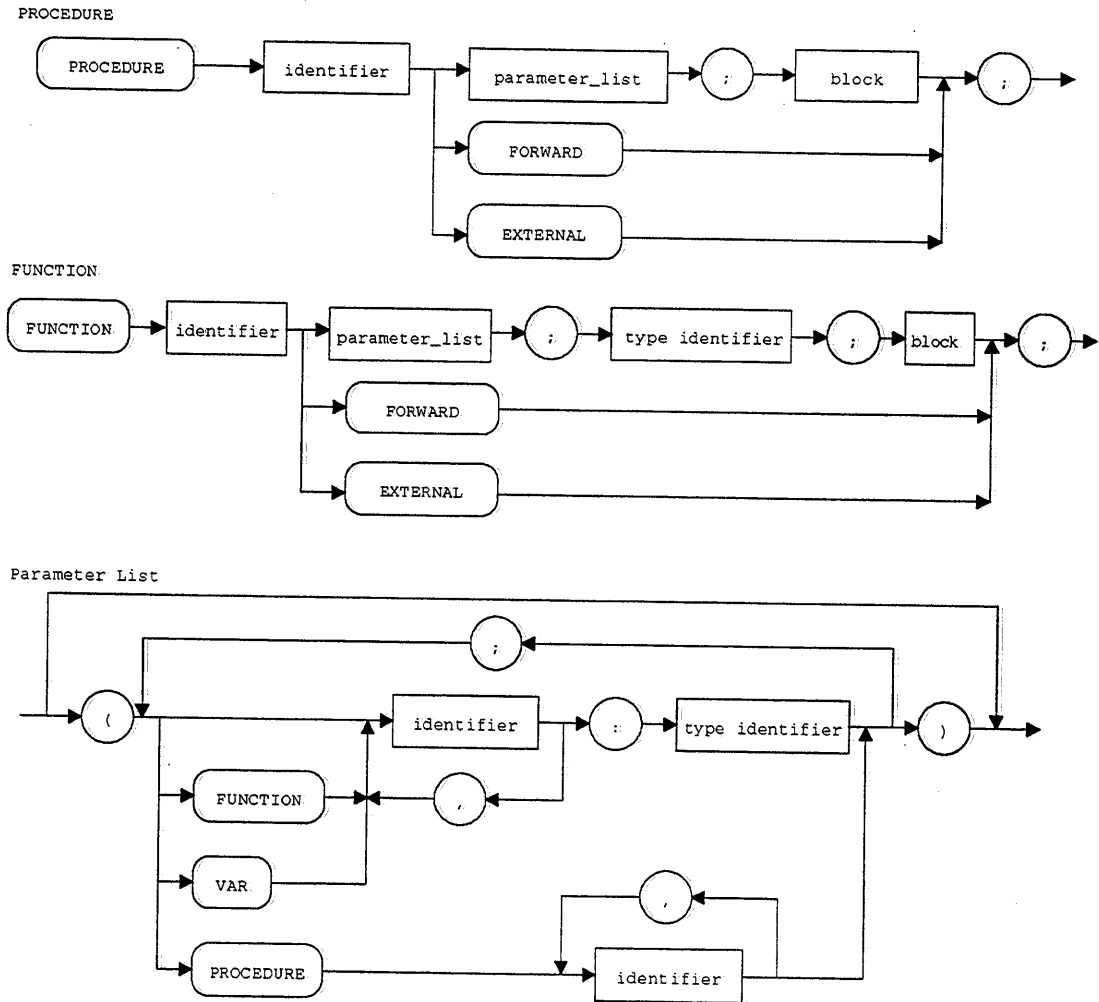


Figure 8-1 Syntax Diagrams: Procedure, Function and Parameter List

Procedures and functions are subprograms that are contained within a program and may be nested within one another. Procedures can be thought of as adding statements to a language, while functions increase the ability to manipulate data in expressions. A procedure is begun or invoked by using a procedure statement. A function is invoked by referencing it in an expression within a statement. Procedures and functions return data to the calling routine, either by variable (VAR) parameters, or through data common to both the caller and callee. Functions also return a value to the caller.

The format of a procedure or function consists of a heading, declarations, and a body. This format is identical to that of a program except for the heading. The heading specifies a unique identifier used to name the routine and contains a list of objects called the formal parameter list. The declaration section is optional and consists of definitions that are to be local to this routine. The body consists of a single compound statement: begin..end. It is the body of the procedure or function that gets executed when the routine is invoked.

Explicit type names must be used when declaring identifiers in a procedure or function heading. This includes value, variable, and function parameters. Either a predefined type or a used defined type may be used for this purpose. For example, the following procedure declaration is illegal:

```
procedure sum (x: array[1..5] of real);
```

and must be replaced with the following:

```
type
  a5 = array [1..5] of real;
procedure sum (x: a5);
```

A function declaration has the same form as a procedure declaration, except that the heading gives the type of value returned by the function. A function may return any scalar value or pointer type. There must be at least one assignment statement that assigns a value to the function identifier within the body of the function.

Procedure and function names must be declared before their use. The unique name in the procedure or function heading is used to invoke the subprogram. The scope of a procedure or function identifier is the block in which the routine is declared. The syntax of a block is given in *Section 2.1*. The use of the EXTERNAL directive is described in *Section 3.6*, and the FORWARD directive is described in *Section 8.4*.

In this reference manual, the word routine will be used to refer to either a procedure or a function.

## 8.2 Parameter Transmission

The formal parameter list contains a fixed number of data objects containing the name and data type of each parameter. When a procedure or function is invoked, the arguments in the procedure or function call are substituted for the formal parameters. The arguments in the procedure or function call are called the actual parameters, and are substituted in the order in which they occur for the formal parameters, or formals. The number of actuals must equal the number of formals, and be assignment compatible with them.

The formal parameter list may contain three types of parameters:

1. value parameter,
2. variable or reference parameter, and
3. procedure or function.

Transmission by value is assumed by default, while transmission by variable is indicated by prefixing each designated parameter with the keyword VAR. Procedures or functions passed as parameters are called formal procedures or formal functions. Notice that functions may return results through variable parameters.

The undesirable side effects due to aliasing can occur in two forms during parameter transmission. The first occurs when the name of an identifier is used more than once as an actual variable parameter. For example, in the following code fragment, x becomes an alias for both s and t:

```
procedure p (VAR s,t: real);
begin
  { body of p }
end;
begin
  ...
  p (x, x);
```

The second instance of aliasing occurring in parameter transmission is a global identifier being used as an actual variable parameter. For example, in the following code fragment, the global variable *y* becomes an alias for the parameter *v*:

```
VAR
  y: real;

procedure q (VAR v: real);
begin
  { body of q }
end;

begin
  ...
  q (y);
```

### 8.2.1 Value Parameters

A value parameter is a formal parameter that is a local variable in the procedure or function and is used to transmit data to the called routine. The corresponding actual parameter determines the initial value of a value parameter when the procedure or function is invoked. A pass-by-value parameter may be manipulated as an ordinary variable in the procedure or function, but any modification to it is not reflected back to the calling routine.

A value parameter may be a constant, a variable, or an expression of any type except the file type. The actual value parameter must be assignment compatible with its corresponding formal value parameter.

### 8.2.2 Variable Parameters

A variable parameter is a formal parameter that is prefixed by the reserve word *VAR* in the routine heading, and is used to transmit and receive data from the called routine. A variable parameter may be initialized in the calling routine, and may be manipulated as an ordinary variable within the procedure or function. Any change made to a variable parameter is reflected in the actual parameter in the calling routine. Hence, variable parameters permit results to be passed back to the calling routine.

Only variables may be passed as variable parameters. Constants, expressions, elements of a packed array, and fields of a packed record may not be passed as variable parameters. A variable parameter must be the same type as its corresponding formal parameter.

### 8.2.3 Formal Routine Parameters

A formal routine parameter is a procedure or function that is passed as a parameter to another procedure or function. Within the called routine, the formal parameter may be used as any other procedure or function. A formal routine parameter must include the complete procedure or function heading, including the number and type of parameters required by the routine. This is required so that the compiler can verify that any call of the routine using the formal parameter name is correct. The actual parameter expression consists of the procedure or function heading.

Non-local variables used by a formal routine parameter are those in effect at the time the formal procedure or function is passed as a parameter, not those in effect when it is activated.

Predefined functions and procedures may not be passed as actual parameters.



### 8.3 Function Results

results A function returns a value that must be either a pointer or a scalar type. The function assigns a value to the function name before leaving the function. This value must be assignment compatible with the type of the function and is returned to the calling expression at the place where the function was invoked.

If the function name occurs on the right hand side of an expression within the function, then it is interpreted as a recursive call.

### 8.4 The FORWARD Directive

The purpose of the FORWARD directive is to allow a procedure or function name to be used before it is defined. This is necessary when two routines call each other at the same level of nesting. Such routines are said to be mutually recursive.

The FORWARD directive informs the compiler that the routine heading just given will be separated from its declarations and body. That is, the declarations and body will be declared somewhere forward in the program.

When used, the FORWARD directive simply follows the procedure or function heading. When time comes to supply the body of the routine, the heading is repeated without the parameter list (otherwise the formal parameters would be declared twice and flagged as an error). When the body of a function, which has been declared forward, is provided, then both the parameter list and return type are omitted.

#### EXAMPLE 1

The following example illustrates how to pass a function as a parameter to a procedure. The program calculates the area under two different curves, a simple step function, and the cosine function, and prints the following result:

```

Area under step function from 0 to 2 is 1.00166666666666737e+00
Area under cosine curve from 0 to pi/2 is 1.0000000002113842e+00
-----
program formal;

procedure integrate (function f(x:real): real;
                    a,b: real;
                    n: integer;
                    var area: real);

var w, sum: real;
    i: integer;

begin
  w := (b-a) / n;
  sum := 0.0;
  for i := 1 to n do
    sum := sum + w * (f(a+(i-1)*w) + 4.0*f(a-w/2+i*w) + f(a+i*w))/6.0;
  area := sum;
end;

function unitStep (x:real):real;
begin
  unitStep := 1.0 + trunc(x);
end;

function cosine (t:real):real;
begin
  cosine := cos(t);
end;

```

```

var
  result : real;
begin
  integrate(unitStep, 0.0, 1.0, 100, result);
  writeln('Area under step function from 0 to 2 is ', result);
  integrate(cosine, 0.0, 3.1415927/2, 100, result);
  writeln('Area under cosine curve from 0 to pi/2 is ', result);
end.

```

**EXAMPLE 2 (ex005.p)**

The following program illustrates some of the intricacies in parameter transmission, and is from the book *Programming Languages: Design and Implementation*, by Terrance W. Pratt, Prentice-Hall, Inc., second edition, 1984.

In this example, the global variable *x* is passed as a VAR parameter to procedure *q*, which has a local variable with the same name. This results in a hole in scope for the global *x*. In addition, the function *f* is passed as a parameter, and has the side effect of changing the global variable *x*. To understand this program, the reader should be familiar with the lexical scope of identifiers as described in Section 2.2, and how environments are passed with formal parameters, described in Section 8.2.3. That is, the value of *x* that will be used within the function *f* will be the value that *x* had when *f* was used as an actual parameter.

```

program forall1;
var x: integer;

procedure q( var i: integer; function r(j:integer):integer);
var x: integer;
begin
  x := 4;
  i := r(i);
end;

procedure p;
var i: integer;
function f(k: integer): integer;
begin
  x := x + k;
  f := i + k;
end;

begin
  i := 7;
  q(x, f);
end;

begin
  x := 7;
  p;
  writeln('x= ', x:3);
end.

```

The program produces the following output:

```
x = 9
```

Line numbers are printed with the program so that the program statements can be referred to in the following discussion.

**Example (ex006.p)**

An execution trace of this program is presented by way of explanation.

Line #	Trace	Remarks
22	x = 7	
23	p	
18	i = 2	
19	q (x, f)	x=7 and i=2 in f's environment
3	q (i=7, f with x=7 and i=2)	
6	x = 4	this x is local to q because of the hole in scope
7	i = f(7)	
12	first formal is k=7	
14	x = 7 + 7	x is from line 19, k is from line 12
	x = 14	
15	f = 2 + 7	i is from line 19, k is from line 12
	f = 9	
8	i = 9	i is a VAR parameter so value 9 passed to actual in calling routine on line 19 setting x = 9
16	x = 9	x is global to procedure p
24	print x = 9	

### EXAMPLE 3 (ex007.p)

The following example illustrates mutual recursion and the FORWARD directive. This program checks the syntax of a simple expression against the following simplified grammar for expressions:

```

<expression> := <term> | <term> + <expression>
<term>       := <factor> | <factor> <term>
<factor>    := <expression> | letter | <factor>

```

An expression consists of a term, or a term followed by a plus symbol, followed by an expression. A term consists of a factor, or a factor followed by a term. A factor consists of an expression, a letter, or a factor. Variables are restricted to a single letter in this example.

Since the grammar is recursive, that is, an expression is defined in terms of an expression, etc., then the expression can be recognized, or parsed, by using recursive procedures. Further, since the components of grammar are defined in terms of one another, then the procedures implementing this grammar will be mutually recursive. The program presented below implements this grammar. The expression has been hard coded into the example to simplify input. The following output is produced by the program:

```

((a+b)*(c+d))+f <<is a valid expression>>
program parse;
type
  c20 = array[1..20] of char;
procedure expression(s:c20; var t:integer; var error:boolean); forward;
procedure term(s:c20; var t:integer; var error:boolean); forward;
procedure getInput(var s:c20; var t:integer; var error:boolean);
var
  i: integer;

```

```

begin
  s[1] := '(';
  s[2] := '(';
  s[3] := 'a';
  s[4] := '+';
  s[5] := 'b';
  s[6] := ')';
  s[7] := 'w';
  s[8] := '(';
  s[9] := 'c';
  s[10] := '+';
  s[11] := 'd';
  s[12] := ')';
  s[13] := ')';
  s[14] := '+';
  s[15] := 'f';
  t := 1;
  error := false;
  for i := 1 to 15 do
    write (s[i]);
  end;

procedure factor(s:c20; var t:integer; var error:boolean);
begin
  if s[t] = '(' then begin
    t := t + 1;
    expression(s, t, error);
    if s[t] = ')' then
      t := t + 1
    else
      error := true
    end
  end
  else
    if s[t] in ['a'..'z'] then
      t := t + 1
    else
      error := true;
    if s[t] = '*' then
      t := t + 1;
    end;
  end;

procedure expression;
begin
  term(s, t, error);
  if s[t] = '+' then begin
    t := t + 1;
    expression(s, t, error);
  end;
end;

procedure term;
begin
  factor(s,t,error);
  if (s[t] = '(') or (s[t] in ['a'..'z']) then
    term(s,t,error);
  end;
end;

var
  s: c20;

```

```
t: integer;
error: boolean;

begin
  getInput(s,t,error);
  expression(s, t, error);
  if error then
    writeln(' <<is an illegal expression>>')
  else
    writeln(' <<is a valid expression>>');
end.
```

# 9

# Input and Output

## 9.1 Overview

A file is a sequence of identical objects, each object consisting of any simple or structured type, except the file type. The goal of the Pascal file system is to provide an abstraction of a peripheral input/output device which embodies the idea of a sequence of arbitrary length and upon which operations on this sequence are natural and familiar. Of the many devices commonly attached to computer systems, the magnetic tape unit is generally chosen to act as a model since it is simple yet general enough to describe the operations commonly performed on sequential files. This model of the Pascal file system is briefly mentioned to provide some intuition behind the characteristics and limitations of Pascal's files. The Pascal file system is an implementation of sequential files on disk. As such, it inherits the advantages and disadvantages of magnetic tape files.

The only operations allowed on a file are sequential reading and sequential writing. Random access is not possible. Since all operations are sequential, the word sequential is usually omitted when referring to Pascal files.

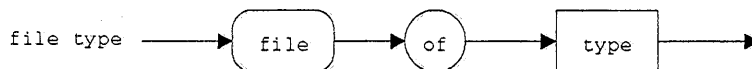
Assignment and comparison of files are not possible using the assignment or comparison operators. Both of these operations must be done component by component. The only way to extend a file is to copy it to another file, and then add the additional records to the new file. No operator is provided to concatenate two files or selectively to update a single component in a file. These operations, like assignment and comparison, require that a program be written to do the required task.

A file is created by writing records to it in the sequence in which it is desired to save the records. A file is accessed by reading the records in the sequence in which they were written from the beginning until the desired record is found, or until the end of file is reached. It is not possible to mix the reading and writing operations on a file without first issuing a command to close and open the file (RESET or REWRITE), and then starting from the beginning of the file.

The Pascal file system maintains a pointer to the current position in the file. This pointer is automatically advanced by reading or writing to the file. The component of the file to which the file pointer points is called the buffer variable. That is, the buffer variable corresponds to the current record in the file. The buffer variable is the name of the current component in the file and is referenced as an ordinary variable.

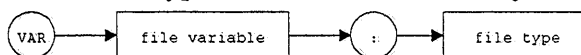
## 9.2 File Declaration and Initialization

A file type has the following syntax:



where *type* may not directly or indirectly refer to a file.

A file variable is a variable of the file type and is declared like any other variable:



The file variable is a pointer that keeps track of two pieces of information: the file name and the current location in the file. The file variable is sometimes called the file pointer for this reason, although it is declared as an ordinary variable. A physical file on disk is associated with the file variable by using the RESET or REWRITE procedures.

The predefined procedures `RESET` and `REWRITE` initialize a file variable and optionally assign it to a physical disk file. `RESET` initializes the file variable for reading an existing file, and `REWRITE` initializes a file variable for writing to an existing (or nonexisting) file. In both cases, the file pointer is positioned to the beginning of the file. Note that the contents of an existing file are lost if the file is opened using `REWRITE`. This is true even if the file is not written to and the program successfully terminates.

The second argument to `RESET` and `REWRITE`, the file name, is generally omitted on subsequent calls unless the file variable is being assigned to a different disk file. Additional calls are made to `RESET` and `REWRITE` for two reasons: in order to change the direction of file access, that is, input to output or output to input, or simply to reposition the file pointer to the beginning. For example, consider the code fragment:

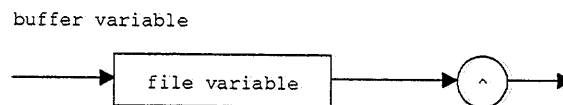
```
rewrite (temp, 'scratch.dat');
...
reset (temp);
```

The first statement assigns the file variable `temp` to the physical disk file `scratch.dat`. If `scratch.dat` exists, then it is opened, otherwise it is created. In either case, `scratch.dat` is opened for writing to its first component. The second statement closes and then opens `scratch.dat` for reading, beginning with its first component. The file, `scratch.dat`, now exists, even though it may be empty.

The declaration of a file variable causes the implicit declaration of a buffer for use when accessing the file. This buffer is called the file buffer, or buffer variable. The file buffer holds one item of the file's base type and is the only portion of the file that is directly accessible. The file variable is a pointer to the current component in the file, and the file buffer contains the value of this component.

The file variable is sometimes compared to a sliding window through which the file can be seen. The file is scanned by moving the window across the file. The position of the window corresponds to the file pointer, and by looking into the window the contents of the file can be seen.

The file buffer, which is implicitly declared in a file declaration, is treated like any other variable. It is referenced by appending a pointer (^) to the file variable. The general syntax is:



For example, consider the following code fragment:

```
type
  dailyRec = record
    day : integer;
    temp: real;
  end;

var
  weather: file of dailyRec;
  ...
  rewrite (weather, 'climate.dat');
```

The call to `REWRITE` associates the physical file named `climate.dat` with the file variable, `weather`. The assignment statements,

```
weather^.day := 21;
weather^.temp:= 34.5;
```

set the value of the buffer variable to a record whose `day` field is 21, and whose temperature field is 34.5.

### 9.3 Input and Output Processing using GET and PUT

The fundamental I/O routines in Pascal are GET and PUT. GET increments the current file pointer to the next component and copies this component to the buffer variable. If the file pointer was already positioned at the last component in the file, then the value of the function EOF becomes TRUE and the value of the buffer variable is undefined. PUT appends the value of the buffer variable to the end of the file.

Reading is accomplished by accessing the buffer variable. This is usually followed by a call to GET to advance the file pointer in preparation for the next read. The following statements illustrate this:

```
component := f^;
get (f);
```

The call to GET follows the assignment statement because of the way buffer variables are initialized by the RESET procedure. RESET sets the file pointer to the beginning of the file and copies the first component into the buffer variable. Thus the buffer variable already contains the contents of the first component of the file when it is used for the first time by the program.

The above sequence of Pascal statements also works when the file being read is connected to an interactive device, such as the user's terminal. Notice that this requires a slight modification to the file initialization process since the first component of the file is not available. If RESET were to demand input when called, then programs would have to distinguish between interactive and non-interactive files, otherwise the user would be required to reply before the program had a chance to print any prompts. To avoid this problem, the Pascal standard allows RESET to delay reading the first component of the file until it is actually used by the program. This allows the treatment of interactive and non-interactive files to be uniform and allows the I/O commands to operate as usual. The technique of delaying a request for data until it is needed is called lazy evaluation and is transparent to the programmer.

Writing is done by assigning a value to the buffer variable and using PUT to add the component to the end of the file. The following statements illustrate this:

```
f^ := component;
put (f);
```

PUT appends the contents of the file buffer to the file and increments the file pointer so that the added item becomes the current component. Technically, the contents of the buffer variable are no longer defined after being moved to the file. Since the file variable now points to the last component in the file, the function EOF is TRUE.

#### EXAMPLE 9.1 (ex008.p)

This example illustrates the use of the GET and PUT procedures to make an identical copy of a file. Buffer variables are used to access the data in the source and destination files. This permits the components of the file to be moved from the source to the destination very efficiently and without the use of a temporary variable to hold the current component. The program can be made to copy any file type by changing the value of *fileType* in the type declaration.

```
program copy1(output);
type
  fileType = file of char;
procedure copy(var src, des: fileType);
begin
  while not (eof(src)) do begin
    des^ := src^;
    put(des);
    get(src);
```



```

        end;
    end;

var
    inp, out: fileType;

begin
    writeln('copy1 started');
    reset(inp, 'ex008.inp');
    rewrite(out, 'ex008.out');
    copy(inp, out);
    writeln('copy1 finished');
end.

```

## 9.4 Buffer Variable Restrictions

The Pascal standard does not allow the buffer variable to be accessed while the file variable is in a position to be altered. This rule is designed to prevent data from being unintentionally modified and is consistent with Pascal's concern for data security. The three situations in which it is possible to violate this rule are given below and they are rather obscure. This rule is not enforced by NDP Pascal. The following describes the three situations in which aliasing is possible with the buffer and file variables:

1. The buffer variable is used as an actual variable parameter to a routine that modifies the corresponding buffer variable;
2. The buffer variable is used in the left-hand side of an assignment statement whose expression on the right-hand side contains a function that modifies the position of the file;
3. The buffer variable is used in a WITH clause, where the component statement modifies the position of the file.

### EXAMPLE 1

This example illustrates the behavior of a program that alters the value of a file variable while a reference exists to the corresponding buffer variable (as in case 1, above). This is not an example to emulate and is only presented to hint at the difficulties in debugging a program when this is done.

Two versions of the same program are given to accomplish the same task: one using PUT and the other using WRITE. The first program uses PUT and is easy to understand. The second uses WRITE and is more difficult. Both programs generate the same output as shown below:

```

    The initial buffer variable is a
    The final buffer variable is 3

```

#### Program 1 (ex009.p)

```

program bv1a(output);

var
    out: file of char;

procedure write3(var c: char);
begin
    c := 'z';
    out^ := '1'; put(out);
    out^ := '2'; put(out);
    out^ := '3'; put(out);
end;

begin
    rewrite(out, 'ex009.out');

```

```

out^ := 'a';
writeln('The initial buffer variable is ', out^);
write3(out^);
writeln('The final buffer variable is ', out^);
end.

```

**Program 2 (ex010.p)**

```

program bv1b(output);
var
  out: file of char;

procedure write3(var c:char);
begin
  c := 'z';
  write(out, '1');
  write(out, '2');
  write(out, '3');
end;

begin
  rewrite(out, 'ex010.out');
  out^ := 'a';
  writeln('The initial buffer variable is ', out^);
  write3(out^);
  writeln('The final buffer variable is ', out^);
end.

```

**EXAMPLE 2 (ex011.p)**

This example illustrates the behavior of a program that alters the value of a file variable while a reference exists to the corresponding buffer variable (as in case 3, above). Again, this is not an example to emulate. This program produces the following output:

```

The initial buffer variable is  1  2  3
The final buffer variable is   101 102 103

```

```

program bv2(output);
type
  rec = record
    a,b,c : integer;
  end;

var
  buf: file of rec;
  data: rec;

begin
  rewrite(buf, 'ex011.out');
  buf^.a := 1;  data.a := 101;
  buf^.b := 2;  data.b := 102;
  buf^.c := 3;  data.c := 103;
  writeln('The initial buffer variable is ', buf^.a:4, buf^.b:4, buf^.c:4);
  with buf^ do begin
    a := 10; b := 11; c := 12;
    write(buf, data);
  end;
  writeln('The final buffer variable is ', buf^.a:4, buf^.b:4, buf^.c:4);
end.

```

## 9.5 Input and Output Processing with READ and WRITE

The procedures READ and WRITE are simply abbreviations for the sequence of commands needed to do input and output with GET and PUT. Both READ and WRITE take an argument that is identical in type to the file's type. Since GET and PUT work with any file type, so do READ and WRITE.

The predefined procedure READ(*f*, *component*) is equivalent to the compound statement:

```
begin
  component := f^;
  get (f)
end;
```

The statement READ(*f*, *v1*, . . . , *vn*) is equivalent to the following sequence of individual reads:

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

Similarly, the predefined procedure WRITE(*f*, *component*) is equivalent to the compound statement:

```
begin
  f^ := component;
  put (f)
end;
```

The statement WRITE (*f*, *v1*, . . . , *vn*) is equivalent to the following sequence of individual writes:

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

### EXAMPLE 9.4 (ex012.p)

This example illustrates the use of the READ and WRITE procedures in making an identical copy of a file, and is very similar to *Example 9.1*. The program here is less efficient than example 9.1 because of the need to store the source file buffer variable into the variable *ch*, and then to move *ch* into the destination file buffer variable. The penalty for these extra copies increase with a more complicated type.

```
program copy2(output);
type
  fileType = file of char;
procedure copy(var src, des: fileType);
var
  ch: char;
begin
  while not (eof(src)) do begin
    read(src, ch);
    write(des, ch);
  end;
end;
var
  inp, out: fileType;
begin
  writeln('ex012 started');
  reset(inp, 'ex012.inp');
  rewrite(out, 'ex012.out');
  copy (inp, out);
```

```
writeln('ex012 finished');
end.
```

**EXAMPLE 9.5 (ex013.p)**

The program in this example merges two files of integers into a third. The integers in the two input files are assumed to be in ascending order. This example illustrates several different ways in which buffer variables may be used. Buffer variables are used to avoid declaring temporary variables when accessing the input files and provide lookahead when comparing the current file components. Notice that GET is used to access the two input files, while WRITE is used to transfer data to the output file.

```
program merge1(output);

type
  integerFile = file of integer;

procedure merge(var input1, input2, result: integerFile);
begin
  reset(input1); reset(input2); rewrite(result);
  while not (eof(input1) or eof(input2)) do
    if input1^ < input2^ then begin
      write(result, input1^);
      get(input1)
    end
    else begin
      write(result, input2^);
      get(input2)
    end;
  while not eof(input1) do begin
    write(result, input1^);
    get(input1);
  end;

  while not eof(input2) do begin
    write(result, input2^);
    get(input2);
  end;
end;

var
  file1, file2, file3: integerFile;
  i: integer;

begin
  writeln('begin ex013.p');
  reset(file1, 'ex013a.inp');
  reset(file2, 'ex013b.inp');
  rewrite(file3, 'ex013.out');
  merge(file1, file2, file3);

  reset(file3);
  while not eof(file3) do begin
    read(file3, i);
    writeln(i);
  end;
  writeln('ex013 finished');
end.
```



# 10

## Predefined Functions and Procedures

The following is an alphabetic list of the predefined functions and procedures in NDP Pascal.

### ABS (x)

returns the absolute value of x

#### Definition:

```
function abs ( i : integer ) : integer;  
function abs ( r : real ) : real;  
function abs ( d : double ) : double;  
function abs ( f : float ) : float;
```

where

*i* is an expression of type integer,  
*r* is an expression of type real,  
*d* is an expression of type double,  
*f* is an expression of type float.

The ABS function returns an integer, real, double or float value depending upon the type of its parameter. The result is the absolute value of the input parameter.

### ARCTAN (x)

returns the arctangent of x

#### Definition

```
function arctan ( i : integer ) : double;  
function arctan ( r : real ) : double;  
function arctan ( d : double ) : double;  
function arctan ( f : float ) : double;
```

where

*i* is an expression of type integer,  
*r* is an expression of type real,  
*d* is an expression of type double,  
*f* is an expression of type float.

The ARCTAN function converts the input parameter to a temporary 64-bit floating point number and returns the arctangent of this value. The input parameter is assumed to be expressed in radians.

### ARGC

returns the number of command line arguments

#### Definition

```
function argc : integer;
```

The ARGC function has no arguments and returns an integer equal to the number of command line arguments specified when the program was run. The number of command line arguments

includes the command name, so ARGV is at least one. An example is included after the ARGV function.

## ARGV(*i*, *s*)

copies the *i*th command line argument into the variable *s*

### Definition

```
CONST n=12 {for example}
TYPE str=packed array [1..n] of char
procedure argv (i:integer; VAR s:str);
```

where

*i* is an integer in the range 1 to the number of command line arguments (which corresponds to the value of the ARGV).

*s* is a character array that will receive the *i*th command line argument.

The ARGV(*i*, *s*) function copies the *i*th command line argument into the variable *s*. The *i*th command line argument is truncated if the receiving variable is not large enough to store all its characters.

### EXAMPLE (ex020.p)

```
program arg1(output);
var i: integer;
    arglist: packed array [1..8] of char;
begin
  writeln('argc = ',argc:2);
  for i := 0 to argc - 1 do begin
    argv(i,arglist);
    writeln('arg ',i:2,' = ', arglist);
  end;
end.
```

The above program illustrates the use of the ARGV and ARGV functions. In this program, the loop index is zero originated so that the first command line argument will be accessed when the loop index is 1, and so on. When the example is executed without any arguments on the command line, the following output is generated. That is, the command:

```
386 Loader:      ndprun arg1
860 Loader:      run860 arg1
```

produces the following output:

```
argc = 1
arg 0 = arg1
```

When the example is executed with several command line arguments, the following output is generated. That is, the command:

```
386 Loader:      ndprun arg1 this is a test abcdefghijkl 11 22
860 Loader:      run860 arg1 this is a test abcdefghijkl 11 22
```

produces the following output (notice that the fifth argument is truncated):

```
argc = 8
arg 0 = arg1
arg 1 = this
arg 2 = is
arg 3 = a
```

```

arg 4 = test
arg 5 = abcdefgh
arg 6 = 11
arg 7 = 22

```

## CHR (n)

returns the ASCII character whose ordinal value is n

### Definition

```
function chr ( i : integer ) : char;
```

where *i* is an integer expression.

The CHR function returns the ASCII character whose ordinal value is equal to the result of the integer expression *i*.

This function is the inverse of the ORD function. That is, ORD(CHR(*i*)) = *i*, if *i* = 0 to 127.

## COS (x)

returns the cosine of x

### Definition

```

function cos ( i : integer ) : double;
function cos ( r : real ) : double;
function cos ( d : double ) : double;
function cos ( f : float ) : double;

```

where

*i* is an expression of type integer,  
*r* is an expression of type real,  
*d* is an expression of type double,  
*f* is an expression of type float.

The COS function converts the input parameter to a temporary 64-bit floating point number and returns the cosine of this value. The input parameter is assumed to be expressed in radians.

### EXAMPLE (ex036.p)

```

program cos1(output);

function cos(d: double): double; external;

const
  pi = 3.14159265358979323846;

var
  i: integer;
  x: double;

begin
  x := 0.0;
  for i := 1 to 5 do begin
    writeln('x = ', x, ' cos(x) = ', cos(x));
    x := x + pi/4.0;
  end;
end.

```



This program generates the following output:

```
x = 0.0000000000000000e+00   cos(x) = 1.0000000000000000e+00
x = 7.85398163397448286e-01   cos(x) = 7.07106781186547462e-01
x = 1.57079632679489657e-01   cos(x) = 0.0000000000000000e+00
x = 2.35619449019234486e+00   cos(x) = -7.07106781186547550e-01
x = 3.14159265358979311e+00   cos(x) = -1.0000000000000000e+00
```

## DISPOSE (p)

deallocates a dynamic variable

### Definition

```
procedure dispose (var p : pointer);
procedure dispose (var p, t1, t2, . . . : scalar);
```

where

*p* is a pointer variable with base type *T*,

*t1, t2, . . .* are scalar constants representing the tag fields if the base type *T* is a variant record.

DISPOSE releases storage assigned to a dynamic variable and sets the pointer to NIL.

## EOF (f)

returns TRUE if file *f* is at end of file

### Definition

```
function eof ( f : filetype ) : boolean;
function eof : boolean;
```

where *f* is a variable of a file type.

EOF is a boolean function that returns TRUE if the file is positioned at the end of file. On a file opened for input, this occurs when an attempt is made to read past the last record in the file. On a file opened for output, this function always returns TRUE.

## EOLN (f)

returns TRUE if file *f* is at end of line

### Definition

```
function ( f : TEXT ) : BOOLEAN;
function eoln : BOOLEAN;
```

where *f* is a TEXT file opened to input. If *f* is omitted then the file INPUT is used.

The EOLN function returns TRUE if file *f* is positioned at an end of line character, and FALSE otherwise. Notice that EOLN is applicable only to TEXT files.

If EOLN (*f*) is true, then the file variable *f*<sup>^</sup> has the value of a blank, i.e., *f*<sup>^</sup> does not return the end of line character. The blank is not in the file but will appear as if it were. This generally does not matter to most applications. If the physical layout of the data in a file is significant, then the programmer must be sensitive to the EOLN condition.

## EXP (x)

returns the base of the natural log (e) raised to the power x

### Definition

```
function exp ( i : integer ) : double;
function exp ( r : real ) : double;
function exp ( d : double ) : double;
function exp ( f : float ) : double;
```

where

*i* is an expression of type INTEGER,  
*r* is an expression of type REAL,  
*d* is an expression of type DOUBLE,  
*f* is an expression of type FLOAT.

The EXP function converts the input parameter to a temporary 64-bit floating point number, and returns the base of the natural logarithms, e, raised to this power. If the return value is too large to be represented, the global variable `errno` will be set to ERANGE.

## GET (f)

advances file pointer to the next component of file *f*

### Definition

```
procedure get ( f : filetype );
```

where *f* is a file variable.

GET positions the file pointer of a file to the next component in the file, then assigns the value of this component to the associated buffer variable.

If the predicate `EOF(f)` is FALSE before the execution of `GET(f)`, then GET advances the current file position to the next component and assigns the value of this component to the buffer variable *f*<sup>^</sup>. If no next component exists, then `EOF(f)` is set to TRUE, and the value of *f*<sup>^</sup> is not defined. An error occurs if `EOF(f)` is TRUE before execution of `GET(f)`.

The file *f* must be opened for input.

## LN (x)

returns the natural logarithm of x

### Definition

```
function ln ( i : integer ) : double;
function ln ( r : real ) : double;
function ln ( d : double ) : double;
function ln ( f : float ) : double;
```

where

*i* is an expression of type INTEGER,  
*r* is an expression of type REAL,  
*d* is an expression of type DOUBLE,  
*f* is an expression of type FLOAT.

The LN function converts the input parameter to a temporary 64-bit floating point number and returns the natural logarithm of this value.

## NEW

allocates a dynamic variable

### Definition

```
procedure NEW (var p : pointer );
procedure NEW (var p : pointer; t1, t2, . . . : scalar);
```

where

*p* is a pointer variable with base type *T*,

*t1, t2, . . .* are scalar constants representing the tag fields if the base type *T* is a variant record.

The NEW procedure allocates a dynamic variable and initializes the pointer to point to the variable. The first form of the NEW procedure allocates an amount of storage that is necessary to represent a value of the type to which the pointer refers. If the type contains a variant record, then the amount of space allocated corresponds to what will be needed by the largest variant of the record.

The second form of the NEW procedure allocates space for a particular instance of a variant record. This requires specifying the tag field for each sub-variant in the record in the tag field list in the procedure call. With this information, the NEW procedure will allocate the exact amount of space needed for this record variant.

## ODD (n)

returns TRUE if the integer *n* is odd

### Definition

```
function odd (i : integer) : boolean;
```

where *i* is an expression of type INTEGER.

The ODD function returns TRUE if the input parameter *i* is odd, and FALSE if it is even.

## ORD (x)

converts a scalar value *x* to an integer

### Definition

```
function ord ( x : scalar_type ) : integer;
```

where *x* is a character, boolean, enumerated or SUBRANGE type.

The ORD function returns an integer value that corresponds to the scalar *x*. If *x* is of type CHAR, then ORD returns the corresponding value of *x* in the ASCII character set. If *x* is an enumerated type, then ORD returns the position in the enumeration, beginning at zero. Since type BOOLEAN is defined as BOOLEAN = (FALSE, TRUE), we have ORD(FALSE) = 0 and ORD(TRUE) = 1. If *x* is a subrange type, then ORD returns the value of the integer despite the relative location of the integer within the subrange.

## PACK (a, i, z)

packs array *a*, beginning at index *i*, into array *z*

### Definition

```
procedure pack ( a : packed_array_type ;
                i : integer ;
                var z : unpacked_array_type );
```

where

*a* is the source unpacked array,  
*i* is an expression that is compatible with the index type of *a*,  
*z* is the destination packed array.

PACK copies elements from the unpacked array *a* to the packed array *z*, beginning with the *i*-th element of *a*. The element types of the two arrays must be identical. There must be enough elements in the *z* array to receive the elements copied from *a*.

Note that PACK is defined for one-dimensional arrays only.

PACK is equivalent to the following definition:

```

type
  t1 = array [m..n] of T;
  t2 = packed array [u..v] of T;
procedure pack (      a : t1;
                    i : integer;
                    var z : t2      );
var j, k:integer;
begin
  k := i;
  for j := u to v do begin
    z [j] := a [k];
    k := k + 1;
  end;
end;
...

```

## PAGE (f)

writes an ASCII form feed to file *f*

### Definition

```
procedure page (var f : TEXT);
```

where *f* is a TEXT file open for output. If *f* is omitted, then file OUTPUT is used.

This procedure writes a form feed character to file *f*. This is control-L at present, or ord(12). When the file is printed, this causes a page eject.

## PRED (x)

returns the predecessor value of the scalar *x*

### Definition

```
function pred ( x : scalar_type ) : scalar_type;
```

where *x* is an expression formed from one of the following types: BOOLEAN, CHARACTER, INTEGER, ENUMERATED or SUBRANGE.

The PRED function returns the predecessor value of the parameter expression. The first element in an enumeration list does not have a predecessor. The PRED of an integer is the integer minus one. The PRED of a REAL argument is not allowed and results in an error.

## PUT (f)

advances the file pointer to the next component of the file *f*

### Definition

```
procedure put ( f : filetype);
```

where *f* is a file variable.

PUT copies the value of the buffer variable to the end of the specified file.

If the predicate EOF(*f*) is TRUE before the execution of PUT(*f*), then PUT appends the buffer variable *f*<sup>^</sup> to the file *f*, EOF(*f*) remains TRUE, and the value of *f*<sup>^</sup> becomes undefined. An error occurs if EOF(*f*) is FALSE before execution of PUT(*f*).

The file *f* must be opened for output.

## READ and READLN (for TEXT files only)

### Definition

```
procedure read (f : text; vList : see_below);
procedure readln (f : text; vList : see_below);
procedure readln (f : text);
```

where

*f* is an optional text file to be used for input, file INPUT is assumed if this parameter is omitted.

*vList* is a list of variables, separated by commas, of any combination of the following types: INTEGER, CHAR, DOUBLE, REAL, FLOAT.

The READ procedure reads character data from the text file *f* and converts it to match the data type of each parameter.

The READLN procedure reads and converts data in the same manner as READ if any variables are present. READLN then positions the file pointer to the beginning of the next line.

CHAR data is read by reading the next character in the file. The READ procedure will return chr(10) for the end of line character.

INTEGER data is read by skipping leading blanks, processing the optional sign and converting all digits up to the first non-numeric character. An end of line will terminate an INTEGER.

DOUBLE, FLOAT and REAL data is read by skipping leading blanks, processing the optional sign, and converting all characters (digits, decimal point, sign, e or E) up to the first non-numeric character that falls outside the syntax of a DOUBLE, FLOAT or REAL number. An end of line will terminate a DOUBLE, FLOAT or REAL number.

READ and READLN will accept numbers up to a line boundary, i.e., a number cannot be placed on two separate lines.

READ and READLN will not convert BOOLEAN or hexadecimal formatted integers.

Errors will cause the global variable *errno* to be set appropriately.

## READ for non-TEXT files

### Definition

```
type ft : file of t
procedure read (f : ft; var v : t);
```

where

$f$  is a file variable,  
 $v$  is a variable whose type is the base type of the file  $f$ .

The READ procedure reads one file component from file  $f$  and assigns this element to the variable  $v$ .

Errors will cause the global variable `errno` to be set appropriately.

READ ( $f$ ,  $v$ ) is equivalent to the following:

```
begin v:= f^; get(f) end;
```

## RESET ( $f$ , $s$ )

opens a file for input

**Definition**

```
procedure reset ( f : filetype; s : string );
```

where

$f$  is a variable of a file type,  
 $s$  is a variable or quoted string. This parameter is optional.

RESET initializes the file pointer to the first component of the file and prepares the file for input. This procedure is equivalent to the following:

1. closing the file if it is open,
2. rewinding the file,
3. opening the file for input,
4. getting the first component of the file.

The second argument to the RESET procedure is the name of the file to be opened. This may be specified as a string constant, that is, a file name embedded in single quotes, or as a variable. If the second argument is a variable, then the file name must be terminated by the value `chr(0)`.

RESET positions the file to the beginning and so is equivalent to rewinding the file. If the specified file  $f$  is not empty, then RESET assigns the buffer variable  $f^$  to the value of the first component of the file, and sets `EOF(f)` to FALSE. If the file  $f$  is empty or does not exist, then  $f^$  is undefined, and `EOF(f)` is set to TRUE.

Except for the predefined file `INPUT`, RESET must be used on every file before using `GET`, `READ`, or `READLN` to obtain data from the file.

For interactive files, advancing the file pointer and assigning the buffer variable is deferred by a technique called lazy evaluation. Without lazy evaluation, the execution of the RESET procedure will cause the program to wait for input to become available, as step 4 above shows. This makes it difficult for a program to display messages or prompts before requesting input. Lazy evaluation allows physical input to be deferred until the input is actually needed by access to a buffer variable. This permits a program to handle interactive terminal input in a natural way.

In Standard Pascal and NDP Pascal, there is no procedure for closing a file. Files are closed automatically when program execution terminates.

## REWRITE ( $f$ , $s$ )

opens a file for output

**Definition**

```
procedure rewrite ( f : filetype; s : string );
```

where

*f* is a variable of the file type,  
*s* is a variable or quoted string. This parameter is optional.

REWRITE positions the file pointer to the beginning of the file and prepares the file for output. This procedure is equivalent to the following:

1. closing the file if it is open,
2. rewinding the file,
3. opening the file for output.

The second argument to the REWRITE procedure is the name of the file to be opened. This may be specified as a string constant, that is, a file name embedded in single quotes or as a variable. If the second argument is a variable, then the file name must end with the value `chr(0)`. If this parameter is omitted, a file name with the prefix "PASRT" is created, and associated with the file variable *f*.

REWRITE positions a file to the beginning in preparation for writing to it, and so any existing data in the file is lost. `EOF(f)` is set to TRUE, and the buffer variable *f*<sup>^</sup> is undefined.

Except for the predefined file OUTPUT, REWRITE must be used on every file before using PUT, WRITE, or WRITELN to transfer data to the file.

In Standard Pascal and NDP Pascal, there is no procedure for closing a file. Files are closed automatically when program execution terminates.

## ROUND (x)

converts a floating point *x* to an integer by rounding

### Definition

```
function round ( d : double ) : integer;
function round ( f : float ) : integer;
function round ( r : real ) : integer;
```

where

*d* is an expression of type DOUBLE,  
*f* is an expression of type FLOAT,  
*r* is an expression of type REAL.

The ROUND function converts a DOUBLE, FLOAT or REAL expression to an INTEGER by rounding. Positive values less than 0.5 are rounded down to the next integer. The following table illustrates the ROUND function:

```
round (1.0) = 1   round (-1.0) = -1.0
round (1.1) = 1   round (-1.1) = -1.0
round (1.2) = 1   round (-1.2) = -1.0
round (1.3) = 1   round (-1.3) = -1.0
round (1.4) = 1   round (-1.4) = -1.0
round (1.5) = 2   round (-1.5) = -2.0
round (1.6) = 2   round (-1.6) = -2.0
round (1.7) = 2   round (-1.7) = -2.0
round (1.8) = 2   round (-1.8) = -2.0
round (1.9) = 2   round (-1.9) = -2.0
round (2.0) = 2   round (-2.0) = -2.0
```

The ROUND function is equivalent to the following:

```
if d > 0.0 then round := TRUNC (d+0.5)
else round := TRUNC (d-0.5);
```

## SIN (x)

returns the sine of x

### Definition

```
function sin ( i : integer ) : double;
function sin ( r : real ) : double;
function sin ( d : double ) : double;
function sin ( f : float ) : double;
```

where

*i* is an expression of type INTEGER,  
*r* is an expression of type REAL,  
*d* is an expression of type DOUBLE,  
*f* is an expression of type FLOAT.

The SIN function converts the input parameter to a temporary 64-bit floating point number, and returns the sine of this value. The input parameter is assumed to be expressed in radians. If the argument is large, some loss of significance in the result may occur, and the global variable `errno` is set to `ERANGE`.

### EXAMPLE (ex074.p)

```
program sin1(output);
function sin(f: double): double; external;
const
  pi = 3.14159265358979323846;
var
  i: integer;
  x: double;
begin
  x := 0.0;
  for i:= 1 to 5 do begin
    writeln(' x = ', x, ' sin(x) = ', sin(x));
    x := x + pi/4.0;
  end;
end.
```

This program generates the following output:

```
x = 0.000000000000000000e+00 sin(x) = 0.000000000000000000e+00
x = 7.85398163397448286e-01 sin(x) = 7.07106781186547550e-01
x = 1.57079632679489657e+00 sin(x) = 1.000000000000000000e+00
x = 2.35619449019234486e+00 sin(x) = 7.07106781186547462e-01
x = 3.14159265358979311e+00 sin(x) = 0.000000000000000000e+00
```

## SQRT (x)

returns the square root of x

### Definition

```
function sqrt ( i : integer ) : double;
function sqrt ( r : real ) : double;
function sqrt ( d : double ) : double;
function sqrt ( f : float ) : double;
```



where

*i* is an expression of type INTEGER,  
*r* is an expression of type REAL,  
*d* is an expression of type DOUBLE,  
*f* is an expression of type FLOAT.

The `SQRT` function converts the input parameter to a temporary 64-bit floating point number, and returns the square root of this value. If the argument is negative, the global variable `errno` is set to `EDOM`, and the function returns 0.

## SQR (x)

returns the square of x

**Definition**

```
function sqr ( i : integer ) : integer;
function sqr ( r : real   ) : real;
function sqr ( d : double ) : double;
function sqr ( f : float  ) : float;
```

where

*i* is an expression of type integer,  
*r* is an expression of type real,  
*d* is an expression of type double,  
*f* is an expression of type float.

The `SQR` function returns either an INTEGER, REAL, DOUBLE or FLOAT value depending upon the type of its parameter. The result is the value of the square of the input parameter.

## SUCC (x)

returns the successor of the scalar x

**Definition**

```
function succ ( x : scalar_type ) : scalar_type
```

where *x* is an expression formed from one of the following types: char, boolean, integer, enumerated or subrange type.

The `SUCC` function returns the successor value of the parameter expression. The last item in an enumerated list has no successor. The `SUCC` of an INTEGER is equivalent to adding one. `SUCC` of a REAL argument is not allowed and results in an error.

## TRUNC (x)

converts a floating point x to an integer by truncating

**Definition**

```
function trunc ( d : double ) : integer;
function trunc ( f : float  ) : integer;
function trunc ( r : real   ) : integer;
```

where

*d* is an expression of type DOUBLE,  
*f* is an expression of type FLOAT,  
*r* is an expression of type REAL.

The TRUNC function converts a DOUBLE, FLOAT or REAL expression to an INTEGER by truncating. The following table gives some values of this function.

```
trunc (1.0) = 1   trunc (-1.0) = -1.0
trunc (1.1) = 1   trunc (-1.1) = -1.0
trunc (1.2) = 1   trunc (-1.2) = -1.0
trunc (1.3) = 1   trunc (-1.3) = -1.0
trunc (1.4) = 1   trunc (-1.4) = -1.0
trunc (1.5) = 1   trunc (-1.5) = -1.0
trunc (1.6) = 1   trunc (-1.6) = -1.0
trunc (1.7) = 1   trunc (-1.7) = -1.0
trunc (1.8) = 1   trunc (-1.8) = -1.0
trunc (1.9) = 1   trunc (-1.9) = -1.0
trunc (2.0) = 2   trunc (-2.0) = -2.0
```

## UNPACK (z, a, i)

copies packed array z, to array a, beginning at index i

**Definition:**

```
procedure unpack ( z : packed_array_type;
                  var a : unpacked_array_type;
                  i : integer );
```

where

*z* is the source packed array,  
*a* is the destination unpacked array,  
*i* is an expression that is compatible with the index type of *z*.

UNPACK copies elements from the packed array *z*, to the unpacked array *a*, beginning with the *i*-th element of *z*. The element types of the two arrays must be identical. There must be enough elements in the *a* array to receive the elements copied from *z*.

Note that UNPACK is defined for one-dimensional arrays only.

UNPACK is equivalent to the following definition:

```
type
  t1 = array [m..n] of T;
  t2 = packed array [u..v] of T;

procedure unpack ( z : t2;
                  var a : t1;
                  i : integer);

var j,k:integer;

begin
  k := i;
  for j := u to v do begin
    a [k] := z [j];
    k := k + 1;
  end;

end;
```

# WRITE and WRITELN for TEXT files.

## Definition

```
procedure write (f : text; exp : see_below);
procedure WRITELN (f : text; exp : see_below);
procedure WRITELN (f : text);
```

## where

*f* is an optional TEXT file opened for output, if omitted then file OUTPUT is assumed.

*exp* is a list of expressions, separated by commas, of any combination of the following types: BOOLEAN, CHAR, INTEGER, DOUBLE, FLOAT, and REAL.

The WRITE procedure writes character data to the text file *f*. Each expression in *exp* is evaluated and converted to character data.

The WRITELN procedure writes data in the same manner as WRITE if any variables are present. WRITELN then writes an end of line marker to file *f*. Note that WRITELN is only applicable to TEXT files.

Errors will cause the global variable *errno* to be set appropriately.

Formatting capability is provided for the data generated by the WRITE and WRITELN procedures.

The WRITE and WRITELN procedure allows the length of the output to be controlled by specifying additional options following the expression. The options take the form

```
exp : width : fraction
```

## where

*exp* is the parameter to the WRITE or WRITELN procedure as described above,  
*width* is an expression that must evaluate to an integer,  
*fraction* is an expression that must evaluate to an integer.

*width* indicates the length of the field into which the result of the expression *exp* will be placed. The data is placed left-justified in the case of type BOOLEAN and CHARACTER, and right-justified in the case of type INTEGER, DOUBLE, FLOAT and REAL.

*fraction* is only applicable to type DOUBLE, FLOAT and REAL, and indicates the number of digits to be printed after the decimal point (within the bounds of the *width* parameter).

The following table indicates the field widths used by default:

type	field width
BOOLEAN	6
CHAR	1
character string	actual size
INTEGER	12
DOUBLE	24
FLOAT	14
REAL	14 or 24

Except for CHAR and character string data, each data type is printed with a leading space. This space is included in the field width given in the above table.

BOOLEAN data is printed as either "TRUE" or "TRUE" right-justified in a field of 6 characters.

Numeric data is right-justified in a field whose size is given in the above table. The sign is printed as a space for positive numbers, and the minus sign is used for negative numbers. The data type DOUBLE, FLOAT and REAL are printed in scientific notation by default, i.e., a number in the form

```
1.12345678e+12
```

where the number of digits after the decimal point is 8 for FLOAT, 17 for DOUBLE, and 8 or 17 for REAL, depending upon the compiler options.

### Writing CHAR data

The value of *width* indicates the length of the field in which the character is to be placed. If *width* is not specified, a value of 1 is used. The character data is right justified, i.e., it is padded on the left with blanks.

In the following examples, b represents a blank space in the output result.

write statement	output
writeln ('x' : 1);	x
writeln ('x' : 2);	bx
writeln ('x' : 3);	bbx
writeln ('x' : 4);	bbbx

### Writing BOOLEAN data

The value of *width* indicates the length of the field in which the boolean data is to be placed. If *width* is not specified, a value of 6 is used. The data is right justified.

In the following examples, b represents a blank space in the output result.

write statement	output
writeln (true : 3);	true
writeln (true : 4);	true
writeln (true : 5);	btrue
writeln (true : 6);	bbtrue

### Writing INTEGER data

The value of *width* indicates the length of the field in which the INTEGER data is to be placed. The data is converted to character format and placed right-justified into this field. If the length of the field is shorter than necessary, then the field is extended as needed. In the following examples, b represents a blank space in the output result.

write statement	output
writeln (123 : 1);	123
writeln (123 : 2);	123
writeln (123 : 3);	123
writeln (123 : 4);	b123
writeln (123 : 5);	bb123

### Writing DOUBLE, FLOAT, and REAL data

WRITE and WRITELN allow a parameter expression to be formatted by an option placed after the expression. This has the form:

*exp: width: fraction*

The value of *width* indicates the length of the field in which the data is to be placed. The value of *ndigits* indicates the number of digits to be printed after the decimal point. The number will be formatted in scientific notation unless the parameter *ndigits* is included. When *ndigits* is specified, then the number is printed in fixed format.

When a number is printed in scientific notation, the width of the print field is extended if it is insufficient to contain the entire number. This is not the case for numbers printed in fixed format. When a number is printed in fixed format, the number is truncated if the width of the print field will not hold the number. Caveat programmer.

In the following examples, b represents a blank space in the output result.

write statement	output
writeln (-1.23456e-10 : 8);	b-1.2e-10
writeln (-1.23456e-10 : 9);	b-1.23e-10

```

writeln (-1.23456e-10 :10);      b-1.235e-10
writeln (-1.23456e-10 :11);      b-1.2346e-10
writeln (-1.23456e-10 :12);      b-1.23456e-10
writeln (-1.23456e-10 :13);      b-1.234560e-10
writeln (-1.23456e-10 :14);      b-1.2345600e-10
writeln (-1.23456e-10 : 8 : 2);    bbb-0.00
writeln (-1.23456e-10 : 9 : 2);    bbbb-0.00
writeln (-1.23456e-10 :10 : 2);    bbbbb-0.00
writeln (-1.23456e-10 :11 : 2);    bbbbbb-0.00
writeln (-1.23456e-10 :12 : 2);    bbbbbbb-0.00
writeln (-1.23456e-10 :13 : 2);    bbbbbbbb-0.00
writeln (-1.23456e-10 :14 : 2);    bbbbbbbbb-0.00

writeln (-1.23456e-10 :22 :12);    bbbbbbbb-0.000000000123
writeln (-1.23456e-10 :22 :13);    bbbbbbb-0.0000000001235
writeln (-1.23456e-10 :22 :14);    bbbbbbb-0.00000000012346
writeln (-1.23456e-10 :22 :15);    bbbbb-0.000000000123456
writeln (-1.23456e-10 :22 :16);    bbb-0.0000000001234560
writeln (-1.23456e-10 :22 :17);    bb-0.00000000012345600
writeln (-1.23456e-10 :22 :18);    b-0.000000000123456000

```

### Writing string data

The value of *width* indicates the length of the field in which the string is to be placed. The string will be right-justified in the field. If the length of the field is shorter than necessary, the string will be truncated on the right.

In the following examples, *b* represents a blank space in the output result.

write statement	output
writeln ('xyz':1);	x
writeln ('xyz':2);	xy
writeln ('xyz':3);	xyz
writeln ('xyz':4);	bxyz
writeln ('xyz':5);	bbxyz

### WRITE to non-TEXT files

#### Definition

```
procedure write ( f : ft; exp : t);
```

where

*f* is a file variable,  
*exp* is an expression that evaluates to type *t*.

Procedure WRITE writes the value of the expression *exp* to the file *f*.

WRITE (*f*, *exp*) is equivalent to the following:

```
begin f^ := exp; PUT (f) end
```

# 11

## Preprocessor Commands

The following is a list of commands interpreted by the preprocessor.

### #DEFINE

replace a name with a string of characters

#### Syntax

```
#DEFINE n s  
#DEFINE n(n1, n2, . . . ) s
```

where

*n* is the macro name,  
*n1*, *n2*, . . . are the formal parameters of the macro,  
*s* is the text that is to be substituted for the name *n*.

The #DEFINE directive is a macro definition. The first form is used to replace an identifier *n* with an arbitrary sequence of characters *s*. The second form allows arguments to be substituted into the replacement text.

The macro name is an identifier with the same syntax as a variable, while the string *s* is arbitrary. No spaces are allowed between the macro name and the open parenthesis. The scope of the macro name is from its point of definition to the end of the file being compiled. A definition may use previous definitions. Substitutions are only made for identifiers, and do not take place within strings. For example, if *Ver* is a defined name, then there is no substitution in *VERSION*, or in the string '*Ver*'.

Strings can be declared for the preprocessor by using the `-D` option on the compiler driver.

### #UNDEF

cancels previous #DEFINE

#### Syntax

```
#UNDEF n
```

where *n* is the name of the symbol to be undefined.

The #UNDEF directive removes the name *n*, which was previously defined with the #DEFINE directive, from the preprocessor symbol table.

### #INCLUDE

redirects compiler input to a supplementary file

#### Syntax

```
#INCLUDE f
```

where *f* is the name of the file to be included. This must be a complete file name in single quotes.

The #INCLUDE command directs the compiler to begin reading its input from the file *f*. The file that is included is placed immediately after the current line. When the end of the file is reached, the compiler will resume reading from the file containing the last #INCLUDE command. INCLUDE statements may be nested 16 levels deep.

## #IF

alter preprocessor control flow based upon result of expression

### Syntax

```
#IF e
```

where *e* is a constant expression.

The #IF directive evaluates a constant integer expression. If the result is nonzero, then the lines following the #IF are evaluated until an #ELSE or #ENDIF directive occurs. If the result is zero, then the lines following the #IF directive are skipped until an #ELSE occurs, and the lines within the #ELSE clause are evaluated.

## #IFDEF

alter preprocessor control flow based upon presence of a symbol

### Syntax

```
#IFDEF n
```

where *n* is an identifier name.

The #IFDEF directive determines if the name *n* is currently defined with a #DEFINE statement. If the name is defined, then the lines following the #IFDEF are evaluated until an #ELSE or #ENDIF directive occurs. If the name is not defined, then the lines following the #IFDEF directive are skipped until an #ELSE occurs, and the lines within the #ELSE clause, if present, are evaluated.

## #IFNDEF

alter preprocessor control flow based upon the absence of a symbol

### Syntax

```
#IFNDEF n
```

where *n* is an identifier name.

The #IFNDEF directive determines if the name *n* is currently NOT defined with a #DEFINE statement. This is the opposite of the #IFDEF directive, and works in the same way.

## #ELSE

alternative clause for #IF, #IFDEF or #IFNDEF directive.

### Syntax

```
#ELSE
```

This directive is an optional clause in an #IF, #IFDEF, or #IFNDEF construct. The text following the #ELSE directive is evaluated if the result of the previous #IF, #IFDEF, or #IFNDEF was zero or FALSE.

## #ENDIF

terminator for an #IF, #IFDEF, or #IFNDEF statement

### Syntax

```
#ENDIF
```

The #ENDIF directive is used to end an #IF, #IFDEF and #IFNDEF statement.

# #LINE

set line number and file name for error messages

## Syntax

```
#LINE c
#LINE c f
```

where

*c* is an integer that will be reported as a line number,  
*f* is an identifier that will be reported as a file name.

The #LINE directive renumbers the lines in the input file to simplify of error reporting. The lines are numbered sequentially beginning with the line following the #LINE directive, and starting with the value *c*. Any syntax errors reported by the compiler occurring after a #LINE directive, use the re-sequenced line number.

The second form of the #LINE directive renames the source file besides renumbering the lines as described above. Any syntax errors reported by the compiler occurring after a #LINE *c f* directive, use the re-sequenced line number and the renamed source file name.

### EXAMPLE 1 (ex021.p)

```
program pre10;
#define swap(a, b)  t := a; a := b; b := t;
#define swap3(a, b, c) swap(a,b); swap(b,c);

var
  t, x, y, z: integer;

begin
  x := 1;
  swap(x, y);
  writeln('x = ', x:2, ' y = ', y:2);
  x := 1; y := 2; z := 3;
  swap3(x, y, z);
  writeln('x = ', x:2, ' y = ', y:2, ' z = ', z:2);
end.
```

The above program uses the #DEFINE directive to create two simple macros. The program produces the following output:

```
x = 2  y = 1
x = 2  y = 3  z = 1
```

### EXAMPLE 2

```
program pre11;
{ example of preprocessor directive }
begin
#line 1234
  error1;
  error2;
end.
```

The above program illustrates the #LINE directive. Since the routines error1 and error2 are not defined, the compiler will print the following error message when compiling this program:

```
"pre11.p", line 1234: Undefined symbol:  error1
"pre11.p", line 1234: Undefined symbol:  error2
```



**EXAMPLE 3**

The following program fragment illustrates the use of the #DEFINE directive to set a debugging flag, and to establish parameters in a data type.

```
program pre12;

{ #define example }
#define DEBUGGING      1
#define STACKSIZE      100
#define ELEMENT_TYPE   real

type
  stack = record
    top : integer;
    elements : array [1..STACKSIZE] of ELEMENT_TYPE
  end;

procedure pop (var s:stack; var e:ELEMENT_TYPE);
begin
  if s.top > STACKSIZE then
    writeln ('stack is empty')
  else begin
    e := s.elements [s.top];
    s.top := s.top +1;
#ifdef DEBUGGING
    writeln ('pop stack');
    writeln ('  stack index   =', s.top-1);
    writeln ('  stack contents =', e);
#endif
  end;
end;

...
```

# A

## Selected Bibliography

American National Standards Committee, *IEEE Standard Pascal Computer Programming Language*, 1983, The Institute of Electrical and Electronic Engineers, Inc.

This book contains the ANSI/IEEE Pascal standard, 770X3.97-1983, which is implemented by Microway.

Cooper, Doug, and Michael Clancy, *Oh! Pascal!*, second edition, W.W. Norton & Company, 1985

An excellent introductory textbook for learning Pascal. Each feature of the language is illustrated with realistic examples, along with sound software engineering principles. The book contains 16 chapters, a glossary, and answers to selected exercises.

Grogono, Peter, *Programming in Pascal*, second edition, Addison-Wesley, 1984

Another excellent textbook for learning Pascal. While this book is smaller than the one by Cooper and Clancy, the lucid explanations of the language elements rival that of any text on any computer language. The book contains 10 chapters, many appendices, and an annotated bibliography. Regrettably, the random number generator presented in this book does not have a full period, yet has appeared in many other Pascal books. See the article "Random Number Generators: Good Ones are Hard to Find" by Park and Miller CACM, Vol 31, Num 10, Oct., 1988 for a correction.

Jensen, Kathleen, and Niklaus Wirth, *Pascal User Manual and Report*, third edition, Springer-Verlag, 1985.

The second edition of this book was used as the definition of the Pascal language until the British and American standards were finalized. The third edition was revised by Andrew Mickel and James Miner who have greatly improved the text, examples, index, and typography.

Ledgard, Henry, *The American Pascal Standard: With Annotations*, Springer-Verlag, 1984.

This book contains the same text as the ANSI/IEEE standard but with notes in the margins. These notes contain explanations for the terse prose used in the standard and, often, the problem that motivated the rule under discussion.

Wood, Derick, *Paradigms and Programming with Pascal*, Computer Science Press, 1984

An excellent book that methodically develops solutions to many classical programming problems. Many of the algorithms receive mathematical analysis so that the performance of the resulting program can be understood.



# B

## Interface to C and Math Libraries

### Overview

Three different libraries are included with NDP Pascal: the Pascal standard library, the math library, and the C library. Each library comes in a different version depending upon its contents and the coprocessor with which it is intended to be run.

The compiler driver automatically requests that the linker include the correct version of each library depending upon the options provided. The following gives a brief description of the libraries. See the *NDP User's Manual* for more details on the use of the compiler driver.

### The Standard Pascal Library

The routines in the standard library are built into the Pascal language and are documented in *Chapter 10*. No special provision need be taken in order to use the routines in the Pascal library in a program.

### The Math and C Libraries

The routines in these libraries are used by the compiler and several of them may be called from a Pascal program. In order to use a routine in the Math or C libraries, the Pascal program must contain the function definition that is given in the documentation for each routine described in this appendix.

### Contents of the Math and C libraries

The following is a summary of the contents of the math and C libraries that are available to the NDP Pascal programmer. The remainder of this appendix contains a complete description of each routine.

#### General string, file, and I/O routines

access	determine file accessibility
atof	ASCII to floating point conversion
atoi	ASCII to integer conversion
bcmp	byte string compare
bcopy	copy sequence of bytes
bufcpy	copy sequence of bytes
bzero	byte string zero
clearn	clear bytes in memory
date	return date in ASCII format
dosdat	return date from DOS or UNIX
dostim	return time from DOS or UNIX
ffs	find first set bit
filln	fill <i>n</i> bytes of memory with a specific character
getenv	get environment variable
idate	return date in integer format
index	index of a character in a string
mapdev	map physical memory to program's data segment
modf	split a 64-bit number into integer and fractional parts
sec_100_	return hundredths of a second since midnight
secnds_	return number of seconds from an origin

<code>sprintf</code>	print formatted output to a string
<code>sscanf</code>	read formatted input from a string
<code>system</code>	shell to DOS
<code>time</code>	return time in ASCII format
<code>timedate_</code>	return date and time in integer format

**Bessel Functions**

First Kind	Second Kind	Description
<code>j0</code>	<code>y0</code>	Order 0
<code>j1</code>	<code>y1</code>	Order 1
<code>jn</code>	<code>yn</code>	Order n

**Trigonometric Functions**

Double Precision	Single Precision	Description
<code>acos</code>	<code>racos</code>	arc cosine
<code>asin</code>	<code>rasin</code>	arc sine
<code>atan</code>	<code>ratan</code>	arc tangent
<code>atan2</code>	<code>ratan2</code>	arc tangent of a quotient
<code>cos</code>	<code>rcos</code>	cosine
<code>cosh</code>	<code>rcosh</code>	hyperbolic cosine
<code>sin</code>	<code>rsin</code>	sine
<code>sinh</code>	<code>rsinh</code>	hyperbolic sine
<code>tan</code>	<code>rtan</code>	tangent
<code>tanh</code>	<code>rtanh</code>	hyperbolic tangent

**Miscellaneous Mathematical Functions**

<code>cabs</code>	absolute value of a complex number
<code>ceil</code>	ceiling function
<code>erf</code>	error function
<code>erfc</code>	complementary error function
<code>floor</code>	floor function
<code>fmod</code>	floating point modulo
<code>gamma</code>	log gamma function
<code>hypot</code>	hypotenuse of a right triangle
<code>ldexp</code>	load exponent
<code>log10</code>	base 10 logarithm
<code>pow</code>	x raised to the power y
<code>srand</code>	seed random number generator
<code>rand</code>	random integer

**access**

check file accessibility

**Definition**

```
type cp = ^char;
function access (path: cp; mode: integer): integer;
```

where

*path* is a pointer to a null terminated character string containing the complete path and file name,

*mode* indicates the type of access desired and is the logical OR of the following values:

- 0: check for existence of file
- 2: check if file can be written to
- 4: check if file is readable.

The `access(path, mode)` function checks the accessibility of the file named `path` in the manner specified by `mode`. `access` returns 0 if all the specified operations are available. If any of the specified operations are not possible, then `access` returns -1 and the global variable `errno` is set appropriately.

#### EXAMPLE (ex022.p)

```

program access1(output);
type cp = ^char;
function access(path: cp; mode: integer): integer; external;
var
  fn: packed array[1..10] of char;
begin
  fn := 'acc.dat';
  fn[8] := chr(0);
  writeln('file name = ', fn);
  writeln('mode 0 access indicator = ', access(&fn[1], 0));
  writeln('mode 2 access indicator = ', access(&fn[1], 2));
  writeln('mode 4 access indicator = ', access(&fn[1], 4));
end.

```

This program generates the following output when the file `acc.dat` is present in the directory in which `access` is run:

```

file name =acc.dat
mode 0 access indicator =      0
mode 2 access indicator =      0
mode 4 access indicator =      0

```

This program generates the following output when the file `acc.dat` is NOT present in the directory in which `access` is run:

```

file name =acc.dat
mode 0 access indicator =     -1
mode 2 access indicator =     -1
mode 4 access indicator =     -1

```

## acos

Arc cosine

#### Definition

```
function acos (d: double): double;
```

where `d` is an expression of type `DOUBLE`.

The `acos(d)` function returns the principal value of the arc cosine of `d`. `acos` takes an argument in the range -1 to 1, and returns a result in the range 0 to pi, expressed in radians. If the input argument is outside of -1 to 1, then `acos` returns 0, and the global variable `errno` is set to `EDOM`.

#### EXAMPLE (ex023.p)

```

program acos1(output);
function acos(d: double): double; external;

```

```

begin
  writeln('acos (0) = ', acos(0.0));
  writeln('acos (-1) = ', acos(-1.0));
  writeln('acos (1) = ', acos(1.0));
  writeln('acos (2) = ', acos(2.0));
  writeln('acos (-3) = ', acos(-3.0));
end.

```

This program generates the following output:

```

acos ( 0) = 1.57079632679489657e+00
acos (-1) = 3.14159265358979311e+00
acos ( 1) = 0.00000000000000000e+00
acos ( 2) = 0.00000000000000000e+00
acos (-3) = 0.00000000000000000e+00

```

## acosf

Single precision arc cosine

### Definition

```
function acosf (f: float): float;
```

where  $f$  is an expression of type `FLOAT`.

The `acosf(f)` function returns the principal value of the arc cosine of  $f$ . `acosf` takes an argument in the range  $-1$  to  $1$ , and returns a result in the range  $0$  to  $\pi$ , expressed in radians. If the input argument is outside of  $-1$  to  $1$ , then `acosf` returns  $0$ , and the global variable `errno` is set to `EDOM`.

## acosh

Inverse hyperbolic cosine

### Definition

```
function acosh (d: double): double;
```

where  $d$  is an expression of type `DOUBLE`.

The `acosh(d)` function returns the value of the inverse hyperbolic cosine of  $d$ .

## asin

Arc sine

### Definition

```
function asin (d: double): double;
```

where  $d$  is an expression of type `DOUBLE`.

The `asin(d)` function returns the principal value of the arc sine of  $d$ . `asin` takes an argument in the range  $-1$  to  $1$ , and returns a result in the range  $-\pi/2$  to  $\pi/2$ , expressed in radians. If the input argument is outside of  $-1$  to  $1$ , then `asin` returns  $0$ , and the global variable `errno` is set to `EDOM`.

### EXAMPLE (ex024.p)

```

program asin1(output);
  function asin(d: double): double; external;

```

```

begin
  writeln('asin (0) = ', asin(0.0));
  writeln('asin (-1) = ', asin(-1.0));
  writeln('asin (1) = ', asin(1.0));
  writeln('asin (2) = ', asin(2.0));
  writeln('asin (-3) = ', asin(-3.0));
end.

```

This program generates the following output:

```

asin ( 0) = 0.0000000000000000e+00
asin (-1) = -1.57079632679489657e+00
asin ( 1) = 1.57079632679489657e+00
asin ( 2) = 0.0000000000000000e+00
asin (-3) = 0.0000000000000000e+00

```

## asinf

Single precision arc sine

### Definition

```
function asinf (f: float): float;
```

where  $f$  is an expression of type `FLOAT`.

The `asinf(f)` function returns the principal value of the arc sine of  $f$ . `asinf` takes an argument in the range  $-1$  to  $1$ , and returns a result in the range  $-\pi/2$  to  $\pi/2$ , expressed in radians. If the input argument is outside of  $-1$  to  $1$ , then `asinf` returns  $0$ , and the global variable `errno` is set to `EDOM`.

## asinh

Inverse hyperbolic sine

### Definition

```
function asinh (d: double): double;
```

where  $d$  is an expression of type `DOUBLE`.

The `asinh(d)` function returns the value of the inverse hyperbolic sine of  $d$ .

## atan

arc tangent

### Definition

```
function atan (d: double): double;
```

where  $d$  is an expression of type `DOUBLE`.

The `atan(d)` function returns the principal value of the arc tangent of  $d$ . `atan` returns a result in the range  $-\pi$  to  $\pi$ , expressed in radians.

### EXAMPLE (ex025.p)

```

program atan1(output);
function atan(d: double): double; external;
begin
  writeln('atan (0) = ', atan(0.0));
  writeln('atan (-1) = ', atan(-1.0));

```



```
writeln('atan (1) = ', atan(1.0));
writeln('atan (2) = ', atan(2.0));
writeln('atan (-3) = ', atan(-3.0));
end.
```

This program generates the following output:

```
atan ( 0) = 0.0000000000000000e+00
atan (-1) = -7.85398163397448286e-01
atan ( 1) = 7.85398163397448286e-01
atan ( 2) = 1.10714871779409042e+00
atan (-3) = -1.24904577239825442e+00
```

## atan2

arc tangent of a quotient

### Definition

```
function atan2 (d1, d2: double): double;
var errno: integer;
```

where

*d1* is an expression of type DOUBLE,  
*d2* is an expression of type DOUBLE, which has a nonzero value.

The `atan2(d1, d2)` function returns the principal value of the arc tangent of  $d1/d2$ . `atan2` returns a value between  $-\pi$  and  $\pi$ , expressed in radians. The signs of both arguments are used to determine the quadrant of the result. If the second argument is zero, then `atan2` returns 0 and the global variable `errno` is set to `EDOM` (which is represented by a zero).

The `atan2` function is used to avoid computation with large numbers that might overflow. It permits the expression of large tangent values as the quotient of two double precision numbers.

### EXAMPLE (ex026.p)

```
program atan2(output);
function atan2(d1,d2: double): double; external;
begin
  writeln('atan2 (0,1) = ', atan2(0.0,1.0));
  writeln('atan2 (-1,1) = ', atan2(-1.0,1.0));
  writeln('atan2 (1,1) = ', atan2(1.0,1.0));
  writeln('atan2 (20,0.1) = ', atan2(20.0,0.1));
end.
```

This program generated the following output:

```
atan2 ( 0.1) = 0.0000000000000000e+00
atan2 (-1.1) = -7.85398163397448286e-01
atan2 ( 1.1) = 7.85398163397448286e-01
atan2 ( 20,0.1) = 1.56579636846093819e+00
```

## atan2f

Single precision arc tangent of a quotient

### Definition

```
function atan2f (f1, f2: float): float;
var errno: integer;
```

where

$f1$  is an expression of type `FLOAT`,  
 $f2$  is an expression of type `FLOAT`, which has a nonzero value.

The `atan2f( $f1$ ,  $f2$ )` function returns the principal value of the arc tangent of  $f1/f2$ . `atan2f` returns a value between  $-\pi$  and  $\pi$ , expressed in radians. The signs of both arguments are used to determine the quadrant of the result. If the second argument is zero, then `atan2f` returns 0 and the global variable `errno` is set to `EDOM` (which is represented by a zero).

The `atan2f` function is used to avoid computation with large numbers that might overflow. It permits the expression of large tangent values as the quotient of two single precision numbers.

## atanf

Single precision arc tangent

### Definition

```
function atanf (f: float): float;
```

where  $f$  is an expression of type `FLOAT`.

The `atanf( $f$ )` function returns the principal value of the arc tangent of  $f$ . `atanf` returns a result in the range  $-\pi$  to  $\pi$ , expressed in radians.

## atanh

Inverse hyperbolic tangent

### Definition

```
function atanh (d: double): double;
```

where  $d$  is an expression of type `DOUBLE`.

The `atanh( $d$ )` function returns the value of the inverse hyperbolic tangent of  $d$ .

## atof

ASCII to floating point conversion

### Definition

```
type cp = ^char;
function atof (str: cp): float;
```

where  $str$  is a pointer to a null terminated string.

The `atof( $str$ )` function converts the null terminated string pointed to by  $str$  to a double-precision floating point value. `atof` starts at the beginning of the string and converts each character in turn. Conversion stops when the character is not recognizable as part of a floating point number. `atof` returns the value converted, even if the end of the string has not been reached.

### EXAMPLE (ex027.p)

```
program atof(output);
type
  cp = ^char;
function atof(arr: cp): float; external;
var
  a: packed array [1..28] of char;
```

```

begin
  a := '12345e-17';
  a[10] := chr(0);
  writeln('The string 12345e-17 has the value ', atof(&a[1]));
end.

```

This program generates the following output:

```
The string 12345e-17 has the value 1.2345000e-13
```

## atoi

### ASCII to integer conversion

#### Definition

```

type cp = ^char;
function atoi(str: cp): integer;

```

where *str* is a pointer to a null terminated string.

The `atoi(str)` function converts the null terminated string pointed to by *str* into an integer value. `atoi` does not recognize decimal points or exponents. It stops converting the input string when it encounters a character that is not recognizable as part of an integer.

#### EXAMPLE (ex028.p)

```

program atoi1(output);
type
  cp = ^char;
function atoi(arr: cp): integer; external;
var
  a: packed array [1..10] of char;
begin
  a := '123456789';
  a[10] := chr(0);
  writeln('character array = ', a);
  writeln('Number beginning at column 5 = ',atoi(&a[5]));
  writeln('Number beginning at column 2 = ',atoi(&a[2]));
  writeln('Number beginning at column 1 = ',atoi(&a[1]));
end.

```

This program generates the following output:

```

character array = 123456789
Number beginning at column 5 =      56789
Number beginning at column 2 =    23456789
Number beginning at column 1 = 123456789

```

## atol

### ASCII to long integer conversion

#### Definition

```

type cp = ^char;
function atol(str: cp): integer;

```

where *str* is a pointer to a null terminated string.

The `atol(str)` function converts the null terminated string pointed to by `str` into an integer value. `atol` does not recognize decimal points or exponents. It stops converting the input string when it encounters a character that is not recognizable as part of an integer. In NDP Pascal, this function is identical to `atoi` since integers are 4-byte by default.

## bcmp

byte string compare

### Definition

```
type cp = ^char;
function bcmp (a1,a2:cp, n:integer): integer;
```

where

`a1` and `a2` are pointers to an array of characters,  
`n` is an integer less than the length of the arrays at `a1` and `a2`.

The `bcmp(a1,a2,n)` function compares the first `n` characters beginning at `a1` (call these characters `list1`) with the first `n` characters beginning at `a2` (call these characters `list2`) and returns one of the following values indicating their relationship:

```
if list1 < list2 then return a negative number
if list1 = list2 then return zero
if list1 > list2 then return a positive number
```

### EXAMPLE (ex029.p)

```
program bcmp1(output);

const
  c1 = 'economy';
  c2 = 'ecology';
  c3 = 'eclipse';
  c4 = 'eclogue';

type
  cp = ^char;
  s10 = packed array [1..10] of char;

function bcmp(a,b:cp; n:integer): integer; external;

procedure bcompare(a,b:s10; n:integer);
begin
  writeln('comparison on first ', n:2,' characters = ',bcmp(&a[1],&b[1],n):3);
end;

begin
  writeln('string 1 = ',c1);
  writeln('string 2 = ',c2);
  bcompare(c1, c2, 3);
  bcompare(c1, c2, 4);
  bcompare(c1, c2, 7);
  writeln('string 3 = ',c3);
  writeln('string 4 = ',c4);
  bcompare(c3, c4, 3);
  bcompare(c3, c4, 4);
  bcompare(c3, c4, 5);
end.
```

This example generates the following output:

```
string 1 =economy
string 2 =ecology
comparison on first 3 characters = 0
comparison on first 4 characters = -2
comparison on first 7 characters = -2
string 3 =eclipse
string 4 =eclogue
comparison on first 3 characters = 0
comparison on first 4 characters = 6
comparison on first 7 characters = 6
```

## bcopy

copy sequence of bytes

### Definition

```
type cp = ^char
function bcopy (a1,a2: cp; n: integer): integer;
```

where

*a1* and *a2* are pointers to an array of characters,  
*n* is an integer less than the length of the arrays at *a1* and *a2*.

The `bcopy(a1, a2, n)` function copies *n* bytes from the address pointed to by *a1* to the address pointed to by *a2*.

Caution: It is the programmer's responsibility to ensure that the receiving buffer is large enough for what is written to it. If it is not large enough, adjacent buffers may be overwritten. Also, no check is made to determine if the source and destination buffers are overlapping.

### EXAMPLE (ex030.p)

```
program bcopy1(output);
type
  cp = ^char;
function bcopy(src, des: cp; n:integer): integer; external;
var
  a, b: packed array [1..10] of char;
begin
  a := 'abcdefghij';
  b := '1234567890';
  writeln('initial string 1 = ', a);
  writeln('initial string 2 = ', b);
  bcopy(&a[2], &b[3], 4);
  writeln;
  writeln('altered string 2 = ', b);
end.
```

This program generates the following output:

```
initial string 1 = abcdefghij
initial string 2 = 1234567890
altered string 2 = 12bcde7890
```

# bufcpy

copy sequence of bytes

## Definition

```
type cp = ^char;
function bufcpy (a1,a2: cp; n: integer): integer;
```

where

*a1* and *a2* are pointers to an array of characters,  
*n* is an integer less than the length of the arrays at *a1* and *a2*.

The `bufcpy(a1, a2, n)` function copies *n* bytes from the address pointed to by *a2* to the address pointed to by *a1*.

Caution: It is the programmer's responsibility to ensure that the receiving buffer is large enough for what is written to it. If it is not large enough, adjacent buffers may be overwritten. Also, no check is made to determine if the source and destination buffers are overlapping.

## EXAMPLE (ex031.p)

```
program bufcpy1(output);
type
  cp = ^char;
function bufcpy(des, src:cp; n:integer): integer; external;
var
  a, b: packed array [1..10] of char;
begin
  a := 'abcdefghij';
  b := '1234567890';
  writeln('initial string 1 = ', a);
  writeln('initial string 2 = ', b);
  bufcpy(&b[2], &a[3], 4);
  writeln;
  writeln('altered string 2 = ', b);
end.
```

This program generates the following output:

```
initial string 1 = abcdefghij
initial string 2 = 1234567890
altered string 2 = 1cdef67890
```

# bzero

byte string zero

## Definition

```
type cp = ^char;
function bzero (a1:cp; n:integer): integer;
```

where

*a1* is a pointer to an array of characters,  
*n* is an integer less than the length of the array at *a1*.

The `bzero(a1, n)` function stores binary zeros in the *n* bytes pointed to by *a1*.

Caution: It is the programmer's responsibility to ensure that the receiving buffer is large enough for what is written to it. If it is not large enough, adjacent buffers may be overwritten.

**EXAMPLE (ex032.p)**

```

program bzer1(output);
type
  cp = ^char;
function bzero(a1:cp; n:integer): integer; external;
type
  pa = packed array [1..10] of char;
var
  a1: pa;
  f: file of pa;
  i: integer;
begin
  a1 := 'abcdefghij';
  writeln('initial array = ', a1);
  bzero(&a1[2], 4);
  write(' final array = ');
  for i := 1 to 10 do
    write (a1[i]);
  rewrite(f, 'bzer01.dat');
  f^ := a1;
  put(f);
  writeln;
end.

```

This program generates the following output and writes the contents of the final array to the file named BZERO1.DAT:

```

initial array = abcdefghij
final array = a fghij

```

Notice that NDP Pascal prints the binary zeros in the array *a1* as blanks. The following hexadecimal dump of the file BZERO1.DAT verifies that *a1* contains the binary zeros at the correct locations:

```

0000: 61 00 00 00 00 66 67 68-69 6a          a....fghij

```

## cabs

absolute value of a complex number.

**Definition**

```
function cabs (d1, d2: double): double;
```

where *d1* and *d2* are expressions of type DOUBLE.

The *cabs(d1, d2)* function returns the absolute value of the complex number (*d1, d2*).

**EXAMPLE (ex033.p)**

```

program cabs1(output);
function cabs(d1,d2: double): double; external;
begin
  writeln('cabs(1,1) = ', cabs( 1.0, 1.0));
  writeln('cabs(3,-4) = ', cabs( 3.0, -4.0));
  writeln('cabs(-3,4) = ', cabs( -3.0, 4.0));
end.

```

```
writeln('cabs(-6,-8) = ', cabs(-6.0, -8.0));
end.
```

This program generates the following output:

```
cabs( 1, 1) = 1.41421356237309515e+00
cabs( 3,-4) = 5.0000000000000000e+00
cabs(-3, 4) = 5.0000000000000000e+00
cabs(-6,-8) = 1.0000000000000000e+01
```

## calloc

calloc

### Definition

```
procedure calloc(nmemb, size: integer);
```

where

*nmemb* is the number of items to be stored;  
*size* is the size of each item.

calloc allocates a block of zero filled memory large enough to hold the number of items specified in the first argument of a size specified in the second argument.

## ceil

ceiling

### Definition

```
function ceil (d: double): double
```

where *d* is an expression of type DOUBLE.

The `ceil(d)` function returns a 64-bit floating point result representing the smallest integer that is greater than or equal to *d*.

### EXAMPLE (ex034.p)

```
program ceil(output);
function ceil(d: double): double; external;
var
  i: integer;
  x: double;
begin
  x := 1.0;
  for i := 1 to 10 do begin
    writeln('x = ', x:5:1, ' ceil(x) = ', ceil(x):6:2);
    x := x + 0.1;
  end;
end.
```

This program generates the following output:

```
x = 1.0  ceil(x) = 1.00
x = 1.1  ceil(x) = 2.00
x = 1.2  ceil(x) = 2.00
x = 1.3  ceil(x) = 2.00
x = 1.4  ceil(x) = 2.00
x = 1.5  ceil(x) = 2.00
```



```

x = 1.6  ceil(x) = 2.00
x = 1.7  ceil(x) = 2.00
x = 1.8  ceil(x) = 2.00
x = 1.9  ceil(x) = 2.00

```

## clearn

clear *n* bytes of memory

### Definition

```

type cp = ^char;
function clearn (n: integer; a1: cp): integer;

```

where

*n* is an integer expression less than the length of the array at *a1*,  
*a1* is a pointer to an array of characters.

The `clearn(n, a1)` function stores binary zeros in the *n* bytes pointed to by *a1*.

**Caution:** It is the programmer's responsibility to ensure that the receiving buffer is large enough for what is written to it. If it is not large enough, adjacent buffers may be overwritten.

### EXAMPLE (ex035.p)

```

program clearn1(output);
type
  cp = ^char;
function clearn(n: integer; a1:cp): integer; external;
type
  pa = packed array [1..10] of char;
var
  a1: pa;
  f: file of pa;
  i: integer;
begin
  a1 := 'abcdefghij';
  writeln('initial array = ',a1);
  clearn(4, &a1[2]);
  write(' final array = ');
  for i:= 1 to 10 do
    write (a1[i]);
  rewrite(f, 'clearn1.dat');
  f^ := a1;
  put(f);
  writeln;
end.

```

This program generates the following output and writes the contents of the final array to the file named `CLEARN1.DAT`:

```

initial array = abcdefghij
final array = a fghij

```

Notice that NDP Pascal prints the binary zeros in the array *a1* as blanks. The following hexadecimal dump of the file `CLEARN1.DAT` verifies that *a1* contains the binary zeros at the correct locations:

```

0000: 61 00 00 00 00 66 67 68-69 6A          a....fghij

```

## clock

Show CPU time

### Definition

```
function clock: integer;
```

This function returns as an unsigned integer the number of clock ticks the CPU has spent on the current process.

## clrndpex

Clear the exception bits in the NDP status word.

### Definition

```
procedure clrndpex;
```

`clrndpex_` clears the exception bits of the status word of the active NDP. Nothing else in the coprocessor is affected. This function is available for the DOS, OS/2, and Windows 386/486 compilers.

## cosf

Single precision cosine

### Definition

```
function cos ( f : float ) : float;
```

where  $f$  is an expression of type `float`.

The `cosf` function converts the input parameter to a temporary 64-bit floating point number and returns the cosine of this value. The input parameter is assumed to be expressed in radians. This function differs from `cos` in that it returns a `float` rather than a `double`.

## cosh

Hyperbolic cosine

### Definition

```
function cosh (d: double): double;
```

where  $d$  is an expression of type `DOUBLE`.

The `cosh(d)` function returns the hyperbolic cosine of  $d$ . If  $d$  is too large, infinity is returned and the global variable `errno` is set to `ERANGE`.

### EXAMPLE (ex037.p)

```
program cosh1(output);
function cosh(d: double): double; external;
const
  pi = 3.14159265358979323846;
var
  i: integer;
  x: double;
begin
  x := 0.0;
  for i := 1 to 5 do begin
```

```

    writeln('x = ', x, ' cosh(x) = ', cosh(x));
    x := x + pi/4.0;
  end;
end.

```

This program generates the following output:

```

x = 0.0000000000000000+00 cosh(s) = 1.0000000000000000e+00
x = 7.85398163397448286e-01 cosh(s) = 1.32460908925200593e+00
x = 1.57079632679489657e+00 cosh(s) = 2.50917847865805618e+00
x = 2.35619449019234486e+00 cosh(s) = 5.32275214951995857e+00
x = 3.14159265358979311e+00 cosh(s) = 1.15919532755215169e+01

```

## coshf

Single precision hyperbolic cosine

### Definition

```
function coshf (f: float): float;
```

where  $f$  is an expression of type FLOAT.

The `coshf(f)` function returns the hyperbolic cosine of  $f$ . If  $f$  is too large, infinity is returned and the global variable `errno` is set to `ERANGE`.

## date\_

return date in ASCII format

### Definition

```

type s9 = packed array [1..9] of char;
procedure date_ (VAR date: s9); external;

```

where `date` is an array of at least nine characters.

The `date_` procedure returns a nine character ASCII string denoting the current date known to DOS, in the format `dd-mm-yy`.

### EXAMPLE (ex038.p)

```

program date1 (output);
type
  s9 = packed array [1..9] of char;
procedure date_ (var date: s9); external;
var
  today: s9;
begin
  date_(today);
  writeln('Today''s date is ',today);
end.

```

This program generates the following output:

```
Today's date is 08-MAY-89
```

# difftime

Difference between two calendar times

## Definition

```
function difftime(time1, time0: integer): double;
```

where *time1* and *time0* represent the stopping and starting times to be measured.

*difftime* returns the difference between its two arguments as a double.

# dosdat

return date from DOS

## Definition

```
procedure dosdat (VAR month, day, year, dayofw: integer); external;
```

where *month*, *day*, *year*, and *dayofw* must be integer variables.

The *dosdat* procedure returns four integers that correspond to the date maintained by DOS. The return parameters have the following range of values (*dayofw* is an abbreviation for day of week):

```
month   :      1 to 12
day     :      1 to 31
dayofw  :      1 to 7
```

## EXAMPLE (ex039.p)

```
program date (output);
procedure dosdat(var month, day, year, dayofw: integer); external;
var
  month, day, year, dayofw: integer;
begin
  dosdat(month, day, year, dayofw);
  writeln('month = ', month:4);
  writeln('day   = ', day:4);
  writeln('year  = ', year:4);
  writeln('dayofw = ', dayofw:4);
end.
```

This program generates the following output:

```
month =   4
day   =  24
year  =1989
dayofw =   1
```

# dostim

return time from DOS

## Definition

```
procedure dostim (VAR hours, minutes, seconds, hundredths: integer); external;
```

where *hours*, *minutes*, *seconds*, and *hundredths* must be integer variables.

The `dostim` procedure returns four integers that correspond to the time of day maintained by DOS. This is a 24-hour clock, so the return parameters have the following range of values:

```
hours       :      0 to 23
minutes     :      0 to 59
seconds     :      0 to 59
hundredths  :      0 to 99
```

#### EXAMPLE (ex040.p)

```
program time (output);
procedure dostim( var hours, minutes, seconds, hundredths: integer); external;
var
  hours, minutes, seconds, hundredths: integer;
begin
  dostim(hours, minutes, seconds, hundredths);
  writeln('hours       = ', hours);
  writeln('minutes     = ', minutes);
  writeln('seconds     = ', seconds);
  writeln('hundredths = ', hundredths);
end.
```

This program generates the following output:

```
hours       = 14
minutes     = 58
seconds     = 55
hundredths  = 84
```

## ecvt

Floating point to ASCII conversion

### Definition

```
type
  s9 = packed array [1..9] of char;
function ecvt(value: double; ndig, decpt, var sign: integer): s9
```

where

*value* represents the floating-point value to be converted;

*ndig* is the number of digits to which the number is to be rounded;

*decpt* is the decimal point position relative to the first character of the returned string. If *decpt* is zero the decimal point is immediately to the left of the first character of the returned string. If *decpt* is positive, the decimal point is to the left of the character numbered *decpt* (the first character being numbered zero). If *decpt* is negative, leading zeros have been suppressed and the decimal point is *decpt* characters to the left of the first character of the string.

*sign* is set on return to nonzero if the value is negative and zero otherwise.

`ecvt` returns an ASCII string equivalent to the first argument in `%e` format. Leading zeros may be suppressed.

# erf

error function

## Definition

```
function erf (d: double): double;
```

where  $d$  is an expression of type DOUBLE.

The `erf(d)` function returns the error function of  $d$ . For large arguments, `erfc`, the complementary error function, should be used to maintain accuracy.

## EXAMPLE (ex041.p)

```

program erf1 (output);
function erf (d: double): double; external;
var
  x: double;
  i: integer;
begin
  x := 0.0;
  for i := 1 to 6 do begin
    writeln(' x = ', x:5:2, ' erf(x) = ', erf(x));
    x := x + 0.2;
  end;
end.

```

This program generates the following output:

```

x = 0.00 erf(x) = 0.000000000000000000e+00
x = 0.20 erf(x) = 2.22702589210478451e-01
x = 0.40 erf(x) = 4.28392355046668436e-01
x = 0.60 erf(x) = 6.03856090847926019e-01
x = 0.80 erf(x) = 7.42100964707660362e-01
x = 1.00 erf(x) = 8.42700792949714914e-01

```

# erfc

complementary error function

## Definition

```
function erfc (d: double): double;
```

where  $d$  is an expression of type DOUBLE.

The `erfc(d)` function returns the complementary error function of  $d$ .

## EXAMPLE (ex042.p)

```

program erfc1 (output);
function erfc (d: double): double; external;
var
  x: double;
  i: integer;
begin
  x := 0.0;
  for i := 1 to 6 do begin
    writeln(' x = ', x:5:2, ' erfc(x) = ', erfc(x));
    x := x + 0.2;
  end;
end.

```

```

    end;
end.

```

This program generates the following output:

```

x = 0.00  erfc(x) = 1.0000000000000000e+00
x = 0.20  erfc(x) = 7.77297410789521411e-01
x = 0.40  erfc(x) = 5.71607644953331470e-01
x = 0.60  erfc(x) = 3.96143909152074024e-01
x = 0.80  erfc(x) = 2.57899035292339548e-01
x = 1.00  erfc(x) = 1.57299207050285108e-01

```

## **\_\_errno**

Returns value of `errno`

### Definition

```
function __errno: integer;
```

The global variable `errno` is set in many functions to indicate what sort of error occurred. `__errno` returns the value of `errno` so a program can respond appropriately to errors.

## **execl**

Executes a file

### Definition

```

type cp = ^char;
function execl(path, arg1, . . . char_zero: cp): integer;

```

where

*path* is the pointer to the path name for the new file to be executed;

*arg1* . . . is the pointer to the first argument. After the last argument, include a character (0) to let the function know there are no more arguments.

This function executes a file and does not return. On failure, the function returns -1 and the global variable `errno` is set appropriately. This function is identical to `execl` under UNIX.

## **execle**

Executes a file

### Definition

```

type cp = ^char;
function execle(path, arg1, . . . , argn, char_zero: cp): integer;

```

where

*path* is the pointer to the path name for the new file to be executed;

*arg1* . . . is the pointer to the first argument to be passed to the new file.

*argn* is the pointer to an array of pointers to the environment strings. After the last argument, include a character (0) to let the function know there are no more arguments.

This function executes a file and does not return. On failure, the function returns -1 and the global variable `errno` is set appropriately. This function is identical to `execle` under UNIX.

## execv

Executes a file

### Definition

```
type cp = ^char;
function execv(path, arg: cp): integer;
```

where

*path* is the pointer to the path name for the new file to be executed;  
*arg* is a pointer to an array of arguments to be passed to the new file.

This function executes a file and does not return. On failure, the function returns -1 and the global variable `errno` is set appropriately. This function is identical to `execv` under UNIX.

## exit

Terminate program.

### Definition

```
procedure exit(status: integer);
```

where *status* is the return value to the parent process.

`exit` ends the program, flushing all buffers and returning its integer argument to the parent process.

## fabs

Double precision absolute value.

### Definition:

```
function fabs ( d : double ) : double;
```

where *d* is an expression of type double,

The `FABS` function returns a double. The result is the absolute value of the input parameter. If the argument is out of range, the global variable `errno` will be set to `EDOM`. If the return value is out of range, `errno` will be set to `ERANGE`.

## \_fcvt

Convert F format to string

### Definition:

```
type cp= ^char;
function _fcvt( value: double; number_of_digits, decimal_point, var sign:
integer): cp;
```

where

*value* is the value to be converted;

*number\_of\_digits* is the number of digits to which *value* is to be rounded;

*decimal\_point* specifies the decimal point position relative to the first character of the returned string. If *decimal\_point* is zero the decimal point is immediately to the left of the first character of the returned string. If *decimal\_point* is positive, the decimal point is to the left of the character numbered *decimal\_point* (the first character being numbered zero). If *decimal\_point* is negative, leading zeros have been suppressed and the decimal point is *decimal\_point* characters to the left of the first character of the string.



*sign* is set to nonzero if the value is negative; otherwise it is set to zero.

`_fcvt` returns a pointer to a string containing the first argument in fixed point (%f) format. Leading zeros may be suppressed.

## ffs

find first set bit

### Definition

```
function ffs (i: integer): integer;
```

where *i* is an expression of type INTEGER.

The `ffs(i)` function returns the place of the first bit in *i* that is set, counting the least significant bit as 1.

### EXAMPLE (ex043.p)

```
program ffs1 (output);
function ffs(i: integer): integer; external;
var
  i,j: integer;
begin
  i := 1;
  for j := 1 to 16 do begin
    writeln(' i = ', i:6, ' ffs(i) = ', ffs(i):6);
    i := 2 * i;
  end;
end.
```

This program generates the following output:

```
i =      1 ffs(i) =      1
i =      2 ffs(i) =      2
i =      4 ffs(i) =      3
i =      8 ffs(i) =      4
i =     16 ffs(i) =      5
i =     32 ffs(i) =      6
i =     64 ffs(i) =      7
i =    128 ffs(i) =      8
i =    256 ffs(i) =      9
i =    512 ffs(i) =     10
i =   1024 ffs(i) =     11
i =   2048 ffs(i) =     12
i =   4096 ffs(i) =     13
i =   8192 ffs(i) =     14
i =  16384 ffs(i) =     15
i =  32768 ffs(i) =     16
```

## filln

fill *n* bytes of memory with a specific character

### Definition

```
type cp = ^char;
function filln (n: integer; a1: cp; ch: char): integer;
```

where

*n* is an integer less than the length of the array at *a1*,  
*a1* is a pointer to an array of characters,  
*ch* is the fill character.

The `filln(n, a1, ch)` function fills *n* bytes of memory pointed to by *a1* with the character *ch*.

**Caution:** It is the programmer's responsibility to ensure that the receiving buffer is large enough for what is written to it. If it is not large enough, adjacent buffers may be overwritten.

#### EXAMPLE (ex044.p)

```

program filln1(output);
type
  cp = ^char;
function filln(n:integer; des: cp; ch: char): integer; external;
var
  a: packed array [1..10] of char;
begin
  a := 'abcdefghij';
  writeln('initial array = ',a);
  filln(4, &a[2], '-');
  writeln(' final array = ', a);
end.

```

This program generates the following output:

```

initial array = abcdefghij
final array = a----fghij

```

## floor

floor

#### Definition

```
function floor (d: double): double;
```

where *d* is an expression of type DOUBLE.

The `floor(d)` function returns a 64-bit value that represents the largest integer that is less than or equal to *d*.

#### EXAMPLE (ex045.p)

```

program floor(output);
function floor(d:double): double; external;
var
  i: integer;
  x: double;
begin
  x := 1.0;
  for i:= 1 to 11 do begin
    writeln(' x = ', x:5:1, ' floor(s) = ', floor(x):6:2);
    x := x + 0.1;
  end;
end.

```

This program returns the following output:

```
x = 1.0 floor(x) = 1.00
x = 1.1 floor(x) = 1.00
x = 1.2 floor(x) = 1.00
x = 1.3 floor(x) = 1.00
x = 1.4 floor(x) = 1.00
x = 1.5 floor(x) = 1.00
x = 1.6 floor(x) = 1.00
x = 1.7 floor(x) = 1.00
x = 1.8 floor(x) = 1.00
x = 1.9 floor(x) = 1.00
x = 2.0 floor(x) = 2.00
```

## fmod

floating point modulo

### Definition

```
function fmod (d1, d2: double): double;
```

where *d1* and *d2* are expressions of type DOUBLE.

The `fmod(d1, d2)` function returns the floating point remainder of its arguments, such that  $d1 = z + n * d2$ , where *n* is the largest integer value for which the equation can be true for a non-negative *n*. If the input value is out of range, the global variable `errno` will be set to EDOM.

### EXAMPLE (ex046.p)

```
program fmod1 (output);
function fmod(d1, d2: double): double; external;
var
  x, y: double;
  i: integer;
begin
  x := 16.5;
  y := 1.5;
  for i := 1 to 16 do begin
    writeln( 'x = ', x:5:1, ' y = ', y:5:1, ' fmod(x,y) = ', fmod(x,y):6:2);
    y := y + 1;
  end;
end.
```

This program generates the following output:

```
x = 16.5 y = 1.5 fmod(x, y)= 0.00
x = 16.5 y = 2.5 fmod(x, y)= 1.50
x = 16.5 y = 3.5 fmod(x, y)= 2.50
x = 16.5 y = 4.5 fmod(x, y)= 3.00
x = 16.5 y = 5.5 fmod(x, y)= 0.00
x = 16.5 y = 6.5 fmod(x, y)= 3.50
x = 16.5 y = 7.5 fmod(x, y)= 1.50
x = 16.5 y = 8.5 fmod(x, y)= 8.00
x = 16.5 y = 9.5 fmod(x, y)= 7.00
x = 16.5 y = 10.5 fmod(x, y)= 6.00
x = 16.5 y = 11.5 fmod(x, y)= 5.00
x = 16.5 y = 12.5 fmod(x, y)= 4.00
x = 16.5 y = 13.5 fmod(x, y)= 3.00
```

```
x = 16.5 y = 14.5 fmod(x, y) = 2.00
x = 16.5 y = 15.5 fmod(x, y) = 1.00
x = 16.5 y = 16.5 fmod(x, y) = 0.00
```

## frexp

exponent and mantissa of a floating-point number

### Definition

```
function frexp (d: double, VAR exptr: integer): double;
```

This double precision function returns the mantissa of the first argument (double) and places its exponent into the second argument (integer). If the return value is out of range, the global variable `errno` will be set to `ERANGE`.

## frexpf

exponent and mantissa of single-precision float

### Definition

```
function frexpf (f: float, VAR exptr: integer): float;
```

This single precision function of `frexp` returns the mantissa of the first argument and places its exponent into the second argument. If the return value is out of range, the global variable `errno` will be set to `ERANGE`.

## gamma

log gamma function

### Definition

```
function gamma (d: double): double
```

where  $d$  is an expression of type `DOUBLE`.

The `gamma (d)` function returns the natural logarithm of the absolute value of the gamma function of  $d$ . The sign of the gamma function of  $d$  is returned in `signgam`. If the input value is out of bounds, the global variable `errno` will be set to `EDOM`.

### EXAMPLE (ex047.p)

```
program gamma1(output);
function gamma(d: double): double; external;
var
  signgam: integer; external;
  i: integer;
  x: double;
begin
  x := 1.0;
  for i := 1 to 5 do begin
    writeln(' x = ', x, ' gamma(x) = ', gamma(x));
    x := x * 10;
  end;
end.
```

This program generates the following output:

```
x = 1.0000000000000000e+00 gamma(x) = 0.0000000000000000e+00
x = 1.0000000000000000e+01 gamma(x) = 1.28018274800814673e+01
```

```

x = 1.0000000000000000e+02  gamma(x) = 3.59134205369575341e+02
x = 1.0000000000000000e+03  gamma(x) = 5.90522042320918081e+03
x = 1.0000000000000000e+04  gamma(x) = 8.20997174964423612e+04

```

## gcvt

convert floating-point to G format string

### Definition

```

type cp = ^char
function gcvt (d: double, ndig: integer, buf: cp): cp

```

This function takes three arguments, *d* (double), *ndig* (integer), and *buf* (character pointer). It converts the first argument to a properly rounded ASCII string placed both in the third argument and returned by the function. If possible it generates *number\_of\_digits* digits after the decimal point in %f format, otherwise in %e format.

## getdat

get date

### Definition

```

procedure getdat(VAR month, date, year, dayofweek: integer)

```

This procedure changes its four integer arguments to return the month, date, year, and day of week.

## getenv

get environment variable

### Definition

```

type cp = ^char;
function getenv (str:cp): cp;

```

where *str* is a pointer to a null terminated character string,

The `getenv(str)` function returns a pointer to the environment variable pointed to by *str*. The user should remember that DOS converts environment variables to uppercase and should allow for case matching. If no entry is found, then a null pointer is returned.

### EXAMPLE (ex048.p)

```

program getenv1(output);
type
  cp = ^char;
  s70 = packed array [1..70] of char;
  s70p = ^s70;
function getenv(str:cp): s70p; external;
var
  str: packed array [1..10] of char;
  p: s70p;
  i: integer;
begin
  str := 'PATH';
  str[5] := chr(0);
  p := getenv(&str[1]);

```

```

writeln('path = ', p^);
str := 'PROMPT';
str[7] := chr(0);
p := getenv(&str[1]);
write('prompt = ');
i := 1;
while p^[i] <> chr(0) do begin
  write(p^[i]);
  i := i + 1;
end;
writeln;
end.

```

This example illustrates two different techniques to access the result pointed to by the `getenv` function. This program generates the following output:

```

DOS:
path = C:;C:\;;C:\DOS;C:\PHAR;C:\EPSILON;C:\BATCH;C:\NORTON;C:\MSC\BIN
prompt = $p$g

UNIX:
path = ./ndp/bin/sysv_0:/usr/bin:/bin:
prompt =

```

## gettim

get time

### Definition

```

procedure gettim(VAR hour, minute, second, hsec: integer);

```

This procedure changes its four integer arguments to return the hour, minute, second, and hundredth of a second.

## hypot

hypotenuse of a right triangle

### Definition

```

function hypot (d1,d2: double): double;

```

The `hypot (d1, d2)` function returns the length of the hypotenuse of a right triangle with sides of length `d1` and `d2`.

### EXAMPLE (ex049.p)

```

program hypot1(output);

function hypot(d1, d2: double): double; external;

begin
  writeln('hypot(3,4) = ', hypot(3.0, 4.0));
  writeln('hypot(5,6) = ', hypot(5.0, 6.0));
  writeln('hypot(6,8) = ', hypot(6.0, 8.0));
  writeln('hypot(7,9) = ', hypot(7.0, 9.0));
end.

```

This program generates the following output:

```

hypot(3, 4) = 5.0000000000000000e+00
hypot(5, 6) = 7.81024967590665400e+00

```

```
hypot(6, 8) = 1.0000000000000000e+01
hypot(6, 9) = 1.14017542509913792e+01
```

## idate\_

return date in integer format

### Definition

```
procedure idate_ (VAR month, day, year: integer); external;
```

where *month*, *day*, and *year* are integer variables.

The `idate_` procedure returns the current date known to DOS or UNIX in three integer variables. Each value returned contained at most two digits.

### EXAMPLE (ex050.p)

```
program idate1 (output);
procedure idate_ (var month, day, year: integer); external;
var
  month, day, year: integer;
begin
  idate_(month, day, year);
  writeln(' month = ', month);
  writeln(' day   = ', day);
  writeln(' year  = ', year);
end.
```

This program generates the following output:

```
month =      5
day   =      8
year  =     89
```

## index

index of a character in a string

### Definition

```
type cp = ^char;
function index (str:cp; ch:char):cp;
```

where

*str* is a pointer to a null terminated character string.  
*ch* is the character to match.

The `index(str, ch)` function returns a pointer to the first instance of the character *ch* in the string pointed to by *str*. If *ch* is not found, then `index` returns a null pointer.

### EXAMPLE (ex051.p)

```
program index1(output);
type
  cp = ^char;
  s10 = packed array [1..10] of char;
  s10p = ^s10;
function index(str:cp; ch:char): s10p; external;
```

```

var
  str: s10;
  p: s10p;
begin
  str := 'invisible';
  str[10] := chr(0);
  p := index(&str[1], 'v');
  writeln('The initial string = invisible');
  writeln('The substring beginning with v = ', p^);
  p := index (&str[1], 'w');
  writeln('The substring beginning with w = ', p^);
end.

```

This program generates the following output:

```

The initial string = invisible
The substring beginning with v =   visible
The substring beginning with w =   (null)

```

## int\*

Generate a software interrupt

### Definition

```

type dwordregs = packed record eax,ebx,ecx,edx,esi,edi,eflag: integer; end;
type segregs = record es,cs,ss,ds: short; end;
type regs16 = record ax,bx,cx,dx,si,di,cflag: short; end;
procedure int386( interrupt: integer; var inregs, outregs: dwordregs);
  external;
procedure int386x( interrupt: integer; var inregs, outregs: dwordregs; var
  sregs: segregs); external;
function int86( interrupt: integer; var inregs, outregs: regs16): integer;
  external;
function int86x( interrupt: integer; var inregs, outregs: regs16; var sregs:
  segregs): integer; external;
function intdos( var inregs, outregs: regs16): integer; external;
function intdosx( var inregs, outregs: regs16; var sregs: segregs): integer;
  external;

```

where

*interrupt* is the number of the interrupt desired;

*inregs* records how the registers should be set just before the interrupt;

*outregs* is set on return to the contents of the registers just after the interrupt;

*sregs* records how the segment registers should be set just before the interrupt, and is set on return to the contents of the segment registers just after the interrupt.

These functions generate software interrupts. *interrupt* is the number of the interrupt to generate (in *intdos* and *intdosx*, the interrupt generated is always 21h). *inregs* is a pointer to a structure containing the values to be placed in the registers before the interrupt is issued. *outregs* is a pointer to a structure that, on return, will contain the values of the registers immediately following the interrupt.

*int386x*, *int86x*, and *intdosx* also require the argument *sregs*, which points to a structure containing the segment register values to be set before the interrupt (the user can set DS and ES). On return, this structure will contain the segment register values immediately following the interrupt.



`int386` and `int386x` differ from the others in that they expect 32-bit register structures, while the others expect packed 16-bit register structures like Microsoft C.

To use these functions, you must understand the details of the interrupt you wish to execute. Failure to adhere to the guidelines of the system documentation will produce unpredictable results. Consult a technical manual for further details.

The functions return the value left in the `eax` register by the interrupt service routine. The procedures have no return value.

## isalnum

Is a character alphanumeric?

### Definition

```
function isalnum (c: integer): boolean;
```

This boolean function ascertains whether its character argument, represented as an integer (i.e., 65 for A, etc.), is alphanumeric.

## isalpha

Is a character alphabetic?

### Definition

```
function isalpha (c: integer): boolean;
```

This boolean function ascertains whether its character argument, represented as an integer (i.e., 65 for A, etc.), is alphabetic.

## iscntrl

Is a character a control character?

### Definition

```
function iscntrl (c: integer): boolean;
```

This boolean function ascertains whether its character argument, represented as an integer (i.e., 65 for A, etc.), is a control character.

## isdigit

Is a character a digit?

### Definition

```
function isdigit (c: integer): boolean;
```

This boolean function ascertains whether its character argument, represented as an integer (i.e., 65 for A, etc.), is a digit. Note that `isdigit(9)` returns false, while `isdigit(ord('9'))` returns true.

## isgraph

Is a character graphical?

### Definition

```
function isgraph (c: integer): boolean;
```

This boolean function ascertains whether its character argument, represented as an integer (i.e., 65 for A, etc.), is a graphics character (letter, numeral, or punctuation).

## isinf

Is the argument infinity?

### Definition

```
function isinf (d: double): boolean;
```

This boolean function ascertains whether its argument (double) is infinity.

## islower

Is a character a lowercase letter?

### Definition

```
function islower (c: integer): boolean;
```

This boolean function ascertains whether its character argument, represented as an integer (i.e., 65 for A, etc.), is a lower case letter.

## isnan

Is the argument Not A Number?

### Definition

```
function isnan (d: double): boolean;
```

This boolean function ascertains whether its argument (double) is a NAN (not a number).

## isprint

Is a character printable?

### Definition

```
function isprint (c: integer): boolean;
```

This boolean function ascertains whether its character argument, represented as an integer (i.e., 65 for A, etc.), is a printable character.

## ispunct

Is a character a punctuation mark?

### Definition

```
function ispunct (c: integer): boolean;
```

This boolean function ascertains whether its character argument, represented as an integer (i.e., 65 for A, etc.), is a punctuation mark.

## isspace

Is a character a space?

### Definition

```
function isspace (c: integer): boolean;
```

This boolean function ascertains whether its character argument, represented as an integer (i.e., 65 for A, etc.), is a space.

## isupper

Is a character upper case?

### Definition

```
function isupper (c: integer): boolean;
```

This boolean function ascertains whether its character argument, represented as an integer (i.e., 65 for A, etc.), is an upper case letter.

## isdigit

Is a character a hexadecimal digit?

### Definition

```
function isdigit (c: integer): boolean;
```

This boolean function ascertains whether its character argument, represented as an integer (i.e., 65 for A, etc.), is a hexadecimal digit.

## j0

Bessel function of the first kind, order 0.

### Definition

```
function j0 (d: double): double;
```

where  $d$  is an expression of type DOUBLE.

The  $j_0(d)$  function returns the Bessel function of the first kind, order 0, of  $d$ . This corresponds to  $J_0(d)$  in the usual notation. If the input argument is out of range, the global variable `errno` will be set to EDOM.

### EXAMPLE (ex052.p)

```
program j0a (output);
function j0( d: double): double; external;
var
  x: double;
  i: integer;
begin
  x := 0.0;
  for i := 1 to 7 do begin
    writeln(' x = ', x:6:2, ' j0(x) = ', j0(x):20:14);
    x := x + 2.5;
  end;
end.
```

This program generates the following output:

```
x = 0.00 j0(x) = 1.000000000000000
x = 2.50 j0(x) = -0.04838377646820
x = 5.00 j0(x) = -0.17759677131434
x = 7.50 j0(x) = 0.26633965788038
x = 10.00 j0(x) = -0.24593576445135
```

```
x = 12.50  j0(x) =  0.14688405470042
x = 15.00  j0(x) = -0.01422447282678
```

## j1

Bessel function of the first kind, order 1.

### Definition

```
function j1 (d: double): double;
```

where  $d$  is an expression of type DOUBLE.

The  $j_1(d)$  function returns the Bessel function of the first kind, order 1, of  $d$ . This corresponds to  $J_1(d)$  in the usual notation. If the input argument is out of range, the global variable `errno` will be set to EDOM.

### EXAMPLE (ex053.p)

```
program j1a (output);
function j1( d: double): double; external;
var
  x: double;
  i: integer;
begin
  x := 0.0;
  for i := 1 to 7 do begin
    writeln(' x = ', x:6:2, ' j1(x) = ', j1(x):20:14);
    x := x + 2.5;
  end;
end.
```

This program generates the following output:

```
x = 0.00  j1(x) =  0.000000000000000
x = 2.50  j1(x) =  0.49709410246427
x = 5.00  j1(x) = -0.32757913759147
x = 7.50  j1(x) =  0.13524842757971
x = 10.00 j1(x) =  0.04347274616886
x = 12.50 j1(x) = -0.16548380461476
x = 15.00 j1(x) =  0.20510403861352
```

## jn

Bessel function of the first kind, order  $i$ .

### Definition

```
function jn (i: integer; d: double): double;
```

where

$i$  is an expression of type INTEGER,  
 $d$  is an expression of type DOUBLE.

The  $j_n(i, d)$  function returns the Bessel function of the first kind, order  $i$ , of  $d$ . This corresponds to  $J_n(i, d)$  in the usual notation. If the input argument is out of range, the global variable `errno` will be set to EDOM.

### EXAMPLE (ex054.p)

```
program jn1 (output);
```

```

function jn(i: integer; d:double): double; external;
var
  x: double;
  i,j: integer;
begin
  for i := 1 to 3 do begin
    x := 2.50;
    for j := 1 to 3 do begin
      writeln('i = ', i:2, ' x = ', x:6:2, ' jn(i,x) = ', jn(i,x):20:14);
      x := x + 2.5;
    end;
    writeln;
  end;
end.

```

This program generates the following output:

```

i = 1 x = 2.50 jn(i,x) = 0.49709410246427
i = 1 x = 5.00 jn(i,x) = -0.32757913759147
i = 1 x = 7.50 jn(i,x) = 0.13524842757971
i = 2 x = 2.50 jn(i,x) = 0.44605905843962
i = 2 x = 5.00 jn(i,x) = 0.04656511627775
i = 2 x = 7.50 jn(i,x) = -0.23027341052579
i = 3 x = 2.50 jn(i,x) = 0.21660039103911
i = 3 x = 5.00 jn(i,x) = 0.36483123061367
i = 3 x = 7.50 jn(i,x) = -0.25806091319346

```

## labs

returns the long absolute value

### Definition:

```
function labs ( i : integer ) : integer;
```

where *i* is an expression of type integer.

The labs function takes and returns an integer. The result is the absolute value of the input parameter.

## ldexp

load exponent

### Definition

```
function ldexp (d: double; i: integer): double;
```

where

*d* is an expression of type DOUBLE.

*i* is an expression of type INTEGER.

The ldexp(*d*, *i*) function calculates the value of  $d \cdot 2^i$ . If the input value is out of range, the global variable *errno* will be set to EDOM. On overflow *errno* is set to ERANGE.

### EXAMPLE (ex055.p)

```

program ldexp1(output);
function ldexp( d:double; i:integer): double; external;

```

```

var
  i: integer;
begin
  writeln('  i  ldexp(2,i)  ldexp(3,i)');
  writeln(' - -----');
  for i := 1 to 5 do
    writeln(i:4, ldexp(2.0,i):12:6, ldexp(3.0,i):12:6);
  end.

```

This program generates the following output:

```

i ldexp(2,i) ldexp(3,i)
- -----
1  4.000000  6.000000
2  8.000000 12.000000
3 16.000000 24.000000
4 32.000000 48.000000
5 64.000000 96.000000

```

## ldexpf

load single precision exponent

### Definition

```
function ldexpf (f: float; i: integer): float;
```

where

$f$  is an expression of type FLOAT,  
 $i$  is an expression of type INTEGER.

The `ldexpf( $f$ ,  $i$ )` function calculates the value of  $f \cdot 2^i$ . On overflow the global variable `errno` is set to `ERANGE`. `errno` is not set for domain errors.

## log

returns the double precision natural logarithm

### Definition

```
function log ( d : double ): double;
```

where  $d$  is an expression of type double.

The LOG function converts the input parameter to a temporary 64-bit floating point number and returns the natural logarithm of this value. If the argument is not positive, the return value is zero and the global variable `errno` is set to `EDOM`.

## log10

base 10 logarithm

### Definition

```
function log10 ( d: double): double;
```

where  $d$  is an expression of type DOUBLE.

The `log10( $d$ )` function returns the base 10 logarithm of  $d$ . If  $d$  is not positive, the return value is 0 and the global variable `errno` is set to `EDOM`.

**EXAMPLE (ex056.p)**

```

program log10a(output);
function log10(d: double): double; external;
var
  x: double;
  i: integer;
begin
  x := 1.1;
  for i := 1 to 5 do begin
    writeln(' x = ', x:14, ' log10(x) = ', log10(x):16);
    x := x * 100.0;
  end;
end.

```

This program generates the following output:

```

x = 1.1000000e+00  log10(x) = 4.139268516e-02
x = 1.1000000e+02  log10(x) = 2.041392685e+00
x = 1.1000000e+04  log10(x) = 4.041392685e+00
x = 1.1000000e+06  log10(x) = 6.041392685e+00
x = 1.1000000e+08  log10(x) = 8.041392685e+00

```

## log10f

base 10 logarithm

**Definition**

```
function log10f (f: float): float;
```

where  $f$  is an expression of type FLOAT.

The  $\text{log10f}(f)$  function returns the base 10 logarithm of  $f$ . If  $f$  is not positive, the return value is 0 and the global variable `errno` is set to EDOM.

## logf

returns the single precision natural logarithm

**Definition**

```
function logf ( f : float ): float;
```

where  $f$  is an expression of type float.

The LOGF function converts the input parameter to a temporary 64-bit floating point number and returns the natural logarithm of this value. If the argument is not positive, the return value is zero and the global variable `errno` is set to EDOM.

## mapdev (DOS only)

map physical memory to program's data segment

**Definition**

```

type
  n   = user_specified;
  sn  = array [0..n] of char;
  sp  = ^sn;

function mapdev (address, nBytes: integer): sp; external;

```

where

*n* is an integer representing the upper dimension of the zero originated array type *sn*.

*sn* is a type definition for an array of characters. The size of this array is specified by the parameter *n*. Note that this array definition is zero originated.

*sp* is a pointer type to an array of characters.

*address* is the physical address of the device.

*nBytes* is the size of the device in bytes.

The `mapdev(address, nBytes)` function returns a pointer to a physical device that can be used by the program. The device is mapped into virtual memory and does not occupy any usable memory. The value returned is not related to the physical address in any obvious way. The program treats the device as an array of characters and accesses physical locations in the device by reading or writing to the character array.

`mapdev` returns a null pointer to indicate an error. However, a non-null value does not guarantee that the mapped device or memory is present. Therefore, testing for presence of the device is recommended if there is any doubt.

#### EXAMPLE (ex057.p)

```

program mapdev1 (output);
{ Interface to mapdev function in C library. }
const
    monochrome = 0xb0000;
    normal = 0x07;
    under = 0x01;
    reverse = 0x70;

type
    m2048 = array [0..2047] of record data, attr: char; end;
    m2048p = ^m2048;

function mapdev (address, nbytes: integer): m2048p; external;
procedure box (mem:m2048p; sym: char; attr, srow, scol, erow, ecol: integer);
{ Fill in region of screen from start_row to end_row, start_col to end_col
  with the character sym.
  Legal values: srow and erow from 0 to 23, scol and ecol from 0 to 79.
}
var row, col, i:integer;
begin
    for col := scol to ecol do
        for row := srow to erow do begin
            i := row * 80 + col;
            mem^[i].data := sym;
            mem^[i].attr := chr(attr);
        end;
    end;

procedure clearmem (mem: m2048p);
{ Erase the screen by filling it with blanks. }
var i: integer;
begin
    for i:= 0 to 79*25 do begin
        mem^[i].data := ' ';
        mem^[i].attr := chr (normal);
    end;
end;

```



```

var
  videomem: m2048p;

begin
  videomem := mapdev (monochrome, 4096);
  clearmem (videomem);
  box (videomem, 'a', normal, 4, 0, 4+2, 0+4);
  box (videomem, 'b', under, 4, 10, 4+3, 10+6);
  box (videomem, 'c', reverse, 4, 23, 4+4, 23+8);
end.

```

This example uses the `mapdev` function to access video memory directly on a machine equipped with a monochrome adapter. The program clears the screen and then generates the following output:

(normal)	(underlined)	(inverse)
aaaaa	bbbbbbb	cccccccc
aaaaa	bbbbbbb	cccccccc
aaaaa	bbbbbbb	cccccccc
	bbbbbbb	cccccccc
		cccccccc

## memchr

Locate a character in an object

### Definition

```

type cp = ^char;
function memchr(s: cp; c, n: integer): cp;

```

where

*s* is a pointer to the block of memory to be searched;  
*c* is an integer representation of the character to be sought;  
*n* is the length of the memory block.

`memchr` finds the first occurrence of a character in a block of memory. It returns a pointer to the character if found and `NIL` otherwise.

## memcmp

Compare two memory buffers

### Definition

```

type cp: ^char;
function memcmp(s1, s2: cp; n: integer);

```

where

*s1* and *s2* are pointers to the blocks of memory to be compared;  
*n* is the number of bytes to be compared.

`memcmp` compares two blocks of memory pointed to by its first and second arguments. The number of bytes to be compared is given in its third argument. If the first block is lexically prior to the second, the function returns a negative integer. If the blocks are the same, the function returns zero. If the second block is lexically prior to the first argument, the function returns a positive integer.

## memcpy

Copy characters from one buffer to another

### Definition

```
type cp = ^char;
function memcpy(s1, s2: cp; n: integer): cp;
```

where

*s1* and *s2* are pointers to the destination and source buffers, respectively;  
*n* is the number of bytes to be copied.

`memcpy` copies the number of bytes specified in its third argument from the buffer pointed to by its second argument to the buffer pointed to by its first argument. The return value is a pointer to the first buffer.

## memmove

Copy characters from one buffer to another, checking for overlap

### Definition

```
type cp = ^char;
function memmove(s1, s2: cp; n: integer): cp;
```

where

*s1* and *s2* are pointers to the destination and source buffers, respectively;  
*n* is an expression of type `INTEGER` representing the number of bytes to be copied.

`memmove` is the same as `memcpy` except that it transfers bytes in reverse order as a result of which the move will be correct even if the two buffers overlap.

## memset

Fill an object with a character

### Definition

```
type cp = ^char;
function memset(s: cp; c, n: integer): cp;
```

where

*s* points to the memory to be filled;  
*c* represents the character with which the memory is to be filled;  
*n* is the number of bytes to fill.

`memset` sets the number of bytes specified in its third argument of the block of memory pointed to by its first argument to the value stored in its second argument. The return value is the same as the first argument.

## mktemp

Make temporary name

### Definition

```
type cp = ^char;
function mktemp(str: cp): cp;
```

where *str* is the prefix of the name to be made.

`mktemp` returns a unique file name for use as a temporary file. It takes a string argument that contains the desired file name prefix and six extra characters for internal use.

## mktime

Convert broken-down time to calendar time

### Definition

```
type tm = record sec,min,hour,mday,mon,year,wday,yday,isdst: integer; end;
function mktime(timeptr: tm): integer
```

where

*sec* is seconds in the minute (0-59),  
*min* is minutes in the hour (0-59),  
*hour* is hour of the day (0-23),  
*mday* is day of the month (1-31),  
*mon* is months since January (0-11),  
*year* is years since 1990,  
*wday* is days since Sunday (0-6),  
*yday* is days since January 1 (0-365),  
*isdst* is whether it is Daylight Savings Time.

`mktime` returns an unsigned integer that can be used in other timing functions.

## modf

split a 64-bit number into integer and fractional parts

### Definition

```
type dp = ^double;
function modf (d: double; dptr: dp): double;
```

where

*d* is the 64-bit value that will be decomposed,  
*dptr* is a pointer to the integral part of *d* returned by `modf`.

The `modf(d, dptr)` function splits the value *d* into its integer and fractional parts, each of which has the same sign as *d*. The fractional part is returned in the `modf` function name. The integral part is stored in the object pointed to by *dptr*.

### EXAMPLE (ex058.p)

```
program modf1(output);
type
  dp = ^double;
function modf( value: double; iptr: dp): double; external;
var
  x: double;
  integral: dp;
  fractional: double;
begin
  x := -123.4567;
  new(integral);
  fractional := modf(x, integral);
  writeln('x = ', x:14:6);
  writeln('integral part of x = ', integral^:14:6);
```

```
writeln('fractional part of x = ', fractional:14:6);
end.
```

This program generates the following output:

```
x = -123.456700
integral part of x = -123.000000
fractional part of x = -0.456700
```

## perror

Print error message

### Definition

```
type cp = ^char;
procedure perror(s: cp);
```

where *s* is the beginning of the text to be printed.

`perror` sends to the standard error device its string argument followed by a colon and the text associated with the current setting of the global variable `errno`.

## pow

*x* raised to the power *y*

### Definition

```
function pow (d1,d2: double): double;
```

where *d1* and *d2* are expressions of type `DOUBLE`.

The `pow(d1,d2)` function returns the value of *d1* raised to the power *d2*. If the return value is out of range, the global variable `errno` will be set to `ERANGE`. If at least one of the arguments is out of range, `errno` will be set to `EDOM`. In the case of undefined results, `pow` returns 0 but `errno` is not set.

### EXAMPLE (ex059.p)

```
program pow1(output);
function pow(d1, d2: double): double; external;
var
  i: integer;
begin
  writeln('      i      2**i      3**i');
  writeln('      -      ----      ----');
  for i := 1 to 10 do
    writeln(i,pow(2.0,i):9:1,pow(3.0,i):9:1);
  end.
```

This program generates the following output:

```

i      2**i      3**i
-      ----      ----
1      2.0      3.0
2      4.0      9.0
3      8.0      27.0
4     16.0     81.0
5     32.0    243.0
6     64.0    729.0
7    128.0   2187.0
```

```

8   256.0   6561.0
9   512.0  19683.0
10  1024.0  59049.0

```

## powf

x raised to the power y

### Definition

```
function powf (f1, f2: float): float;
```

where *f1* and *f2* are expressions of type FLOAT.

The `powf (f1, f2)` function returns the value of *f1* raised to the power *f2*. If at least one of the arguments is out of range, the global variable `errno` will be set to `EDOM`. If the return value is out of range, `errno` is not set. In the case of undefined results, `powf` returns 0 but `errno` is not set.

## racos

single precision arc cosine

### Definition

```
function racos (f: float): float;
```

where *f* is an expression of type FLOAT whose value is between -1 and 1.

The `racos (f)` function returns the principal value of the arc cosine of *f*. `acos` takes an argument in the range -1 to 1 and returns a result in the range 0 to pi, expressed in radians. If the input argument is outside of -1 to 1, then `racos` returns 0.

### EXAMPLE (ex060.p)

```

program racos1(output);
function racos( f:float):
  float; external;
begin
  writeln('racos (0) = ', racos(0.0));
  writeln('racos (-1) = ', racos(-1.0));
  writeln('racos (1) = ', racos(1.0));
  writeln('racos (2) = ', racos(2.0));
  writeln('racos (-3) = ', racos(-3.0));
end.

```

This program generates the following output:

```

racos ( 0) = 1.5707964e+00
racos (-1) = 3.1415927e+00
racos ( 1) = 0.0000000e+00
racos ( 2) = 0.0000000e+00
racos (-3) = 0.0000000e+00

```

## raise

Send a signal

### Definition

```
function raise(sig: integer): integer;
```

where *sig* is an expression of type INTEGER.

raise is used in conjunction with a prior call to the UNIX function `signal` and will return -1 if the argument is out of range. If the argument is one of the following, it will call `_exit`:

SIGABRT	SIGHUP	SIGQUIT	SIGURG
SIGALRM	SIGILL	SIGSEGV	SIGVTALRM
SIGBUS	SIGINT	SIGSYS	SIGXCPU
SIGEMT	SIGKILL	SIGTERM	SIGXFSZ
SIGFPE	SIGPIPE	SIGTRAP	

If the argument is one of the following, it will return zero:

SIGCHLD	SIGPROF	SIGTTIN	SIGUSR2
SIGCONT	SIGSTOP	SIGTTOU	SIGWINCH
SIGIO	SIGTSTP	SIGUSR1	

## rand

random integer

### Definition

```
function rand: integer;
```

The `rand` function returns a pseudo-random integer between 0 and 2147483647 (MAX-INT). The `rand` function seeds itself unless the `srand` function is used.

### EXAMPLE (ex061.p)

```
program rand1(output);
function rand: integer; external;
var
  i: integer;
begin
  for i:=1 to 5 do
    writeln(rand);
  end.
```

This program generates random output.

## rasin

single precision arc sine

### Definition

```
function rasin (f: float): float;
```

where  $f$  is an expression of type `FLOAT` whose value is between -1 and 1.

The `rasin( $f$ )` function returns the principal value of the arc cosine of  $f$ . `rasin` takes an argument in the range -1 to 1 and returns a result in the range 0 to pi, expressed in radians. If the input argument is outside of -1 to 1, then `rasin` returns 0.

### EXAMPLE (ex062.p)

```
program rasin1(output);
function rasin(f: float): float; external;
begin
  writeln('rasin(0) = ', rasin(0.0));
  writeln('rasin(-1) = ', rasin(-1.0));
  writeln('rasin(1) = ', rasin(1.0));
```

```
writeln('rasin(2) = ', rasin(2.0));
writeln('rasin(-3) = ', rasin(-3.0));
end.
```

This program generates the following output:

```
rasin ( 0) = 0.0000000e+00
rasin (-1) = -1.5707964e+00
rasin ( 1) = 1.5707964e+00
rasin ( 2) = 0.0000000e+00
rasin (-3) = 0.0000000e+00
```

## ratan

single precision arc tangent

### Definition

```
function ratan (f: float): float;
```

where  $f$  is an expression of type `FLOAT` whose value is between -1 and 1.

The `ratan( $f$ )` function returns the principal value of the arc tangent of  $f$ . `ratan` takes an argument in the range -1 to 1 and returns a result in the range 0 to pi, expressed in radians. If the input argument is outside of -1 to 1, then `ratan` returns 0.

### EXAMPLE (ex063.p)

```
program ratan1(output);
function ratan(f: float): float; external;
begin
  writeln('ratan(0) = ', ratan(0.0));
  writeln('ratan(-1) = ', ratan(-1.0));
  writeln('ratan(1) = ', ratan(1.0));
  writeln('ratan(2) = ', ratan(2.0));
  writeln('ratan(-3) = ', ratan(-3.0));
end.
```

This program generates the following output:

```
ratan ( 0) = 0.0000000e+00
ratan (-1) = -7.8539824e-01
ratan ( 1) = 7.8539824e-01
ratan ( 2) = 1.1071488e-01
ratan (-3) = -1.2490458e+00
```

## ratan2

single precision arc tangent of a quotient

### Definition

```
function ratan2 (f1, f2: float): float;
```

where  $f1$  is an expression of type `FLOAT`, and  $f2$  is an expression of type `FLOAT` that has a nonzero value.

The `atan2( $f1$ ,  $f2$ )` function returns the principal value of the arc tangent of  $f1/f2$ . `atan2` returns a value between  $-\pi$  and  $\pi$ , expressed in radians. The signs of both arguments are used to determine the quadrant of the result. If the second argument is zero, then `atan2` returns 0 and the global variable `errno` is set to `EDOM` (which is represented by a zero).

**EXAMPLE (ex064.p)**

```

program ratanx(output);
function ratan2(f, g: float): float; external;
begin
  writeln('ratan2 (0,1) = ', ratan2( 0.0, 1.0));
  writeln('ratan2 (-1,1) = ', ratan2( -1.0, 1.0));
  writeln('ratan2 (1,1) = ', ratan2( 1.0, 1.0));
  writeln('ratan2 (20,0.1) = ', ratan2( 20.0, 0.1));
end.

```

This program generates the following output:

```

ratan2 ( 0,1) = 0.0000000e+00
ratan2 (-1,1) = -7.8539824e-01
ratan2 ( 1,1) = 7.8539824e-01
ratan2 (20,0.1) = 1.5657964e+00

```

**rcos**

single precision cosine

**Definition**

```
function rcos (f: float): float;
```

where  $f$  is an expression of type FLOAT.

The  $\text{rcos}(f)$  function returns the cosine of  $f$ , where  $f$  is expressed in radians.

**EXAMPLE (ex065.p)**

```

program rcos1(output);
function rcos(f: float): float; external;
const
  pi = 3.14159265358979323846;
var
  i: integer;
  x: float;
begin
  x := 0.0;
  for i:= 1 to 5 do begin
    writeln(' x = ', x, " rcos(x) = ", rcos(x));
    x := x + pi/4.0;
  end;
end.

```

This program generates the following output:

```

x = 0.0000000e+00  rcos(x) = 1.0000000e+00
x = 7.8539819e-01  rcos(x) = 7.0710683e-01
x = 1.5707964e+00  rcos(x) = 0.0000000e+00
x = 2.3561945e+00  rcos(x) = -7.0710683e-01
x = 3.1415927e+00  rcos(x) = -1.0000000e+00

```



## rcosh

single precision hyperbolic cosine

### Definition

```
function rcosh (f: float): float;
```

where  $f$  is an expression of type FLOAT.

The `rcosh(f)` function returns the hyperbolic cosine of  $f$ .

### EXAMPLE (ex066.p)

```
program rcosh1(output);
function rcosh(f: float): float; external;
const
  pi = 3.14159265358979323846;
var
  i: integer;
  x: float;
begin
  x := 0.0;
  for i:= 1 to 5 do begin
    writeln(' x = ', x, ' rcosh(x) = ', rcosh(x));
    x := x + pi/4.0;
  end;
end.
```

This program generates the following output:

```
x = 0.0000000e+00 rcosh(x) = 1.0000000e+00
x = 7.8539819e-01 rcosh(x) = 1.3246090e+00
x = 1.5707964e+00 rcosh(x) = 2.5091784e+00
x = 2.3561945e+00 rcosh(x) = 5.3227520e+00
x = 3.1415927e+00 rcosh(x) = 1.1591953e+01
```

## remove

Delete a file

### Definition

```
type cp = ^char;
function (filename: cp): integer;
```

where *filename* is the name of the file to be deleted.

`remove` deletes the file named in its string argument. It returns 0 if successful and -1 otherwise. In the case of failure, the global variable `errno` will be set either to `ENOENT`, indicating the file could not be found, or to `EACCESS`, indicating the file could not be deleted. Under UNIX the function calls `unlink`, the return value is that of `unlink`.

## rename

Rename a file

### Definition

```
type cp = ^char;
function rename(old, new: cp): integer;
```

where *old* is the original name of the file, and *new* is its new name.

This integer function renames the file named in its first string argument to the name given in its second string argument. It returns 0 if successful and -1 otherwise. In the case of failure, the global variable `errno` will be set either to `ENOENT`, indicating the file could not be found, or `EACCESS`, indicating the file could not be renamed.

## rexp

single-precision exponential function

### Definition

```
function rexp (f: float): float;
```

where *f* is a single-precision float. `rexp` returns  $e^f$ .

## rfrexp

compute single-precision exponent and mantissa

### Definition

```
function rfrexp (f: float, VAR exptr: integer): float;
```

where

*f* is a single-precision float.

*exptr* contains, on return, the exponent part of *f*.

The return value is the mantissa of *f*. This is a single precision version of `frexp`.

## rindex

reverse search for character in string

### Definition

```
type cp = ^char;
function rindex (str:cp; ch:char): cp;
```

where

*str* is a pointer to a null terminated string to be searched,

*ch* is the character being searched for.

The `rindex(str, ch)` function returns a pointer to the location of the character *ch* in the null terminated string pointed to by *str*, or a null pointer if no match occurs.

### EXAMPLE (ex067.p)

```
program rindex1(output);
type
  cp = ^char;
  s40 = packed array [1..40] of char;
  s40p = ^s40;
function rindex(str:cp; ch:char): s40p; external;
var
  str: s40;
  ptr: s40p;
procedure printstring(c:char);
begin
```

```

writeln('the last substring beginning with ',c,' = ',ptr^);
if ptr = nil then
  writeln('(',c,' is not present)');
end;

begin
  str := 'A little learning is a dangerous thing.';
  str[40] := chr(0);
  writeln('The initial string = ',str);
  writeln;
  ptr := rindex(&str[1], 'l'); printstring('l');
  ptr := rindex(&str[1], 'd'); printstring('d');
  ptr := rindex(&str[1], 't'); printstring('t');
  ptr := rindex(&str[1], 'x'); printstring('x');
end.

```

This program generates the following output:

```

The initial string = A little learning is a dangerous thing.
The last substring beginning with l = learning is a dangerous thing.
The last substring beginning with d =           dangerous thing.
The last substring beginning with t =           thing.
The last substring beginning with x =           (null)
(x is not present)

```

## rldexp

single-precision multiplication by a power of two

### Definition

```
function rldexp (f: float, exp: integer): float;
```

where

$f$  is a single-precision float.  
 $exp$  is an integer.

The return value is  $f * 2^{exp}$ . This is a single precision version of `ldexp`. The global variable `errno` is not set for domain errors for this function.

## rlog

single-precision natural logarithm

### Definition

```
function rlog (f: float): float;
```

where  $f$  is a single-precision float.

`rlog` returns  $\ln(f)$ . It is a single precision version of `ln`.

## rlog10

single-precision common logarithm

### Definition

```
function rlog10 (f: float): float;
```

where  $f$  is a single-precision float.

`rlog10` returns the common (base 10) logarithm of  $f$ . It is a single precision version of `log10`.

## rpow

single-precision power function

### Definition

```
function rpow (f: float): float;
```

where  $f$  is a single-precision float.

This is a single precision version of `pow`. If the return value is out of range, the global variable `errno` is not set.

## rsin

single precision sine

### Definition

```
function rsin (f: float): float;
```

where  $f$  is an expression of type `FLOAT`.

The `rsin(f)` function returns the sine of  $f$ , where  $f$  is expressed in radians.

### EXAMPLE (ex068.p)

```

program rsin1(output);
function rsin(f: float): float; external;
const
  pi = 3.14159265358979323846;
var
  i: integer;
  x: float;
begin
  x := 0.0;
  for i:= 1 to 5 do begin
    writeln(' x = ', x, ' rsin(x) = ', rsin(x));
    x := x + pi/4.0;
  end;
end.

```

This program generates the following output:

```

x = 0.0000000e+00  rsin(x) = 0.0000000e+00
x = 7.8539819e-01  rsin(x) = 7.0710683e-01
x = 1.5707964e+00  rsin(x) = 1.0000000e+00
x = 2.3561945e+00  rsin(x) = 7.0710683e-01
x = 3.1415927e+00  rsin(x) = 0.0000000e+00

```

## rsinh

single precision hyperbolic sine Definition

```
function rsinh (f: float): float;
```

where  $f$  is an expression of type `FLOAT`.

The `rsinh(f)` function returns the hyperbolic sine of  $f$ .

### EXAMPLE (ex069.p)

```

program rsinh1(output);

```

```

function rsinh(f: float): float; external;

const
  pi = 3.14159265358979323846;

var
  i: integer;
  x: float;

begin
  x := 0.0;
  for i:= 1 to 5 do begin
    writeln(' x = ', x, '  rsinh(x) = ', rsinh(x));
    x := x + pi/4.0;
  end;
end.

```

This program generates the following output:

```

x = 0.0000000e+00  rsinh(x) = 0.0000000e+00
x = 7.8539819e-01  rsinh(x) = 8.6867094e-01
x = 1.5707964e+00  rsinh(x) = 2.3012989e+00
x = 2.3561945e+00  rsinh(x) = 5.2279720e+00
x = 3.1415927e+00  rsinh(x) = 1.1548738e+01

```

## rsqrt

single-precision square root

### Definition

```
function rsqrt (f: float): float;
```

where  $f$  is a single-precision float.

This is a single precision version of sqrt.

## rtan

single precision tangent Definition

```
function rtan (f: float): float;
```

where  $f$  is an expression of type FLOAT.

The  $rtan(f)$  function returns the principal value of the arc tangent of  $f$ .  $rtan$  returns a result in the range  $-\pi$  to  $\pi$ , expressed in radians. If the result is out of range, the global variable `errno` will be set to `ERANGE`. `errno` is not set if the input argument is out of range.

### EXAMPLE (ex070.p)

```

program rtan1(output);

function rtan(f: float): float; external;

const
  pi = 3.14159265358979323846;

var
  i: integer;
  x: float;

begin
  x := 0.0;
  for i:= 1 to 5 do begin

```

```

        writeln(' x = ', x, ' rtan(x) = ', rtan(x));
        x := x + pi/4.0;
    end;
end.

```

This program generates the following output:

```

x = 0.0000000e+00  rtan(x) = 0.0000000e+00
x = 7.8539819e-01  rtan(x) = 9.9999988e-01
x = 1.5707964e+00  rtan(x) = -2.2877332e+07
x = 2.3561945e+00  rtan(x) = -1.0000001e+00
x = 3.1415927e+00  rtan(x) = 8.7422784e-08

```

## rtanh

single precision hyperbolic tangent

### Definition

```
function rtanh (f: float): float;
```

where  $f$  is an expression of type FLOAT.

The `rtanh(f)` function returns the hyperbolic tangent of  $f$ .

### EXAMPLE (ex071.p)

```

program rtanh1(output);
function rtanh(f: float): float; external;

const
    pi = 3.14159265358979323846;

var
    i: integer;
    x: float;

begin
    x := 0.0;
    for i:= 1 to 5 do begin
        writeln(' x = ', x, ' rtanh(x) = ', rtanh(x));
        x := x + pi/4.0;
    end;
end.

```

This program generates the following output:

```

x = 0.0000000e+00  rtanh(x) = 0.0000000e+00
x = 7.8539819e-01  rtanh(x) = 6.5579420e-01
x = 1.5707964e+00  rtanh(x) = 9.1715235e-01
x = 2.3561945e+00  rtanh(x) = 9.8219335e-01
x = 3.1415927e+00  rtanh(x) = 9.9627209e-01

```

## sec\_100\_ (DOS only)

return hundredths of a second since midnight

### Definition

```
function sec_100_ : integer; external;
```

The `sec_100_` function returns the number of hundredths of a second since the previous midnight. This function has no arguments.

**EXAMPLE (ex072.p)**

```

program sec100(output);
function sec_100_: integer; external;
var
  hsecs: integer;
begin
  hsecs := sec_100_;
  writeln('The number of hundredths of a second since midnight = ',hsecs);
end.

```

This program generates the following output:

The number of hundredths of a second since midnight =

## secnds\_

return number of seconds from an origin

**Definition**

```
function secnds_ (VAR lastTime: integer): integer; external;
```

where *lastTime* is a variable of type INTEGER.

The *secnds\_* function returns the numbers of seconds since the previous midnight, less the value of *lastTime*. A day's worth of seconds are added to the result if the calculated value is less than the value of *lastTime*.

**EXAMPLE (ex073.p)**

```

program secnds1(output);
function secnds_ (var lasttime: integer): integer; external;
var
  ten_am, ten_pm, thistime, lasttime: integer;
begin
  lasttime := 0;
  thistime := secnds_(lasttime);
  writeln('Seconds since midnight = ', thistime);

  ten_am := 10 * 60 * 60; { seconds since 10 AM }
  ten_pm := 22 * 60 * 60; { seconds since 10 PM }
  writeln('Seconds since yesterday at 10 am = ', secnds_(ten_am));
  writeln('Seconds since yesterday at 10 pm = ', secnds_(ten_pm));
end.

```

This program generates the following output:

```

Seconds since midnight =
Seconds since yesterday at 10 am =
Seconds since yesterday at 10 pm =

```

## setndpsw

set ndp status word

**Definition**

```
procedure (sw: integer);
```

where *sw* is an integer.

This procedure takes an integer as its argument, and sets the NDP Status Word, simulating exceptions.

## sinf

single-precision sine

### Definition

```
function sinf (f: float): float;
```

where *f* is a single-precision float.

This is a single precision version of `sin`.

## sinh

hyperbolic sine

### Definition

```
function sinh (d: double): double;
```

where *d* is an expression of type `DOUBLE`.

The `sinh(d)` function returns the hyperbolic sine of *d*. If the result is too large, the global variable `errno` is set to `ERANGE` and the function returns `HUGE_VAL`.

### EXAMPLE (ex075.p)

```
program sinh1(output);
function sinh(f: double): double; external;
const
  pi = 3.14159265358979323846;
var
  i: integer;
  x: double;
begin
  x := 0.0;
  for i:= 1 to 5 do begin
    writeln(' x = ', x, ' sinh(x) = ', sinh(x));
    x := x + pi/4.0;
  end;
end.
```

This program generates the following output:

```
x = 0.000000000000000000e+00  sinh(x) = 0.000000000000000000e+00
x = 7.85398163397448286e-01  sinh(x) = 8.68670961486009747e-01
x = 1.57079632679489657e+00  sinh(x) = 2.30129890230729427e+00
x = 2.35619449019234486e+00  sinh(x) = 5.22797192467780292e+00
x = 3.14159265358979311e+00  sinh(x) = 1.15487393572577464e+01
```

## sinhf

single-precision hyperbolic sine

### Definition

```
function sinhf (f: float): float;
```



where  $f$  is a single-precision float.

This is a single precision version of `sinh`.

## sprintf

print formatted output to a string

### Definition

```
type cp = ^char;
function sprintf (str, fmt: cp;
                 arg: (variable length argument list, see text)): integer;
```

where

$str$  is a pointer to a string that is to receive the character data from the argument list  $arg$ ,  
 $fmt$  is a pointer to a null terminated string containing formatting instructions,  
 $arg$  is a variable length argument list representing the data to be transferred to string  $str$ .

The `sprintf(str, fmt, arg)` function reads the data in the variable length argument list  $arg$ , converts it according to the format specification in  $fmt$  and writes the result as a null terminated string to the array  $str$ . `sprintf` returns the number of items written to the output string  $str$ , not counting the terminating null.

A description of the syntax of the descriptors in the format specification is best left to a book on the C programming language, for example, *The C Programming Language* by Brian Kernighan and Dennis Ritchie, published by Prentice Hall, 1988. The explanation given here will necessarily be concise, and not do justice to the full power of this function.

### How the `sprintf` function works

The format string,  $fmt$ , indicates how the data in the argument list is to be interpreted and formatted when copied to the output string  $str$ . The format string may contain text and format specifiers. A format specifier is a character sequence that begins with a percent sign (%) and ends with a single character abbreviation (listed below) for the type of conversion to do. A format specifier is required for each item in the argument list. Any text within the format string is copied to the output string without any format conversion.

The `sprintf` function proceeds in the following manner. The format string is read from left to right. Any text within the format string is immediately copied to the output string. When a format specifier is encountered (identified by the leading percent sign), the next item from the argument list is read and copied to the output string while doing the conversion required by the format specifier. The `sprintf` function then continues with where it left off in the format string. Each format specifier causes the next item in the argument list to be formatted and copied to the output string. This process stops when the end of the format string is encountered.

As a simple example, the format string "The answer is %f" will copy the text "The answer is " to the output string, then read one number of type `double` from the argument list, convert it to an ASCII string, and copy the ASCII representation of the number to the output string.

### Declaring the `sprintf` function

The following shows the declaration of the `sprintf` function when  $n$  different data items are to be formatted and copied to the output string (pointed to by)  $str$ . The format string,  $fmt$ , must contain  $n$  format specifiers compatible with the corresponding data types  $t1$  through  $tn$ .

```
type
  cp = ^char;
  t1 = ..
  t2 = ..
  tn = ..
```

```
function sprintf (str,fmt:cp; d1:t1; d2:t2; . . . ; dn:tn): integer;
```

## The format Specifiers

The format specifier begins with the percent sign, and ends with the format character. The format conversion may be modified by placing additional characters between the percent sign and the format specifier. These modifiers provide additional control over the conversion process. If present, they must be placed between the percent sign and the format conversion character in the following order:

### 1. Flags

- Left justifies the data item in the field.
- + Forces a plus or minus sign before any numeric data. By default, the plus sign is omitted for positive values, and the minus sign is placed before negative values.
- space Places one space before positive values, and the minus sign before negative values. By default, the space is omitted for positive values.
- # The actual result depends upon the conversion character. For `o`, the first digit will be zero; for `x` or `X`, the string "0x" or "0X" will be prefixed to a nonzero value. For `e`, `E`, `f`, `g`, and `G`, a decimal point will be placed in the result. For `g` and `G`, trailing zeros will not be removed.
- 0 Specifies that the padding characters to be used with numeric data is a zero. The default padding character is a space.

### 2. Minimum field width

The converted value will occupy a field in the output string at least this wide, and wider if necessary. The converted value is padded on the left (or right if the `-` flag is present) to fill up the field width. The `0` flag controls the padding character.

### 3. Period

A period is used to separate the minimum field width from the precision. This is only necessary if the precision modifier will be used.

### 4. Precision

For `s`, this is the number of characters to place in the output string; for `e`, `E`, or `f`, this is the number of digits after the decimal point; for `g` or `G`, this is the number of significant digits; for integers, this has the same meaning as the minimum field width.

The following table is a list of format identifiers, or conversion characters:

- |             |   |
|-------------|---|
| d,i,o,u,x,X | Formats data stored in integer form. <code>d</code> and <code>i</code> converts the data into decimal form; <code>o</code> converts the data into octal form; <code>u</code> converts the data into an unsigned number; <code>x</code> and <code>X</code> convert the data into hexadecimal form, <code>x</code> uses lowercase hexadecimal characters while <code>X</code> uses uppercase hexadecimal characters.  |
| e,E         | Formats data stored in floating point form. Data is converted into scientific notation, for example, <code>1.2F34e+56</code> . One digit is always placed before the decimal point. The precision specifies the number of digits after the decimal point, the default is 6. The decimal point is not printed if the precision is zero. The case of the letter <code>e</code> in the exponent matches the case of the format specifier. The exponent contains at least two digits, and then as many digits as necessary. |
| f           | Formats data stored in floating point form. Data is converted into decimal notation, for example <code>12.34</code> .   |

- g,G Causes the e or E format to be used if the exponent of the data item is less than -4, or greater than the precision. Otherwise f format is used. Trailing zeros are not copied to the output string, and a decimal point is used only if it is followed by a digit.
- c Copies a single character to the output string.
- s Copies a string of characters until the null terminator (chr(0)) is found, or until the number of characters copied equals the precision.
- % Copies a percent sign to the output string.

**EXAMPLE 1 (ex076.p)**

```

program sprintf1(output);
type
  cp = ^char;
function sprintf(des, fmt: cp; d:double): integer; external;
var
  des, fmt: packed array [1..40] of char;
  i, n: integer;
  d: double;
begin
  fmt := 'The answer is %f' ;
  fmt[17] := chr(0);
  d := -123.456;
  n := sprintf(&des[1], &fmt[1], d);
  writeln('Format string = ', fmt:20);
  writeln('Number of characters transferred = ', n:4);
  write('The destination string = ');
  for i := 1 to n do
    write(des[i]);
  writeln;
end.

```

This example creates a null terminated string that contains the message "The answer is" followed by the value of a variable. The function `sprintf` obtains this result by copying two strings to the destination array. The first string is the text "The answer is ". The second string is the ASCII representation of the value stored in a variable of type `DOUBLE` (so the format specifier is `f`). This example generates the following output:

```

Format string = The answer is %f
Number of characters transferred = 25
The destination string = The answer is -123.456000

```

**EXAMPLE 2 (ex077.p)**

```

program sprintf2(output);
type
  cp = ^char;
function sprintf (des, fmt: cp; i1, i2, i3: integer): integer; external;
var
  des, fmt: packed array [1..100] of char;
  i, n, x: integer;
begin
  fmt := 'x (in decimal) = %d, (in hex) = %x, (in octal) %o';
  fmt[52] := chr(0);

```

```

x := 0x7fff0000;
n := sprintf(&des[1], &fmt[1], x, x, x);
writeln('Source data = ', x);
writeln('Format string = ', fmt:52);
writeln('Number of characters transferred = ',n:4);
writeln;
writeln('Destination string = ');
for i := 1 to n do
  write(des[i]);
writeln;
end.

```

This example converts a number into its decimal, octal and hexadecimal equivalents, and copies these values into a character array. Brief titles also placed into the array to identify the numbers when printed. This example illustrates the `d`, `x` and `o` format specifiers and generates the following output:

```

Source data = 2147418112
Format string = x (in decimal) = %d, (in hex) = %x, (in octal) = %o
Number of characters transferred = 74
Destination string = x (in decimal) = 2147418112, (in hex) = 7fff0000, (in octal) =
17777600000

```

## sqrtf

single-precision square root

### Definition

```
function sqrtf (f: float): float;
```

where  $f$  is a single-precision float.

This is a single precision version of `sqrt`.

## srand

seed random number generator

### Definition

```
function srand (i: integer): integer;
```

where  $i$  is the value of the seed.

The `srand` function initializes the random number generator used by `rand`.

### EXAMPLE (ex078.p)

```

program srand1(output);

function rand: integer; external;
function srand(i: integer): integer; external;

var
  j, seed: integer;

begin
  seed := 1009;
  srand(seed);
  writeln('seed = ',seed);
  for j:= 1 to 5 do
    writeln(rand);
  writeln;
end.

```

```

seed := 155921;
srand(seed);
writeln('seed = ', seed);
for j:= 1 to 5 do
  writeln(rand);
writeln;
end.

```

This program generates the following output: (This will vary from machine to machine.)

```

seed =      1009
  16152143
 1084389312
  48409366
  515415905
  681807129
seed =      155921
  313258710
  551196409
  263068568
 1703377437
 1715819866

```

## sscanf

read formatted input from a string

### Definition

```

type cp = ^char;
function sscanf (str, fmt: cp;
                arg: {variable length argument list, see text}): integer;

```

where

*str* is a pointer to a null terminated string containing the character data to be read.

*fmt* is a pointer to a null terminated string containing formatting instructions.

*arg* is a variable length argument list; each argument is a pointer to an item that is to receive the converted data from *str*.

The `sscanf(str, fmt, arg)` function reads formatted input from the string *str*, converts it according to the format specification in *fmt* and writes the results to the data items pointed to by the argument list *arg*. `sscanf` returns the number of arguments that were converted and assigned. The return value does not include input characters that were read but not assigned.

The `sscanf` function is nearly the reverse of the `sprintf` function and works in a similar manner. Note that although many of the abbreviations used for the format conversions are identical to those used by the `sprintf` function, they often have slightly different meanings.

A description of the syntax of the descriptors in the format specification is best left to a book on the C programming language, for example, *The C Programming Language* by Brian Kernighan and Dennis Ritchie, published by Prentice Hall, 1988. The explanation given here will necessarily be concise, and not do justice to the full power of this function.

### How the `sscanf` function works

The format string, *fmt*, indicates how the character data in the source string is to be converted. Note that the argument list contains pointers to the variables that are to receive the results of this conversion. The format string contains text and format specifiers. A format specifier is a character sequence that begins with a percent sign (%) and ends with a single character abbreviation (listed below) for the type of conversion to do. A format specifier is required for

each argument in the destination argument list. Characters other than a format specifier or a space must match the characters found in the input string. A space causes leading spaces to be skipped.

The `sscanf` function maintains two pointers to its current location in the format and source strings. Both pointers initially point to the beginning of these strings. The pointers are advanced until the end of the format string is reached, the end of the argument list is reached, or a conflict occurs. The data in the format string is processed as follows:

1. A space causes `sscanf` to skip over any spaces in the source string.
2. Text in the format string is compared against data in the source string. If the character data is identical then it is skipped, otherwise `sscanf` terminates.
3. A format specifier causes `sscanf` to convert the appropriate number of characters from the source string to the specified representation, and place the result at the location indicated by the corresponding argument.

Any format specifier may be preceded by a maximum field width or an assignment suppression character. The field width is a decimal digit that specifies the maximum number of characters to read from the source string for this particular format specifier. The assignment suppression character is an asterisk (\*) that causes a field in the source string to be read but not assigned to any variable in the argument list. The item in the input string is simply skipped. For example, `.*s`, or `.*i`.

As a simple example, the format string "The answer is %f" will read and skip over the characters "The answer is" in the source string, then skip over any blanks until a number is found. This number will be converted and stored in the location pointed to by the corresponding argument.

### Declaring the `sscanf` function

The following shows the declaration of the `sscanf` function when  $n$  different values are to be converted from the input string, `str`, and stored in the locations pointed to by the variables `d1` through `dn`. The format string, `fmt`, must contain  $n$  format specifiers compatible with the corresponding data types to which `t1` through `tn` point.

```

type
  cp = ^char;
  t1 = ^ {some type}
  t2 = ^ {some type}
  tn = ^ {some type}

function sscanf (src,fmt:cp; d1:t1; d2:t2; . . . ; dn:tn): integer;

```

### The Format Specifiers

A format specifier begins with the percent sign, and ends with the format character. It may be modified by the maximum field width or assignment suppression flag.

The following table is a list of format identifiers, or conversion characters:

d,u	Converts a decimal integer. The corresponding data argument must be a pointer to an integer.
i	Converts a decimal integer with an optional prefix. 0x or 0X denote a hexadecimal constant; 0 denotes an octal constant. The corresponding data argument must be a pointer to an integer.
o	Converts an octal integer with or without a leading 0. The corresponding data argument must be a pointer to an integer.
x,X	Converts a hexadecimal integer with or without a leading 0x or 0X. The corresponding data argument must be a pointer to an integer.

e, E, f, g, G	Converts a floating point constant. The input is an optional sign, a string of numbers with an optional decimal point, and an optional exponent field containing an e or E followed by a possibly signed integer.
s	Reads a string of characters until a space is encountered. A null character ( <code>chr(0)</code> ) is appended to the end of the string. The corresponding data argument must be a pointer to an array of type <code>char</code> .
c	Reads a character; does not skip over spaces or a null terminator. The corresponding data argument for <code>%c</code> must be a pointer to type <code>char</code> , for <code>%wc</code> , where <code>w</code> is the field width, the data argument must be a pointer to an array of type <code>char</code> .
n	Writes the number of characters read so far by <code>sscanf</code> to the corresponding argument, which must be a pointer to type <code>INTEGER</code> . No characters are read from the input string with this specifier.
[ <i>char.string</i> ]	Matches the character string within the square brackets to the longest sequence of identical characters from the input string. The matching characters are copied to an array of type <code>char</code> and end with the null character, <code>chr(0)</code> . The corresponding data argument must be a pointer to this array. For example, <code>[xyz]</code> will match with the three strings <code>x</code> , <code>xy</code> or <code>xyz</code> .
[ <i>^char.string</i> ]	Matches any characters not in the character string within the square brackets to the longest sequence of characters from the input string. The matching characters are copied to an array of type <code>char</code> and are terminated with the null character, <code>chr(0)</code> . The corresponding data argument must be a pointer to this array. For example <code>[^xyz]</code> will match with any sequence of characters except <code>x</code> , <code>xy</code> or <code>xyz</code> .
%	Reads past a percent sign in the input string without making any assignment.

**EXAMPLE 1 (ex079.p)**

```

program sscanf1(output);
type
  cp = ^char;
  fp = ^float;
function sscanf(src,fmt: cp; des: fp): integer; external;
var
  src, fmt: packed array [1..40] of char;
  des: float;
  n: integer;
begin
  src := 'The answer is -123.456';
  src[23] := chr(0);
  fmt := 'The answer is %f';
  fmt[17] := chr(0);
  n := sscanf (&src[1], &fmt[1], &des);
  writeln('Source string is = ', src:25);
  writeln('Format string is = ', fmt:19);
  writeln('number of items transferred = ', n:4);
  writeln('Destination = ', des);
end.

```

This example illustrates how a number may be obtained from a character string and converted into a numeric format. The characters preceding the number are discarded, and the number is copied into a variable of type FLOAT. This example produces the following output:

```
Source string is =   The answer is -123.456
Format string is =   The answer is %f
Number of items transferred =   1
Destination = -1.2345600e+02
```

### EXAMPLE 2 (ex080.p)

```
program sscanf2(output);

type
  cp = ^char;
  ip = ^integer;

function sscanf(src, fmt: cp;
               sign: cp; d1:ip; ch:cp; d2:ip): integer; external;

var
  src, fmt: packed array [1..20] of char;
  sign, dot: char;
  d1, d2, n: integer;

begin
  src := '-123.456';
  fmt := '%1s%u%1s%u';
  fmt[11] := chr(0);
  writeln('Source string = ', src:10);
  writeln('Format string = ', fmt:10);
  n := sscanf(&src[1], &fmt[1], &sign, &d1, &dot, &d2);
  writeln('Number of items transferred = ', n:4);
  writeln('      sign = ', sign);
  writeln('  integer part = ', d1:4);
  writeln(' decimal point = ', dot);
  writeln('fractional part = ', d1:4);
end.
```

This example converts a string representing a decimal number into pieces that represent the sign, integer and fractional parts of the number. This example generates the following output:

```
Source string = -123.456
Format string = %1s%u%1s%u
Number of items transferred =   4
      sign = -
  integer part = 123
 decimal point = .
fractional part = 456
```

## strcat

concatenate two strings

Definition

```
type cp = ^char;
function strcat (str1,str2: cp): cp;
```

where *str1* and *str2* are pointers to null terminated strings.



The `strcat (str1, str2)` function copies the null terminated string, `str1`, onto the end of the null terminated string, `str2`. The first character of `str1` replaces the null terminating `str2`. No test is made for overflowing `str2`. `strcat` returns a pointer to the concatenated string.

#### EXAMPLE (ex081.p)

```

program strcat1(output);
type
  cp = ^char;
function strcat(str1, str2: cp): cp; external;
var
  s, t: packed array [1..75] of char;
  p: cp;
  i: integer;
begin
  s := 'Go in and out the window as';
  s[28] := chr(0);
  t := ' you have done before.';
  t[23] := chr(0);
  p := strcat(&s[1], &t[1]);
  writeln('String 1 = ', s:27);
  writeln('String 2 = ', t:22);
  writeln;
  write('Combined strings = ');
  for i:= 1 to 75 do
    write(s[i]);
  writeln;
end.

```

This program generates the following output:

```

String 1 = Go in and out the window as
String 2 =  you have done before.

```

## strchr

index of a character in a string

#### Definition

```

type cp = ^char;
function strchr (str:cp; ch:char): cp;

```

where

`str` is a pointer to the null terminated character string to be searched,  
`ch` is the character to match.

The `strchr (str, ch)` function returns a pointer to the first instance of the character `ch` in the string pointed to by `str`. If `ch` is not found, then `strchr` returns a null pointer.

#### EXAMPLE (ex082.p)

```

program strchr1(output);
type
  cp = ^char;
  s40 = packed array [1..40] of char;
  s40p = ^s40;
function strchr(str:cp; ch:char): s40p; external;

```

```

var
  str: s40;
  ptr: s40p;

procedure printstring(c:char);
begin
  writeln('The last substring beginning with ',c,' = ',ptr^);
  if ptr = nil then
    writeln('(',c,' is not present)');
  end;

begin
  str := 'A little learning is a dangerous thing.';
  str[40] := chr(0);
  writeln('The initial string = ',str);
  writeln;
  ptr := strchr(&str[1], 'l'); printstring('l');
  ptr := strchr(&str[1], 'd'); printstring('d');
  ptr := strchr(&str[1], 't'); printstring('t');
  ptr := strchr(&str[1], 'x'); printstring('x');
  end.

```

This program generates the following output:

```

The initial string = A little learning is a dangerous thing.
The last substring beginning with l = little learning is a dangerous thing.
The last substring beginning with d =                dangerous thing.
The last substring beginning with t = ttle learning is a dangerous thing.
The last substring beginning with x =                (null)
(x is not present)

```

## strcmp

string compare

### Definition

```

type cp = ^char;
function strcmp (str1,str2: cp): integer;

```

where *str1* and *str2* are pointers to null terminated strings.

The `strcmp (str1,str2)` function compares two null terminated strings and returns one of the following values indicating their relationship:

```

if str1 < str2 then return a negative number
if str1 = str2 then return zero
if str1 > str2 then return a positive number

```

### EXAMPLE (ex083.p)

```

program strcmp(output);

const
  s1 = 'economy';
  s2 = 'ecology';
  s3 = 'eclipse';
  s4 = 'eclogue';

type
  cp = ^char;
  s10 = packed array [1..10] of char;

```

```

function strcmp(str1,str2:cp; n:integer): integer; external;
procedure compare(a,b:s10; n:integer);
begin
  a[n+1] := chr(0);
  b[n+1] := chr(0);
  writeln('Comparison on first ', n:2, ' characters = ', strcmp(&a[1],&b[1],n));
end;

begin
  writeln('String 1 = ',s1);
  writeln('String 2 = ',s2);
  compare(s1, s2, 3);
  compare(s1, s2, 4);
  compare(s1, s2, 20);
  writeln('String 3 = ',s3);
  writeln('String 4 = ',s4);
  compare(s3, s4, 3);
  compare(s3, s4, 4);
  compare(s3, s4, 20);
end.

```

This program generates the following output:

```

String 1 = economy
String 2 = ecology
Comparison on first 3 characters =      0
Comparison on first 4 characters =      2
Comparison on first 20 characters =     2

String 3 = eclipse
String 4 = eclogue
Comparison on first 3 characters =      0
Comparison on first 4 characters =     -6
Comparison on first 20 characters =     -6

```

## strcoll

compare two strings based on a program's locale

### Definition

```

type cp = ^char;
function strcoll (str1: cp, str2: cp): integer;

```

where *str1* and *str2* are pointers to null terminated strings.

In the present implementation, this function behaves identically to `strcmp`.

## strcpy

string copy

### Definition

```

type cp = ^char;
function strcpy (str1,str2: cp): integer;

```

where *str1* and *str2* are pointers to null terminated strings.

The `strcpy` function copies the null terminated string *str2* to the address pointed to by *str1*, up to an including the null terminating *str2*.

Caution: It is the programmer's responsibility to ensure that the receiving buffer is large enough for what is written to it. If it is not large enough, adjacent buffers may be overwritten.

**EXAMPLE (ex084.p)**

```

program strcpy1(output);
type
  cp = ^char;
function strcpy(des, src: cp): integer; external;
const
  a = 'The gardener planted the flower in fresh potting soil.';
  b = 'The search continues for new subatomic particles.';
var
  des, src: packed array [1..100] of char;
  i: integer;
begin
  src := a;
  src[55] := chr(0);
  des := b;
  des[50] := chr(0);
  writeln('String 1 = ', a);
  writeln('String 2 = ', b);
  writeln;
  strcpy(&des[26], &src[36]);
  write('Altered string 2 = ');
  i := 1;
  while (des[i] <> chr(0)) do begin
    write(des[i]);
    i := i + 1
  end;
  writeln;
end.

```

This program generates the following output:

```

String 1 = The gardener planted the flower in fresh potting soil.
String 2 = The search continues for new subatomic particles.
Altered string 2 = The search continues for fresh potting soil.

```

## strcspn

compute the length of the initial portion of a string consisting of characters that do not occur in a second string

**Definition**

```

type cp = ^char;
function strcspn (str1: cp, str2: cp): integer;

```

where *str1* and *str2* are null terminated strings.

*strcspn* returns the number of consecutive characters in its first string argument that are not found in its second string argument.

# strerror

convert an error number into an appropriate message

## Definition

```
type cp = ^char;
function strerror (errnum: integer): cp;
```

where *errnum* is an integer error number.

*strerror* returns the error text associated with its argument as found in the array of strings *sys\_errlist*. If the argument is out of range, the string returned says simply that no further information is available.

# strftime

Convert broken-down time to string

## Definition

```
type cp = ^char;
type tm = record sec,min,hour,mday,mon,year,wday,yday,isdst: integer; end;
function strftime(var S: cp; maxsize: integer; Format: cp; timeptr: tm):
  integer;
```

where

*sec* is seconds in the minute (0-59),

*min* is minutes in the hour (0-59),

*hour* is hour of the day (0-23),

*mday* is day of the month (1-31),

*mon* is months since January (0-11),

*year* is years since 1990,

*wday* is days since Sunday (0-6),

*yday* is days since January 1 (0-365),

*isdst* is whether it is Daylight Savings Time,

*S* is the string into which the information is to be placed,

*maxsize* is the maximum number of characters to be placed into *S*,

*Format* is the format controlling what is to go into *S*, consisting of zero or more conversion specifiers and ordinary multibyte characters. A conversion specifier consists of a % character followed by a character that determines its behavior. All ordinary multibyte characters, including the terminating null character, are copied unchanged into *S*. Each conversion specifier is replaced by appropriate characters, as determined by the *LC\_TIME* category of the current locale and by the values contained in the structure pointed to by *timeptr*.

*strftime* places characters into the array pointed to by *S* as controlled by *Format*. The return value is the length of *S*. Following is a list of conversion specifiers allowed.

%a abbreviated weekday name

%A full weekday name

%b abbreviated month name

%B full month name

%c date and time  
 %d day of the month as a decimal number (01 - 31)  
 %H hour (24-hour clock) as a decimal number (00 - 23)  
 %I hour (12-hour clock) as a decimal number (01 - 12)  
 %j day of the year as a decimal number (001 - 366)  
 %m month as a decimal number (01 - 12)  
 %M minute as a decimal number (00 - 59)  
 %p AM/PM designation  
 %S second as a decimal number (00 - 61)  
 %U Week number of the year as a decimal number (00-53), where first Sunday is the first day of week 01  
 %w weekday as a decimal number (0-6) where Sunday is 0  
 %x date  
 %X time  
 %y year without century as a decimal number (00 - 99)  
 %Y year with century.

## strindex

index of a substring within a string

### Definition

```

type cp = ^char;
function strindex (str1, str2: cp): integer;

```

where *str1* and *str2* are pointers to null terminated strings.

The `strindex (str1, str2)` function finds the first occurrence of string *str2* (not including the terminating null character) in the string *str1*. It returns a pointer to the located string in *str1*, or -1 if no match occurs.

### EXAMPLE (ex085.p)

```

program strindex(output);
type
  cp = ^char;
function strindex(str, sub: cp): integer; external;
const
  s1 = 'Twinkle, twinkle little star, how I wonder what you are.';
  s2 = 'Twin';
  s3 = 'ink';
  s4 = 'hat';
  s5 = 'xxx';
var
  str, sub: packed array [1..60] of char;
begin
  str := s1;
  str[57] := chr(0);
  sub := s2;

```

```

str[5] := chr(0);
writeln('String = ', s1);
writeln;
writeln('The index of substring ', s2, ' = ', strindex(&str[1], &sub[1]):3);
sub := s3;
str[4] := chr(0);
writeln('The index of substring ', s3, ' = ', strindex(&str[1], &sub[1]):3);
sub := s4;
str[4] := chr(0);
writeln('The index of substring ', s4, ' = ', strindex(&str[1], &sub[1]):3);
sub := s5;
str[4] := chr(0);
writeln('The index of substring ', s5, ' = ', strindex(&str[1], &sub[1]):3);
end.

```

This program generates the following output:

```

Index          1          2          3          4          5          6
              01234567890123456789012345678901234567890123456789
String = Twinkle, twinkle little star, how I wonder what you are.
The index of substring Twin = 0
The index of substring ink = 2
The index of substring hat = 44
The index of substring xxx = -1

```

## strlen

string length

### Definition

```

type cp = ^char;
function strlen (str:cp): integer;

```

where *str* is a pointer to a null terminated string.

The `strlen (str)` function returns the length of the string pointed to by *str*. The terminating null character is not counted when determining the length.

### EXAMPLE (ex086.p)

```

program strlen1(output);
type
  cp = ^char;
function strcat(str1, str2: cp): cp; external;
function strlen(str:cp): integer; external;
var
  str1, str2: packed array[1..100] of char;
  i, len: integer;
  p: cp;
begin
  str1 := 'At the end of a row';
  str1[20] := chr(0);
  str2 := ' I stepped on the toe';
  str2[22] := chr(0);
  p := strcat(&str1[1], &str2[1]);
  str2 := ' of an unemployed hoe.';
  str2[23] := chr(0);

```

```

p := strcat(&str1[1], &str2[1]);
i := 1;
while str1[i] <> chr(0) do begin
  write(str1[i]);
  i := i + 1;
end;
writeln;
writeln;
len := strlen(&str1[1]);
writeln('The number of characters in this string is ', len:3);
end.

```

This program generates the following output:

```

At the end of a row I stepped on the toe of an unemployed hoe.
The number of characters in this string is 62

```

## strncat

string concatenate with maximum length

### Definition

```

type cp = ^char;
function strncat (str1, str2: cp; n: integer): cp;

```

where

*str1* and *str2* are pointers to null terminated strings,  
*n* is an integer less than the length of the strings at *str1* or *str2*.

The `strncat (str1, str2)` function copies the null terminated string, *str1*, onto the end of the null terminated string, *str2*, until *n* characters are copied or a null is encountered in *str1*. The first character of *str1* replaces the null terminating *str2*. If the terminating null character in *str1* is found before *n* characters are copied, then the null is added to *str2* and no other characters are written. If *n* characters are written before a terminating null is found, then `strncat` places a terminating null onto *str2*. `strncat` returns a pointer to the concatenated string.

### EXAMPLE (ex087.p)

```

program strncat1(output);
type
  cp = ^char;
function strncat(des, src: cp; n: integer): cp; external;
var
  a, b, c: packed array[1..80] of char;
  p: cp;
  i: integer;
procedure printc;
begin
  i := 1;
  while c[i] <> chr(0) do begin
    write (c[i]);
    i := i + 1;
  end;
  writeln;
end;

```



```

begin
  a := 'Where never is heard a discouraging word ';
  a[42] := chr(0);
  b := 'and the skies are not cloudy all day.';
  b[38] := chr(0);
  writeln('String 1 = ',a:42);
  writeln('String 2 = ',b:42);
  writeln;
  p := strncat(&c[1], &a[1], 20);
  write('First 20 characters from string 1 = ');
  printc;
  p := strncat(&c[1], &a[21], 100);
  write('Remaining characters from string 1 = ');
  printc;
  p := strncat(&c[1], &b[1], 50);
  writeln;
  writeln('String 1 concatenated with string 2 = ');
  printc;
end.

```

This program generates the following output:

```

String 1 = Where never is heard a discouraging word
String 2 = and the skies are not cloudy all day.
First 20 characters from string 1 = Where never is heard
Remaining characters from string 1 = Where never is heard a discouraging word
String 1 concatenated with string 2 =
Where never is heard a discouraging word and the skies are not cloudy all day.

```

## strncmp

string compare with maximum length

### Definition

```

type cp = ^char;
function strncmp (str1, str2: cp; n: integer): integer;

```

where

*str1* and *str2* are pointers to null terminated strings,  
*n* is an integer specifying the maximum number of characters to compare.

The `strncmp (str1, str2)` function compares the first *n* characters of two null terminated strings and returns one of the following values indicating their relationship:

```

if str1 < str2 then return a negative number
if str1 = str2 then return zero
if str1 > str2 then return a positive number

```

The comparison stops if a null is encountered before *n* characters, since that marks the end of the string.

### EXAMPLE (ex088.p)

```

program strncmp(output);
const
  s1 = 'economy';
  s2 = 'ecology';
  s3 = 'eclipse';
  s4 = 'eclogue';

```

```

type
  cp = ^char;
  s10 = packed array [1..10] of char;

function strcmp(str1,str2:cp; n:integer): integer; external;

procedure compare(a,b:s10; n:integer);
begin
  a[n+1] := chr(0);
  b[n+1] := chr(0);
  writeln('Comparison on first ', n:2,' characters = ', strcmp(&a[1],&b[1],n));
end;

begin
  writeln('String 1 = ',s1);
  writeln('String 2 = ',s2);
  compare(s1, s2, 3);
  compare(s1, s2, 4);
  compare(s1, s2, 20);
  writeln('String 3 = ',s3);
  writeln('String 4 = ',s4);
  compare(s3, s4, 3);
  compare(s3, s4, 4);
  compare(s3, s4, 20);
end.

```

This program generates the following output:

```

String 1 = economy
String 2 = ecology
Comparison on first 3 characters = 0
Comparison on first 4 characters = 2
Comparison on first 20 characters = 2
String 3 = eclipse
String 4 = eclogue
Comparison on first 3 characters = 0
Comparison on first 4 characters = -6
Comparison on first 20 characters = -6

```

## strncpy

string copy with maximum length

### Definition

```

type cp = ^char;
function strncpy (str1,str2:cp; n:integer): cp;

```

where

*str1* and *str2* are pointers to null terminated strings,  
*n* is the maximum number of characters to copy.

The `strncpy (str1, str2, n)` function copies up to *n* characters from the string *str2* to the string *str1*. The copy involves the following two situations:

1. if a null character is encountered in *str2* before *n* characters have been copied, then *str1* is filled with nulls until *n* characters have written.
2. If *str2* is longer than *n* characters, then a null character is not copied to *str1*. `strncpy` returns a pointer to the resulting string.

Caution: It is the programmer's responsibility to ensure that the receiving buffer is large enough for what is written to it. If it is not large enough, adjacent buffers may be overwritten.

#### EXAMPLE (ex089.p)

```

program strncpy1 (output);
{ Interface to strncpy function in c library }

type cp = ^char;

function strncpy (des,src :cp; n:integer): cp; external;

var
  des,s1,s2: packed array [1..100] of char;
  i: integer;
  p: cp;

begin
  s1 := 'Say not of a thing which cannot be understood'; s1[46]:=chr(0);
  s2 := 'that in the end it will be understood.'; s2[39]:=chr(0);
  writeln ('String 1 = ',s1:45);
  writeln ('String 2 = ',s2:30);
  writeln;
  p := strncpy (&des[1], &s1[1], 25);
  p := strncpy (&des[26], &s2[20], 50);
  write ('Final string = '); i := 1;
  while (des[i] <> chr(0)) do begin write(des[i]); i:= i+1 end;
  writeln;

end.

```

This program generates the following output:

```

String 1 = Say not of a thing which cannot be understood
String 2 = that in the end it will be understood.
Final String = Say not of a thing which will be understood.

```

## strpbrk

Find first occurrence of any character from a given string in another string

#### Definition

```

type cp = ^char;
function (s1, s2: cp): cp;

```

where

*s1* is the string being searched;  
*s2* is a list of characters being sought.

strpbrk returns the location in *s1* of the first character that appears in *s2*. If no character from *s2* appears in *s1*, NIL is returned.

## strrchr

reverse index of a character in a string

#### Definition

```

type cp = ^char;
function strrchr (str:cp; ch:char): cp;

```

where

*str* is a pointer to the null terminated string to be searched,  
*ch* is the character to match.

The `strrchr(str, ch)` function returns a pointer to the last instance of the character *ch* in the string pointed to by *str*. If *ch* is not found, then `strrchr` returns a null pointer.

#### EXAMPLE (ex090.p)

```

program strrchr1(output);

type
  cp = ^char;
  s40 = packed array [1..40] of char;
  s40p = ^s40;

function strrchr(str:cp; ch:char): s40p; external;

var
  str: s40;
  ptr: s40p;

procedure printstring(c:char);
begin
  writeln('The last substring beginning with ',c,' = ',ptr^);
  if ptr = nil then
    writeln('(',c,' is not present)');
  end;

begin
  str := 'A little learning is a dangerous thing.';
  str[40] := chr(0);
  writeln('The initial string = ',str);
  writeln;
  ptr := strrchr(&str[1], 'l'); printstring('l');
  ptr := strrchr(&str[1], 'd'); printstring('d');
  ptr := strrchr(&str[1], 't'); printstring('t');
  ptr := strrchr(&str[1], 'x'); printstring('x');
  end.

```

This program generates the following output:

```

The initial string = A little learning is a dangerous thing
The last substring beginning with l =      learning is a dangerous thing.
The last substring beginning with d =                dangerous thing.
The last substring beginning with t =                                thing.
The last substring beginning with x =                                (null)
(x is not present)

```

## strrindex

reverse index of a substring within a string

#### Definition

```

type  cp = ^char;

function strrindex (str1,str2: cp): integer;

```

where *str1* and *str2* are pointers to null terminated strings.

The `strrindex (str1, str2)` function finds the last occurrence of string `str2` (not including the terminating null character) in the string `str1`. It returns a pointer to the located string in `str1`, or -1 if no match occurs.

### EXAMPLE (ex091.p)

```

program strrindex(output);
type
  cp = ^char;
function strrindex(str, sub: cp): integer; external;
const
  s1 = 'Twinkle, twinkle little star, how I wonder what you are.';
  s2 = 'Twin';
  s3 = 'ink';
  s4 = 'hat';
  s5 = 'xxx';
var
  str, sub: packed array [1..60] of char;
begin
  str := s1;
  str[57] := chr(0);
  sub := s2;
  str[5] := chr(0);
  writeln('String = ', s1);
  writeln;
  writeln('The index of substring ', s2, ' = ', strrindex(&str[1], &sub[1]):3);
  sub := s3;
  str[4] := chr(0);
  writeln('The index of substring ', s3, ' = ', strrindex(&str[1], &sub[1]):3);
  sub := s4;
  str[4] := chr(0);
  writeln('The index of substring ', s4, ' = ', strrindex(&str[1], &sub[1]):3);
  sub := s5;
  str[4] := chr(0);
  writeln('The index of substring ', s5, ' = ', strrindex(&str[1], &sub[1]):3);
end.

```

This programs generates the following output:

```

Index          1          2          3          4          5          6
              012345678901234567890123456789012345678901234567890123456789
String = Twinkle, twinkle little star, how I wonder what you are.
The index of substring Twin = 0
The index of substring ink = 11
The index of substring hat = 44
The index of substring xxx = -1

```

## strsave

Save a copy of a string

### Definition

```

type cp = ^char;
function strsave(str: cp): cp;

```

where `str` is the string to be copied.

strsave returns a pointer to a new copy of its string argument.

## strspn

Compute the length of the initial match between two strings

### Definition

```
type cp = ^char;
function strspn(s1, s2: cp): integer;
```

where *s1* and *s2* are the two strings being compared.

strspn returns the number of consecutive characters in its first string argument that are also in its second string argument.

## strstr

Locate a string within another string

### Definition

```
type cp = ^char;
function strstr(s1, s2: cp): cp;
```

where

*s1* is the string being searched;  
*s2* is the string being sought.

strstr returns a pointer to the first occurrence of *s2* in *s1*. If *s2* isn't found in *s1*, NIL is returned.

## strtod

Interpret a string representation of a double value

### Definition

```
type cp = ^char;
function strtod(nptr: cp; var endptr: cp): double;
```

where *nptr* is an expression representing a pointer to a char.

The return value of *strtod* is a double representing the contents of *nptr*. If *endptr* is not already set to NIL, it is set on return to point to the first character of *nptr* that cannot be interpreted. If the number is too large to store as a double, the constant *HUGE\_VAL* is returned and the global variable *errno* is set to *ERANGE*. If the number is too small to store as a double, zero is returned and *errno* is set to *ERANGE*.

## strtok

Tokenize a string

### Definition

```
type cp = ^char;
function strtok(var s1: cp; s2: cp): cp;
```

where

*s1* is a pointer to a char;  
*s2* is an expression representing a pointer to a char.

`strtok` returns the portion of `s1` that precedes the first instance of a character appearing also in `s2`. If the strings have no characters in common, `NIL` is returned. If `NIL` is passed as the first argument, the next token of the original string is returned.

## strtol

Interpret a string representation of a long integer value

### Definition

```
type cp = ^char;
function strtol(nptr: cp; var endptr: cp; base: integer): integer;
```

where

`nptr` is an expression representing a pointer to a char;  
`endptr` is a pointer to a char;  
`base` is an expression representing an integer.

`strtol` converts `nptr` into a long value, assuming the base specified in `base`. `endptr` is set on return to the end of the converted string.

## strtoul

Interpret a string representation of an unsigned long integer value.

### Definition

```
type cp = ^char;
function strtoul(nptr: cp; var endptr: cp; base: integer): integer;
```

where

`nptr` is an expression representing a pointer to a char;  
`endptr` is a pointer to a char;  
`base` is an expression representing an integer.

`strtoul` converts `nptr` into an unsigned long value, assuming the base specified in `base`. Leading spaces and zeros are permitted, and if the base is 16 the number may be preceded by the sequence `0x` or `0X`. If the base is zero the form is expected to be that of a standard integer, i.e., `0x` or `0X` for hexadecimal, leading `0` for octal, and leading nonzero digit for decimal. If `endptr` is not `NIL`, it will return a pointer to the portion of the string that could not be interpreted. The string to be interpreted may contain a sign, which is applied to the result before converting it to unsigned. If the number is too large to be expressed by an unsigned long integer, the return value is `ULONG_MAX` and the global variable `errno` is set to `ERANGE`.

## strxfrm

Transform a string into another based upon the program's locale

### Definition

```
type cp = ^char;
function strxfrm(str1, str2: cp; n: integer): cp;
```

where

`str1` and `str2` are pointers to null terminated strings,  
`n` is the maximum number of characters to copy.

The `strxfrm (str1, str2, n)` function copies up to  $n$  characters from the string `str2` to the string `str1`. The copy involves the following two situations:

- 1) if a null character is encountered in `str2` before  $n$  characters have been copied, then `str1` is filled with nulls until  $n$  characters have been written.
- 2) If `str2` is longer than  $n$  characters, then a null character is not copied to `str1`. `strxfrm` returns a pointer to the resulting string.

Caution: It is the programmer's responsibility to ensure that the receiving buffer is large enough for what is written to it. If it is not large enough, adjacent buffers may be overwritten.

In this implementation, this function is identical to `strncpy`.

## swab

swap bytes

### Definition

```
type cp = ^char;
function swab (str1, str2: cp; n: integer): integer;
```

where

`str1` and `str2` are pointers to null terminated strings,  
 $n$  is an integer representing the maximum number of characters to copy.

The `swab (str1, str2, n)` function copies  $n$  bytes in pairs from `str1` to `str2`, reversing the pairs in the process. If  $n$  is odd, then it is rounded down.

### EXAMPLE (ex092.p)

```
program swab2 (output);
type
  cp = ^char;
function swab( src, des: cp; n: integer): integer; external;
var
  src, des: packed array [1..20] of char;
begin
  src := '12345678badcfefhgji';
  src[19] := chr(0);
  writeln('Initial string = ', src);
  writeln;
  swab(&src[1], &des[1], 18);
  writeln('    Swab once = ', des);
  src := des;
  swab(&src[1], &des[1], 18);
  writeln('    Swab twice = ', des);
end.
```

This program generates the following output:

```
Initial string = 12345678adcfefhgji
    Swab once = 21436587abcdefghij
    Swab twice = 12345678adcfefhgji
```



# system (DOS only)

## shell to DOS Definition

```
type cp = ^char;
function system (c: cp): integer; external;
```

where *c* is a pointer to a null terminated character string.

The `system(c)` function shells to DOS and executes the command contained in the null terminated character string pointed to by *c*. The normal return value of `system` is zero; a non-zero return value indicates failure.

The `system` function works by loading a copy of `COMMAND.COM` into memory and passing it *c*, the pointer to the command to be executed. Both internal and external DOS commands may be executed. In order for the `system` function to work correctly, sufficient memory must be available for loading `COMMAND.COM` and any program it might load. There are several ways to accomplish this using the Phar Lap DOS extender, `RUN386`. For example, the following command will load and run the compiled program `fn.exp`, leaving the maximum conventional memory available for loading `COMMAND.COM`:

```
run386 -maxreal ffffh fn.exp
```

### EXAMPLE (ex093.p)

The program in this example uses the `system` function to execute three DOS commands. These commands 1) get a directory of the current drive and redirect the listing to a file, named *dirlst*; 2) print the file *dirlst* on the screen; 3) echo a message on the terminal.

```
program system1(output);

type
  cp = ^char;
function system(c: cp): integer; external;

var
  cmd: packed array[1..20] of char;

begin
  cmd := 'dir > dirlst';
  cmd[20] := chr(0);
  if (system (&cmd[1]) <> 0) then
    writeln(cmd, 'failed!');

  cmd := 'type dirlst ';
  cmd[20] := chr(0);
  if (system (&cmd[1]) <> 0) then
    writeln(cmd, 'failed!');

  cmd := 'echo -- all done.';
  cmd[20] := chr(0);
  if (system (&cmd[1]) <> 0) then
    writeln(cmd, 'failed!');
end.
```

This program was executed with the command:

```
run386 -maxreal ffffh system1
```

and generated the following output:

```
Volume in drive D is DISK2_VOL1
Directory of D:\PAS\SYSTEM
```

```

.          <DIR>      5-12-89   3:08p
..         <DIR>      5-12-89   3:08p
SYSTEM    DOC      1920   5-12-89   2:45p
SYSTEM1   P        548   5-12-89   3:30p
SYSTEM1   MAP     9212   5-12-89   3:31p
SYSTEM1   EXP    22807   5-12-89   3:31p
ZSYSTEM1  TXT         0   5-12-89   3:31p
DIRLST    0         0   5-12-89   3:31p
      8 File(s) 44498944 bytes free
-- all done.

```

## tan

tangent

### Definition

```
function tan (d: double): double;
```

where  $d$  is an expression of type DOUBLE.

The  $\tan(d)$  function returns the value of the tangent of  $d$ . If the input argument is out of range, the global variable `errno` is set to `EDOM`.

### EXAMPLE (ex094.p)

```

program tan1(output);
function tan(d: double): double; external;
const
  pi = 3.14159265358979323846;
var
  i: integer;
  x: double;
begin
  x := 0.0;
  for i:= 1 to 5 do begin
    writeln(' x = ', x, ' tan(x) = ', tan(x));
    x := x + pi/4.0;
  end;
end.

```

This program generates the following output:

```

x = 0.000000000000000000e+00 tan(x) = 0.000000000000000000e+00
x = 7.85398163397448286e-01 tan(x) = 1.000000000000000000e+00
x = 1.57079632679489657e+00 tan(x) = -1.700000000000000149e+308
x = 2.35619449019234486e+00 tan(x) = -1.000000000000000000e+00
x = 3.14159265358979311e+00 tan(x) = 0.000000000000000000e+00

```

## tanf

Compute single-precision tangent

### Definition

```
function (f: float): float;
```

where  $f$  is an expression of type FLOAT.

The `tanf(f)` function returns the value of the tangent of  $f$ . If the result is out of range, the global variable `errno` will be set to `ERANGE`. `errno` is not set if the input argument is out of range.

## tanh

hyperbolic tangent

### Definition

```
function tanh (d: double): double;
```

where  $d$  is an expression of type `DOUBLE`.

The `tanh(d)` function returns the hyperbolic tangent of  $d$ .

### EXAMPLE (ex095.p)

```
program tanh1(output);
function tanh(f: double): double; external;
const
  pi = 3.14159265358979323846;
var
  i: integer;
  x: double;
begin
  x := 0.0;
  for i:= 1 to 5 do begin
    writeln(' x = ', x, ' tanh(x) = ', tanh(x));
    x := x + pi/4.0;
  end;
end.
```

This program generates the following output:

```
x = 0.000000000000000000e+00 tanh(x) = 0.000000000000000000e+00
x = 7.85398163397448286e-01 tanh(x) = 6.55794202632672541e-01
x = 1.57079632679489657e+00 tanh(x) = 9.17152335667274345e-01
x = 2.35619449019234486e+00 tanh(x) = 9.82193380007238656e-01
x = 3.14159265358979311e+00 tanh(x) = 9.96272076220750153e-01
```

## tanhf

Compute single-precision hyperbolic tangent

### Definition

```
function tanhf (f: float): float;
```

where  $f$  is an expression of type `FLOAT`.

The `tanhf(f)` function returns the hyperbolic tangent of  $f$ .

## time\_ (DOS only)

return the current time in ASCII format

### Definition

```
type s8 = packed array [1..8] of char;
procedure time_ (VAR timeStr: s8); external;
```

where *timeStr* is an array variable of type CHAR containing at least eight elements.

The `time_` function returns the current time, known to DOS, as an eight character ASCII string in the form *hh:mm:ss*.

**EXAMPLE (ex096.p)**

```

program time1(output);
type
  s8 = packed array [1..8] of char;
procedure time_(var timeStr: s8); external;
var
  now: s8;
begin
  time_(now);
  writeln('The time is ', now);
end.

```

This program generates the following output:

```
The time is 14:40:40
```

## timedate\_

return date and time in integer format

**Definition**

```

procedure timedate_(VAR year, month, day, hour, minute, second, mcsec: integer);
external;

```

where *year*, *month*, *day*, *hour*, *minute*, *second*, and *mcsec* are variables of type INTEGER.

The `timedate_` procedure returns the current date and time as these items are known to DOS. Remember, the DOS time granularity is approximately 5 hundredths of a second.

**EXAMPLE (ex097.p)**

```

program timdat(output);
procedure timedate_
  (var year, month, day, hour, minute, second, mcsec: integer); external;
var
  year, month, day, hour, minute, second, mcsec: integer;
begin
  timedate_(year, month, day, hour, minute, second, mcsec);
  writeln('year = ', year);
  writeln('month = ', month);
  writeln('day = ', day);
  writeln('hour = ', hour);
  writeln('minute = ', minute);
  writeln('second = ', second);
  writeln('mcsec = ', mcsec);
end.

```

This program generates the following output:

```

year      =      89
month     =      4
day       =      8

```

```
hour    =      14
minute  =      34
second  =      40
mcsec   =    750000
```

## tmpnam

Create a file name

### Definition

```
type cp = ^char;
function tmpnam(s: cp): cp;
```

where *s* is an expression representing a pointer to a char.

tmpnam creates a file in the /tmp directory (UNIX) or current working directory (DOS, OS/2, Windows) beginning with X\_ and a character and ending with a five digit number that does not have the same name as an existing file. If *s* is not NIL, the return value is *s*. If *s* is NIL, the return value is a pointer to a string containing the new file name.

## tolower

Convert character to lower case

### Definition

```
function tolower(c: integer): integer;
```

where *c* is an expression of type INTEGER.

tolower returns the lower case equivalent of its character argument.

## toupper

Convert character to upper case

### Definition

```
type cp = ^char;
function toupper(c: integer): integer;
```

where *c* is an expression of type INTEGER.

toupper returns the upper case equivalent of its character argument.

## y0

Bessel function of the second kind, order 0.

### Definition

```
function y0 (d: double): double;
```

where *d* is an expression of type DOUBLE.

The  $y_0(d)$  function returns the Bessel function of the second kind, order 0, of *d*. This corresponds to  $Y_0(d)$  in the usual notation. If the input argument is out of range, the global variable *errno* will be set to EDOM.

### EXAMPLE (ex098.p)

```
program y0a(output);
function y0 (d: double): double; external;
```

```

var
  x: double;
  i: integer;

begin
  x := 0.0;
  for i := 1 to 7 do begin
    writeln(' x = ', x:6:2, ' y0(x) = ', y0(x):20:14);
    x := x + 2.5;
  end;
end.

```

This program generates the following output:

```

x = 0.00 y0(x) = 0.000000000000000
x = 2.50 y0(x) = 0.49807035961523
x = 5.00 y0(x) = -0.30851762524903
x = 7.50 y0(x) = 0.11731328614821
x = 10.00 y0(x) = 0.05567116728360
x = 12.50 y0(x) = -0.17121430684467
x = 15.00 y0(x) = 0.20546429603892

```

## y1

Bessel function of the second kind, order 1.

### Definition

```
function y1 (d: double): double;
```

where  $d$  is an expression of type DOUBLE.

The  $y_1(d)$  function returns the Bessel function of the second kind, order 1, of  $d$ . This corresponds to  $Y_1(d)$  in the usual notation. If the input argument is out of range, the global variable `errno` will be set to `EDOM`.

### EXAMPLE (ex099.p)

```

program y1a(output);

function y1 (d: double): double; external;

var
  x: double;
  i: integer;

begin
  x := 0.0;
  for i := 1 to 7 do begin
    writeln(' x = ', x:6:2, ' y1(x) = ', y1(x):20:14);
    x := x + 2.5;
  end;
end.

```

This program generates the following output:

```

x = 0.00 y1(x) = 0.000000000000000
x = 2.50 y1(x) = 0.14591813796679
x = 5.00 y1(x) = 0.14786314339123
x = 7.50 y1(x) = -0.25912851048611
x = 10.00 y1(x) = 0.24901542420695
x = 12.50 y1(x) = -0.15383825653750
x = 15.00 y1(x) = 0.02107362803687

```

# yn

Bessel function of the second kind, order  $i$ .

## Definition

```
function yn (i: integer; d: double): double;
```

where

$i$  is an expression of type INTEGER,  
 $d$  is an expression of type DOUBLE.

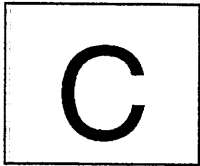
The  $yn(i, d)$  function returns the Bessel function of the second kind, order  $i$ , of  $d$ . This corresponds to  $Y_n(i, d)$  in the usual notation. If the input argument is out of range, the global variable `errno` will be set to EDOM.

## EXAMPLE (ex100.p)

```
program yna(output);
function yn(i: integer; d:double): double; external;
var
  x: double;
  i, j: integer;
begin
  for i := 1 to 3 do begin
    x := 2.50;
    for j := 1 to 3 do begin
      writeln(' i = ', i:2, ' x = ', x:6:2, ' yn(i,x) = ', yn(i,x):20:14);
      x := x + 2.50;
    end;
    writeln;
  end;
end.
```

This program generates the following output:

```
i = 1  x =  2.50  yn(i,x) =  0.14591813796679
i = 1  x =  5.00  yn(i,x) =  0.14786314339123
i = 1  x =  7.50  yn(i,x) = -0.25912851048611
i = 2  x =  2.50  yn(i,x) = -0.38133584924180
i = 2  x =  5.00  yn(i,x) =  0.36766288260552
i = 2  x =  7.50  yn(i,x) = -0.18641422227784
i = 3  x =  2.50  yn(i,x) = -0.75605549675367
i = 3  x =  5.00  yn(i,x) =  0.14626716269319
i = 3  x =  7.50  yn(i,x) =  0.15970759193793
```



# NDP Pascal Error Messages

## Overview

*Section C.1* of this appendix contains a listing and explanation of the compile time error messages. *Section C.2* contains a listing and explanation of the execution time error messages. Other listings of errors may be found in Appendix A of the *NDP User's Manual* and the *NDP Tools Manual*.

## C.1. Compile Time Error Messages

### 1 Selector must be constant

This occurs when the selector in a variant record is not a constant. The variant record may be defined in either a type definition or a variable declaration. This error is sometimes caused by misspelling a previously defined enumerated constant.

### 2 Tag field must be a scalar type

This occurs in a type definition or variable declaration that makes use of a variant record with a tag field. The constant or variable used in the tag field must be a scalar. It may not be an array, record, set, pointer, or file type.

### 3 Initial value for tag must be constant

This occurs when the predefined procedure `NEW` is called to allocate space for a dynamic variable with a variant record. The value for the tag field must be a constant.

### 4 Statement expected: function call illegal

A function call appears where a statement is expected.

### 5 Ordinal type expected

The indexes of an array type must be of type ordinal. This error occurs when an array is being declared without indexes, or the indexes are not ordinal values.

### 6 Type not defined

A type identified is being used that has not yet been defined. This is probably caused by omitting the type definition, placing the type definition in the wrong place, or a spelling mistake.

### 7 Scalar value expected

The upper and lower limits of a subrange type must be scalar values.



## 8 Incompatible scalar values

The upper and lower limits of a subrange type were specified as scalar values, but not of the same base type.

## 9 Labels must be in the range 0..9999

Labels must be integer values in the range 0 to 9999.

## 10 Illegal use of a procedure as a function

## 11 Multiple definition of identifier:

An identifier may only be defined once. This is probably a spelling mistake.

## 12 Ordinal type required in case statement

The expression used as the selector in a case statement must be of ordinal type.

## 21 Must be a record

A variable is being used as if it were part of a record. The variable is not a record, hence, it may not be qualified with a field name.

## 26 Illegal variable or expression

This message has a variety of causes and probably results from a typing mistake.

An invalid type in a type definition. For example:

```
type t1 = nil; or type t2 = ;
```

An unrecognizable operator in an expression. For example:

```
:=* or x := :
```

Providing an incorrect type to a predefined function. For example, using `mod(r, r)` for real `r`.

## 27 This object has no defined size

## 29 Operand must be a variable

The dyadic operators taken from the C language, `+=`, `-=`, etc., may not be used in an expression together with the assignment operator.

## 30 REAL operand not allowed here

A variable of type `REAL` is being used in an expression in which the real type is not allowed. For example, using the integer divide operator, `div`, with real arguments.

## 31 Cannot take the address of this object

The address of (`&`) operator is being incorrectly used. This usually occurs when the address of operator is used on an entire array, instead of an array element. For example, if `x` is an array, then `&x[10]` is legal, while `&x` is not.

### 33 Type mismatch

The variable or constant being referenced is not of the expected type. This message has a variety of causes and probably results from a typing mistake.

- can't pass field of record as actual parameter unless it's a pointer (array)  
 - 2 types spelled the same but, ones uppercase & the other is lower case

### 35 Can only index arrays

Square brackets are used to designate array subscripts. Check to see that the identifier is an array. If the identifier is a procedure or function, then parentheses must be used to delimit the argument list.

### 36 This is not a variable

The identifier being referenced is not a variable. This occurs when an identifier is declared in two different ways: as a constant in a CONST declaration, and later as a field identifier within a record definition. This conflict in usage is reported when the qualified variable name is used on the left hand side of an assignment statement.

### 37 Illegal operation

The operator symbol is not recognized or is being incorrectly used. This is usually a typing mistake, for example, using \*\* for \*, or using // for /.

Using an operator on a data type for which the operator is not defined also produces this message. For example, using the integer divide function, div, to divide two real numbers.

### 38 This is a binary file

The compiler is reading a binary file. Check that the file name and extension on the compiler command line is correct.

### 40 This must be a procedure or function

Attempt to use an identifier as a procedure or function. This error may result from enclosing array subscripts with parentheses instead of square brackets [ and ].

This error also may occur in a procedure or function call when a formal parameter is a procedure or function, and the actual parameter is a constant, variable, or expression.

- may be non-standard pascal (SUN) proced. or func. call.

### 41 A procedure may not return a value

A procedure name may not occur on the left side of an assignment statement.

### 42 No return type specified for function

The heading of a function definition does not contain a return type.

### 43 Unpacked variable required for actual parameter

Actual variable parameters used in function or procedure calls must be unpacked. This issue is discussed in Section 4.11.

Using a constant or a type identifier as an actual variable parameter also produces this error.

- You can't pass a field of a record as a parameter unless it's a pointer (or array).

**44 Not enough arguments given**

The number of arguments in a procedure or function call is less than the number of parameters declared in the procedure or function definition.

**45 New requires a pointer variable**

The argument to the predefined procedure NEW requires a variable whose type is a pointer.

**46 Argv requires a string variable**

The second argument to the predefined procedure ARGV must be a string, that is, a packed array [1..n] of char.

**47 Pointer type undefined****48 Scalar value required**

A variable whose type is a scalar is required by the syntax. For example, the lower and upper values in the definition of a subrange type must be scalar values.

**49 Program ends before end of file**

The end of the input file has been encountered without finding the final period in a program, or final semicolon in a separately compiled module.

This message often occurs when a syntax error is encountered and the parser is unable to recognize the next statement being scanned because of the previous error.

*- nested comments, - an extra parenthesis, - extra semicolon in type or var sect*

**50 Record variable expected**

A variable whose type is a record is required by the syntax. For example, the WITH statement requires a record variable.

**51 Type expected**

A type identifier is required; an anonymous type is not allowed. For example, `ch: ^packed array [1..5] of char;` is not legal since a type name must follow the pointer.

**52 No assignment to return variable in function**

The function name is not assigned a value within the body of the function.

**53 Dispose requires a pointer value**

The argument to the predefined procedure DISPOSE requires a variable whose type is a pointer.

**54 Cannot write this type of expression**

The predefined procedures WRITE and WRITELN can only print expressions of type BOOLEAN, CHAR, INTEGER, DOUBLE, FLOAT, and REAL.

**55 Cannot read into this type of variable**

The predefined procedures `READ` and `READLN` can only read data of type `BOOLEAN`, `CHAR`, `INTEGER`, `DOUBLE`, `FLOAT`, and `REAL`.

**56 Text file expected**

The predefined procedures `READLN` and `WRITELN` require a text file.

**57 Text variable expected**

The predefined function `EOLN` and `PAGE` require an actual parameter that is of type `TEXTFILE`.

**58 File expected**

A variable of `FILE` type was expected.

**59 Set constructor elements must be ordinal values****60 Unexpected end of file**

The end of the input file has been encountered before the end of the program has been reached. This is probably because of a mismatched `BEGIN/END` pair, or a semicolon prematurely terminating an `IF` statement.

**61 Type illegal in expression**

A type identifier may not be used in an expression. An expression may only contain constants, variables, functions calls, and operators.

**62 Procedure call illegal in expression**

This occurs when a procedure name is used in an expression as if it were a function.

**64 Built in operation may not be an actual parameter**

A predefined Pascal routine may be used as an actual parameter. To get around this, create another function that simply returns the required value of the predefined Pascal routine. See *Example 1* in *Section 8.4* for an example using the `cos` function.

**65 Subrange type expected**

A subrange type identifier is required. For example, the definition of an array required the use of a subrange type.

**66 Packed array variable expected**

The predefined procedures `PACK` and `UNPACK` require a packed actual argument.

**67 Array component types incompatible**

The array parameters to the predefined `UNPACK` procedure are not compatible. This error only occurs when the `-ansi` compile time switch is used.

**68 Unpacked array variable expected**

The predefined procedures PACK and UNPACK require an unpacked actual argument.

**69 Duplicate case**

There is a duplicate constant value in the alternatives of a case statement.

**71 Cannot open file: <filename>**

The file indicated by <filename> cannot be found.

**72 expected: <symbol1> got: <symbol2>**

The scanner expected to see <symbol1>. Instead <symbol2> was encountered.

**74 File name too long**

The file name specified is greater than 131 characters.

*- 14 errors for 1986 machines.*

**78 Too many -I options**

A maximum of sixteen -I options are allowed to the compiler.

**79 Illegal option: <string>**

The option specified by <string> is not recognized by the compiler.

**80 Type name expected**

A formal parameter in a procedure or function is missing a type identifier, or the identifier provided is not a type identifier.

**82 File of file not allowed**

A file may not be made up of files.

**83 Input or output not defined**

The file identifiers INPUT or OUTPUT do not occur in the program heading and the program contains a READ or WRITE statement. This only occurs if the -ansi compiler option is used.

**85 Low bound must be less than high bound**

The bounds in a subrange type must be given with the lower bound first. This error also can be caused by entering an integer constant larger than MAXINT, which causes the value to be stored as a negative number.

**86 Null string**

The null string is not allowed in standard Pascal. This error only occurs if the -ansi compiler option is used.

**87 A number may not be followed by a letter**

An integer constant contains a letter. This only occurs when the `-ansi` compiler option is used.

**88 File may not be assigned to**

A file variable may not be used on the left hand side of an assignment statement.

**89 File comparison not legal**

Two file variables may not be used in an expression. File comparison must be done on a component by component basis.

**90 Subrange not legal in record tag/case**

In standard Pascal, a subrange may not be used as the constant selector in a case statement. This occurs in two situations: in a variant record definition or in a CASE statement. Get around this restriction by explicitly listing each constant in the subrange. This error only occurs when the `-ansi` compiler option is used.

**91 Tag field used as var parameter**

Standard Pascal does not allow the tag field of a variant record to be passed as a variable parameter to a procedure or function. This error only occurs when the `-ansi` compiler option is used.

**92 Illegal type for comparison**

Standard Pascal does not allow comparison between the types specified. For example, comparison between two record structures is considered illegal and must be done on a component by component basis. This error only occurs when the `-ansi` compiler option is used.

**93 Argument to round or trunc must be real**

The input argument to the predefined functions `ROUND` and `TRUNC` must be of type `REAL`.

**94 Nil may not be in constant**

Standard Pascal does not permit a constant to be defined with the value of `NIL`. This error only occurs if the `-ansi` compiler option is used.

**95 Expression not legal in constant**

Standard Pascal does not permit an expression to be used in a constant definition. This error only occurs if the `-ansi` compiler option is used.

**96 For index may not be var parameter**

The index of a `FOR` loop may not be passed as a variable parameter to a procedure or function.

**97 Assignment to FOR loop index inside loop**

The index of a `FOR` loop may not be altered with the body of the loop.

**98 Not all tag cases specified in variant record**

The list of case selector constants is not exhaustive for the type specified in the selector expression. This only occurs when the `-ansi` option is used, and typically occurs in the form "case integer of", since not all integers are listed.

**99 May not dispose of a value parameter**

Standard Pascal does not allow a parameter that is passed by value to be an argument in the dispose procedure. This only occurs if the `-ansi` compiler option is used.

**100 Goto out of scope**

The target of a `GOTO` statement is not within the current scope. A `GOTO` statement cannot transfer control into a structured statement, or into or out of a procedure or function. Further details are in *Section 7.7*.

**101 Label already defined**

Duplicate definition of a label.

**102 Local variable required**

The index of a `FOR` loop must be an entire variable. It may not be an element of an array, record, or pointer to an integer.

**103 Statement expected**

The Pascal syntax requires a statement here. This message occurs in a variety of ways, mostly caused by typing mistakes. For example, typing the assignment operator without the colon, e.g., `=`, instead of `:=`. *- nested comments*  
*- extra semicolon in type or var section*

**104 Case expression not constant**

The selector in a case expression is not a constant expression.

**105 Label not declared**

A label was found that was not declared.

**106 Label expected**

A label is expected in the `GOTO` statement.

**107 Variable expected**

The Pascal syntax requires a variable here. This can be caused by using a constant identifier on the left hand side of an assignment statement, or qualifying a variable with an undefined field name (i.e., `rec.x := 7;` where the field `x` does not exist in the record `rec`).

**108 Structured type expected**

A structured type is expected after the keyword `pack` in a type declaration.

**109 Function return must be scalar/pointer**

A function can only return a scalar or a pointer value. It cannot return a structured data type.  
- functions can't return records (can return pointer to record) → may want to change to a pointer

**110 Digit required after decimal point**

Standard Pascal requires that numeric constants have a digit after the decimal point. This only occurs when the `-ansi` compiler option is used.

**111 Cannot index string constants**

String constants can only be accessed in their entirety. They may not be indexed like arrays.

**112 Cannot dereference a function call**

A function that returns a pointer cannot be dereferenced in an expression.

**113 Multiple forward declarations**

Only one forward declaration is allowed for any procedure or function identifier.

**114 Second parameter required**

The second argument is missing in a call to the predefined `READ` or `WRITE` procedure.

**115 Only equality and inequality allowed on pointers**

Arithmetic may not be done on pointers.

**116 FOR loop variable assigned in procedure**

Standard Pascal does not permit the loop index to be altered in a function or procedure. The instance detected here occurs when the loop index is altered in the function or procedure since it is global to the routine. This only occurs if the `-ansi` compiler option is used.

**117 Value out of bounds**

An assignment is being made to a subrange variable that is outside of the subrange type. This only occurs if the `-ansi` compiler option is used.

**118 Array size undefined**

This occurs when the size of the array passed as the second argument to the predefined `ARGV` procedure is undefined.

**119 Set too large for representation**

The default number of elements in a set is 32. This may be extended to 256 by using the `-p4` compiler option.

**120 Packed variable may not be passed by reference**

A packed variable may not be passed as a variable actual parameter. See the discussion of packed and unpacked types in *Section 4.11*. This error only occurs when the `-ansi` compiler option is used.



**121 Variable expected**

This occurs when a type identifier is used on the left hand side of an assignment statement.

**122 External declaration only allowed at top level**

Only identifiers declared at the outermost level of a program or separately compiled module may be declared external.

**123 Type size exceeds implementation limit**

The length of a type is zero (the null string is illegal), or the size of a constant exceeds the value allowed for an integer, real, or floating constant.

**130 Internal Compiler Error <number>**

This is generated when an internal consistency check within the compiler has failed. Please send this message, program source code and other pertinent material to Microway.

*- check to make sure your identifiers are spelled correctly  
- check for non-standard Pascal (Sun) use*

**171 Ran out of string space**

There are too many characters in the identifier names and character strings to store in the string table. Space for this table is dynamically allocated, with a maximum of roughly one-half megabyte. Get around this restriction by breaking the program into smaller routines and compiling them separately.

**173 End of line found in string**

A character string is missing its terminating apostrophe.

**176 Include nested too deeply**

INCLUDE files may be nested 16 levels deep.

**177 Illegal character**

A character was encountered that is not valid in the context in which it occurred. For example,  
a := 1 \ 2;

**180 End of file found in IF**

An IF statement has not been terminated with a semicolon. This has caused the remaining portion of the program to be read in as a comment.

**181 Preprocessor expression must be constant**

Preprocessor expressions must evaluate to a constant.

**182 Unmatched #endif**

A preprocessor #IF, #IFDEF, #IFNDEF does not have a matching #ENDIF.

**183 Too many parameters for a macro**

The maximum number of parameters to a macro is 64.

**184 Illegal preprocessor command**

The identifier following the # symbol is not recognized as a preprocessor command.

**185 End of file found in comment**

A comment has not been terminated with its matching symbol. This has caused the remaining portion of the program to be read in as a comment.

**186 #defines nested too deeply**

Macros can be nested to a maximum level of 32.

**188 Wrong number of params in macro call**

The number of arguments in a macro call does not correspond to the number of parameters in the macro definition.

**191 Warning: Cannot take the address of this object**

It is not possible to take the address of a numeric constant, a type identifier, or an array name. To get the address of an array, take the address of the array indexed by its lower bound.

**200 redeclaration of: <name>**

Standard Pascal requires that the identifiers in the program heading be unique. This is caused by a duplicate name, probably a file name, in the program statement. This error only occurs when the -ansi compiler option is used.

**201 Label not defined: <label>**

The label specified by <label> has been declared but not defined in the program, or not used with a statement. Note that the label does not have to appear as the target of a GOTO statement for this error to occur.

**202 Type not declared <t>**

The type specified by <t> has been used in a forward declaration but not been declared.

**203 Used before defined in scope <t>**

The identifier specified by <t> has been previously defined in an outer scope level, (so that it is global to some routines), and then used and redefined in another scope level. This is equivalent to having an identifier with two different meanings in the same scope level, which is not allowed in Pascal.

This also occurs in the definition of a pointer type where the type being pointed to is separated from the type being defined. For example:

```
type ptr = ^t;

procedure p;
begin
end; {p}

type t = {type definition}
```

**204 Parameter not declared: <p>**

The parameter specified by <p> was used in the program statement and not declared in the body of the program. This error only occurs when the `-ansi` compiler option is used.

**205 Parameter is not a variable: <p>**

The parameter specified by <p> was used in a program statement, then later declared as something other than a variable, for example, a type. This error only occurs when the `-ansi` compiler option is used.

**206 Forward procedure not defined: <p>**

A body procedure or function that was declared forward was not found. This error only occurs when the `-ansi` compiler option is used.

**209 No such field in this record: <f>**

A record name is being qualified with an identifier that has not been declared to be part of this record.

**211 Undefined symbol: <s>**

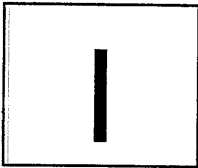
The identifier specified by <s> has not been previously declared.

**212 Undefined type: <t>**

The specified type identifier was used that was not previously defined. This message normally occurs in the parameter list of a procedure or function.

**217 Duplicate field: <f>**

The specified field names occur more than once in a record definition.



# Index

#DEFINE 95  
#ELSE 96  
#ENDIF 96  
#IF 96  
#IFDEF 96  
#ifndef 96  
#INCLUDE 95  
#LINE 97  
#UNDEF 95  
\_errno 120  
ABS 79  
Access 102  
Acos 103  
Acosf 104  
Acosh 104  
Actual parameter 64  
Aliasing 64, 74  
ARCTAN 79  
ARGC 79  
ARGV 80  
Array  
    multi-dimensional 42  
Array, component 33  
Array, index 33  
Array Type 33  
Arrays, multi-dimensional 33  
Asin 104  
Asinf 105  
Asinh 105  
Assignment compatible 48, 51, 55, 58  
Atan 105  
Atan2 106  
Atan2f 106  
Atanf 107  
Atanh 107  
Atof 107  
Atoi 108  
Atol 108  
Bcmp 109  
Bcopy 110  
Block 11  
BOOLEAN Scalar Type 29  
Bufcpy 111  
Buffer variable 71, 83, 86  
Buffer variables  
    variable 37  
Bzero 111  
C library functions  
    \_errno 120  
    access 102  
    acos 103  
    acosf 104  
    acosh 104  
    asin 104  
    asinf 105  
    asinh 105  
    atan 105  
    atan2 106  
    atan2f 106  
    atanf 107  
    atanh 107  
    atof 107  
    atoi 108  
    atol 108  
    bcmp 109  
    bcopy 110  
    bufcpy 111  
    bzero 111  
    cabs 112  
    calloc 113  
    ceil 113  
    clearn 114  
    clock 115  
    clrndpex 115  
    cosf 115  
    cosh 115  
    coshf 116  
    date\_ 116  
    difftime 117  
    dosdat 117  
    dostim 117  
    ecvt 118  
    erf 119  
    erfc 119  
    execl 120  
    execle 120  
    execv 121  
    exit 121  
    fabs 121  
    fcvt 121  
    ffs 122  
    filln 122  
    floor 123  
    fmod 124  
    frexp 125  
    frexpf 125  
    gamma 125  
    gcvt 126  
    getdat 126  
    getenv 126  
    gettim 127

hypot 127  
idate\_ 128  
index 128  
isalnum 130  
isalpha 130  
iscntrl 130  
isdigit 130, 132  
isgraph 130  
isinf 131  
islower 131  
isnan 131  
isprint 131  
ispunct 131  
isspace 131  
isupper 132  
j0 132  
j1 133  
jn 133  
ldexp 134  
ldexpf 135  
log 135  
log10 135  
log10f 136  
logf 136  
mapdev 136  
memchr 138  
memcmp 138  
memcpy 139  
memmove 139  
memset 139  
mktemp 139  
mktime 140  
modf 140  
perror 141  
pow 141  
powf 142  
racos 142  
raise 142  
rand 143  
rasin 143  
ratan 144  
ratan2 144  
rcos 145  
rcosh 146  
remove 146  
rename 146  
rexp 147  
rfrexp 147  
rindex 147  
rldexp 148  
rlog 148  
rlog10 148  
rpow 149  
rsin 149  
rsinh 149  
rsqrt 150  
rtan 150  
rtanh 151  
sec\_100\_ 151  
secsds\_ 152  
sinf 153  
sinh 153  
sinhf 153  
sprintf 154  
sqrtf 157  
srand 157  
sscanf 158  
strcat 161  
strchr 162  
strcmp 163  
strcoll 164  
strcpy 164  
strcspn 165  
strerror 166  
strftime 166  
strindex 167  
strlen 168  
strncat 169  
strncmp 170  
strncpy 171  
strpbrk 172  
strrchr 172  
strrindex 173  
strsave 174  
strspn 175  
strstr 175  
strtod 175  
strtok 175  
strtol 176  
strtoul 176  
strxfrm 176  
swab 177  
system 178  
tan 179  
tanf 179  
tanh 180  
tanhf 180  
time\_ 180  
timedate\_ 181  
tmpnam 182  
tolower 182  
toupper 182  
y0 182  
y1 183  
yn 184  
Cabs 112  
Calloc 113  
Case selector 52  
Ceil 113  
CHAR 30  
CHR 81  
Clearn 114  
Clock 115  
Clrndpex 115

- Compile time error messages 185
- Constant 26
- Control variable 55
- COS 81
- Cosf 115
- Cosh 115
- Coshf 116
- Data type 23
- Date\_ 116
- Declaration order 14
- Difftime 117
- DISPOSE 82
- DOS Interrupts
  - INT 21h 129
- Dosdat 117
- Dostim 117
- Dynamic variable 84
- Dynamic variables 36
- Ecvt 118
- End of line 86
- Entire variable 55
- EOF 82, 83, 86
- EOLN 82
- Equal precedence 46
- Erf 119, 123
- Erfc 119, 123
- Errno 120, 141
- Evaluation order 46
- Execl 120
- Execle 120
- Execv 121
- Exit 121
- EXP 83
- Expressions
  - boolean expressions 47
- EXTERNAL 14
- Fabs 121
- Fcvt 121
- Ffs 122, 123
- File 71
- File pointer 83, 86, 87, 88
  - pointer 37
- File variable 82
- Filln 122, 123
- Floor 123
- Fmod 124
- Formal function 64
- Formal parameter 64
- Formal procedure 64
- Formal routine 65
- FORWARD 66
- Frexp 125
- Frexpf 125
- Function 63
- Function call 48
- Function, results 66
- Functions
  - side effects 48
- Gamma 125
- Gcvt 126
- GET 43, 73, 83
- Getdat 126
- Getenv 126
- Gettim 127
- Heading, function 63
- Heading, procedure 63
- Hole in scope 13, 67
- Hypot 127
- Idate\_ 128
- If statement 54
- Index 128
- INPUT 15
- INTEGER 30
- Isalnum 130
- Isalpha 130
- Iscentrl 130
- Isdigit 130, 132
- Isgraph 130
- Isinf 131
- Islower 131
- Isnan 131
- Isprint 131
- Ispunct 131
- Isspace 131
- Isupper 132
- J0 132
- J1 133
- Jn 133
- Label 54, 56
- Labs 134
- Lazy evaluation 73, 87
- Ldexp 134
- Ldexpf 135
- LN 83
- Log 135
- Log10 135
- Log10f 136
- Logf 136
- Macro 95
- Mapdev 136
- MAXINT 31
- Memchr 138
- Memcmp 138
- Memcpy 139
- Memmove 139
- Memset 139
- Mktemp 139
- Mktime 140
- Modf 140
- Mutually recursive 66
- NDP status word 115
- NEW 43, 84
- NIL 37
- ODD 84

- Operator 46
- Operators
  - boolean operators 47
  - relational operators 47
- Optimizations 47
- ORD 84
- OUTPUT 15
- PACK 33, 84
- PACKED 33
  - records 35
- Packed array 85, 91
- PAGE 85
- Parameter, actual 58
- Parameter, formal 58
- Parameter list 64
- Parameter transmission 64
- Parameters 48
- Parenthesis 46
- Pass by reference 65
- Pass by value 65
- Pass by variable 65
- Perror 141
- Pointer 23, 43
- Pointer data type 23
- Pointers
  - file 43
- Pow 141
- Powf 142
- Precedence 46
- Precision 29
- PRED 55, 85
- Procedure 63
- Procedure call 58
- PUT 43, 73, 86
- Racos 142
- Raise 142
- Rand 143, 149
- Range 29
- Rasin 143, 149
- Ratan 144, 149
- Ratan2 144, 149
- Rcos 145, 149
- Rcosh 146, 149
- READ 76, 86
- READLN 86
- Record 34, 35
- Remove 146
- Rename 146
- RESET 72, 87
- REWRITE 72, 87
- Rexp 147
- Rfrexp 147
- Rindex 147
- Rldexp 148
- Rlog 148
- Rlog10 148
- ROUND 88
- Routine 64
- Rpow 149
- Rsin 149
- Rsinh 149
- Rsqrt 150
- Rtan 150
- Rtanh 151
- Scalar 26
- Scalar data type 23
- Sec\_100\_ 151
- Secnds\_ 152
- Selector, case 52
- Set
  - set constructor 48
- Setndpsw 152
- SIN 89
- Sinf 153
- Sinh 153
- Sinhf 153
- Sprintf 154
- SQR 90
- SQRT 89
- Sqrtf 157
- Sscanf 158
- Statement
  - assignment 51
  - case statement 52
  - compound statement 53
  - empty statement 54
  - for statement 55
  - goto statement 56
  - if statement 57
  - repeat statement 59
  - while statement 60
  - WITH statement 61
- Statement, procedure 63
- STATIC 14
- Strcat 161
- Strchr 162
- Strcmp 163
- Strcoll 164
- Strcpy 164
- Strcspn 165
- Strerror 166
- Strftime 166
- Strindex 167
- String 9
- Strlen 168
- Strncat 169
- Strncmp 170
- Strncpy 171
- Strong typing 24
- Strpbrk 172
- Strrchr 172
- Strrindex 173
- Strsave 174
- Strspn 175

Strstr 175  
Strtod 175  
Strtok 175  
Strtol 176  
Strtoul 176  
Structured data type 23  
Strxfrm 176  
Subprogram 63  
SUCC 55, 90  
Swab 177  
System 178  
Tag field 35  
Tan 179  
Tanf 179  
Tanh 180  
Tanhf 180  
Textfile 38  
Time\_ 180  
Timedate\_ 181  
Tmptnam 182  
Tolower 182  
Toupper 182  
TRUNC 90  
Type  
    Assignment compatibility 24  
    compatibility and conversions 24  
    compatible 24  
    enumerated 26  
    file 37  
    identical 24  
    implicit type conversion 25  
    pack  
        unpack 39  
    pointer 36  
    record 34  
    set 27  
    subrange 27  
    text 38  
Type identifier 23  
UNPACK 33, 91  
Unpacked array 85, 91  
Value parameters 65  
Variable 42  
    buffer 43  
    component 41  
    control variable 55  
    data type 24  
    entire 41  
    file referencing 43  
    indexed 41  
Variable, control 55  
Variable parameter 65  
Variant record 84  
WRITE 76, 92, 94  
WRITELN 92  
YO 182  
Y1 183



