U. Allendörfer D. Cordes

PASCAL–XSC

User's Guide

Numerik Software GmbH
Baden-Baden, Germany

PASCAL–XSC

User's Guide

The programming language PASCAL–XSC (PASCAL eXtension for Scientific Computation) significantly simplifies programming in the area of scientific and technical computing. PASCAL–XSC provides a large number of predefined data types with arithmetic operators and predefined functions of highest accuracy for real and complex numbers, for real and complex intervals, and for the corresponding vectors and matrices. Thus, PASCAL–XSC makes your computer much more powerful at the arithmetic level.
Through an implementation in C, compilers for PASCAL–XSC are available for a large variety of computers such as personal computers, workstations, mainframes and supercomputers. PASCAL–XSC provides modules, an operator concept, functions and operators of arbitrary result type, overloading of functions, procedures and operators, dynamic arrays, access to subarrays, rounding control by the user, and accurate evaluation of expressions. The language is particularly suited for the development of numerical algorithms which deliver highly accurate and automatically verified results. A number of problem-solving routines with automatic result verification have already been implemented. PASCAL–XSC contains Standard PASCAL. It is immediately usable by PASCAL programmers. PASCAL–XSC is easy to learn and ideal for programming education.

Address: Numerik Software GmbH
P.O. Box 2232
W-7570 Baden-Baden
Federal Republic of Germany

Contents

List of Figures

Chapter 1

Introduction

1.1 Typography

Throughout this document the following typing conventions are applied in order to emphasize and distinguish certain words, names, or paragraphs.

italic types	are used for emphasized terms within the text.
"quoted italic types"	are used for C names within the text.
bold-faced types	are used for PASCAL–XSC word symbols like **begin** and **module** within the text.
slanted types	are used for PASCAL–XSC standard names like *integer* and *real* within the text.
`typescript`	is used for PASCAL–XSC program listings, options, and input and output protocols.

Citations are always given in the form [*nr*] where *nr* is the corresponding entry number in the reference list.

1.2 The PASCAL–XSC System

The complete PASCAL–XSC system consists of

the PASCAL–XSC configuration program,
the PASCAL–XSC manager programs,
the PASCAL–XSC compiler,
the PASCAL–XSC standard modules, and
the PASCAL–XSC runtime system library.

The PASCAL–XSC compiler does not contain a code generator for machine specific code. Instead of this, readable C code (conforming to the ANSI C standard [1]) is generated. The idea is to withdraw the high-level language PASCAL–XSC compiler from machine dependencies as far as possible and to rely on the capabilities of existing C compilers to be generators of efficient machine code which take into account low-level routines of operating systems and machine dependent properties.

The required C compiler for the compilation of the generated C code as well as the linker for the generation of an executable program are not part of the PASCAL–XSC system.

Both the PASCAL–XSC compiler and the PASCAL–XSC runtime system are completely implemented in C. Due to differing linkage conventions and differing methods of argument passing, the PASCAL–XSC runtime system library must be compiled with the same C compiler which is applied to the C code generated by the PASCAL–XSC compiler.

In this document, the PASCAL–XSC system and its default settings are described. Local installations may differ from the described PASCAL–XSC system in altered default settings and specific hardware dependencies. These differences are gathered in *local configuration guides*. Possible changes in the default values are marked in this document by a reference to the appropriate *local configuration guide* which applies to the individual installation.

1.3 The PASCAL–XSC Language

The programming language PASCAL–XSC is completely described in [4] and will not be presented in this manual.

The current implementation of a PASCAL–XSC compiler comprises the complete language PASCAL–XSC with some minor exceptions described in Appendix A *Deviations*. On the other hand, some additional features are introduced for the sake of generalization of concepts. The description of these extensions can be found in Chapter 5 *PASCAL–XSC Implementation*.

Chapter 2

Installation

The process of installing the PASCAL–XSC system depends on the operating system which is available on the target machine. Thus, a general guide for installation is not possible. Nevertheless, the installation of the PASCAL–XSC system on a multi-user operating system gives an impression on the tasks that have to be done. Due to its wide distribution, the operating system UNIX is selected for a demonstration of the process of installation in section 2.1. The following sections 2.2 and 2.3 do not depend on a specific operating system.

2.1 Installation on a UNIX System

This chapter describes, how to install the PASCAL–XSC system on a machine with operating system UNIX[1]. For details about the installation for other operating systems refer to your *local configuration guide*.

Goals of the installation are

- to provide easy access to the PASCAL–XSC system for all users who have access to the target machine,
- to enable each user to create and modify his individual PASCAL–XSC configuration, and
- to establish copy protection and write protection for the PASCAL–XSC system.

The following steps shall be performed in the stated order.

1. **Create directories for the PASCAL–XSC system.**
    - Create a main directory for the installation of the complete PASCAL–XSC system, for example:

[1]UNIX is a registered trade mark of Bell Laboratories

```
mkdir /pxsc
```

- Create a subdirectory for the executable programs of the PASCAL–XSC system, for example:

  ```
  mkdir /pxsc/bin
  ```

- Create a subdirectory for all remaining files of the PASCAL–XSC system, for example:

  ```
  mkdir /pxsc/sys
  ```

  In this document, this subdirectory will be called "system directory" or synonymously $PXSC_SYS.

All the created directories must have execution permission for all users. This can be established by using system commands, for example:

```
chmod  go=x  /pxsc  /pxsc/bin  /pxsc/sys
```

2. **Define environment variables.**
   Add the path of the subdirectory for the executable programs of PASCAL–XSC to the PATH variable of all authorized users.

   Define an environment variable called PXSC_SYS which holds the path name of the "system directory" of the PASCAL–XSC system for all users.

   - *The value of PXSC_SYS must end with a path delimiter character "/".*
   - *The name PXSC_SYS must by written with upper case letters.*

   For example using the bourne shell:

   ```
   PXSC_SYS=/pxsc/sys/
   export PXSC_SYS
   ```

   For example using the C shell:

   ```
   setenv PXSC_SYS /pxsc/sys/
   ```

   For example on DOS:

   ```
   set PXSC_SYS=\pxsc\sys\
   ```

   Place the appropriate commands for the definition of the environment variables PATH and PXSC_SYS in a general profile or the profile of each user.

3. **Copy executable programs.**
All executable programs of the PASCAL–XSC system must be copied to the created subdirectory for executable programs. The following table lists the executable program files of the PASCAL–XSC system.

	executable programs
pxsc	PASCAL–XSC compiler
mxsc	batch manager
dxsc	interactive manager
exsc	short listing generator
psclist	long listing generator
l2p	listing to source file conversion
pxsccfg	configuration program
dismod	discompiler for interface files
splitmod	program to split a PASCAL–XSC module
mod2lib	shell-procedure to create a library from a module
mvmod	shell-procedure to move a PASCAL–XSC module

All these files must have execution permission and should be read and write protected. This can be established by using system commands, for example:

```
chmod  go=x  $PXSC_SYS../bin/*
```

4. **Copy all other files.**
All other files of the PASCAL–XSC system must be copied to the subdirectory $PXSC_SYS. The following table lists the remaining files of the PASCAL–XSC system.

	help files, include files, libraries
cxsc.hlp	help file of the configuration program
dxsc.hlp	help file of the interactive manager
errtext.hlp	compiler messages
info.txt	runtime help file
o_msg1.h	runtime messages
p88.env	configuration file
p88rts.ii	runtime include file
p88rts.h	runtime interface file
rts.a	runtime library of PASCAL–XSC

	compiled standard modules	
stdmod.o	stdmod.h	stdmod.mod
iostd.o	iostd.h	iostd.mod
x_intg.o	x_intg.h	x_intg.mod
x_real.o	x_real.h	x_real.mod
x_strg.o	x_strg.h	x_strg.mod

	compiled arithmetic modules	
i_ari.o	i_ari.h	i_ari.mod
c_ariaux.o	c_ariaux.h	c_ariaux.mod
c_ari.o	c_ari.h	c_ari.mod
ci_ari.o	ci_ari.h	ci_ari.mod
mv_ari.o	mv_ari.h	mv_ari.mod
mvi_ari.o	mvi_ari.h	mvi_ari.mod
mvc_ari.o	mvc_ari.h	mvc_ari.mod
mvci_ari.o	mvci_ari.h	mvci_ari.mod

	compiled problem solving modules	
lss_aprx.o	lss_aprx.h	lss_aprx.mod
lss.o	lss.h	lss.mod
ilss.o	ilss.h	ilss.mod

All these files must have read permission and should be write and execution protected. This can be established by using system commands, for example:

```
chmod  go=r  $PXSC_SYS*
```

5. **Create p88rts.i**
   If the runtime include file "p88rts.i" does not exist (this may be possible for the first installation), rename "p88rts.ii" to "p88rts.i":

   ```
   cd $PXSC_SYS
   mv p88rts.ii p88rts.i
   ```

6. **Set write protection.**
   Establish write protection on the files "p88rts.i" and "p88.env", for example by means of the system command

   ```
   chmod -w p88rts.i p88.env
   ```

   in order to protect them against accidential deleting.

All the following sections and chapters are not restricted to UNIX systems.

2.2 Environment Variables

The environment variable PXSC_SYS holds the path name of the "system directory" and should be the same for all users. Nevertheless, it may be altered locally by each user in order to switch to an alternate installation of the PASCAL–XSC system.

Further environment variables may be defined locally by each user of the PASCAL–XSC system.

PXSC_SYS	holds the name of the"system directory" of the installed PASCAL–XSC system (usually the value of this environment variable is not changed by users)
PXSC_USR	holds the name of a directory bearing an individual configuration file "p88.env" and further modules (must end with a path delimiter character!)
PXSC_EDIT	holds the command string for the invocation of the favorite editor program, the user prefers to use
PXSC_LIB	holds a suffix string for the link command containing linker options, object file names, and library names

All names of these environment variables must be written in capital letters.

The variables PCSC_SYS and PXSC_USR are investigated by the PASCAL–XSC compiler, the subroutines of the PASCAL–XSC runtime system library, the interactive manager, and the batch manager. PXSC_EDIT and PXSC_LIB are used by the interactive manager and the batch manager only.

2.3 Testing the Installation

If you have installed the PASCAL–XSC system on a multi-user operating system, for instance on a UNIX system, then you must be logged in as a user other than the owner of the PASCAL–XSC system, in order to test the correct access to the system.

If you have installed the PASCAL–XSC system on a single-user operating system, for instance on a DOS system, then switch to a test directory, e.g.,

```
cd \test
```

in order to test the correct access to the system.

Define the environment variable PXSC_EDIT with your favorite editor command, e.g.,

```
set PXSC_EDIT=c:\dos\edit.exe
```

Enter the command

```
dxsc hello.p
```

If you get an answer like

```
command not found
```

from the operating system, make sure that the command path which is usually represented by the environment variable PATH contains the name of the directory holding the executable programs of the PASCAL–XSC system. If you get the answer

```
Help file ...dxsc.hlp not found.
```

enter the letter 'q' followed by the RETURN key to quit the interactive manager. Make sure that the environment variable PXSC_SYS is defined (see 2.1) and that all files in this directory have read permission.

Otherwise, you will see the main menu of the interactive manager "dxsc" of the PASCAL–XSC system.

In order to test the on-line-help facility of the manager, enter the command 'h'. You will get information about all available commands of the program "dxsc".

Enter 'q' to quit the program. For additional testing perform the actions described in section 3.1.

Chapter 3

Compiling a PASCAL–XSC Program

3.1 First Try

This section describes, how to compile and run a simple PASCAL–XSC program.

When working on a DOS system, define the environment variable PXSC_EDIT with your favorite editor command, e.g.,

```
set PXSC_EDIT=c:\dos\edit.exe
```

Enter the command

```
dxsc hello.p
```

You will see the main menu of the interactive manager ”dxsc” of PASCAL–XSC.

Each command of the interactive manager must be terminated by the RETURN key.

Enter the letter ’`e`’ in order to start the editor program and then enter the following PASCAL–XSC program by means of the editor.

```
program hello ( output ) ;
begin
   writln ('Hello') ;
end.
```

In order to see what happens in case of errors, misspell the procedure name *writeln* as shown above.

Leave the editor by typing the appropriate editor command.

Enter the command ’`c`’ to compile and link program ”hello.p”.

You see the error message

```
    3:   writln ('Hello') ;
         1
 Error at 1: Identifier not declared.
```

Enter the editor again, correct the error by inserting the missing letter 'e', and recompile the program with the 'c' command.

If you see the message

```
Linkage complete.
```

you may enter the command 'rr' to run the linked program without command line arguments.

The output string

```
Hello
```

is displayed on your terminal screen.

3.2 PASCAL–XSC Batch Manager

The batch manager may be used to compile a single PASCAL–XSC module or to compile and link a single PASCAL–XSC program. As the batch manager does not request any user input, this manager may be used in background processes.

The command syntax of the batch manager call is given in Figure 3.1.

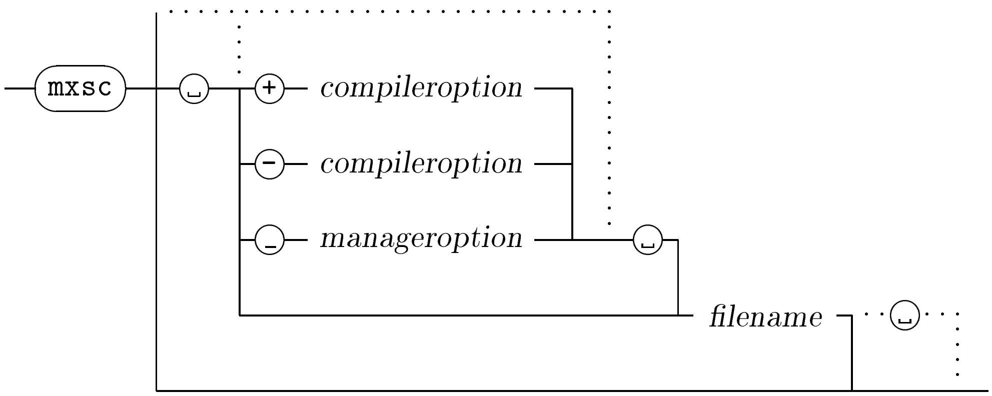


Figure 3.1: Command syntax of the batch manager call

Compiler options are prefixed by a '`+`' or '`-`' and are passed to the PASCAL–XSC compiler. See 3.6 *PASCAL–XSC Compiler Options* for a detailed description.

Manager options are prefixed by the underscore character '`\_`'. Manager options are:

`\_c` *C compiler option*	add a further C compiler option
`\_e`	edit the source file before compilation
`\_x`	run the compiled program after linkage

If the '`_e`' option is applied, the value of the environment variable PXSC_EDIT is used to call the editor. So each user may use his favorite editor. In case of the manager option '`_e`', the PASCAL–XSC source file specified by *filename* need not exist, but it must exist after leaving the editor.

If the '`\_x`' option is applied, the program will be called immediately after linkage without any program parameters. See 4 *Running PASCAL–XSC Programs* for an alternative call of PASCAL–XSC programs with program parameters.

The *filename* immediately following the '`mxsc`' command must be the file name of a PASCAL–XSC source file. The file name extension of the source file name (normally '`.p`') may be omitted. The file name must not start with any of the symbols '`+`', '`-`', or '`\_`'.

3.3 PASCAL–XSC Interactive Manager

The interactive manager "dxsc" improves the development cycles of PASCAL–XSC programs allowing repeated calls of the editor, the PASCAL–XSC compiler, and the compiled and linked PASCAL–XSC programs. Therefore, this manager is called

PASCAL–XSC development system.

The interactive manager communicates with the user. Since the interactive manager is a portable program written in ANSI C, each manager command must be terminated by an end-of-line character (normally generated by pressing the RETURN key), and the manager does not allow the use of any unportable features such as cursor keys, function keys, or mouse control.

All manager commands may be written in lower case letters as well as in upper case letters.

3.3.1 Single Program Development

Enter the command '`dxsc`' followed by a file name. If a PASCAL–XSC source file does not exist, you *must* specify the file name with extension (normally '`.p`'), in contrast to the batch manager. If the PASCAL–XSC source file already exists, then the extension may be omitted.

If you do not specify a file name, you are requested to do so. Subsequently, the current file name of the "dxsc" command is denoted by *filename*.

e **Edit**

After the main menu of the interactive manager appeared on the screen, enter '`e`' to start the editor for *filename*. If you prefer another editor, you may change or add the definition of the environment variable PXSC_EDIT in your profile, see your *local configuration guide* and your operating system manual. The value of PXSC_EDIT must be the command name of the editor call and will be concatenated with the file name before it is passed to the command interpreter.

After editing the current PASCAL–XSC source file, leave the editor with the appropriate editor command.

Press RETURN, if you are requested to do so.

c **Compile**

Enter '`c`' to start the PASCAL–XSC compiler for *filename*.

Let us assume, the PASCAL–XSC source file contains errors that can be detected by the compiler.

The PASCAL–XSC compiler reports errors in a short form by providing an error number, a line number, and a column number indicating the position of the error in the PASCAL–XSC source file. This information may be used, whenever the listing generator fails for some reason.

After the message

`Pascal compilation unsuccessful.`

the short listing generator is called automatically by the interactive manager. The listing is displayed on the terminal. See 3.4 *PASCAL–XSC Listing* for an explanation of the listing.

You may now enter the editor with the 'e' command and correct the error in the PASCAL–XSC source file. Leave the editor and recompile your program using the 'c' command. Repeat this cycle until you see the message

`linkage complete`

or the message

filename `is module: no linker call.`

l — **L**ist edit

If the error listing is too long to fit on the screen, you may use the 'l' command to correct the errors. The 'l' command calls the long listing generator. The long listing contains the full text of the PASCAL–XSCsource file with interspersed error messages. The editor will be called automatically to edit the listing file and to revise the PASCAL–XSC source.

In the long listing, all PASCAL–XSC source lines start with a blank character and all message lines start with an exclamation mark '!'. In order to find error messages, search for lines starting with '!' by means of an appropriate editor command. When revising PASCAL–XSC source lines, do not type a '!' at the beginning of a line. Do not delete a '!' from the beginning of a line.

After leaving the editor, a new PASCAL–XSC source file may be constructed from the listing file by deleting all lines starting with '!'. You are asked, whether you want to do this. The reconstruction of a PASCAL–XSC source file from a listing file is done by the executable program "l2p".

Due to some strange handling of the tabulator character, the indentation of the original PASCAL–XSC source file may be damaged, when using the 'l' command.

r — **R**un

Enter the 'r' command of the manager to execute the PASCAL–XSC program. The letter 'r' may be followed by program parameters, see 4.1 *PASCAL–XSC File Variables* and 4.2 *PASCAL–XSC Runtime Options*. If no program parameters are specified immediately after the letter 'r', you are requested by the interactive manager program to enter the program parameters. Enter an empty line, if the program needs no parameters. Enter the 'rr'command (re-run), if you want to run the program with the same parameters as specified in a previous run of the program in the actual session of the interactive manager.

q **Quit**

Enter the 'q' command to leave the interactive manager program "dxsc".

3.3.2 Multiple File Development

You may specify several file names in the command line on calling "dxsc". On a UNIX system you may use "wild cards", for example:

```
dxsc *.p
```

The last file name specified will be the current file name. The current file is the file, that will be edited, compiled, and executed. The other file names are stored by the manager. The manager "dxsc" can handle at most 10 file names. You may switch between file names by means of the 'f' command of the manager.

f **File name**

Enter the letter 'f' followed by the end-of-line character (pressing the RETURN key). You will see a sub-menu with the file names stored by "dxsc". These file names are identified by digits and letters. The letter 'p' identifies the last main program, the letter 'm' identifies the last PASCAL–XSC module, and the letter 'o' identifies a file not known to be a module or a main program (others).

Entering 'f' followed by a letter or a digit suppresses the sub-menu. With the 'fn' command of "dxsc" you may add a file name to the list of file names stored by the interactive manager.

If you specify more than 10 file names, you are requested to drop previously specified file names. A sub-menu will appear and by entering digits you can specify file names which shall be dropped.

e **Edit**

You may edit a file other than the current file by entering the letter 'e' followed by the file name. This does not change the setting of the current file.

3.3.3 Further Tools

h **Help**

The interactive manager contains an on-line-help facility that informs the user about all manager commands. Enter 'h' or '?' for further help.

d **Display toggle**

If you are tired of repeatedly looking at the main menu and the

```
<<< Press the RETURN key to continue >>>
```

prompt, enter the 'd' command of "dxsc". To display the main menu again, enter the 'd' command a second time.

The 'd' command toggles between displaying and suppressing the main menu.

y **sYstem**

The system command 'y' of the interactive manager allows processing of commands of the operating system without leaving the manager. Enter the letter 'y' followed by a system command to execute the system command. Enter the letter 'y' followed by the end-of-line character to enter a command line interpreter; for example the bourne shell in UNIX or "COMMAND.COM" in DOS. Use an appropriate system command to return to the interactive manager program.

c **Compile**

Compiler options for the PASCAL–XSC compiler and the C compiler may be provided with the 'c' command immediately following the letter 'c'. See 3.6 for an explanation of PASCAL–XSCcompiler options. More details about changing the default options of the PASCAL–XSC compiler can be found in 3.7 *PASCAL–XSC Configuration*.

Options for the C compiler in use must be separated from PASCAL–XSC compiler options by a semicolon ';' in the 'c' command. Refer to the manuals of your C compiler for an explanation of C compiler options.

For example, if you want to merge the text of the PASCAL–XSC source file into the generated C code (compiler option '+m') and you want to add symbolic information to the generated object code produced by the C compiler (C compiler option '-g'), use the "dxsc" command

```
c +m;-g
```

If you want to change linker options, you have to use the 'b' command.

b **Batch**

The 'b' command creates a batch file (DOS) or shell procedure file (UNIX) which can be used for the C compilation and the linkage of a PASCAL–XSCprogram.

The generated file is called "lxsc.bat" and may be used

- to display the C compiler call and the linker call being used, or
- to modify the C compiler call or the linker call, or
- to recompile a program, whenever the compilation with the PASCAL–XSC or C compiler fails, because of insufficient memory.

In order to modify a call, the following steps must be performed:

1. Run the PASCAL–XSC compiler by means of the 'c' command. You may abort the C compilation or the linkage by means of an appropriate break key (if available) since the results of these steps are not needed.
2. Enter the 'b' command of the interactive manager program.
3. Edit the generated batch file "lxsc.bat" by means of the manager command

   ```
   e lxsc.bat
   ```

4. Run the batch file by means of the manager command

   ```
   y lxsc.bat
   ```

In order to compile a program in case of insufficient memory, perform the following steps:

1. Leave the manager with the 'q' command.
2. Start the PASCAL–XSC compiler separately, see 3.5 *PASCAL–XSC Compiler*.
3. Enter the interactive manager "dxsc" again.
4. Enter the 'b' command of "dxsc".
5. Run the generated batch file by means of the manager command

   ```
   y lxsc.bat
   ```

   or leave "dxsc" and run the batch file separately.

In order to compile a module in case of insufficient memory, you have to call the PASCAL–XSC compiler and the C compiler separately.

Since the 'b' command is intended to create a link command, the 'b' command fails after compiling a module.

The 'b' command may also fail after compiling a main program. In order to avoid this error either

- switch off the 'rm' option, see 3.7.2 *Configuration Program*, or
- abort the C compilation or the linkage by means of an appropriate break key, or
- use the "dxsc" command

  `y pxsc` *options filename*

  for the compilation, see 3.5 *PASCAL–XSC Compiler*, or
- enter the 'b' command before the compilation. Ignore the warning message of the manager program. Compile the program with the 'c' command. Enter the 'b' command a second time.

m **Make**

In the current implementation the 'm' command is identical with the 'c' command.

p **Print**

The 'p' command may be used to print a file on a printer. Entering 'p' followed by end-of-line prints the current file. Typing 'p' followed by a file name prints the specified file without changing the current file.

The 'p' command depends on the current installation and the operating system. Refer to your *local configuration guide*.

3.4 PASCAL–XSC Listing

The short listing generator is called "exsc". After a PASCAL–XSC compilation "exsc" may be called without program parameters. The listing is displayed on the terminal. The short listing contains only the source lines in which errors are detected. The source lines are preceded by their line numbers.

The long listing generator is called "psclist". The long listing contains all source lines. They are not preceded by line numbers. The long listing is intended to be edited. See the 'l' command of "dxsc" in 3.3.1.

Under each erroneous source line, there is a line with position digits, that indicate positions in the preceding line of source text. Position digits, that are not separated by spaces, must not be interpreted as numbers. For example in the listing

```
4:     x:= a+b ;
       12   34
```

the symbols 'x', ':=', '+', and 'b' caused some kind of error. Do *not* read 12 as "twelve" and 34 as "thirtyfour".

The corresponding error messages of the PASCAL–XSC compiler are listed below the line of position digits. For example

```
Error at 1: Identifier not declared.
```

refers to the symbol 'x' in line 4, *not* to line 1 or column 1.

Error messages starting with 'Check at' are internal compiler errors. The error text is meaningless for users. If such an error message still occurs after correcting all PASCAL–XSC errors, you should inform the compiler development group about this error.

In some of these internal error messages, the data type of an actual parameter or an actual operand is given. The notations used to represent these data types are listed in Figure 3.2.

Notation	Description
typename	named data type
ARRAY	unnamed static array type
DYNAMIC	unnamed dynamic array type
RECORD	unnamed record type
FILE	unnamed file type
SET	unnamed set type
^*typename*	unnamed pointer type with the name of the refered data type
^	data type of NIL
(*name*,...)	unnamed enumeration type with the name of the first constant
..	unnamed subrange type
,...	indicates that only the data types of the first four parameters are listed
unknown	indicates that an error occurred in the declaration of the actual parameter or operand

Figure 3.2: Notation of data types in listings

In order to get more evident error messages, use named data types, e.g., by introducing *type* definitions in your PASCAL–XSC source file.

In some cases, the type name is succeeded by a level number, indicating the static level of the definition of the type name:

Level	Description
0	a predeclared name such as *real* or *char*
1	an imported name,
2	a name declared on program level or a non-global name declared on module level,
3	a name declared on subroutine level, or
4,...	an inner subroutine level name.

The compiler has to split calls of *read*, *readln*, *write*, and *writeln* into several individual calls. If a position digit is placed below `read`, `readln`, `write`, or `writeln`, then the error message refers to the first parameter group. If a position digit is placed below a comma, then the error message refers to the parameter group immediately following the comma.

3.5 PASCAL–XSC Compiler

Warning: The PASCAL–XSC compiler should not be called directly by the user, because it is a frequent error to forget the required call of the C compiler.

The name of the PASCAL–XSC compiler is ”pxsc”. A call of ”pxsc” without any program parameters displays explanations of all PASCAL–XSCcompiler options and their current default settings, which are defined in the configuration file, see 3.7 *PASCAL–XSC Configuration*.

The command syntax of the PASCAL–XSC compiler call is given in Figure 3.3.

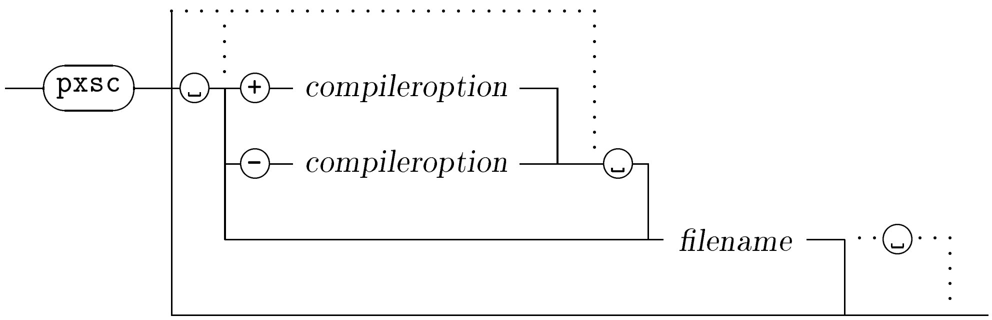


Figure 3.3: Command syntax of PASCAL–XSC compiler call

Compiler options are prefixed by a ’+’ or ’-’. See 3.6 *PASCAL–XSC Compiler Options* for a detailed description.

The optional argument *filename* is the name of the PASCAL–XSC source file with or without extension (normally ’.p’). The file name must not start with ’+’ or ’-’. If several file names are specified, only the last one will be used.

If options are specified but no file name is given, the PASCAL–XSC source is read from standard input. In this case the merge option ’+m’ is disabled and a PASCAL–XSC listing cannot be produced.

3.6 PASCAL–XSC Compiler Options

PASCAL–XSC compiler option are formed by one of the symbols ’+’ or ’-’ followed by an option name without intervening blank characters.

In the current implementation only the first letter of an option name is significant. The ’+’ character switches the compiler option ”on”; the ’-’ character switches the compiler option ”off”. This overrides the default setting of the option in the configuration file, see 3.7 *PASCAL–XSC Configuration*.

Note to UNIX users: the PASCAL–XSC compiler options must not be collected since, e.g., '`-vw`' is equivalent to '`-v`' and not to '`-v -w`'.

3.6.1 Display Options

v **Verbose**

'`+v`' displays useful information:

- the version number of the PASCAL–XSC compiler
- the name of the PASCAL–XSC source file
- the path name of the configuration file being used
- the path name of imported modules
- whether or not the interface of a module has changed
- the name and line number of the subroutine just being compiled

`Compiling` *subroutine name* `at line` *line number*

'`-v`' (quiet) suppresses all the information listed above. Warnings and error messages will still be displayed.

If neither '`+v`' nor '`-v`' is specified, all the information listed above will be displayed except for the subroutine information.

l **List file**

'`+l`' directs all error messages and warnings of the PASCAL–XSCcompiler to the listing file (default file name extension '`.lst`'). Do not use this option when using the batch manager or interactive manager, because both the short and long listing generator rewrite the listing file.

Using the options '`-v +l`' suppresses all terminal output, except for the number of errors, if any.

w **Warnings**

'`-w`' suppresses all warnings given by the compiler. Only a message

n `warnings suppressed`

will indicate that a total number of *n* warnings are suppressed.

3.6.2 Code Generation Options

x **indeX check**

'`+x`' enables the generation of runtime checks, such as index checks, range checks, and pointer checks.

'`-x`' suppresses the generation of runtime checks, thus improving the processing speed. Use '`-x`' only if you are sure, that the program will not raise any error that may be detected by a runtime check.

n **line Numbers**

'`+n`' supports source line information in the generated program. In case of a runtime error, a dynamic function trace back and line number information refering to PASCAL–XSC source files may be given by the runtime system.

'`-n`' suppresses the generation of line number and trace back information, thus improving the processing speed.

c **Code generation**

'`+c`' forces code generation, even if compilation errors occurred. A subsequent C compilation of the generated C code may fail in case of detected compilation errors.

'`-c`' suppresses code generation and interface file generation.

If neither '`+c`' nor '`-c`' is specified, code will be generated, if the PASCAL–XSC compiler does not detect any errors.

s **Source directory**

'`+s`' directs the compiler output to the directory where the source file is found.

'`-s`' directs the compiler output to the current directory.

3.6.3 Debug Options

t **Terminal**

'`+t`' directs the generated C code to standard output instead of to the output file with default extension '`.c`'.

m **Merge**

'`+m`' merges the lines of the PASCAL source file as C comment lines into the generated C code.

$\boxed{\textbf{r}}$ **Rename**

'`-r`' (not rename) preserves the PASCAL–XSC identifier names. This improves debugging and readability of the generated C program, because the original PASCAL–XSC names are used in the C source file. The C compilation and the linkage may fail when using '`-r`'.

$\boxed{\textbf{d}}$ **Dump**

'`+d`' (internal dump) produces a lot of output, that is meaningless to normal users.

3.7 PASCAL–XSC Configuration

When a compilation is started, the PASCAL–XSC compiler reads the PASCAL–XSC configuration file "p88.env", which contains default settings and system dependencies.

The configuration file contains

- default settings of compiler options,
- non-command-line compiler options,
- file name extensions, and
- path names of interface files.

In the "system directory" $PXSC_SYS, a general configuration file is available for all users.

Each user may create individual configuration files without influencing other users. This can be done by generating an individual configuration file in the "user directory" PXSC_USR for general access to a user-defined configuration, or by placing a configuration file in the current directory. Thus, in each directory an individual configuration of the PASCAL–XSC system is possible. For details refer to the following section.

3.7.1 Search Algorithm

The PASCAL–XSC compiler searches for the configuration file "p88.env" in several directories according to the following sequence of steps:

Step 1: Search for the configuration file in the current directory. If the configuration file is not found, then the search is continued with the next step.

Step 2: If the environment variable PXSC_USR is defined, search in the directory given by the value $PXSC_USR. The value of PXSC_USR may be defined individually by each user and must end with a path delimiter character ('/' in UNIX, '\' in DOS). The name PXSC_USR must be written in upper case letters. If the configuration file is not found, then the search is continued with the next step.

Step 3: If the environment variable HOME is defined, then $HOME is concatenated with the name of a "fixed user directory", which is defined in the executable programs of the PASCAL–XSC system (for example "/pxsc/"). Refer to the *local configuration guide* for the setting of the default value. The search is done in the directory which results from the concatenation. If HOME is not defined, then the search is done in the "fixed user directory". If the configuration file is not found, then the search is continued with the next step.

Step 4: If the environment variable PXSC_SYS is defined, then the search is done in directory $PXSC_SYS. The value of PXSC_SYS should be the same for all users. The value of PXSC_SYS must end with a path delimiter character. The name PXSC_SYS must be written in upper case letters. If the configuration file is not found, then the search is continued with the next step.

Step 5: Search is continued in the "fixed system directory" which is defined during the installation of the PASCAL–XSC system (for example '`/pxsc/sys/`'). Refer to the *local configuration guide* for the setting of the default value. If the configuration file is not found, then the search is continued with the next step.

Step 6: Use a default configuration. This might not fit to the operating system or C compiler in use.

This algorithm (except for Step 6) is also used, when searching for interface files of a PASCAL–XSC module and for runtime files (see also 3.8 *Module Concept* and 4.2 *PASCAL–XSC Runtime Options*).

3.7.2 Configuration Program

A configuration file can be created, displayed, and modified by the configuration program. The name of this program is "pxsccfg".

The program "pxsccfg" is an interactive program, that contains an on-line help facility similar to "dxsc" described in 3.3 *PASCAL–XSC Interactive Manager*.

The configuration program "pxsccfg" is invoked without program parameters. Before starting "pxsccfg" change the current directory to be the directory, where you want to store the new configuration file.

The program "pxsccfg" searches for an existing configuration file as described in 3.7.1. The modified configuration file is always written into the current directory, no matter where the original configuration file was found.

After reading an old configuration file, "pxsccfg" displays the actual configuration and a menu on standard output *"stdout"*. The configuration may be modified by "pxsccfg" commands.

h **Help**

The 'h' command of "pxsccfg" enters the on-line help facility.

d **Display**

The 'd' command displays the modified configuration and a menu.

u **Update**

The 'u' command writes a modified configuration to the configuration file in the current directory without leaving "pxsccfg".

e **Exit**

The 'e' command writes a modified configuration to the configuration file in the current directory and leaves "pxsccfg".

k **Kill**

Discards the configuration file that is found in the current directory. Do not apply this command to the system directory.

q **Quit**

The 'q' command leaves "pxsccfg" without saving the configuration file.

o **+** **−** **Option**

The default settings of command line options may be

set	with the	'+' command,
reset	with the	'-' command,
toggled	with the	'o' command.

See 3.6 *PASCAL–XSC Compiler Options* for a description of command line options.

The following default settings should *not* be changed:

```
-l  c  +x  -d  -t  +r
```

The "pxsccfg" commands '+c', '-c', 'oc', '+v', '-v', 'ov' disable the normal usage of the *code* and *verbose* option, respectively. Use the command 'nc'and 'nv', respectively, in order to switch to the normal usage of these options (without preceding '+' or '-').

n Non-command-line

Non-command-line options are options, that can not be controlled via command line options, but only via the configuration program.

The non-command-line option '`+rm`' saves disk space by removing all output files which are no longer needed. '`-rm`' preserves all output files of the compiler.

The non-command-line option '`+src`' directs output files of the compiler to the directory of the source file. '`-src`' directs output files to the current directory.

The non-command-line option '`+proto`' should be set, if the C compiler uses new-style prototypes (see ANSI C standard and your C compiler manual). '`-proto`' must be used, if the C compiler does not accept new-style prototypes.

The non-command-line option '`+y`' uses the path name of the "system directory" in order to search for the runtime include file "p88rts.i". If the option '`-y`' is selected, the path name of the runtime include file"p88rts.i" is the path name specified by the '`r`' command of the configuration program.

The setting of these options may be toggled with the '`nr`', '`ns`', '`np`', and '`ny`' commands of "pxsccfg", respectively.

t Type name

The '`t`' command of "pxsccfg" offers the possibility to change the default file name extensions. The extension of PASCAL–XSC source files '`.p`', the extension of listing files '`.lst`', and the extension of executable program files may be changed by users depending on the operating system. Changing other file name extensions might result in fatal errors.

i Interface

With the '`i`' command the user specifies a directory path where PASCAL–XSC modules may be found. See also 3.8 PASCAL–XSC Module Concept.

r Runtime interface

The '`r`' command sets the directory path of the runtime include file "p88rts.i" of the runtime system. This command automatically sets the non-command-line option '`-y`'. A user explicitly defines a path name, if he uses his own runtime system or PASCAL–XSC cross compilations are intended for target machines with different directory hierarchy.

3.8 The Module Concept

The PASCAL–XSC language identifies modules by identifier names only. These identifier names are associated with file names. Two problems derive from this concept:

1. *the distinction between upper case and lower case letters*, and
2. *the specification of a directory path.*

Some operating systems like UNIX distinguish between upper case letters and lower case letters in file names. In order to avoid any problems concerning file names, it is recommended to the user to use always the same case in typing identifier names at the following positions:

- in the **module** clause of the PASCAL–XSC source file,
- in the **use** clause of the PASCAL–XSC source file, as well as
- in the command line of the compiler call or manager call, and
- in the '`fn`' command of the interactive manager program, see 3.3.2.

In the current implementation, module names must differ within the first 6 characters.

The language PASCAL–XSC does not allow any specification of path names in the **use** clause, because this would result in unportable PASCAL–XSC programs.

Since it is possible to compile PASCAL–XSC modules in different directories or to place modules into different directories, the compiler searches several directories for a module, whenever a **use** clause occurs. The following search algorithm is performed:

Step 1: Search in the current directory. If not found, continue with the next step.

Step 2: Search in the directory of the PASCAL–XSC source file just being compiled. If not found, continue with the next step.

Step 3: Search in the directory, specified in the configuration file, see the '`i`' command in 3.7.2 *Configuration Program*. If not found, continue with the next step.

Step 4 to 7: The compiler uses Step 2 through Step 5 of the search algorithm for the configuration file described in 3.7.1. If not found, the compilation is aborted.

The compiler "pxsc" searches for interface files, which normally have the file name extension '`.mod`'. The PASCAL–XSC compiler and the managers "mxsc" and "dxsc" assume that the accompanying C include file (with file name extension '`.h`') and the object file of the compiled module are placed in the same directory. Therefore, it is mandatory to move the interface file, the include file, and the object file together whenever a compiled PASCAL–XSC module is moved from one directory to another directory. The command "mvmod" may be used to do this. The shell procedure "mvmod" expects one module name as first argument and one target directory as second argument.

The interface file of a PASCAL–XSC module contains the **global** declarations in a compressed and unreadable form. The program "dismod" displays the contents of an interface file. When "dismod" is called without parameters, the program explains its usage. Normally "dismod" is called with the name of the interface file to be displayed. The file name extension '`.mod`' may be ommitted.

In order to reduce the size of executable PASCAL–XSC programs, it is necessary to transform the object files of some PASCAL–XSC modules into object libraries. The command "mod2lib" may be used for this. "mod2lib" is a shell procedure and may not be available for a specific operating system.

"mod2lib" calls the program "splitmod" which splits the C source file, generated by the PASCAL–XSC compiler, into several C source files each containing one PASCAL–XSC subroutine. All these C files must be compiled and their object files must be put into a library. These actions are done by "mod2lib", if available.

For each PASCAL–XSC module that has been imported by a PASCAL–XSC main program, the manager "mxsc" or "dxsc" looks for a file with the file name extension '`.0`' (digit zero) and a file with the extension of library files, for example '`.a`' in UNIX.

If the '`.0`' file exists, the object file of the corresponding module is <u>not</u> linked with the main program. If the '`.0`' file does not exist and a library file exists, the library is linked instead of the object file of the module.

If the files with extension '`.0`' and '`.a`' exist, they must be placed in the directory of the interface file ('`.mod`') of the PASCAL–XSC module.

3.9 Summary of File Usage

This section contains a list of all program and data files which are used or created by the executable programs of the PASCAL–XSC system and the C compiler.

"$\Leftarrow$"	indicates an input file,
"$\Rightarrow$"	indicates an output file,
"$\Leftrightarrow$"	indicates an input/output file,
"??"	indicates check for existence only.

The symbol @ is to be replaced by the name of the program or module just being compiled. If an explicit path precedes the name of the program or module, then the target directory for the generation of output files depends on the setting of the non-command-line option '`src`' and the '`s`' compiler option. The '`s`' compiler option overwrites the non-command-line option '`src`'. Refer to 3.7.2 *Configuration Program* and 3.6.2 *Code Generation Options*.

The symbol $*$ is to be replaced by the names of imported modules.

`./`	file is placed in the current directory
`!/`	file is placed in the "system directory" $PXSC_SYS
`?/`	file is searched for according to the implemented search algorithms described in 3.7.1 and 3.8.

Here, the file name convention of UNIX is used.

<u>pxsccfg</u>		<u>configuration program</u>
⇔	?/p88.env	configuration file
⇐	!/cxsc.hlp	help file
<u>pxsc</u>		<u>PASCAL–XSC compiler</u>
⇐	?/p88.env	configuration file
⇐	@.p	PASCAL–XSC source file
⇐	?/*.mod	imported interface files
⇔	@.mod	exported interface file, modules only
⇒	@.c	generated code
⇒	@.h	C interface file, modules only
⇔	./modmod.tmp	temporary interface file
⇒	./errmess.tmp	error message file, in case of errors or warnings only
⇒	./linkinfo.tmp	linkage information file, programs only

Due to the usage of "errmess.tmp" and "linkinfo.tmp", it is not possible to run the compiler more than once concurrently in the same current directory. "modmod.tmp" is used only, if the C system does not support temporary files.

<u>mxsc and dxsc</u>		<u>manager programs</u>
⇐	?/p88.env	configuration file
⇐	!/dxsc.hlp	help file of interactive manager
??	@.p	PASCAL–XSC source file
⇐	./linkinfo.tmp	linkage information file
⇒	./lxsc.opt	option file for link command, not UNIX
⇒	./lxsc.bat	batch file created by the "b" command
??	?/*.0	null file
??	?/*.a	object libraries
<u>exsc and psclist</u>		<u>listing generator programs</u>
⇐	./errmess.tmp	error message file
⇐	!/errtext.hlp	compiler messages
⇐	@.p	PASCAL–XSC source file
⇒	@.lst	PASCAL–XSC listing file

l2p		listing to source file converter
⇐	@.lst	PASCAL–XSC listing file
⇒	@.p	PASCAL–XSC source file
cc		C compiler
⇐	@.c	generated C code
⇐	@.h	C interface file, modules only
⇐	?/∗.h	C include files
⇐	!/p88rts.i	runtime include file
⇐	!/p88rts.h	runtime interface file
⇒	@.o	object file
cc		linkage
⇐	@.o	object file of main program
⇐	?/∗.o	object files of modules
⇐	?/∗.a	object file libraries
⇐	!/rts.a	runtime system library
⇒	@	executable program file
@		program execution
⇐	!/info.txt	runtime help file
⇐	!/o_msg1.h	runtime messages

Chapter 4

Running PASCAL–XSC Programs

4.1 PASCAL–XSC File Variables

Names of PASCAL–XSC file variables need not be listed as program parameters in the **program** clause. If there are names of file variables in the **program** clause, then an external file name may be given for each program parameter as command line argument. The maximum length of each file name including an optional directory path is restricted to 63 characters even though longer file names may be possible on certain operating systems. The association of file names (external devices) with file variables in the program parameter list is a three stage process:

- **Step 1: Keyword association**
  All file variables which are explicitly named as keywords immediately preceding a file name and separated by the symbol '`=`' are associated first. If no file name is given after the symbol '`=`', then a subsequent *reset* will assign to a *text* file variable the standard input device *"stdin"*, a subsequent *rewrite* will assign to a *text* file variable the standard output device *"stdout"*. A file variable, which is not a *text* file, must not be assigned to standard input or standard output.

- **Step 2: Positional association**
  The remaining command line arguments which are not runtime options (see 4.2 *PASCAL–XSC Runtime Options*) are associated from left to right with those file variables listed in the **program** clause which have not been assigned an external file name yet.

- **Step 3: Association by prompting**
  By default, there is no immediate prompting for an external file name if a program parameter has not been associated with a command line argument.

  If you want prompting at the beginning of the processing of a program, specify the runtime option '`-pr`', see 4.2 *PASCAL–XSC Runtime Options*. In this case, a prompt is displayed for each file variable that has not been assigned an external file name in **Step 1** or **Step 2**. The prompt is displayed on the standard output device *"stdout"* which demands the input of an external file name from standard input device *"stdin"*. There will be no prompting for standard file variables *input* and *output* which, by default, are associated with the standard devices *"stdin"* and *"stdout"*, respectively. The prompting devices *"stdout"* for message displaying and *"stdin"* for input of file names may be altered. Refer to the *local configuration guide* for current settings.

Runtime options are interpreted separately from the association of **program** parameters with command line arguments. All runtime options are identified by a special symbol (the default symbol is '`-`') in order to avoid mix-up with file name conventions. Refer to the *local configuration guide* for the currently implemented special symbol in use.

In Figure 4.1, examples for the association of external file names in the command line with file variables in the program parameter list are given.

An association of an external file name with a file variable is also possible during a PASCAL–XSC *reset* or *rewrite*. The following strategy is applied:

1. If a string specifying a file name is explicitly given as a second argument to *reset* or *rewrite*, then the file variable is associated with the external file specified by the string. If the file variable is of type *text*, then an empty string as second argument assigns the standard input device *"stdin"* or the standard output device *"stdout"* in *reset* and *rewrite*, respectively.

2. If no string is given as a second argument, then a previously assigned external file name is reused in *reset* and *rewrite*.

3. If there is no file name available for a file variable in a *reset* or *rewrite* which is also listed in the program clause, then a prompt is generated on *"stdout"* which demands the input of a file name from standard input *"stdin"*. The process of prompting is identical to that for command line arguments (see above).

   By default, there is no prompting for an external file name in case of local file variables, i.e., file variables that are not listed in the program clause. If you want prompting, specify the runtime option '`-tf`', see 4.2 *PASCAL–XSC Runtime Options*.

   Local file variables without a previously assigned file name in a *rewrite* are associated with a temporary file.

   Temporary files are generated in a temporary file directory (by default this is the current directory). Temporary file names have a length of 6 characters and are completed by the file name extension '`.tmp`'. The file name is constructed by the letter '`t`' and a 5 digit number padded with leading zeros. Temporary files will be removed if the program terminates without a PASCAL–XSC error message. Otherwise, they will be closed before leaving the program, thus freezing the file contents just before processing has been aborted by the PASCAL–XSC runtime system.

Note: In the current implementation, there is no mechanism which prevents multiple write operations by simultaneously running PASCAL–XSC programs.

A local file variable without a previously assigned file name in a *reset* causes a runtime error.

```
PROGRAM PROG(DATA,INPUT,OUTPUT,COPY);
VAR DATA : FILE OF INTEGER;
    COPY : TEXT;
BEGIN END.
```

`PROG file1 file2 file3 file4`

2.	`DATA`	is associated with	`file1`
2.	`INPUT`	is associated with	`file2`
2.	`OUTPUT`	is associated with	`file3`
2.	`COPY`	is associated with	`file4`

`PROG COPY=file4 file1`

1.	`COPY`	is associated with	`file4`
2.	`DATA`	is associated with	`file1`
3.	`INPUT`	is associated with	*"stdin"*
3.	`OUTPUT`	is associated with	*"stdout"*

`PROG -pr INPUT=file1 COPY=`

1.	`INPUT`	is associated with	`file1`
1.	`COPY`	is associated with	*"stdin"* in *reset* *"stdout"* in *rewrite*
3.	`DATA`	is prompted for a file name	
3.	`OUTPUT`	is associated with	*"stdout"*

`PROG`

3.	`DATA`	association done in *reset* or *rewrite*	
3.	`INPUT`	is associated with	*"stdin"*
3.	`OUTPUT`	is associated with	*"stdout"*
3.	`COPY`	association done in *reset* or *rewrite*	

Figure 4.1: Example for the association of file variables with command line arguments

4.2 PASCAL–XSC Runtime Options

The PASCAL–XSC runtime system provides certain options for debugging purposes and documentation. This section describes all runtime options which may be activated in an installation of the PASCAL–XSC system. Refer to the *local configuration guide* for further runtime options.

Runtime options are given as command line arguments. They should be distinguishable from file names according to the local file name conventions in order to avoid misinterpretations. Thus, names of runtime options contain a special character symbol at the first position of the option which is inconvenient in file name notations. The default symbol is a single dash '-' immediately preceding the option.

If an unknown runtime option is used as command line argument, then the argument string is assumed to be a file name. In order to avoid program break-down due to an invalid file name, the existence of the used runtime options should be checked, e.g., by using runtime option **-info**.

The following list explains all possible runtime options and what they do.

-cc **Constant Conversion**

Displays a warning message in case of an inexact conversion of *real* constant data or *real* input data from the decimal format to the internal floating-point number representation. No warning message is displayed for conversions with directed rounding.

-ieee **IEEE trap handling**

Toggles the default setting for the enabled status of IEEE exception handlers. Each of the five IEEE exceptions is characterized by one letter:

d	DIVISION BY ZERO
i	INVALID OPERATION
o	EXPONENT OVERFLOW
u	EXPONENT UNDERFLOW
x	INEXACT RESULT

If one or more letters are immediately following the header part **-ieee** of the option, the default status of the exception handler is toggled from 'enabled' to 'disabled' or from 'disabled' to 'enabled'. For instance, the runtime option **-ieeedu** toggles both the enabled status of the DIVISION BY ZERO and EXPONENT UNDERFLOW exception handlers. The default enabled status of the IEEE exception handlers in the PASCAL–XSC system is given in the following list.

DIVISION BY ZERO	enabled
INVALID OPERATION	enabled
EXPONENT OVERFLOW	enabled
EXPONENT UNDERFLOW	disabled
INEXACT RESULT	disabled

Note that this setting is valid only before processing the first PASCAL–XSC statement which may be placed in the initialization part of a module. Refer to Appendix D *IEEE Exception Handling Environment* for explicitly changing status settings within a PASCAL–XSC program.

-info **runtime INFOrmation**

Displays information about the PASCAL–XSC system and the currently processed PASCAL–XSC program on standard output device *"stdout"*. The information is either explicitly stored in the runtime system or will be read from the default file "info.txt" which is assumed to reside in the "system directory" of the PASCAL–XSC system (refer to 2 *Installation*). Program processing is terminated after displaying all available information.

Instead of the runtime option **-info**, the following alternative notations may be used:

?, **-?**, **-help**, **-h**, **/h**, and **/help**.

Note, that the character **?** in a command line argument may have special meaning in certain operating systems.

Extended notations for the runtime option **-info** are available:

-info:
-info:key
-info@file
-info:key@file

Here, **key** stands for the leading part of a keyword, and **file** stands for a file name to be searched for instead of "info.txt". There must be no intervening blank characters in the extended runtime option.

Keywords in **file** are consecutive sequences of characters which do not contain blank characters and are preceded by a colon '**:**'. Each keyword must be left-adjusted on a single line in **file**. For each matching keyword, all lines in **file** up to the next keyword (or the end of the file) are displayed. The argument string **key** matches with keywords if it coincides to full length with the leading characters of keywords in **file**. If there is no **key** after the colon '**:**' in the runtime option, then all keywords defined in **file** are listed.

-nn Normalized Numbers

The value determined by the predecessor and successor routines *pred* and *succ* for *real* floating-point numbers is forced to be a normalized value. The default setting allows the generation of denormalized floating-point number values. Refer to 5.3.1.2 *REAL* for an explanation of terms.

-pp Program Parameters

Display the names of the PASCAL–XSC file variables which are listed in the program parameter list of the **program** statement of the activated program. Program processing is terminated after displaying the names of the file variables in the program parameter list.

-pr parameter PRompting

Enables the prompting for external file names for program parameters at the beginning of the processing of a program if there are less file names as command line arguments than there are names of file variables in the program parameter list.

-sd System Directory

Displays the path of the "fixed system directory" that is set during the installation of the runtime system.

The extended syntax of the runtime option

-sd:path

can be used to alter the path of the "fixed system directory" to **path** for the time of processing of the program. The colon '**:**' immediately following the option name is ignored and is no part of **path**. The length of string **path** is restricted to 63 characters.

-sz Signed Zero

Enables the generation of a minus sign if the IEEE value for a negative zero is detected in an output operation.

-tb Trace Brief

The output caused by activating runtime option **-tr** is reduced in case of recursive functions if this runtime option is used. Moreover, there will be no function tracing for PASCAL–XSC runtime routines.

-tf **no Temporary Files**

Enables the prompting for file names for local file variables in *reset* and *rewrite* if there is no file name explicitly specified as second argument and no previous association of an external file name has been done.

-tr **TRace**

Enables the generation of a function trace during the processing of the PASCAL–XSC program, provided the PASCAL–XSC compiler option '+n' (see 3.6.2 *Code Generation Options*) was activated. Any available information about entering and leaving a PASCAL–XSC procedure, function, and operator is displayed on the standard error device *"stderr"*. The nesting level of the procedure, function, and operator calls is shown by "indentation". The number of lines of output may be reduced in case of recursive functions when runtime option **-tb** is activated too.

```
program tr_test(output);

function fak(n : integer) : integer;
begin
   if n<=1 then fak:=1 else fak:=n*fak(n-1);
end;

begin
   writeln('fak(5) = ',fak(5));
end.
```

After compilation of program `tr_test` with compiler option '+n', the processing of

```
tr_test -tr
```

yields the following output on a terminal screen if *"stderr"* is redirected to *stdout"*.

```
--- FAK in tr_test.p entered.
--- +FAK in tr_test.p entered.
--- +.FAK in tr_test.p entered.
--- +..FAK in tr_test.p entered.
--- +...FAK in tr_test.p entered.
--- +...FAK in tr_test.p terminated.
--- +..FAK in tr_test.p terminated.
--- +.FAK in tr_test.p terminated.
--- +FAK in tr_test.p terminated.
--- FAK in tr_test.p terminated.
fak(5) = 120
```

-ud User Directory

Displays the path of the "fixed user directory" that is set during the installation of the runtime system.

The extended syntax of the runtime option

-ud:path

can be used to alter the path of the "fixed user directory" to **path** for the time of processing of the program. The colon '**:**' immediately following the option name is ignored and is no part of **path**. The length of string **path** is restricted to 63 characters.

-vn Version Number

Displays the version identification of the PASCAL–XSC runtime system on standard output device *"stdout"* before the processing of the PASCAL–XSC program starts.

Chapter 5

PASCAL–XSC Implementation

In this chapter, technical details about the implementation of the PASCAL–XSC language are described. Section headings and section numbering are chosen analogously to Chapter 2 of the language reference [4]. Thus, section 5.3.2.4 in this document refers to section 2.3.2.4. *Dynamic Arrays* in [4]. Section headings marked by * are given for completeness but do not contain any additional text.

All statements and default values in this chapter refer to the hardware-independent version of the PASCAL–XSC system. Specific details on hardware dependencies and altered default settings are explained in individual *local configuration guides* available for each installation.

5.1 Basic Symbols

The length of PASCAL–XSC source file lines may be unlimited. However, all symbols and comments must start within the first 255 positions of a source line in order to get correct positional information for error messages.

Neither identifier names nor literal constants must be longer than 255 characters.

The character '`$`' is no basic symbol. The notation of hexadecimal *integer* constants is not implemented. The keyword **global** must be written `global`, `GLOBAL`, or `Global` when used after the keyword **use**.

5.2 Identifiers

Identifiers must not be longer than 255 characters. The PASCAL–XSC compiler does not distinguish lower and upper case letters in identifiers. All lower case letters are converted to upper case letters. Exceptions are those identifiers which are preceded by the keywords **module**, **use**, **use global**, and **external**.

There are two reasons for these exceptions:

1. The names of PASCAL–XSC modules are associated with file names. Since operating systems like UNIX distinguish lower and upper case letters, the PASCAL–XSC compiler must distinguish lower and upper case letters in names of modules.

2. After the keyword **external** the entry name of a C function may be specified. Since the programming language C distinguishes between lower and upper case letters, the PASCAL–XSC compiler must distinguish lower and upper case letters in external entry names.

The length of **module** names may be restricted by the maximal length of file names that can be handled by the operating system. In the current implementation, module names must differ within the first 6 characters.

The length of **external** entry names may be restricted by the maximal length of entry names that can be distinguished by the linker. Note, that external entry names of user-defined routines must not coincide with entry names and global variables used by the PASCAL–XSC and C runtime systems. All entry names and global variable names of the PASCAL–XSC runtime system are constructed as indicated by the following summary:

one lower case letter	`a`,. . .,`z`
the underscore character	`_`
2 to 4 characters from	`a`,. . .,`z`, `0`,. . .,`9`, `_`

Thus, PASCAL–XSC runtime entry names and global variable names are formed by 4 to 6 characters. A complete list of external names may be obtained by scanning the object files of the runtime system. It is not recommended to use runtime routines without setting up a correct PASCAL–XSC runtime environment.

5.3 Constants, Types, and Variables

The maximal length of string constants is restricted to 255 and by the length of a source line, i.e., the maximal length of a string constant depends on the editor which is used to generate the PASCAL–XSC source file.

The address of variables as well as the address of components of records depend on the alignment conventions of the C compiler which is used for the compilation of the generated C code. The keyword **packed** is ignored in the current implementation of the PASCAL–XSC compiler.

PASCAL–XSC type names	C type names
boolean	*"a_bool"*
char	*"a_char"*
dotprecision	*"d_otpr"*
integer	*"a_intg"*
real	*"a_real"*
string	*"s_trng"*

Figure 5.1: PASCAL–XSC simple types and related C types

5.3.1 Simple Types

Every PASCAL–XSC simple data type is represented by an individual C data type. The C type names are defined by *"typedef"* statements in file "p88rts.h". The relations between PASCAL–XSC simple types and C data type names are given in Figure 5.1.

In the following additional sections, some notes are made on the data formats used to implement the PASCAL–XSC simple types. It must be emphasized that the described data formats in this document are used by the default implementation which may be altered in an individual installation that uses hardware support. Refer to the *local configuration guide* for hardware dependencies.

5.3.1.1 *integer*

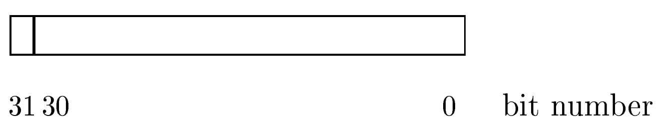


Figure 5.2: *integer* data format

The data type *integer* consists of all values from a consecutive sequence of integers. There is only one signed integer format supported by PASCAL–XSC. For an object n of type *integer* there holds

$$\Leftrightarrow maxint \Leftrightarrow 1 \leq ord(n) \leq maxint.$$

The corresponding C data type is named *"a_intg"*. A variable of type *integer* is 32 bits long and requires four bytes of storage.

The largest positive integer value representable by the data type *integer* is denoted by *maxint*.

$$maxint = 2^{31} \Leftrightarrow 1 = 2147483647.$$

The representation of values of type *integer* is in two's complement notation. Figure 5.2 sketches the bit ordering of an *integer* value.

The notation of hexadecimal *integer* constants is not implemented.

Unsigned integer operations are not supported by the PASCAL–XSC compiler. The PASCAL–XSC module *x_intg* provides operators and functions for the manipulation of bits of an object of type *integer*. Refer to 6.4 *Module x_intg*.

5.3.1.2 *real*

The data type *real* consists of all floating-point numbers and special values which are specified by the IEEE standard [3] for the double floating-point number format. The corresponding C data type is named *"a_real"*. A variable of type *real* is 64 bits long and requires eight bytes of storage. In Figure 5.3, a sketch of the IEEE double floating-point data format is given.

	1	11	52	bit count
(r)	s	e	m	
	63	62 52	51 0	bit number

$$
\mathrm{r} = \begin{cases}
(\Leftrightarrow 1)^s \cdot 1.m \cdot 2^{e-1023} & 1 \le e \le 2046 & \text{normalized} \\
(\Leftrightarrow 1)^s \cdot 0.m \cdot 2^{-1022} & e = 0, m \neq 0 & \text{denormalized} \\
(\Leftrightarrow 1)^s \cdot 0 & e = 0, m = 0 & \text{signed zero} \\
(\Leftrightarrow 1)^s \cdot \infty & e = 2047, m = 0 & \text{signed infinity} \\
\text{NaN} & e = 2047, m \neq 0 & \text{not a number}
\end{cases}
$$

Figure 5.3: IEEE double floating-point format

The largest and the smallest positive real value representable by the data type *real* are denoted by *maxreal* and *minreal*, respectively. The constants *maxreal* and *minreal* are defined in module *x_real*.

The floating-point number format uses a binary representation (base B=2) of the mantissa digits. There is one implicitly defined hidden bit in the representation of normalized and denormalized numbers, thus making a total length of the mantissa m of 53 bits. The exponent field e occupies 11 bits. For normalized floating-point numbers according to the notation in Figure 5.3, the range of exponent values is specified by the maximum exponent $e_{max} = 2046 \Leftrightarrow 1023 = 1023$ and the minimum exponent $e_{min} = 1 \Leftrightarrow 1023 = \Leftrightarrow 1022$. The sign field s occupies 1 bit.

Special values named "not a number" (NaN) may be signaling or quiet. By default the PASCAL–XSC runtime assumes that a signaling NaN is identified by bit 51 of the

Hexadecimal Representation	Decimal Value
0000000000000001 *minreal*	$4.9406564584124654 \cdot 10^{-324}$ Shortest decimal number which yields the specified hexadecimal value with rounding to the nearest floating-point value. Decimal to binary conversion of this number with rounding towards $-\infty$ yields 0.0.
7FEFFFFFFFFFFFFF *maxreal*	$1.7976931348623158 \cdot 10^{308}$ Shortest decimal number which yields the specified hexadecimal value with rounding to the nearest floating-point value. Decimal to binary conversion of this number with rounding towards $+\infty$ yields $+\infty$.

Figure 5.4: *real* constants *minreal* and *maxreal*

representation of the floating-point number being set. A quiet NaN is identified by bit 51 of the representation of the floating-point number being not set. The sign of a NaN is ignored. The described interpretation of bit 51 is the default setting and may be altered when hardware operations are used for the current installation. Refer to the *local configuration guide* for the actual setting.

A quiet NaN is generated instead of a *real value* if an exception occurred in an operation that does not produce any reasonable arithmetic result due to the exception (invalid operation) and the trap handler is disabled for this exception. The structure of a generated quiet NaN is given in Figure 5.5.

bit	63	= 0 or 1	(sign is ignored)
bit	62-52	= 2047	(all bits are set)
bit	51	= 0	(identifies quiet NaN)
bit	32-50	= 0	(reserved)
bit	0-31	= *integer*	(exception code)

Figure 5.5: Structure of a quiet NaN

Representations of special values of the *real* data format are listed in Figure 5.6.

The order in which the bytes of a *real* value are stored depend on the storage conventions used by the hardware. This is most important in those cases where hardware support for arithmetic operations is used. Refer to the *local configuration guide* for details.

Refer to 6.5 *Module* x_real for additional *real* routines.

Hexadecimal representation	Decimal value
`FFF0000000000000`	$\Leftrightarrow\infty$
`FFEFFFFFFFFFFFFF`	$\Leftrightarrow 1.7976931348623158 \cdot 10^{308}$ Shortest decimal number which yields the specified hexadecimal value with rounding to the nearest floating-point value. Decimal to binary conversion of this number with rounding towards $-\infty$ yields $-\infty$.
`BFF0000000000000`	$\Leftrightarrow 1.0$
`8010000000000000`	$\Leftrightarrow 2.2250738585072013 \cdot 10^{-308}$ Shortest decimal number which yields the specified hexadecimal value with rounding to the nearest floating-point value. Decimal to binary conversion of this number with rounding towards $+\infty$ yields the smallest denormalized number.
`8000000000000001`	$\Leftrightarrow 4.9406564584124654 \cdot 10^{-324}$ Shortest decimal number which yields the specified hexadecimal value with rounding to the nearest floating-point value. Decimal to binary conversion of this number with rounding towards $+\infty$ yields -0.0.
`8000000000000000`	$\Leftrightarrow 0.0$
`0000000000000000`	0.0
`0000000000000001` *minreal*	$4.9406564584124654 \cdot 10^{-324}$ Shortest decimal number which yields the specified hexadecimal value with rounding to the nearest floating-point value. Decimal to binary conversion of this number with rounding towards $-\infty$ yields 0.0.
`0010000000000000`	$2.2250738585072013 \cdot 10^{-308}$ Shortest decimal number which yields the specified hexadecimal value with rounding to the nearest floating-point value. Decimal to binary conversion of this number with rounding towards $-\infty$ yields the largest denormalized number.
`3FF0000000000000`	1.0
`7FEFFFFFFFFFFFFF` *maxreal*	$1.7976931348623158 \cdot 10^{308}$ Shortest decimal number which yields the specified hexadecimal value with rounding to the nearest floating-point value. Decimal to binary conversion of this number with rounding towards $+\infty$ yields $+\infty$.
`7FF0000000000000`	$+\infty$

Figure 5.6: Special *real* values

5.3.1.3 *boolean*

The data type *boolean* consists of the values *false* and *true* with

$$ord(false)=0 \quad \text{and} \quad ord(true)=1.$$

The corresponding C type is named *"a_bool"*. A variable of type *boolean* requires one byte of storage.

5.3.1.4 *char*

The data type *char* consists of all values from a prescribed collating sequence (see for instance the ASCII collating sequence in Figure E.1 on page 98). For an object h of type *char* there holds

$$0 \leq ord(h) \leq 255.$$

The corresponding C type is named *"a_char"*. A variable of type *char* is 8 bits long and requires one byte of storage.

5.3.1.5 Enumeration Types

The maximal number of enumeration constants is restricted by the maximal value for *"enum"* values of the C compiler in use, i.e., for an ANSI C compiler this number is restricted by the value of *"INT_MAX"* defined in *"limits.h"*.

5.3.1.6 *dotprecision*

The values that can be represented by the data type *dotprecision* consist of all real values which may be generated by exact dot product evaluations

$$\sum_{i=1}^{n} a_i \cdot b_i$$

for *real* values a_i and b_i which are not NaNs, and any number n within the range

$$1 \leq n \leq maxint.$$

The corresponding C data type is named *"d_otpr"*. Throughout an installation of the runtime system, the variables of type *dotprecision* for the evaluation of dot products with *real* operands in the IEEE binary double format (refer to 5.3.1.2 *real*) have a fixed length with at least

$$\begin{array}{rcl} 2\cdot \mathrm{e}_{max} + \lceil log_2(maxint)\rceil + 2\cdot(\text{ mantissa length} \Leftrightarrow \mathrm{e}_{min}) &=& \\ 2\cdot 1023 + \quad 31 \quad + 2\cdot(\quad 53 \quad + 1022) &=& \\ 2077 \quad + \quad 2150 &=& 4227 \text{ bits.} \end{array}$$

The term $\lceil log_2(maxint) \rceil$ results from the maximum number of carry bits that may occur during the evaluation of a dot product of vectors with at most *maxint* components.

Additional information like invalid operands and technical flags (temporary value indicator, signed zero indicator) may be stored within an object of type *dotprecision* which will increase the number of reserved bits by a fixed amount.

The component of a **file** type and the components of the variant part of a **record** must not contain a component of type *dotprecision* neither directly nor indirectly.

5.3.2 Structured Types

The keyword **packed** is ignored in type specifications and declarations.

5.3.2.1 Arrays

In PASCAL–XSC static and dynamic arrays may be defined. The element with smallest index has the smallest address of all elements of an array. For multiple dimensional arrays, the elements of the last dimension occupy consecutive storage positions. The maximal number of indices for a static array may be restricted by the C compiler in use.

The component type of an array type must not be a **file** type.

5.3.2.2 Subarrays*

5.3.2.3 Access to Index Bounds*

5.3.2.4 Dynamic Arrays

The maximum number of indices (dimension) of a dynamic array is restricted to 255. There is no error message in case of violating this upper bound.

Dynamic arrays are represented by a C structure which contains administrative information and a C pointer to the allocated array of elements of the dynamic array. The administrative information needed for the implementation of a dynamic array consists of

- the dimension of the array,
- the size of one array element in bytes,
- the total number of array elements,
- the lower bound of each index range,
- the upper bound of each index range,

- the stride value of each index range,
- the "subarray" indicator, and
- the "temporary array" indicator.

The stride value of the n^{th} dimension is the distance between array elements with consecutive index values in the n^{th} dimension. Thus, according to the storage convention of PASCAL the stride of the last dimension of a main array is always 1.

5.3.2.5 Strings*

5.3.2.6 Dynamic Strings

The PASCAL–XSC data type *string* consists of a (possibly empty) sequence of characters of type *char*. Two kinds of variables of type *string* are distinguished.

(A) *string* variables with a specified maximum length, e.g.,

```
VAR f :  string[9];
```

Variables of this type can hold at most the specified number of characters. The standard function *maxlength* returns the specified maximum length in the variable definition (`maxlength(f)` is 9). The standard function *length* yields the length of the actually stored string of characters which may be set explicitly by standard procedure *setlength* within the range from `0` to `maxlength(f)`, or by *string* assignment. An ALLOCATION exception may occur if the requested storage space is not available when the *string* variable is defined.

(B) *string* variables with an unspecified maximum length, e.g.,

```
VAR v :  string;
```

Variables of this type have a default maximum length of *maxint* which, of course, stands for a virtually reserved amount of storage. The standard function *maxlength* always returns *maxint* (`maxlength(v)` is 2147483647). The standard function *length* yields the length of the actually stored string of characters which may be set explicitly by standard procedure *setlength* within the range from `0` to *maxint*. An ALLOCATION exception may occur if the requested storage space is not available during a *string* operation. The size of the actually used storage space increases during processing, if

- a "longer" string value is assigned to the *string* variable,
- an indexed access to a character of a *string* variable is done with an index greater than the value returned by *length*, or

- standard procedure *setlength* is used with an argument value which is greater than the value returned by *length*.

There will be no loss of information unless string positions are explicitly changed. Refer to 6.6 *Module x_strg* for additional routines.

The corresponding C type is named *"s_trng"*. A value of type *string* is represented by a structure and a sufficiently large amount of storage space to hold the characters of the actual string value. The administrative information in *"s_trng"* consists of

- the size of the allocated array holding the string characters,
- the actual length of the string value that is stored in the allocated array,
- the "fixed length" indicator,
- the "substring" indicator, and
- the "temporary string" indicator.

The components of a **file** type and the components of the variant part of a **record** must not contain (dynamic) *string* components neither directly nor indirectly. Examples for invalid and correct definitions of file types are given in Figure 5.7. The component type `STRING[10]` denotes a dynamic type of at most 10 characters but not a static string type of exactly 10 characters.

```
{ Example for invalid FILE types }
FILE OF STRING [10];
FILE OF RECORD R : REAL;
               S : STRING [10];
        END

{ Example for a correct FILE type }
FILE OF RECORD
        string_length : INTEGER;
        string_char   : ARRAY [1..10] OF CHAR;
        END;
```

Figure 5.7: Example for invalid and correct definitions of file types with strings

5.3.2.7 Records*

5.3.2.8 Records with Variants

A variant part of a **record** must not contain components of type *string*, *dotprecision*, and dynamic array neither directly nor indirectly.

5.3.2.9 Sets

The ordinal numbers of the **set** elements and the bounds of **set** types must be within the range from 0 to 255. The upper bound for the range of variables of **set** type may be altered in the current installation. Refer to the *local configuration guide* for the actual value of the upper bound of **set** types.

5.3.2.10 Files

File types are not allowed as component types and must not be referenced by pointers. The component type of a **file** type must not be of type *string*, *dotprecision*, and dynamic array neither directly nor indirectly.

5.3.2.11 Text Files

The standard file variables *input* and *output* are predeclared variables and, thus, may be used within a PASCAL–XSC **module**.

If *output* is missing in the program parameter list, then it is automatically associated with the standard output device *"stdout"*. If *input* is missing in the program parameter list, then it is automatically associated with the standard input device *"stdin"*. After a *reset* for *text* files associated with *"stdin"* the standard function *eoln* for these files yields *true* and the buffer variable of the file has the value '␣' which stands for a blank character.

Output written to text files should always be terminated by a call of procedure *writeln*, thus generating an end-of-line delimiter at the end of an output line. Otherwise, only a partial line may be written to the text file.

5.3.3 Structured Arithmetic Standard Types

5.3.3.1 The Type *complex*

The corresponding C data type is called *"a_cmpx"* and is implemented as a structure with two *"a_real"* components *"RE"* and *"IM"* representing the real part and the imaginary part of a complex number, respectively.

5.3.3.2 The Type *interval*

The corresponding C data type is called *"a_intv"* and is implemented as a structure with two *"a_real"* components *"INF"* and *"SUP"* representing the infimum (lower bound) and the supremum (upper bound) of a real interval, respectively.

5.3.3.3 The Type *cinterval*

The corresponding C data type is called *"a_cinv"* and is implemented as a structure with two *"a_intv"* components *"RE"* and *"IM"* representing the real part and the imaginary part of a complex interval, respectively.

5.3.3.4 Vector Types and Matrix Types

The dynamic array data types *rvector*, *cvector*, *ivector*, *civector*, *rmatrix*, *cmatrix*, *imatrix*, and *cimatrix* in PASCAL–XSC are associated with the names of C data types defined in "p88rts.h" according to Figure 5.8.

PASCAL–XSC type name	C type name
rvector	*"a_rvty"*
cvector	*"a_cvty"*
ivector	*"a_ivty"*
civector	*"a_civt"*
rmatrix	*"a_rmty"*
cmatrix	*"a_cmty"*
imatrix	*"a_imty"*
cimatrix	*"a_cimt"*

Figure 5.8: PASCAL–XSC vector and matrix types and related C types

5.3.4 Pointers

Procedures *mark* and *release* are not implemented.

The pointer constant *nil* has the value *"NULL"* as defined by the C compiler in use.

Forward declared types must be **record** types as demonstrated in Figure 5.9. Forward declared data type names are those type names that occur immediately after the symbol '^' and before their definition.

```
{ Example for invalid declared types }
TYPE RPTR = ^RTYP;
     RTYP =  REAL;
TYPE APTR = ^ATYP;
     ATYP = ARRAY[1..9] OF RECORD n : APTR END;

{ Example for correctly declared types }
TYPE RTYP =  REAL;
     RPTR = ^RTYP;
TYPE APTR = ^ATYP;
     ATYP =  RECORD
             a : ARRAY[1..9] OF RECORD n : APTR END;
             END;
```

Figure 5.9: Example for invalid definitions of types with pointers

5.3.5 Compatibility of Types

The rules of type compatibility as defined by Standard PASCAL may process ambiguities in connection with the concept of overloading of subroutines. Therefore, the PASCAL–XSC compiler restricts the compatibility of types:

- Different **set** types are not compatible. But, a set constructor with elements of type T is compatible with all types **set of** T.
- Different subrange types are not compatible. But every subrange type is compatible with its base type. The correct use of subrange types is illustrated in Figure 5.10.

```
{ Example for invalid use of subrange types }
TYPE range = 1..10;
VAR      y : 1..20;
PROCEDURE p ( VAR x : range );
   BEGIN {...} END;
BEGIN
   p ( y );
END.

{ Example for correct use of subrange types }
TYPE range = 1..10;
VAR      y : 1..20;
PROCEDURE p ( VAR x : range );
   BEGIN {...} END;
BEGIN
   p ( range(y) );
END.
```

Figure 5.10: Example for use of subrange types

5.3.5.1 Compatibility of Array Types*

5.3.5.2 Compatibility of Strings*

5.4 Expressions

Any type identifier may be used explicitly as the name of a type conversion function which is predeclared.

If		T1 is a type identifier
	and	T2 is a PASCAL–XSC data type
	and	X is an expression of type T2
	and	T1 and T2 are assignment compatible
then		T1(X) is a legal expression of type T1

Nevertheless, if a function T1 with one argument of type T2 is declared, then the declared function is used, because the predeclared type conversion function has been redefined.

5.4.1 Standard Expressions

5.4.1.1 Integer Expressions

The operators *div* and *mod* are defined according to the specifications of standard PASCAL. The integer operations `+`, `-`, and `*` as well as the standard functions *succ*, *pred*, and *sqr* are passed directly to the C compiler and usually will not cause runtime exceptions. In module *x_intg* described in 6.4 *Module x_intg*, the *integer* operations `+`, `-`, and `*` and the standard functions *succ*, *pred*, and *sqr* are redefined by runtime routines which perform an overflow checking. In case of an integer overflow, a runtime exception is signaled by these routines.

The standard function *ival* causes a runtime exception if the leading non-blank characters of the *string* argument are not part of a valid representation of an *integer* value.

5.4.1.2 Real Expressions

Exception handling for *real* operations `+`, `-`, `*`, and `/` is done according to the specifications of the IEEE standard [3]. Refer to Appendix D *IEEE Exception Handling Environment* for more details about default settings.

Standard function *rval* causes a runtime exception if the leading non-blank characters of the *string* argument are not part of a valid representation of a *real* value.

A set of 25 mathematical functions with *real* arguments are part of the PASCAL–XSC runtime system with a guaranteed accuracy of less than 2 ulp (1 ulp = one unit in the last place of the mantissa).

The supported domains of the mathematical functions are listed in Figure 5.11. Domain intervals marked by * are smaller than the maximum domain intervals which are possible due to the real data format in use.

Function	Domain of valid *real* arguments
sqr(r)	[⇔*sqrmax*, *sqrmax*]
sqrt(r)	[0, *maxreal*]
exp(r)	[⇔*maxreal*, *expmax*]
exp2(r)	[⇔*maxreal*, *exp2max*]
exp10(r)	[⇔*maxreal*, *exp10max*]
ln(r)	[*minreal*, *maxreal*]
log2(r)	[*minreal*, *maxreal*]
log10(r)	[*minreal*, *maxreal*]
sin(r)	[⇔*trimax*, *trimax*]*
cos(r)	[⇔*trimax*, *trimax*]*
tan(r)	[⇔*trimax*, *trimax*]*
cot(r)	[⇔*trimax*, ⇔*cotmin*]* or [*cotmin*, *trimax*] *
arcsin(r)	[⇔1, 1]
arccos(r)	[⇔1, 1]
arctan(r)	[⇔*maxreal*, *maxreal*]
arctan2(r1,r2)	*r1* = *r2* = 0 not allowed
arccot(r)	[⇔*maxreal*, *maxreal*]
sinh(r)	[⇔*hypmax*, *hypmax*]
cosh(r)	[⇔*hypmax*, *hypmax*]
tanh(r)	[⇔*maxreal*, *maxreal*]
coth(r)	[⇔*maxreal*, ⇔*cotmin*] or [*cotmin*, *maxreal*]
arsinh(r)	[⇔*maxreal*, *maxreal*]
arcosh(r)	[1, *maxreal*]
artanh(r)	[⇔*(one–eps)*, *(one–eps)*]
arcoth(r)	[⇔*maxreal*, ⇔*(one+eps)*] or [*(one+eps)*, *maxreal*]

Figure 5.11: Domains of *real* functions with a priori error estimation

The term *maxreal* stands for the largest finite *real* value, and the term *minreal* stands for the smallest positive *real* value. Refer to Figure 5.6 for details about special *real* values of the IEEE double data format.

The decimal values of named constants in Figure 5.11 are listed in the following table.

Constant	decimal	hexadecimal
hypmax	$7.104\,758\,600\,739\,439 \cdot 10^{2}$	`408633ce8fb9f87d`
sqrmax	$1.340\,780\,792\,994\,259\,6 \cdot 10^{154}$	`5fefffffffffffff`
trimax	$9.223\,372\,036\,854\,776 \cdot 10^{18}$	`43e0000000000000`
expmax	$7.097\,827\,128\,933\,84 \cdot 10^{2}$	`40862e42fefa39ef`
exp2max	$1.023\,999\,999\,999\,999\,9 \cdot 10^{3}$	`408fffffffffffff`
exp10max	$3.082\,547\,155\,599\,167 \cdot 10^{2}$	`40734413509f79fc`
cotmin	$5.562\,684\,646\,268\,008 \cdot 10^{-309}$	`0004000000000001`
maxreal	$1.797\,693\,134\,862\,315\,8 \cdot 10^{308}$	`7fefffffffffffff`
minreal	$4.940\,656\,458\,412\,465\,4 \cdot 10^{-324}$	`0000000000000001`
(one–eps)	$0.999\,999\,999\,999\,999\,9$	`3fefffffffffffff`
(one+eps)	$1.000\,000\,000\,000\,000\,2$	`3ff0000000000001`

The implementation of *real* mathematical functions with a posteriori error estimations and maximum domain intervals is available by using the definitions in module *x_real*. Refer to 6.5 *Module x_real*.

Standard function *mant* yields the signed mantissa of a given *real* value. In case of a non-zero value, the mantissa is normalized such that for the base 2 of the implemented IEEE floating-point number system holds:

$$0.5 \leq |\text{mantissa}| < 1$$

If the *real* value is zero, then the mantissa is zero. If the *real* value is infinity, then the mantissa is infinity.

Standard function *expo* yields the *integer* exponent of a given *real* value with respect to the base 2 of the implemented floating-point number system. In case of a non-zero finite value, the exponent of a *real* number for the implemented IEEE double format satisfies:

$$\Leftrightarrow 1021 \leq \text{exponent} \leq 1024$$

If the *real* value is zero, then ⇔*maxint* is returned. If the *real* value is infinity, then *maxint* is returned.

For non-zero finite floating-point numbers x the following identity holds:

$$x = comp(mant(x), expo(x))$$

Alternative functions with normalization according to the IEEE standard are available in module *x_real* via procedure *x_comp* and functions *x_mant* and *x_expo*.

Standard functions *succ* and *pred* return the "next" floating-point number which is larger or smaller than the actual argument. They cause a runtime exception if the symmetric range of floating-point numbers from *–maxreal* to *maxreal* is left. By default, all normalized and denormalized floating-point numbers are considered by the standard functions *succ* and *pred*. A restriction to normalized floating-point numbers is possible by using runtime option **-nn**. Refer to 4.2 *PASCAL–XSC Runtime Options*.

5.4.1.3 Boolean Expressions

Standard functions *succ* and *pred* cause a runtime exception if the range *false..true* is left.

5.4.1.4 Character Expressions

Standard function *chr* causes a runtime exception if the actual argument of type *integer* is not within the range from 0 to 255.

Standard functions *succ* and *pred* cause a runtime exception if the range from *chr(0)* to *chr(255)* is left.

5.4.1.5 Enumeration Expressions

Standard functions *succ* and *pred* cause a runtime exception if the range of the enumeration type is left.

5.4.1.6 Subrange Expressions

The standard functions *succ* and *pred* return their argument type as result type. It is not an error, if the result of these functions exceeds the range of the subrange type but not the range of the base type.

5.4.2 Accurate Expressions (#-Expressions)*

5.4.3 Expressions for Structured Types and Pointer Expressions

5.4.3.1 Array Expressions*

5.4.3.2 String Expressions*

5.4.3.3 Record Expressions*

5.4.3.4 Set Expressions*

5.4.3.5 Pointer Expressions

If the PASCAL–XSC source was compiled with the '`+x`' option (see 3.6.2), then the runtime system checks any pointer access for the pointer value *nil*. Dereferencing other invalid pointers can not be checked by the runtime system. A memory violation error may occur, which can not be handled by the runtime system. Consequently, there is no positional information available by the runtime system.

5.4.4 Extended Accurate Expressions (#-Expressions)*

5.5 Statements

5.5.1 Assignment Statement*

5.5.2 Input/Output Statements

The PASCAL–XSC default procedure *write* is overloaded for different data types and for different format specifications. The formal procedure headers of the default *write* procedures are listed without the required keyword **procedure**.

1. ```
   WRITE (VAR f: TEXT; n: INTEGER);
   WRITE (VAR f: TEXT; n: INTEGER; w: INTEGER);
   ```
2. ```
   WRITE (VAR f: TEXT; h: CHAR);
   WRITE (VAR f: TEXT; h: CHAR; w: INTEGER);
   ```
3. ```
   WRITE (VAR f: TEXT; b: BOOLEAN);
   WRITE (VAR f: TEXT; b: BOOLEAN; w: INTEGER);
   ```
4. ```
   WRITE (VAR f: TEXT; r: REAL);
   WRITE (VAR f: TEXT; r: REAL; w: INTEGER);
   WRITE (VAR f: TEXT; r: REAL; w, f: INTEGER);
   WRITE (VAR f: TEXT; r: REAL; w, f, m: INTEGER);
   ```
5. ```
   WRITE (VAR f: TEXT; s: STRING);
   WRITE (VAR f: TEXT; s: STRING; w: INTEGER);
   ```

For each of these routines a brief description is given. Procedures for positive arguments '`w`' and '`f`' that are already available in Standard PASCAL are unchanged. An interpretation for negative arguments '`w`' and '`f`' as well as for the rounding argument '`r`' is added.
   ```

1. write integer value
   If field width w is not specified or w is greater than or equal to zero, then standard PASCAL (right-adjusted) output is generated. If w is less than zero, then left-adjusted output is generated, i.e., trailing blanks are generated instead of preceding blanks.

   Example: The PASCAL–XSC *write* statements

   ```
   WRITELN ( 1234:6,',',  1234:-6,',',  1234);
   WRITELN (-1234:6,',', -1234:-6,',', -1234);
   ```

   produce the following output with blank characters represented by ␣.

   ```
   ␣␣1234,␣1234␣,1234
   ␣-1234,-1234␣,-1234
   ```

2. write character value
   If field width w is not specified, then w is assumed to be one. If field width w is greater than zero, then $(w \Leftrightarrow 1)$ blanks are preceding the character value according to standard PASCAL. If field width w is zero, then nothing is output. If w is less than zero, then left-adjusted output is generated, i.e., $(\Leftrightarrow w \Leftrightarrow 1)$ trailing blanks are generated instead of preceding blanks.

   Example: The PASCAL–XSC *write* statement

   ```
   WRITELN ('a':-3, 'b', 'c', 'd':3);
   ```

   produces the following output with blank characters represented by ␣.

   ```
   a␣␣bc␣␣d
   ```

3. write boolean value
   If field width w is not specified, then the complete text strings representing the *boolean* values *true* and *false* are output, respectively. If field width w is greater than zero, then the appropriate text beginning with its first character is right-adjusted in a field of width w. If field width w is less than zero, then the appropriate text beginning with its first character is left-adjusted in a field of width $(\Leftrightarrow w)$. If field width w is zero, then no characters are output.

   The default text for the *boolean* value *true* is '`TRUE␣`'. The default text for the *boolean* value *false* is '`FALSE`'. The default text may be altered in certain installations. Refer to the *local configuration guide* for details.

   Example: The PASCAL–XSC *write* statements

   ```
   WRITELN ( TRUE,',',  TRUE:-7,',',  TRUE:7,',',  TRUE:1);
   WRITELN (FALSE,',', FALSE:-7,',', FALSE:7,',', FALSE:1);
   ```

produce the following output with blank characters represented by ␣.

```
TRUE␣,TRUE␣␣␣,␣␣TRUE␣,T
FALSE,FALSE␣␣,␣␣FALSE,F
```

4. <u>write real value</u>

   If field width w is not specified, i.e., the *write* statement is of the form `write(r)`, then an equivalent notation is `write(r:23:0:0)`. If the number of fraction digits f is not specified, i.e., the *write* statement is of the form `write(r:w)`, then an equivalent notation is `write(r:w:0:0)`. If rounding mode m is not specified, i.e., the *write* statement is of the form `write(r:w:f)`, then an equivalent notation is `write(r:w:f:0)`.

   If field width w is greater than or equal to zero, then the output is right-adjusted according to standard PASCAL with a minimum field width of $w = 9$ in case of a floating-point number representation ($f = 0$). The minimum field width is composed of

   – one character for the minus sign '`-`' (or blank '␣')

   – one digit before the decimal point

   – one character for the decimal point '`.`'

   – one digit after the decimal point

   – one character for the exponent delimiter '`E`'

   – one character for the sign of the exponent '`+`' or '`-`'

   – three digits for the exponent eventually padded with leading zeros (value 0.0 has exponent '`E+000`').

   If w is less than zero, then the output is left-adjusted in case of a fixed-point number representation. In case of a floating-point number representation the field width is set to $(\Leftrightarrow w)$.

   If the number of fraction digits f is zero, then a floating-point number representation is output. If the number of fraction digits f is less than zero, then f is set to $(\Leftrightarrow f)$. If the number of fraction digits f is greater than zero, then a fixed-point number representation with f digits in the fraction part is output in a field of minimum width $(f + 2)$. The minimum field width is composed of

   – one digit before the decimal point

   – one character for the decimal point '`.`'

   – f digits after the decimal point.

   The rounding mode used for the representation of real values is identified by the sign of m. If m is negative, then the decimal representation of the real value is rounded towards −infinity. If m is positive, then the decimal representation of the real value is rounded towards +infinity. If m is zero, then the decimal representation of the real value is rounded to the nearest decimal number with respect to the exact value. In case of a tie, the decimal representation of the real value has an even least significant decimal digit.

Example: The PASCAL–XSC *write* statements

```
WRITELN (1.250001: 9:0:-1, 1.250001: 9, 1.250001:9:0:1, 1.25:9);
WRITELN (1.250001:20:0:-1, 1.250001:20:0:1);
WRITELN (1.250001:30);
```

produce the following output with blank characters represented by ␣.

```
␣1.2E+000␣1.3E+000␣1.3E+000␣1.2E+000
␣1.250000999999E+000␣1.250001000000E+000
␣1.2500009999999999177334E+000
```

Note: The value of the decimal number 1.250001 cannot be represented exactly using IEEE double format. The string '`1.250001`' is converted to an IEEE double value using the rounding to the nearest IEEE floating-point number representation before the *write* procedure starts. For the decimal number 1.250001 the converted IEEE double value is smaller than the exact decimal value but still greater than 1.25.

5. write string value
   For the default *write* procedures the actual argument for the *string* argument may be
   A) an "*array [ ] of char*" with arbitrary but fixed index range
   B) a "*string [ ]*" with arbitrary but fixed length
   C) a "*string*" of default size

   Subsequently, the corresponding actual arguments are denoted by `a`, `b`, and `c`. The notations `lb` and `ub` stand for the lower bound and the upper bound of the index range specified in the definition of an "*array [ ] of char*", respectively.

   For each of the argument types in A), B) and C) the output is (slightly) different.

   If the field width w is not specified, then all of the characters in the declared array `a` and all of the characters from 1 to the value determined by standard function *length* of `b` or `c` are displayed. If `length(b)`=0 or `length(c)`=0, then no characters are output.

   If the field width w is greater than or equal to zero, then q blank characters with

$$\text{A)} \quad q := \begin{cases} w \Leftrightarrow \texttt{ub} + \texttt{lb} \Leftrightarrow 1 & \text{if } w > \texttt{ub} \Leftrightarrow \texttt{lb} + 1 \\ 0 & \text{otherwise} \end{cases}$$
$$\text{B)} \quad q := \begin{cases} w \Leftrightarrow maxlength(b) & \text{if } w > maxlength(b) \\ 0 & \text{otherwise} \end{cases}$$
$$\text{C)} \quad q := \begin{cases} w \Leftrightarrow length(c) & \text{if } w > length(c) \\ 0 & \text{otherwise} \end{cases}$$

are output followed by r characters with

$$\begin{array}{ll} \text{A)} & r := w \Leftrightarrow q \\ \text{B)} & r := \min\{length(\text{b}),\, w \Leftrightarrow q\} \\ \text{C)} & r := w \Leftrightarrow q \end{array}$$

of the actual argument beginning with the first character. In case B), an additional number of $(w \Leftrightarrow q \Leftrightarrow r)$ trailing blank characters are output.

If the field width w is negative, then the output is left adjusted. If number q

$$\begin{array}{ll} \text{A)} & q{:=}\ \texttt{ub} \Leftrightarrow \texttt{lb} + 1 \\ \text{B)} & q{:=}\ length(\text{b}) \\ \text{C)} & q{:=}\ length(\text{c}) \end{array}$$

is greater than or equal to $(\Leftrightarrow w)$, then $(\Leftrightarrow w)$ characters of the actual argument beginning with the $(q+w\Leftrightarrow 1)^{th}$ character (relative to the beginning of the string) are output followed by $s = \Leftrightarrow w \Leftrightarrow q$ blanks if s is greater than zero.

In Figure 5.12, a summary of equivalent PASCAL notations for the PASCAL–XSC procedure *write* with string arguments are given.

Example: For variables `a`, `b`, and `c` defined by

```
var a : packed array [1..10] of char;
    b : string[10];
    c : string;
```

the PASCAL–XSC statements

```
a := 'PASCAL-XSC';
b := 'PASCAL-XSC system';
c := 'PASCAL-XSC system';
WRITELN ('a : ',a:6,',',a:-6,',',a:20,',',a);
WRITELN ('b : ',b:6,',',b:-6,',',b:20,',',b);
WRITELN ('c : ',c:6,',',c:-6,',',c:20,',',c);
b := 'PASCAL';
WRITELN ('b : ',b:6,',',b:-6,',',b:20,',',b);
```

produce the following output with blank characters represented by '␣'.

```
a␣:␣PASCAL,AL-XSC,␣␣␣␣␣␣␣␣␣␣PASCAL-XSC,PASCAL-XSC
b␣:␣PASCAL,AL-XSC,␣␣␣␣␣␣␣␣␣␣PASCAL-XSC,PASCAL-XSC
c␣:␣PASCAL,system,␣␣␣PASCAL-XSC␣system,PASCAL-XSC␣system
b␣:␣PASCAL,AL␣␣␣␣,␣␣␣␣␣␣␣␣␣␣PASCAL␣␣␣␣,PASCAL-XSC
```

A)
```
var a : array [lb..ub] of char;
size := ub-lb+1;
```

write(a) ⟹ write(a:size)		
write(a:w)⟹		
size<w	: write(' ':w-size, a:size)	right adjusted
0≤w≤size	: FOR i:=0 TO w-1 DO write(a[lb+i])	leading characters
-size≤w≤⇔1	: FOR i:=w+1 TO 0 DO write(a[ub+i])	trailing characters
w<-size	: write(a:size, ' ':-w-size)	left adjusted

B)
```
var b : string[size];
len := LENGTH(b);
```

write(b) ⟹ write(b:len)		
write(b:w)⟹		
size<w	: write(' ':w-size,b:len,' ':size-len)	right adjusted
len<w≤size	: write(b:len,' ':size-w)	leading characters
0≤w≤len	: FOR i:=1 TO w DO write(b[i])	
-len≤w≤⇔1	: FOR i:=w+1 TO 0 DO write(b[len+i])	trailing characters
w<-len	: write(b:len,' ':-w-len)	left adjusted

C)
```
var c : string;
len := LENGTH(c);
```

write(c) ⟹ write(c:len)		
write(c:w)⟹		
len<w	: write(' ':w-len, c:len)	right adjusted
0≤w≤len	: FOR i:=1 TO w DO write(c[i])	leading characters
-len≤w≤⇔1	: FOR i:=w+1 TO 0 DO write(c[len+i])	trailing characters
w<-len	: write(c:len, ' ':-w-len)	left adjusted

Figure 5.12: Equivalent notations for procedure *write* with *string* arguments

5.5.3 Empty Statement*

5.5.4 Procedure Statement*

5.5.5 goto-Statement

The destination of a **goto** statement must not be outside of the current block. In order to terminate the processing of a PASCAL–XSC program, the procedure *exit* may be used which is declared in module *iostd*. Refer to 6.3 *Module iostd*.

5.5.6 Compound Statement*

5.5.7 Conditional Statements*

5.5.8 Repetitive Statements*

5.5.9 with-Statement

Between the keyword **with** and the matching keyword **do** a maximum of 15 names of variables is allowed. Nested **with**-statements may be used to remove this restriction.

5.6 Program Structure*

5.7 Subroutines

5.7.1 Procedures*

5.7.2 List of Predefined Procedures and I/O Statements

The standard procedures *new* and *dispose* ignore all tag marks in connection with variant-records. The standard procedures *mark* and *release* are not implemented.

5.7.3 Functions

The assignment of a value to the result variable of a function (function name) must be within the statement part of the function and not an inner procedure, function, or operator.

The name of a function may be any type name. If the result type of a function is identical with the name of the function and the function has exactly one argument, then a coersion is defined. A coersion is a type conversion function that is automatically generated by the compiler. Even though coersions may be defined without restriction, it is recommended that coersions should not be used, since the concept of coersions conflicts with the concept of overloading.

```
{Example for coersion }
PROGRAM PROG;
TYPE T1 = ...;
TYPE T2 = ...;
FUNCTION T2 ( x : T1 ) : T2;
  BEGIN T2 := ...; END;
PROCEDURE P ( arg : T2 );
  BEGIN END;
VAR y : T1;
BEGIN
  P ( y );
END.
```

Figure 5.13: Example for a type conversion function

Let `T2` be the function name and `T1` the argument type. Then the function `T2` is called automatically, if an expression of type `T1` is passed to a formal value argument of type `T2`. In the sample program given in Figure 5.13, the PASCAL–XSC compiler automatically produces the procedure call `P(T2(y))`.

5.7.4 Functions with Arbitrary Result Type*

5.7.5 List of Predefined Functions

Function *loc* is not implemented.

5.7.6 Operators

The **priority** definition for an operator is valid until the end of the PASCAL–XSC source code, i.e., it is even valid outside the defining block. Thus, there will be a warning message for local **priority** definitions.

5.7.7 Table of Predefined Operators*

5.7.8 forward- and external-Declaration

In PASCAL–XSC the keyword **external** is used to declare entry names of subroutines which have been implemented in another programming language. Subsequently it is assumed, that the programming language C has been used.

After the keyword **external** the entry name of an external C function is specified. This specification either is a PASCAL name according to the syntax of PASCAL or a string constant. Lower and upper case letters in the identifier name are distinguished after the keyword **external**. The entry name given after the keyword **external** should contain at least one lower case letter in order to avoid name conflicts with PASCAL–XSC names. Name conflicts with entry names used by the PASCAL–XSC runtime system and the C runtime system must not occur. Usually the C compiler and the linker will give adequate warnings or error messages in case of name conflicts. If there is no name listed after the keyword **external**, then the PASCAL–XSC subroutine name is taken as the entry name of an externally defined subroutine. Note, that lower case letters in PASCAL–XSC names are always converted to upper case letters.

5.7.9 Modified Call by Reference for Structured Types

The modified reference call is allowed for all data types. If the modified reference call is applied to scalar types and pointer types, then the PASCAL–XSC compiler gives a warning message pointing out that this construction is not valid in Standard PASCAL.

5.7.10 Overloading of Procedures, Functions, and Operators

The rules of type compatibility have been changed. Refer to 5.3.5 *Type Compatibility*.

The assignment compatibility is extended to user-defined coersions. Refer to section 5.7.3 *Functions* for more details.

The rules of selecting subroutines have been changed:

> The PASCAL–XSC compiler detects an error, if there is more than one subroutine with the same minimum number of applicable coersions in the same block. *The position of subroutine arguments is not taken into account.* This is different to the language description [4].

The example in Figure 5.14 is taken from [4] and differs in the third assignment statement due to this restriction.

```
OPERATOR +* ( a : INTEGER; b : REAL    ) ir_res : REAL;
...
OPERATOR +* ( a : REAL;    b : INTEGER ) ri_res : REAL;
...
VAR     i : INTEGER;
    r,res : REAL;
...
res := i +* r;  { first  operator is applied }
res := r +* i;  { second operator is applied }
res := i +* i;  { statement is not possible, two operators available }
res := r +* r;  { statement is not possible, no  operator  available }
...
```

Figure 5.14: Example for the selection of subroutines

5.7.11 Overloading of *read* and *write**

5.7.12 Overloading of the Assignment Operator :=*

5.8 Modules*

5.9 String Handling and Text Processing*

5.10 How to Use Dynamic Arrays*

Chapter 6

PASCAL–XSC Modules

6.1 Module *stdmod*

The module *stdmod* is imported automatically without an explicit **use** clause. This module contains definitions of identifiers and operators that are predeclared in the language PASCAL–XSC.

6.2 Arithmetic Modules

The implementation dependent parts of the arithmetic modules are concerned with output operations and the domains of mathematical functions. Output operations for vectors and matrices as specified in the modules *mv_ari*, *mvc_ari*, *mvi_ari*, and *mvci_ari* are mapped to the output operations of the corresponding component types *real*, *complex*, *interval*, and *cinterval*, respectively. There are no mathematical functions defined for vectors and matrices.

6.2.1 Module *i_ari*

The supported domains of the mathematical functions for arguments of type *interval* are listed in Figure 6.1.

The arcus tangent of the quotient of two interval arguments is not included in module *i_ari* but is implemented separately in a module named *iatan2*.

The decimal values of named constants in Figure 6.1 are listed in the following table.

Function	Domain of valid *interval* arguments
sqr(i)	$i \subset [-sqrmax, sqrmax]$
sqrt(i)	$i \subset [0, maxreal]$
exp(i)	$i \subset [-maxreal, expmax]$
exp2(i)	$i \subset [-maxreal, exp2max]$
exp10(i)	$i \subset [-maxreal, exp10max]$
ln(i)	$i \subset [minreal, maxreal]$
log2(i)	$i \subset [minreal, maxreal]$
log10(i)	$i \subset [minreal, maxreal]$
sin(i)	$i \subset [-maxreal, maxreal]$
cos(i)	$i \subset [-maxreal, maxreal]$
tan(i)	$i \subset [-maxreal, maxreal] \wedge \frac{\pi}{2} + k\pi \notin i$
cot(i)	$i \subset [-maxreal, -cotmin] \cup [cotmin, maxreal] \wedge k\pi \notin i$
arcsin(i)	$i \subset [-1, 1]$
arccos(i)	$i \subset [-1, 1]$
arctan(i)	$i \subset [-maxreal, maxreal]$
arctan2(i1,i2)	$0 \in i1$ and $0 \in i2$ not allowed
arccot(i)	$i \subset [-maxreal, maxreal]$
sinh(i)	$i \subset [-hypmax, hypmax]$
cosh(i)	$i \subset [-hypmax, hypmax]$
tanh(i)	$i \subset [-maxreal, maxreal]$
coth(i)	$i \subset [-maxreal, -cotmin] \cup [cotmin, maxreal]$
arsinh(i)	$i \subset [-maxreal, maxreal]$
arcosh(i)	$i \subset [1, maxreal]$
artanh(i)	$i \subset [-(one-eps), (one-eps)]$
arcoth(i)	$i \subset [-maxreal, -(one + eps)] \cup [(one+eps), maxreal]$

Figure 6.1: Domains of *interval* functions

	decimal	hexadecimal
hypmax	$7.104\,758\,600\,739\,439 \cdot 10^{2}$	`408633ce8fb9f87d`
sqrmax	$1.340\,780\,792\,994\,259\,6 \cdot 10^{154}$	`5fefffffffffffff`
expmax	$7.097\,827\,128\,933\,84 \cdot 10^{2}$	`40862e42fefa39ef`
exp2max	$1.023\,999\,999\,999\,999\,9 \cdot 10^{3}$	`408fffffffffffff`
exp10max	$3.082\,547\,155\,599\,167 \cdot 10^{2}$	`40734413509f79fc`
cotmin	$5.562\,684\,646\,268\,008 \cdot 10^{-309}$	`0004000000000001`
(one–eps)	$0.999\,999\,999\,999\,999\,9$	`3fefffffffffffff`
(one+eps)	$1.000\,000\,000\,000\,000\,2$	`3ff0000000000001`
maxreal	$1.797\,693\,134\,862\,315\,8 \cdot 10^{308}$	`7fefffffffffffff`
minreal	$4.940\,656\,458\,412\,465\,4 \cdot 10^{-324}$	`0000000000000001`

The output operation for values of type *interval*

```
procedure write(var f : text, i : interval);
```

produces a decimal representation of the lower and upper bounds of the interval value rounded downwards and upwards, respectively. Only significant digits of the decimal representations of the lower and upper bound are displayed, i.e., all leading coinciding digits and a small number of differing digits are displayed. The default output format consists of 52 characters.

1	left bracket
1	blank
23	characters for the lower bound
1	komma
1	blank
23	characters for the upper bound
1	blank
1	right bracket

An example for the described output format is given in Figure 6.2.

6.2.2 Module *c_ari*

Mathematical functions for data type *complex* based on the IEEE binary double format and an accuracy of 2 ulps are not yet implemented.

The output operation for values of type *complex*

```
procedure write(var f : text, c : complex);
```

produces a decimal representation of the real and imaginary part of the complex value rounded to the nearest decimal representation. The default output format consists of 52 characters.

1	left parenthesis
1	blank
23	characters for the real part
1	komma
1	blank
23	characters for the imaginary part
1	blank
1	right parenthesis

An example for the described output format is given in Figure 6.2.

```
program p(output);

use i_ari,c_ari,ci_ari;

var i : interval;
    c : complex;
    z : cinterval;
begin
   i.inf := 1.0;    i.sup := 1.0;    writeln(i);
   c.re  := 1.0;    c.im  := 1.0;    writeln(c);
   z.re  := i;      z.im  := i;      writeln(z);
   i.inf := 1.235;  i.sup := 1.236;  writeln(i);
   z.re  := i;      z.im  := i;      writeln(z);
end.
```

Output of program p:

```
[  1.0000000000000000E+000, 1.0000000000000000E+000 ]
(  1.0000000000000000E+000, 1.0000000000000000E+000 )
( [  1.0000000000000000E+000, 1.0000000000000000E+000 ],
  [  1.0000000000000000E+000, 1.0000000000000000E+000 ] )
[               1.235E+000,             1.236E+000 ]
( [               1.235E+000,             1.236E+000 ],
  [               1.235E+000,             1.236E+000 ] )
```

Figure 6.2: Output format for structured arithmetic types

6.2.3 Module *ci_ari*

Mathematical functions for data type *cinterval* based on the IEEE binary double format and an accuracy of 1 ulp with respect to the evaluation of bounds are not yet implemented.

The output operation for values of type *cinterval*

```
procedure write(var f : text, z : cinterval);
```

produces a decimal representation of the real and imaginary part of the complex interval value. Both the real and imaginary part are output via the interval output procedure described in 6.2.1 *Module i_ari*. The default output format consists of 2 lines separated by a newline character and a total of 111 displayable characters (55 characters on the first line and 56 characters on the second line).

1	left parenthsis
1	blank
52	characters for the real part
1	komma immediately followed by a newline character
2	blanks
52	characters for the imaginary part
1	blank
1	right parenthesis

An example for the described output format is given in Figure 6.2.

6.3 Module *iostd*

The module *iostd* contains declarations of some additional input and output routines, and constant definitions.

```
const
   stdin  = 0 ; { standard input }
   stdout = 1 ; { standard output }
   stderr = 2 ; { standard error }
   stdcon = 3 ; { Terminal, ''/dev/tty'' ? }
   stdprn = 4 ; { Printer }
   stdrdr = 5 ; { Reader (CPM-device) }
   stdpun = 6 ; { Puncher (CPM-device) }
   stdtmp = 8 ; { temporary file, will be deleted by close }
   stdorg = 9 ; { File, associated originally by command line }
{ use these constant names for the nr parameter
  in reset and rewrite }

procedure reset  (var t:text; nr:integer);
procedure rewrite(var t:text; nr:integer);
procedure close  (var t:text);
procedure flush  (var t:text);
function  filexists (s:string) : boolean;
function  getenv    (s:string) : string;
procedure exit (retcode : integer);
```

After importing module "*iostd*" the procedures *reset* and *rewrite* may be called with one of the constants

`stdin`, `stdout`, `stderr`, `stdcon`, `stdprn`, `stdrdr`, `stdpun`, `stdtmp`, or `stdorg` as second parameter. "*stdin*", "*stdout*", and "*stderr*" are standard devices of C. The meaning of `stdcon`, `stdprn`, `stdrdr`, and `stdpun` depends on the operating system. Refer to the *local configuration guide* for details.

The function *filexists* returns *false*, if a call of *reset* with the same file name parameter would fail.

The procedure *close* deletes a temporary *text* file or makes a file accessible to other programs and file variables. Files are closed automatically when leaving the block of their declaration.

The procedure *flush* writes the buffer of a file. This routine may be used to display output on the terminal without using calls to *writeln*.

Function *getenv* is equivalent to the function "*getenv*" in ANSI C, i.e., the contents of an existing environment variable is returned. If the environment variable is not defined, an empty string is returned.

The procedure *exit* may be used to terminate the processing of a PASCAL–XSC program passing a return code to the calling program or shell. The integer value zero should be returned in order to indicate that no error has occurred.

6.4 Module *x_intg*

Module *x_intg* contains definitions of additional *integer* operators and procedures. In Figure 6.3, a brief description of the additional named operators is given. The operators provide a possibility to access and manipulate individual bits of the *integer* format described in 5.3.1.1 *integer*. For these operators the operands of type *integer* are interpreted as fields of 32 bits and not as a single signed *integer* value.

and – Bitwise logical AND operation.

eqv – Bitwise logical EQV operation.
'`a EQV b`' is equivalent with '`NOT(a XOR b)`'.

not – Bitwise logical NOT operation.

or – Bitwise logical OR operation.

xor – Bitwise logical XOR operation.

bclr – Clear a single bit.

bset – Set a single bit.

btest – Test a single bit to be set.

msb – Bit number of most significant bit that is set.

Operator call				Result type	Priority
a	*and*	b	$\Longrightarrow$	*integer*	$*$
a	*bclr*	m	$\Longrightarrow$	*integer*	$*$
a	*bset*	m	$\Longrightarrow$	*integer*	$*$
a	*btest*	m	$\Longrightarrow$	*boolean*	$*$
a	*eqv*	b	$\Longrightarrow$	*integer*	$+$
	msb	a	$\Longrightarrow$	*integer*	ˆ
	not	a	$\Longrightarrow$	*integer*	ˆ
	ones	a	$\Longrightarrow$	*integer*	ˆ
a	*or*	b	$\Longrightarrow$	*integer*	$+$
a	*rotate*	s	$\Longrightarrow$	*integer*	$*$
a	*shift*	s	$\Longrightarrow$	*integer*	$*$
a	*xor*	b	$\Longrightarrow$	*integer*	$+$
a	*integer* operand interpreted as field of 32 bits.				
b	*integer* operand interpreted as field of 32 bits.				
m	*integer* value within the range from 0 to 31.				
s	signed *integer* value for *shift* and *rotate*.				

Figure 6.3: Additional named operators in module *x_intg*

ones – Number of bits that are set.

shift – Shift bits.
If the right operand is positive, then a bit shift to the left is done. Vacated bit positions are cleared.

rotate – Rotate bits.

If the right operand is positive, then a bit rotation to the left is done. Vacated bit positions get the value of the bits that are shifted out.

An output procedure *write* for the bits of an *integer* value is available. The *write* procedure is declared by

```
procedure write(var f : text; i : integer;
                                mode : char;
                                m,n : integer);
```

and writes a representation of bits `m` through `n` of *integer* `i` to the *text* file `f` using format `mode`. The values of `m` and `n` must be within the range from 0 to 31. Valid `mode` characters are '`b`' and '`B`' for binary representation, and '`x`' and '`X`' for hexadecimal representation using lower and upper case letters, respectively. In order to display bit 5 through bit 2 of *integer* variable `v` from left to right the following PASCAL–XSC statements may be used

```
v := 32+8+2+1;
writeln('Bits from ',v,' = ',v:'b':5:2)
writeln('Bits from ',v,' = ',v:'x':5:2)
writeln('Bits from ',v,' = ',v:'X':5:2)
```

which yield the output lines

```
Bits from 43 = 1010
Bits from 43 = a
Bits from 43 = A
```

Module *x_intg* also contains redefinitions for the arithmetic *integer* operators $+$, $\Leftrightarrow$, $*$, and $/$. These operators perform an overflow checking.

6.5 Module *x_real*

Module *x_real* contains additional constants, types, functions, and procedures for an extended or alternative processing of *real* values.

6.5.1 Classification of *real* values

The specification of the *real* data type in section 5.3.1.2 suggests a classification of *real* values according to the represented value. A total of 10 different classes of *real* values can be distinguished. The data type *x_ccode* is introduced which enumerates the classification codes using enumeration constants.

```
type x_ccode = ( x_sNaN,  { signaling NaN          }
                 x_qNaN,  { quiet NaN              }
                 x_minf,  { minus infinity         }
                 x_mnor,  { negative normalized    }
                 x_mden,  { negative denormalized }
                 x_mnul,  { minus zero             }
                 x_pnul,  { plus zero              }
                 x_pden,  { positive denormalized }
                 x_pnor,  { positive normalized    }
                 x_pinf   { plus infinity          }
               );


function x_class ( r : real ) : x_ccode;
```

The return code of function *x_class* is the classification code of type *x_ccode* of the given *real* value argument.

```
function x_value ( c : x_ccode ) : real;
```

The *real* value returned by function `x_value` is a special value corresponding to the given classification code of type *x_ccode*. The returned value for the classification codes are listed in the following table.

code	hexadecimal value	description
x_sNaN	7ff80000ffffffff	signaling NaN
x_qNaN	7ff00000ffffffff	quiet NaN
x_minf	fff0000000000000	minus infinity
x_mnor	ffefffffffffffff	negative normalized
x_mden	8000000000000001	negative denormalized
x_mnul	8000000000000000	minus zero
x_pnul	0000000000000000	plus zero
x_pden	0000000000000001	positive denormalized
x_pnor	7fefffffffffffff	positive normalized
x_pinf	7ff0000000000000	plus infinity

Note, that the values returned for `x_sNaN` and `x_qNaN` depend on the installation of your system. Refer to the *local configuration guide*.

6.5.2 Composition and Decomposition of *real* Values

The default functions *comp*, *expo*, and *mant* assume an abstract representation of a non-zero normalized floating-point number with a normalized mantissa m satisfying $B^{-1} \leq |m| < 1$. Here, B stands for the base which is used for the representation of the digits of mantissa m.

In the special case $B = 2$, the abstract representation of a non-zero normalized floating-point number with a mantissa m satisfying $1 \leq |m| < 2$ is possible, too. The binary IEEE data formats are defined by such a formulation. The functions *x_comp*, *x_expo*, and *x_mant* handle *real* values according to this abstract representation.

```
function x_comp ( m : real; e : integer ) : real;
function x_expo ( r : real ) : integer;
function x_mant ( r : real ) : real;
```

6.5.3 Mathematical Functions

Two different implementations of the mathematical functions are provided. The main reason for two different implementations is the significant loss of performance in case of a posteriori error estimations as compared with implementations using a priori error estimations.

On the other hand, the implementation using a posteriori error estimations can be applied to other *real* data formats including multiple presicion formats whereas the implementation using a priori error estimations is restricted to the *real* format as specified in 5.3.1.2 *real*.

By default, the implementation of *real* mathematical functions with a priori error estimations is used. The entry names of the alternative implementation of mathematical functions which uses "a posteriori error" estimations instead of "a priori" error estimations are:

```
function x_sqrt (r : real) : real;
function x_exp  (r : real) : real;
function x_exp2 (r : real) : real;
function x_exp10(r : real) : real;
function x_sin  (r : real) : real;
function x_cos  (r : real) : real;
function x_tan  (r : real) : real;
function x_cot  (r : real) : real;
function x_sinh (r : real) : real;
function x_cosh (r : real) : real;
function x_tanh (r : real) : real;
function x_coth (r : real) : real;
function x_arcsin (r : real) : real;
function x_arccos (r : real) : real;
function x_arctan (r : real) : real;
function x_arccot (r : real) : real;
function x_arsinh (r : real) : real;
function x_arcosh (r : real) : real;
function x_artanh (r : real) : real;
function x_arcoth (r : real) : real;
function x_arctan2(x,y : real) : real;
```

The supported domains of the mathematical functions are listed in Figure 6.4.

The term *maxreal* stands for the largest finite *real* value, and the term *minreal* stands for the smallest positive *real* value. Refer to Figure 5.6 for details about special *real* values of the IEEE double data format.

The decimal values of named constants in Figure 6.4 are listed in the following table.

Function	Domain of valid *real* arguments
x_sqrt(r)	[0, *maxreal*]
x_exp(r)	[⇔*maxreal*, *expmax*]
x_exp2(r)	[⇔*maxreal*, *exp2max*]
x_exp10(r)	[⇔*maxreal*, *exp10max*]
x_ln(r)	[*minreal*, *maxreal*]
x_log2(r)	[*minreal*, *maxreal*]
x_log10(r)	[*minreal*, *maxreal*]
x_sin(r)	[⇔*maxreal*, *maxreal*]
x_cos(r)	[⇔*maxreal*, *maxreal*]
x_tan(r)	[⇔*maxreal*, *maxreal*]
x_cot(r)	[⇔*maxreal*, ⇔*cotmin*] or [*cotmin*, *maxreal*]
x_arcsin(r)	[⇔1, 1]
x_arccos(r)	[⇔1, 1]
x_arctan(r)	[⇔*maxreal*, *maxreal*]
x_arctan2(r1,r2)	*r1* = *r2* = 0 not allowed
x_arccot(r)	[⇔*maxreal*, *maxreal*]
x_sinh(r)	[⇔*hypmax*, *hypmax*]
x_cosh(r)	[⇔*hypmax*, *hypmax*]
x_tanh(r)	[⇔*maxreal*, *maxreal*]
x_coth(r)	[⇔*maxreal*, ⇔*cotmin*] or [*cotmin*, *maxreal*]
x_arsinh(r)	[⇔*maxreal*, *maxreal*]
x_arcosh(r)	[1, *maxreal*]
x_artanh(r)	[⇔*(one–eps)*, *(one–eps)*]
x_arcoth(r)	[⇔*maxreal*, ⇔*(one + eps)*] or [*(one+eps)*, *maxreal*]

Figure 6.4: Domains of *real* functions with a posteriori error estimation

	decimal	hexadecimal
hypmax	$7.104\,758\,600\,739\,439 \cdot 10^{2}$	`408633ce8fb9f87d`
expmax	$7.097\,827\,128\,933\,84 \cdot 10^{2}$	`40862e42fefa39ef`
exp2max	$1.023\,999\,999\,999\,999\,9 \cdot 10^{3}$	`408fffffffffffff`
exp10max	$3.082\,547\,155\,599\,167 \cdot 10^{2}$	`40734413509f79fc`
cotmin	$5.562\,684\,646\,268\,008 \cdot 10^{-309}$	`0004000000000001`
(one–eps)	$0.999\,999\,999\,999\,999\,9$	`3fefffffffffffff`
(one+eps)	$1.000\,000\,000\,000\,000\,2$	`3ff0000000000001`
maxreal	$1.797\,693\,134\,862\,315\,8 \cdot 10^{308}$	`7fefffffffffffff`
minreal	$4.940\,656\,458\,412\,465\,4 \cdot 10^{-324}$	`0000000000000001`

There are two *real* constants *maxreal* and *minreal* defined in module *x_real* that hold the largest and smallest positive real value, respectively.

6.5.4 Formatted Input/Output for *real* Values

The representation of *real* input and output values in a hexadecimal notation is made available by overloading the procedures *read* and *write* for *real* arguments.

```
procedure read ( var f : text; var r : real; mode : char );
procedure write( var f : text;     r : real; mode : char );
```

Both the input format and the output format is a fixed-format 16 digit hexadecimal notation. The left-most hexadecimal digit holds bit 63 through bit 60 of the floating-point number format as specified in Figure 5.3. The right-most hexadecimal digit holds bit 3 through bit 0.

Procedure *read* for hexadecimal input does not distinguish between lower case hexadecimal digits `a`, `b`, `c`, `d`, `e`, `f` and upper case hexadecimal digits `A`, `B`, `C`, `D`, `E`, `F`. Possible `mode` characters are 'x' and 'X'.

Procedure *write* may be called with `mode` character 'x' or 'X', which will produce lower case hexadecimal digits `a`, `b`, `c`, `d`, `e`, `f` or upper case hexadecimal digits `A`, `B`, `C`, `D`, `E`, `F`, respectively.

The PASCAL–XSC statement

```
writeln(1.0:'x',' = ',1.0:'X')
```

yields the following output line:

```
3ff0000000000000 = 3FF0000000000000
```

6.5.5 IEEE Exception Handling Routines

In order to manipulate the exception handling environment of IEEE exceptions, a number of constants are defined which can be used together with the PASCAL–XSC procedures `IEEE_environment` and `IEEE_trap_enable`.

```
IEEE_INV_OP
IEEE_DIV_BY_ZERO
IEEE_OVERFLOW
IEEE_UNDERFLOW
IEEE_INEXACT
IEEE_ALL

IEEE_CONTINUE
```

The constants `IEEE_INV_OP`, `IEEE_DIV_BY_ZERO`, `IEEE_OVERFLOW`, `IEEE_UNDERFLOW`, and `IEEE_INEXACT` characterize the five exceptions specified by the IEEE standard. The constant `IEEE_CONTINUE` is used for changing the exception environment.

```
procedure IEEE_environment(action  : integer;
                           handler : integer;
                           mode    : boolean);
```

Procedure `IEEE_environment` transfers an '`action`' code to the embedding environment of an IEEE exception handler which is selected by the *integer* argument 'handler'. The '`mode`' value activates a characterization if it is *true* or inactivates a characterization if it is *false*.

The '`action`' code

```
IEEE_CONTINUE
```

forces the environment to continue processing after the trap handler has terminated. Other codes may be provided by further releases of the PASCAL–XSC runtime system. The trap handler is identified by the *integer* argument '`handler`' which may have one of the values `IEEE_DIV_BY_ZERO`, `IEEE_INEXACT`, `IEEE_INV_OP`, `IEEE_OVERFLOW`, and `IEEE_UNDERFLOW`. The value of `mode` must be *true* in order to activate this characteristic of the exception handling environment. If `IEEE_CONTINUE` is selected and '`mode`' is *false*, then processing is aborted after leaving the trap handler. Otherwise the processing of the program is continued.

```
procedure IEEE_trap_enable(handler : integer; mode : boolean);
```

Procedure `IEEE_trap_enable` sets the enabled status of an IEEE exception handling routine. If the value of `mode` is *true* then the trap handler is enabled. If the value of `mode` is *false*, then the trap handler is disabled. The selection of the trap handler is done by the value of the *integer* argument `handler` which may have one of the values `IEEE_DIV_BY_ZERO`, `IEEE_INEXACT`, `IEEE_INV_OP`, `IEEE_OVERFLOW`, and `IEEE_UNDERFLOW`.

An example for the usage of the procedures `IEEE_environment` and `IEEE_trap_enable` is given in Appendix D *IEEE Exception Handling Environment*.

```
function IEEE_test(handler : integer) : boolean;
```

Function `IEEE_test` returns the value of the exception flag that is associated with the IEEE exception identified by '`handler`'. The exception flag is not changed by this function. Exceptions flags stay *set* until they are explicitly *reset* via calls to procedure `IEEE_reset`.

Before processing the first PASCAL–XSC statement all exception flags are reset. Note that the first PASCAL–XSC statement may be placed in an included module and may affect the exception flags before processing the first statement of the program body. Thus, if IEEE exception handling routines are used, it is recommended that all exception flags are explicitly reset in the program body before processing is started.

The exception flag is set explicitly by the user via procedure 'IEEE_set' or automatically by all IEEE floating-point operations if the corresponding exceptional conditions are met. Exception flags are set independent of the enabled status of the related trap handler. Valid numbers for the *integer* argument 'handler' are IEEE_DIV_BY_ZERO, IEEE_INEXACT, IEEE_INV_OP, IEEE_OVERFLOW, and IEEE_UNDERFLOW.

```
procedure IEEE_set(handler : integer);
```

Procedure IEEE_set sets the value of the exception flag that is associated with the IEEE exception identified by 'handler'. Valid numbers for the *integer* argument 'handler' are IEEE_ALL, IEEE_DIV_BY_ZERO, IEEE_INEXACT, IEEE_INV_OP, IEEE_OVERFLOW, and IEEE_UNDERFLOW. If IEEE_ALL is selected, then all IEEE flags are set.

```
procedure IEEE_reset(handler : integer);
```

Procedure IEEE_reset resets the value of the exception flag that is associated with the IEEE exception identified by 'handler'. Valid numbers for the *integer* argument 'handler' are IEEE_ALL, IEEE_DIV_BY_ZERO, IEEE_INEXACT, IEEE_INV_OP, IEEE_OVERFLOW, and IEEE_UNDERFLOW. If IEEE_ALL is selected, then all IEEE flags are reset.

```
procedure IEEE_save(var x : integer);
```

Procedure IEEE_save saves the setting of all IEEE exception flags and all IEEE trap enabled flags as value of the integer variable 'x'. The complete flag settings saved in variable 'x' may be restored by calling IEEE_restore with argument 'x'

```
procedure IEEE_restore(x : integer);
```

Procedure IEEE_restore restores the setting of all IEEE exception flags and all IEEE trap enabled flags from the integer value 'x'.

6.6 Module *x_strg*

```
procedure x_release ( var s : string );
```

Procedure x_release is only meaningful for actual arguments that are string variables without static length specification in the declaration. The purpose is to release the unused portion of a string variable, i.e., the length of the allocated string is reduced to the length of the currently stored string. The current string may be shorter due to a call to standard procedure *setlength* or the assignment of a 'shorter' string. Usually an allocated string space is not automatically returned to the heap management during processing since it is not known whether pending information in the string variable is still relevant.

6.7 Modules *lss*, *ilss*, *clss*, *cilss*

Beyond other application modules delivered with the PASCAL–XSC system, there may be modules for the solution of linear systems of equations and the inversion of a matrix.

The modules named *lss*, *ilss*, *clss*, and *cilss* each contain two procedures called `LSS` and `INV` for matrix and vector arguments with component types *real*, *interval*, *complex*, and *cinterval*, respectively.

For modules **x***lss* (**x**=empty, *i*, *c*, *ci*) procedures `LSS` and `INV` are declared in the following form with **y**=`R`, `I`, `C`, `CI` and **z**=`I`, `I`, `CI`, `CI` according to the choice of **x**.

```
procedure LSS (  var A : yMATRIX
                 var B : yVECTOR
                 var Y : zVECTOR
                 var ERRCODE : INTEGER )

procedure INV (  var A : yMATRIX
                 var Y : zMATRIX
                 var ERRCODE : INTEGER )
```

`LSS` solves quadratic and over-determined and under-determined linear systems of equations of the form

$$A \cdot x = b$$

with A an m-by-n matrix, b an m-vector, and x an n-vector. The verified enclosure of the exact solution x is returned by argument `Y`. The contents of arguments `A` and `B` are not changed.

`INV` determines the inverse of the m-by-n matrix A if m is equal to n. Otherwise, the pseudo-inverse (Moore-Penrose-Inverse) is determined. The enclosure of the exact inverse is returned by argument `Y`. The contents of argument `A` are not changed.

Both routines return an exception code via argument `ERRCODE`. The following states are distinguished:

- for `LSS` and `INV`:

  `ERRCODE`=0 : Everything is o.k. and no exceptions occurred.

  `ERRCODE`=1 : No enclosure determined due to ill-conditioned matrix.

  `ERRCODE`=2 : No enclosure determined since matrix possibly is singular (if m=n) or does not possess full rank (if m≠n).

- for `LSS` only:

  `ERRCODE`=3 : Wrong dimensions : number of rows of `A` is different from number of elements of `B`.

`ERRCODE`=4 : Wrong dimensions : number of columns of `A` is different from number of elements of `Y`.

- for `INV` only:

`ERRCODE`=3 : Wrong dimensions : number of columns of `A` is different from number of rows of `Y`.

`ERRCODE`=4 : Wrong dimensions : number of rows of `A` is different from number of columns of `Y`.

Note : Both routines are able to deliver exact solutions, i.e., the diameter of all components of `Y` are zero. This is the case, if the residual of the approximate solution is exactly zero, i.e., the approximate solution is the exact solution. Consequently, the interval iteration process is not performed and the uniqueness of the solution is not guaranteed. Nevertheless, `ERRCODE`=0 is returned, since `Y` represents an exact solution.

Modules *lss_aprx* and *clss_aprx* contain procedures of the following form with **y**=`R` or `C`, respectively.

```
procedure MINV (  var A : yMATRIX
                  var ERR : INTEGER )

procedure MINV1 ( var W : yMATRIX
                  var ERR : INTEGER )
```

These routines are used by all procedures `LSS` and `INV`.

`MINV` determines an approximate inverse of a quadratic matrix A using the Gauß-Jordan-Algorithm (with column pivots). Argment `A` is replaced by the determined approximate inverse. There is no checking for a quadratic matrix.

`MINV1` determines an approximate inverse of the quadratic matrix $A = I + W$ using the Gauß-Jordan-Algorithm (without pivoting). Only the difference $W = A \Leftrightarrow I$ is passed as argument. Analogously, only the difference of the determined approximate inverse from the unit matrix I is returned. If A^{-1} denotes the exact inverse of A, W_{in} the input matrix, and W_{out} the output matrix of `MINV1`, then $A = I + W_{in}$ and $A^{-1} \approx I + W_{out}$. There is no checking for a quadratic matrix.

Both routines return an exception code by argument `ERR`.

- for `MINV` and `MINV1`:

`ERR`=0 : Everything is o.k. and no exception occurred.

`ERR`=1 : No approximate inverse is determined, since the matrix possibly is singular.

Appendix A

Deviations

A.1 Deviations from Standard PASCAL

This section contains deviations of the current implementation of the PASCAL–XSC compiler as well as of the PASCAL–XSC language description in [4] from standard PASCAL. Deviations are those properties, that make programs written in standard PASCAL be uncompilable with the PASCAL–XSC compiler, or will produce different results when standard PASCAL programs compiled with the PASCAL–XSC compiler are executed. This section does not contain PASCAL–XSC extensions and details of the implementation.

The most important deviation of the language PASCAL–XSC from Standard PASCAL results from the PASCAL–XSC concept of overloading names of subroutines.

1. redefinition

   A local subroutine may overload a global subroutine instead of redefining it. For example:

   ```
   procedure p(x:integer) ;  begin {...} end ;  { outer p }

   procedure main ;

     procedure p(x:real) ;  begin {...} end ;  { inner p }

   begin { main }
      p(1) ;                              { call of p }
   end ;
   ```

   In standard PASCAL, the inner procedure `p` is called, because the outer procedure `p` is redefined and, therefore, is not available when `p` is called in procedure `main`. In PASCAL–XSC the inner procedure `p` overloads the outer procedure `p`, thus, both procedures (with different lists of arguments) are available.

The outer procedure `p` is called, because its formal parameter type matches the type of the actual arguments.

2. forward declaration

   When defining a forward declared procedure or function, the formal parameter list must be repeated in order to identify the subroutine uniquely.
   Refer to section 2.7.8 in [4].

3. **goto** statement

   A **goto** statement must not leave the immediately surrounding block.
   Refer to 5.5.5 **goto**-*Statement*.

4. standard procedures *pack*, *unpack*

   The standard procedures *pack* and *unpack* are not recognized.

5. file type

   A file type must not be the component type of an array or the type of a record component. File variables must not be referenced by a pointer.
   Refer to 5.3.2.10 *Files*.

6. set elements

   The ordinal numbers of set elements are restricted to the range from 0 through 255.
   Refer to 5.3.2.9 *Sets*.

7. type compatibility

   Different subrange types as well as different set types can not be converted automatically. An explicit type conversion by means of a type name is required.
   Refer to 5.3.5 *Compatibility of Types*.

8. pointers

   Forward declared data types must be record types.
   Refer to 5.3.4 *Pointers*.

9. functions

   The assignment to a function result must be in the statement part of the function and not in an inner block.
   Refer to 5.7.3 *Functions*.

10. keywords

    New reserved keywords are:
    operator, **use**, **dynamic**, **global**, **priority**, **module**.
    Refer to section 2.1 *Basic Symbols* in [4].

The identifier **external** is not a reserved keyword. **sum** is reserved only immediately after the keywords **to** and **downto** in an accurate expression.

11. unary '+' and unary '-'

    The unary operators '+' and '-' have highest priority.
    Refer to section 2.4.1 *Expressions* in [4].

12. *read* and *write*

    The procedures *read* and *write*, *readln* and *writeln* may be redefined only in a special way.
    Refer to section 2.7.11 *Overloading of read and write* in [4].

A.2 Deviations from PASCAL–XSC

This section contains deviations of the implemented PASCAL–XSC compiler concerning the language definition of PASCAL–XSC. Deviations are those properties, that make PASCAL–XSC programs, which have successfully been processed by an older version of the PASCAL–XSC compiler (Atari code generating version), be uncompilable, or will produce different result when processed by the C generating version of the PASCAL–XSC compiler, or make programs uncompilable that are written strictly conformant with the language description. This section does not contain extensions and implementation details. Deviations from Standard PASCAL are listed separately in the preceding section A.1.

1. file types
   File types must have neither *dotprecision* nor *string* components.
   Refer to 5.3.2.10 *Files*.

2. variant part
   Variant parts of records must have neither *dotprecision* nor *string* components.
   Refer to 5.3.2.8 *Records with Variants*.

3. hexadecimal constants
   Hexadecimal constants are not implemented.
   Refer to 5.3.1.1 *integer*.
   Character $ is not a basic symbol.
   Refer to 5.1 *Basic Symbols*.

4. standard function *loc*
   The standard function *loc* is not implemented.
   Refer to 5.7.5 *List of Predefined Functions*.

5. functions *mark* and *release*
   The functions *mark* and *release* are not implemented.
   Refer to 5.7.2 *List of Predefined Procedures and Input/Output Statements*.

6. Type compatibility
   Refer to 5.3.5 *Type Compatibility*.

7. Selecting routines
   The process of selecting overloaded routines is changed.
   Refer to 5.7.10 *Overloading of Procedures, Functions, and Operators*.

Appendix B

Syntax Diagrams

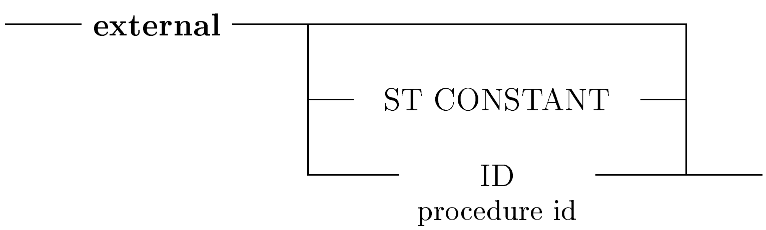
This section briefly summarizes the syntactical differences of the current implementation of the PASCAL–XSC compiler compared with the PASCAL–XSC language description [4].

P7 CONSTANT

The notation of a hexadecimal integer constant is not implemented. The character '$' is not a basic symbol.

P21 BODY

After the compiler directive **external** a syntactically correct PASCAL–XSC identifier name is allowed which represents an external entry name.



P26 STANDARD PROCEDURE CALL

Procedures *mark* and *release* are not implemented.

P37 STANDARD FUNCTION CALL

Functions *loc* is not implemented.

Appendix C

Runtime Messages

During the processing of a PASCAL–XSC program unexpected exceptional conditions may occur. Possible reasons may be, e.g., the occurrence of input/output errors, mathematical errors, or errors caused by the operating system. Exceptions detected by the PASCAL–XSC system are communicated to the user by displaying messages on the standard error device *"stderr"*. The device for displaying messages may be altered. Refer to the *local installation guide* for the actual setting.

```
--- Division by zero.
---    left operand : 0x3ff0000000000000
---   right operand : 0x0000000000000000
---          result : 0x7ff0000000000000
--- ERROR at line 25 in 'DIVIDE'
--- 'DIVIDE' defined in 'mod1.p' is called in 'mod2.p' at line 53.
--- 'DIVMAT' defined in 'mod2.p' is called in 'mymod.p' at line 20.
--- 'DIVMAT' defined in 'mymod.p' is called in 'prog.p' at line 134.
--- 'DIVISION' defined in 'prog.p' is called in 'prog.p' at line 30.
--- 'PROG' defined in 'prog.p' is called by operating system.
```

Figure C.1: Example for an exception message

All messages which are caused by the exception handling routines are preceded by a header string composed of three dashes: '`---␣`'. The symbol ␣ stands for a blank character. A typical reaction on an exceptional condition is composed of three blocks of messages:

1. A descriptive message text.

2. A list of actual PASCAL–XSC values which were used when the exception occurred.

3. A function trace back which reflects the actual (dynamic) nesting of PASCAL–XSC subroutines at the moment when the exception occurred.

An example for displayed messages in case of a DIVISION BY ZERO exception is given in Figure C.1.

C.1 Descriptive Messages

The possible descriptive message text lines of an exception detected by the PASCAL–XSC runtime system are listed in alphabetic order.

`Allocation failed :`
Allocation of dynamic storage failed. Additional text specifies whether an attempt was made to allcoate a *dotprecision* variable, a dynamic array, a dynamic string, or a user-defined object.

`Division by zero`
An attempt is made to divide a non-zero value by zero. In case of a *real* division, the IEEE exception handling environment is active (refer to Appendix D *IEEE exception handling environment* on page 95). The division of zero by zero or infinity by infinity causes an invalid operation exception.

`Equal length of dynamic vectors expected.`
One-dimensional dynamic arrays with different lengths are used as operands where equal length is required.

`Error in I/O operation :`
Input and output errors may be caused by a variety of reasons. Therefore, additional text describes the individual fact that produced a PASCAL–XSC runtime exception.

- Command line errors :
  `Missing command line argument.`
- Open errors :
  `Empty string.`
  `Filename too long.`
  `Invalid file name.`
  `Missing variable name.`
  `No file name has previously been assigned.`
  `No device assigned.`
  `Unable to open file for reading.`
  `Unable to open file for writing.`
  `Standard I/O must not be used for binary I/O.`

– Read and write errors :
  `Device not a binary device.`
  `Device not a TEXT device.`
  `Device not opened for reading.`
  `Device not opened for writing.`
  `Error writing data to file.`
  `Invalid read/write mode.`
  `Invalid syntax of hexadecimal value.`
  `Invalid syntax of integer value.`
  `Invalid syntax of real value.`
  `Unexpected End-Of-File.`
  `Unexpected End-Of-Line.`
  `No digits in string.`

`Evaluation error possibly caused by invalid argument.`
A PASCAL–XSC standard function is processed that caused an exception. Most often one of the actual argument values is invalid.

`Exponent range restricted.`
The exponent range for *real* input operations is restricted to the integers in the interval [-999,999]. Nevertheless, the conversion is processed according to the specified rounding eventually causing an OVERFLOW or UNDERFLOW exception.

`Exponent too large (infinity returned).`
An attempt is made to explicitly generate a *real* number with a non-representable large exponent.

`Exponent too small (zero returned).`
An attempt is made to explicitly generate a *real* number with a non-representable small exponent.

`Function call with matrices of different column lengths.`
Message caused by a mismatching of arguments in a PASCAL–XSC dot product expression (#-expression).

`Function call with matrices of different row lengths.`
Message caused by a mismatching of arguments in a PASCAL–XSC dot product expression (#-expression).

`Function call with vectors of different lengths.`
Message caused by a mismatching of arguments in a PASCAL–XSC dot product expression (#-expression).

`Index out of range.`
An index or element value outside the actually valid range is used.

`Inexact`
The result of an operation is inexact and a rounding operation is applied to the delivered result. In case of the *real* operations $+$, $\Leftrightarrow$, $*$, $/$, the IEEE exception handling environment is active (refer to Appendix D *IEEE exception handling environment* on page 95).

`Inexact conversion of decimal constant.`
The result of a conversion routine is inexact and a rounding operation is applied to the delivered result.

`Inexact conversion of decimal input data.`
The result of a conversion routine is inexact and a rounding operation is applied to the delivered result.

`Internal buffer too small :`
For some internal routines reserved (static) buffers of a certain size are used to handle intermediate data. This error should not occur in a correct PASCAL–XSC program. Additional text is provided for debugging purposes.

```
Dynamic variable.
Dynamic mantissa too long.
Reading a dynamic string.
```

`Invalid operation :`
In case of the *real* operations $+$, $\Leftrightarrow$, $*$, $/$, the IEEE exception handling environment is active (refer to Appendix D *IEEE exception handling environment* on page 95). Additional text specifies the details of an invalid operation exception.

```
0*infinity
0/0
Signaling NaN as operand
infinity-infinity
infinity/infinity
```

`Invalid width of output field.`
A formatted write operation is done with an invalid field width.

`Mantissa bits lost on generating denormalized number.`
An attempt is made to explicitly generate a *real* number with a non-representable mantissa.

`Mantissa out of range (1.0<= |mantissa| <2.0).`
An attempt is made to explicitly generate a *real* number with an invalid mantissa argument.

`Mismatching index ranges.`
The index ranges of dynamic arrays do not match.

`Mismatching inner lengths in a matrix-matrix product.`
Message caused by a mismatching of arguments in a PASCAL–XSC dot product expression (#-expression).

`Mismatching inner lengths in a matrix-vector product.`
Message caused by a mismatching of arguments in a PASCAL–XSC dot product expression (#-expression).

`Mismatching inner lengths of arguments in a function call.`
Message caused by a mismatching of arguments in a PASCAL–XSC dot product expression (#-expression).

`Non-positive modulo value.`
A non-positive modulo value is used.

`One-dimensional dynamic array expected.`
A one-dimensional dynamic array is expected.

`Overflow occurred`
An overflow exception occurred for a *real* or *integer* operation. In case of the *real* operations $+$, ⇔, $*$, $/$, the IEEE exception handling environment is active (refer to Appendix D *IEEE exception handling environment* on page 95).

`------ Processing aborted ------`
Message displayed by the exception handler when processing is aborted.

`Range of integer data type exceeded.`
An integer value exceeding the range from -*maxint*-1 to *maxint* inclusively results from an operation.

`Scalar product of vectors with different lengths.`
Message caused by a mismatching of arguments in a PASCAL–XSC dot product expression (#-expression).

`Substring destination array shorter than required.`
The length of a substring array on the left-hand side of an assignment is smaller than the length of the string value on the right-hand side.

`Summation of matrices with different column lengths.`
Message caused by a mismatching of arguments in a PASCAL–XSC dot product expression (#-expression).

`Summation of matrices with different row lengths.`
Message caused by a mismatching of arguments in a PASCAL–XSC dot product expression (#-expression).

`Summation of vectors with different lengths.`
Message caused by a mismatching of arguments in a PASCAL–XSC dot product expression (#-expression).

`Undefined device specification.`
An invalid device specification is used in an extended open statement.

`Underflow occurred`
An underflow exception occurred for a *real* operation. In case of the *real* operations $+$, $\Leftrightarrow$, $*$, $/$, the IEEE exception handling environment is active (refer to Appendix D *IEEE exception handling environment* on page 95).

`Unexpected infinity operand.`
A *real* operand with value `infinity` is used in a *dotprecision* operation.

`Unexpected NULL pointer.`
A NULL pointer is detected in an operation that requires a pointer value referencing an allocated piece of memory. The term "NULL" stands for the special pointer value which is defined for the C compiler in use.

`Unexpected quiet NaN operand.`
A *real* operand with value `qNaN` is used in a *dotprecision* operation.

C.2 List of Values

After displaying a descriptive message text an optional list of values is given which is related to the context in which an exception occurred. Each value is described on one line by giving

1. a short text which describes the displayed value by its meaning or data type,
2. a hexadecimal representation in case of arithmetic values, and
3. a "readable" representation.

In Figure C.2, the possible short text used for describing the displayed values are listed. The number sign, `#`, stands for the number of the displayed value in the current message.

C.3 Function Trace Back

A complete exception message is terminated by a function trace back. Due to overloading of procedure and function names, the application of recursive calls, and the use of a variety of user-defined and standard PASCAL–XSC modules, it may be difficult

`# char`	`file variable`
`# dot`	`index`
`# integer`	`input data`
`# dynamic`	`left operand`
`# real`	`lower bound`
`# string`	`mantissa`
`argument`	`result`
`basis`	`right operand`
`bit number`	`set element`
`dimension`	`string length`
`error code`	`upper bound`
`exponent`	`vector length`
`file name`	

Figure C.2: Short text used in list of values

to analyze the context in which an exception occurred. Only positional information within the PASCAL–XSC source code may not be sufficient.

The function trace back displays the actual nesting of a routine which caused an exception. For each routine its name and the name of its defining module are listed together with the name of the calling module. Line number informations are available if the appropriate PASCAL–XSC compiler option **n** is specified (refer to 3.6 *PASCAL–XSC Compiler Options*).

Appendix D

IEEE Exception Handling Environment

For the *real* operations according to the IEEE standard [3], the following five classes of exceptions are defined:

- DIVISION BY ZERO
- INVALID OPERATION
- EXPONENT OVERFLOW
- EXPONENT UNDERFLOW
- INEXACT RESULT

Each of these exceptions is handled by its own exception handler which may be enabled or disabled. If the exception handler is enabled, then exception messages are displayed. By default, the exception handlers for the exceptions DIVISION BY ZERO, EXPONENT OVERFLOW, and INVALID OPERATION are enabled and cause the termination of the processing of a program after messages are displayed. The exception handlers for the exceptions EXPONENT UNDERFLOW and INEXACT RESULT are disabled and a standard corrective action (application of the specified rounding operation) is taken before processing is continued.

The default settings of the enabled status of the exception handlers may be altered. Refer to the *local configuration guide* for current settings.

The runtime option **-ieee** is provided for changing the default settings of the enabled status of the exception handlers when the processing of a compiled and linked PASCAL–XSC program is started. Refer to 4.2 *PASCAL–XSC Runtime Options* for more details.

Another possibility of changing the status of the exception handling environment for IEEE exceptions is given by additional runtime support procedures defined in module *x_real*. Refer to 6.5 Module x_real.

Example for the use of procedures `IEEE_trap_enable` and `IEEE_environment` declared in module *x_real*.

```
program ieeetest;

use x_real;

begin
   IEEE_environment(IEEE_CONTINUE,IEEE_DIV_BY_ZERO,true);

   IEEE_trap_enable(IEEE_DIV_BY_ZERO,false);
   writeln('Divide 1 by 0 : ',1.0/0.0); writeln;
   IEEE_trap_enable(IEEE_DIV_BY_ZERO,true);
   writeln('Divide 1 by 0 : ',1.0/0.0); writeln; { line 11 }

   IEEE_environment(IEEE_CONTINUE,IEEE_DIV_BY_ZERO,false);

   IEEE_trap_enable(IEEE_DIV_BY_ZERO,false);
   writeln('Divide 1 by 0 : ',1.0/0.0); writeln;
   IEEE_trap_enable(IEEE_DIV_BY_ZERO,true);
   writeln('Divide 1 by 0 : ',1.0/0.0); writeln; { line 18 }
end.
```

If program `ieeetest` is compiled with compiler option '`+n`' (see 3.6.2 *Code Generation Options*), then the processing of `ieeetest` yields the following output on a terminal screen.

```
Divide 1 by 0 :              +infinity

--- Division by zero
--- ERROR at line 11 in 'ieeetest.p'
Divide 1 by 0 :              +infinity

Divide 1 by 0 :              +infinity

--- Division by zero
--- ERROR at line 18 in 'ieeetest.p'
------ Processing aborted ------
Divide 1 by 0 :
```

Processing is aborted at line 18, since trap handling is enabled for `IEEE_DIV_ZERO` and the environment for the exception handler is set up to terminate the execution of the program after leaving the exception handler.

Appendix E

ASCII Collating Sequence

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

Figure E.1: ASCII collating sequence

Bibliography

[1] American National Standard for Information Systems, *Pogramming Language C, X3.159-1989*, 1989.

[2] G. Bohlender, L. B. Rall, Ch. Ullrich, J. Wolff v. Gudenberg: *PASCAL-SC Wirkungsvoll programmieren, kontrolliert rechnen*, Bibliographisches Institut Mannheim, 1986.

[3] *IEEE Standard for Binary Floating-Point Arithmetic, IEEE Std 754-1985*, 1985.

[4] R. Klatte, U. Kulisch, M. Neaga, D. Ratz, Ch. Ullrich: *PASCAL-XSC, Sprachbeschreibung mit Beispielen*, Springer-Verlag, Heidelberg, 1991.
R. Klatte, U. Kulisch, M. Neaga, D. Ratz, Ch. Ullrich: *PASCAL-XSC, Language Reference with Examples*, Springer-Verlag, Heidelberg, to be published in 1992.

Index

A

B

C

D

E

N

O

P

Q

R