Contents

1 Preface . . . 1

2 Pascal-XT programming system . . . 5
2.1 Call and overview . . . 5
2.2 User guidance . . . 9
2.3 Extension of the programming system . . . 11
2.4 Termination behavior of the programming system . . . 13
2.5 Modes of operation of the programming system . . . 16
2.6 Statements to the programming system . . . 17
2.6.1 Definitions and notes . . . 17
2.6.2 ADD-TOOL . . . 21
2.6.3 CALL-STATEMENT-FILE . . . 22
2.6.4 COMPILE-UNIT . . . 24
2.6.5 DEFINE-PROJECT-FILE . . . 32
2.6.6 EDIT-UNIT . . . 33
2.6.7 END . . . 39
2.6.8 MODIFY-COMPILE . . . 40
2.6.9 MODIFY-EDIT . . . 42
2.6.10 REMOVE-DIRECTORY-ENTRY . . . 43
2.6.11 RUN-PROGRAM . . . 44
2.6.12 SHOW-ATTRIBUTES . . . 48
2.6.13 STEP . . . 56
2.6.14 SYSTEM-COMMAND . . . 57
2.7 Examples . . . 58
Developing a main program . . . 58
Working with PLAM libraries and the project directory . . . 66

3 Project directory . . . 79
3.1 Tasks of the project directory . . . 79
3.2 Defining and processing the project directory . . . 81
3.3 Hints on working with the project directory . . . 83
Example: Working with the project directory . . . 84

4 Pascal-XT compiler 91
4.1 Using the project directory 91
4.2 Implementation-defined attributes 93
4.3 Representation of objects in main memory 96
4.4 Generated object modules 99
4.5 Compiler options 101
4.6 Listings generated by the compiler 103
4.6.1 Controlling listing output 103
4.6.2 Source listing 104
4.6.3 Errors, warnings and notes 106
4.6.4 Assembler listing 108
4.6.5 Cross-reference listing 109
4.6.6 Map listing 111

5 Files 113
5.1 Pascal files 113
5.1.1 External Pascal files 113
5.1.2 Local Pascal files 113
5.2 Supported BS2000 files and libraries 114
5.2.1 Standard files 116
5.2.2 SAM and ISAM files 116
5.2.3 PLAM library elements 118
5.2.4 Temporary files 119
5.3 Assigning BS2000 files to Pascal files 120
5.3.1 Default assignments 120
5.3.2 Assignment with the FILE command 120
5.3.3 Assignment with the predefined procedure assignfile 124
5.3.4 Assignment in the RUN statement 131
5.4 File operations 132

6 Linking and executing object programs 135
6.1 General 135
6.2 Static linking 138
6.2.1 Linking to form a phase 138
6.2.2 Prelinking to form prelinked modules 140
6.2.2.1 Prelinking to form a single prelinked module 141
6.2.2.2 Prelinking code and data modules separately 143
6.2.2.3 Prelinking the runtime system 146
6.2.3 Segmented linking 146
6.3 Dynamic linking 153
6.4 Program termination code 155
6.5 License protection for the Pascal-XT runtime system 156

7 Language interfaces . . . 157
7.1 ILCS program communication interface . . . 158
7.1.1 ILCS register conventions . . . 159
7.1.2 ILCS data structures . . . 160
7.1.3 Initializing the Pascal-XT runtime system . . . 161
7.1.4 Program mask handling by ILCS . . . 161
7.1.5 Parameter passing in ILCS program systems . . . 162
7.1.6 Linking ILCS program systems . . . 163
7.2 Interfacing non-Pascal subprograms . . . 164
7.3 Invocation by programs in other languages . . . 171
7.4 Internal interface . . . 176

8 UTM linkage . . . 189
Language interfaces under UTM . . . 190
Data types and constants . . . 190
Basic structure of UTM program units . . . 191
Requesting memory with "NEW" . . . 192
External files . . . 192
Error handling as of Pascal-XT V2.2A . . . 193
Error handling up to Pascal-XT V2.2A . . . 193
Linking the application . . . 194

9 Debugging aid PATH . . . 195
9.1 Features and characteristics of PATH . . . 195
9.1.1 Command summary . . . 197
9.1.2 Definition of terms . . . 198
9.1.3 Syntax elements . . . 199
9.1.3.1 Debugging aid comments and options . . . 202
9.1.4 Testpoints . . . 203
9.1.4.1 Testpoints before program start . . . 203
9.1.4.2 User-set testpoints . . . 203
9.1.4.3 Postmortem testpoints . . . 206
9.1.4.4 Exception testpoints . . . 207
9.1.4.5 Entry testpoints . . . 209
9.1.5 Scope of identifiers . . . 212
9.1.6 Access to identifier types . . . 214
9.1.7 Generation of test tables . . . 214
9.2 PATH commands . . . 215
9.2.1 Testpoint commands . . . 216
9.2.1.1 AT command . . . 216
9.2.1.2 GETCMD command . . . 218
9.2.1.3 RESUME command . . . 219
9.2.1.4 REMOVE command . . . 220
9.2.1.5 SLEEP command . . . 222
9.2.1.6 AWAKE command . . . 223

9.2.1.7 Example of the interaction between testpoint commands 225
9.2.2 Action commands 226
9.2.2.1 DISPLAY command 226
9.2.2.2 ASSIGN command 230
9.2.2.3 IF command 231
9.2.2.4 Compound command 232
9.2.2.5 SYSTEM command 233
9.2.2.6 EDIT command 234
9.2.2.7 SHOW command 235
9.2.2.8 DUMP command 239
9.2.2.9 KILL command 240
9.2.2.10 SWITCH command 241
9.3 Debugging aid messages 242
Error messages regarding test tables 244
Errors in testpoint commands 244
Errors in action commands 245
9.4 Linking with PATH 248
9.5 Testing with PATH 249

10 Runtime errors and error handling 255
10.1 STXIT events and ILCS 256
10.2 Error handling and output in the event of an error 257
10.2.1 Pascal-XT handling of SEH events 257
10.2.2 The Pascal-XT Break_Error 259
10.2.3 Language interfacing between Pascal-XT and Assembler 259
10.2.4 Output when a runtime error occurs 260
10.3 Detecting runtime errors 273
10.4 System error codes 285

A Appendix 289
A.1 Comparison between Pascal (BS2000) version 3.x and Pascal-XT . . 289
A.2 Compiler listings 301
A.3 Predefined packages 306
A.4 BS2000CALLS 307
A.5 DMSIO 312
A.6 EDTADAPTER 327
A.7 ERRORS 331
A.8 HEAPSUPPORT 333
A.9 MEMORYMANAGER 339
A.10 Compatibility problems between Pascal-XT V2.2 and V3.0 341

References 343

Index 351

1 Preface

The Pascal-XT language is an extension of Standard Pascal. It is available on a number of different processors and with a variety of operating systems. This guide describes how to use the compiler under the BS2000 operating system.

What previous knowledge is required?

You should have a knowledge of the Pascal-XT language, of the BS2000 command language, of the Data Management System (DMS) and of the linkage editors TSOSLNK and DLL.

Literature references appear in the text as figures in square brackets. The precise title of each publication referred to is given under "References".

The Pascal-XT V2.1 language set remains applicable to this version (V2.2A); it is described in the Pascal-XT (SINIX, BS2000) Language Reference Manual [1].

What does this User Guide contain?

This User Guide describes

- operation of the Pascal-XT programming system
- BS2000-specific attributes of the Pascal-XT compiler
- assignment of BS2000 files to Pascal files
- linking and execution of programs
- language interfacing with external (non-Pascal) programs (ILCS interface)
- UTM linkage
- PATH debugging aid and testing of programs
- runtime error messages
- comparison of the languages Pascal (BS2000) V3.x and Pascal-XT
- predefined BS2000-specific packages.

User inputs

In the examples user inputs are highlighted by shading, e.g.:

```
/EXEC $USERID.PASCAL-XT
```

Changes since the last version of the manual

Section	Topic	new	modified	deleted
2.3	Example			x
2.6.3	CALL-STATEMENT-FILE, PLAM version number		x	
2.6.4	Compiler option MESSAGE-LEVEL COMPILE-UNIT, PLAM version number	x	 x	
2.6.6	EDIT-UNIT, PLAM version number		x	
2.6.8	MODIFY-COMPILE, PLAM version number		x	
2.6.9	MODIFY-EDIT, PLAM version number		x	
2.6.11	RUN-PROGRAM for language interfacing	x		
2.6.12	SHOW-ATTRIBUTES, PLAM version number		x	
3	Project directory		x	
4.1	Pascal-XT compiler, using the project directory		x	
4.2	Ulp-precision mathematical routines (5)	x		
4.5	Compiler option MESSAGE-LEVEL Table of compiler options Description of option specification	x	 x x	
4.6	Four-digit year number in compiler listing	x		
4.6.2	Compilation summary: Output of suppressed messages		x	
4.6.3	Output of suppressed messages in the compilation summary and to SYSOUT		x	
5.2.3	PLAM version number Note		x x	 x
5.3.3	ASSIGNFILE, PLAM version number		x	
6.1	Linking, general Pascal-XT runtime system and ILCS Compatibility between different versions	 x x	x	
6.2.2	Prelinking to form prelinked modules		x	
6.5	License protection	x		

Section	Topic	new	modified	deleted
7	Language interfaces, general section	x		
7.1	General			x
	Program communication interface ILCS	x		
7.2	Connecting subprograms written in other languages		x	
7.3	Invocation by programs in other languages		x	
8	UTM linkage		x	
	New ILCS interface	x		
	Language interfaces	x		
	Requesting memory with "New"	x		
	Error handling		x	
	Linking		x	
9	Debugging aid PATH		x	
9.1.4.3	Postmortem testpoints		x	
9.1.4.4	Exception testpoints		x	
9.2.2.7	Four-digit year number in the PATH command SHOW-UNITS	x		
9.2.2.9	PATH command KILL		x	
9.5	Restarting the program under test		x	
10	Runtime errors and error handling		x	
10.1	STXIT events and ILCS		x	
10.2	Error handling and output in the event of an error		x	
10.2.1	Handling of SEH events	x		
10.2.2	Pascal-XT Break_Error		x	
10.2.3	Language interfacing between Pascal-XT and Assembler	x		
10.2.4	Output when a runtime error occurs		x	
	Dynamic call chain		x	
	Restrictions for versions <= 2.1A			x
A.10	Compatibility problems between Pascal-XT versions 2.2 and 3.0	x		

In addition to these changes, program examples have been expanded and/or corrected, text formatting has been improved and any typographical errors detected have been removed.

2 Pascal-XT programming system

2.1 Call and overview

The Pascal-XT programming system provides a convenient environment for interactively creating, compiling and testing programs without having to leave the programming system. Beyond this, the system provides additional functions to facilitate software development. The normal mode of operation of the programming system is interactive. However, it can also be used to its full extent in batch mode.

Fig. 2-1 shows the statements of the programming system and the ways of accessing the various BS2000 files.

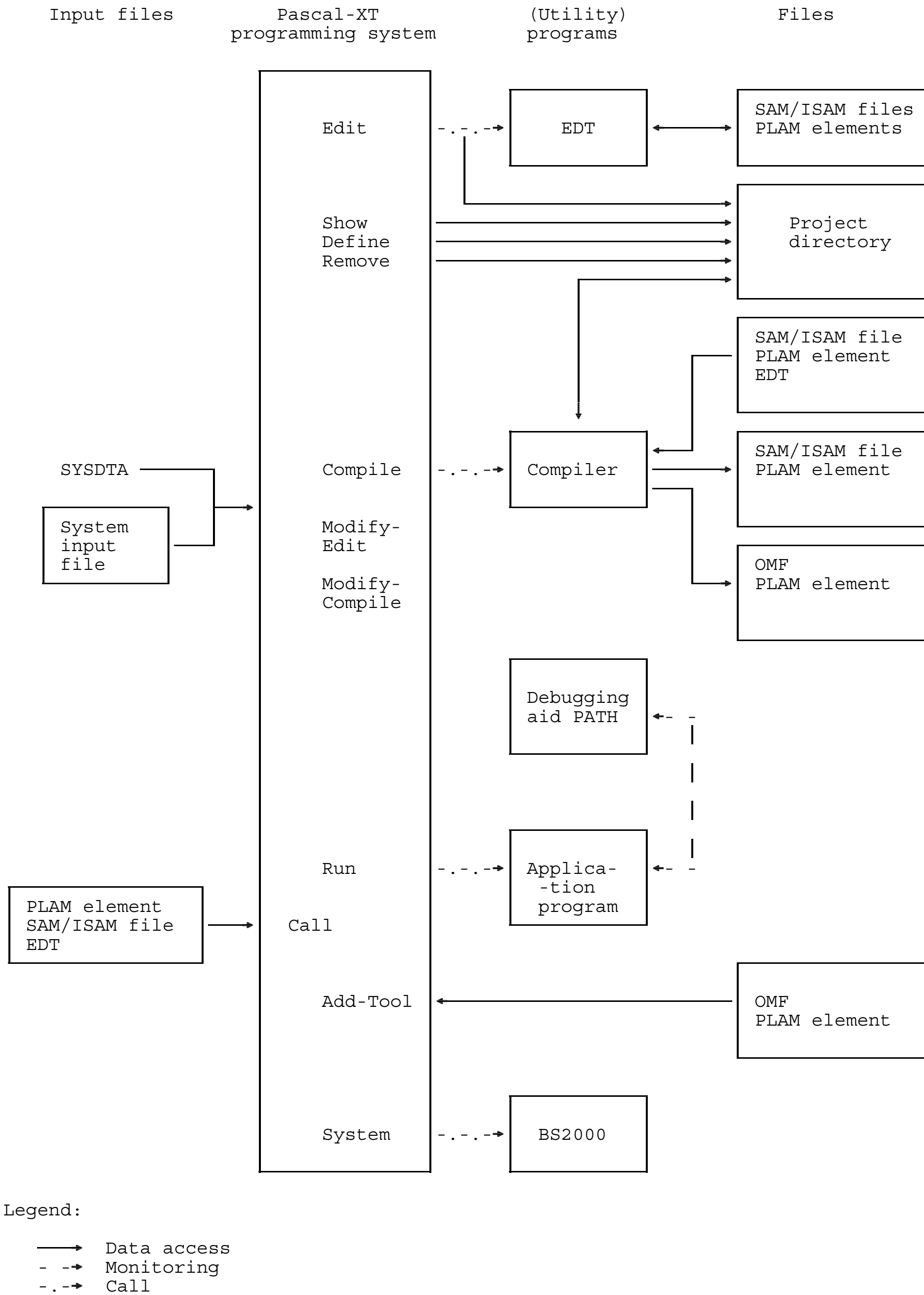

Fig. 2-1 Statements and access capabilities of the Pascal-XT programming system

Calling the Pascal-XT programming system

The Pascal-XT programming system is called with the EXEC command (see [7]). The following assumes that the programming system is available under the system administrator ID $TSOS. In this case the call is:

```
/EXEC $PASCAL-XT
```

If the programming system is available under a different user ID, the call is:

```
/EXEC $USERID.PASCAL-XT
```

Statement input to the programming system is interpreted by the command processor SDF [2] which offers various levels of user prompting (see next section). Depending on the level selected, the programming system issues a prompt (`//` or `STMT`) or displays a menu. The user may then input any of the statements below. A detailed description of their syntax and meaning is given in section 2.6.

ADD-TOOL
: Load Pascal program into main memory. It may be executed by the programming system as often as desired.

CALL-STATEMENT-FILE
: Execute statements to the programming system which are stored in a file.

COMPILE-UNIT
: Call Pascal-XT compiler.

DEFINE-PROJECT-FILE
: Define project directory as required by the compiler in order to access specifications.

EDIT-UNIT
: Call EDT editor (see [13]).

END
: Terminate programming system.

MODIFY-COMPILE
: Temporarily modify operand default values in the compile statement.

MODIFY-EDIT
: Temporarily modify operand default values in the edit statement.

MODIFY-SDF-OPTIONS
: Modify SDF options (see [2]).

REMOVE-DIRECTORY-ENTRY
: Delete entry from current project directory.

RUN-PROGRAM
: Execute tool or program with or without Pascal-XT debugger.

SHOW-ATTRIBUTES
: Output information on programming system and current project directory.

SHOW-SDF-OPTIONS
: Output SDF options (see [2]).

STEP
: Define restart point following errors occurring in batch jobs or DO procedures.

SYSTEM-COMMAND
: Switch to BS2000 command mode, or execute BS2000 system command.

2.2 User guidance

For the analysis of statements input to the programming system, the Pascal-XT programming system uses the command processor SDF (System Dialog Facility). This provides several levels of user guidance. In *expert mode* statements may be entered in the shortest form; in the event of an error, a statement must be repeated completely. *Unguided dialog*, the next level, presents the user with any incorrect operands that have been entered so that they can be corrected; it accepts the correct values. Finally, *guided dialog* presents the user with possible statements in the form of guided dialog menus, in which only the operand values have to be entered. In guided dialog, in turn, one of three levels (minimum, medium, maximum) may be selected. When minimum is specified, only the operands are displayed. With medium the operands are displayed with the possible values, and with maximum, help texts are also displayed.

In this manual only the major attributes of expert mode are described. Additional information may be found under [2].

Input prompt

In expert mode, the input prompt of the Pascal-XT programming system consists of two slashes //. An input prompt at the BS2000 command level takes the form of a single slash /.

Help levels

If you are unsure which statements, operands or operand values are permissible, help is available:

– Which statements are permissible in the Pascal-XT programming system?

  ```
  //?
  ```

  If a question mark is entered following the input prompt, all programming system statements available are listed.

– Which operands can be specified for a statement?

  ```
  //statement?
  ```

  If a question mark is given immediately following the name of a statement, the operands of that statement are displayed in a menu. The user is then to enter only the desired operand values.

– Which operand values can be specified for an operand?

```
//statement operand=?
```

If a question mark is entered instead of an operand value, the permissible operand values for the operand specified are displayed.

Input in block mode

Several statements can be entered in a block if they are separated by a "logical end of line" (LZE key). If an error occurs in the analysis or execution of a statement, the remaining statements of the block will not be carried out in this case.

Function keys

K1 — In guided and unguided dialog the current menu is aborted and control shifts to the next higher menu level.

K2 — The statement entered is interrupted and processing switches to BS2000 command mode. Following input of /RESUME the statement is continued; following input of /INTR it is aborted.
In a current program, inputting K2/INTR generates a Break_Error (see chapter 10), which leads to program abortion if it is not handled properly.

2.3 Extension of the programming system

The Pascal-XT programming system provides only a very small number of statements for the software development process. However, it is designed as an open-ended programming system and permits the user to extend it by adding individual statements he frequently requires. These individual statements are implemented by programs, designated in the following as tools. Before being used, they must be loaded using the statement ADD-TOOL (see section 2.6.2), and be introduced to the programming system. In contrast to the predefined statements of the programming system, these tools cannot be called by specifying their names; they are started by means of the RUN statement (see section 2.6.11).

Requirements on tools

Tools are Pascal programs or programs that have been generated by other language compilers and which are called by a Pascal frame program. Programs that do not have at least a "Pascal frame" cannot be included into the programming system.

Programs that are already present as object modules (phases) cannot be used as tools.

Adding tools to the programming system

Tools are loaded into main memory using the ADD-TOOL statement, and cannot be unloaded until the programming system is left. External references are resolved by the autolink mechanism of the Dynamic Linking Loader.

A tool is automatically known by the name specified as the element name in the TOOL operand of the ADD-TOOL statement. However, it is also possible to refer to the tool using a shorter or mnemonic name, a so-called *alias name*. This alias name may be specified in the ADD-TOOL statement. The tool is then known only by this alias name.

Note

Tools may be loaded from PLAM libraries with effect from BS2000 version 8.

Calling a tool

A tool is executed by means of the RUN-PROGRAM statement (see section 2.6.11), specifying the name of the tool. If the tool has any program parameters, these may be assigned BS2000 files in the PARAMETER operand. The programming system stores these assignments for future calls and makes them the default values unless new assignments are specified in subsequent calls.

Information concerning existing tools

The SHOW-ATTRIBUTES statement (see section 2.6.12) displays a summary of the existing tools. Apart from the names of the tools, the assignments of BS2000 files to the program parameters are shown, to the extent specified in earlier calls.

Use of tools

Tools remain loaded in memory until the programming system is left. This setup makes for fast calling as the tools need not be loaded when called. On the other hand, they permanently occupy storage, which may result in storage bottlenecks when large programs are compiled or executed. It is therefore recommended that only smaller programs should be used as tools.

The first example in section 2.7 (*Developing a main program*) shows how a program which writes a file line by line to standard output can be loaded as a tool in the programming environment using ADD-TOOL (see note (20)).

2.4 Termination behavior of the programming system

The termination behavior of the programming system corresponds to the program termination behavior conventions for user programs in BS2000. The termination behavior is intended to minimize ill effects (destruction of data) resulting from faulty programs.

Normal program termination

The programming system terminated normally. All statements of the programming system have been executed successfully.

Abnormal program termination

An error occurred when a statement of the programming system was called, or an error occurred in the programming system or compiler. The operating system issues the message `ABNORMAL PROGRAM TERMINATION`. In batch mode, or in a DO procedure, the spin-off mechanism is subsequently activated and processing branches to the next job step.

In job variables for program monitoring(see [8]), the termination behavior is logged or explained in more detail. Such a job variable can be specified when the programming system is called (see example at the end of this section).

Job variables for program monitoring consist of a 3-byte status indicator and a 4-byte return code indicator. The first byte in the return code gives the termination code, the remaining 3 bytes provide additional program information.

The termination code and the status indicator are not set until a program or programming system is terminated.

Status indicator

The status indicator is set by the operating system. When a program is started, it is set to the value '$R '. In the case of normal program termination it is set to the value '$T ', with abnormal termination it is set to '$A '.

Termination code

In the programming system, each statement returns a termination code (see section 6.4). The termination code of the RUN statement is the same as the termination code of the executed program. The highest termination code occurring determines the termination behavior of the programming system. The termination code can assume one of the following values:

’0’ = All statements of the programming system were executed without error.

’1’ = All statements of the programming system were executed without error. However, warnings were issued by the compiler during compilation.

’2’ = One (or more) statement(s) of the programming system was (were) either syntactically incorrect or resulted in an error during execution. If an error occurs during compilation, execution of the COMPILE statement is considered erroneous.

’3’ = An error occurred in the programming system. The error should be reported to maintenance.

The termination codes ’0’ and ’1’ result in normal termination of the programming system, codes ’2’ and ’3’ result in abnormal termination.

Program information

The program information provides detailed information on the termination code. Normally, it contains 3 blanks. In the programming system, only the compiler sets the program information. If multiple compilations are performed, the highest program information is returned when the programming system is left.

The compiler provides the following program information, in accordance with the BS2000 convention:

’000’: The compilation proceeded without error.

’002’: The compiler issued warnings.

’004’: The compiler issued error messages.

'005': A user error has occurred. Possible errors are:

- Project directory not defined.
- A specification has been compiled from the EDT.
- Memory overflow (compilation unit too large).
- Interruption with INTR.
- Error in opening the specified files.
- There was no entry for the package or program name in the project directory.

'006': A compiler error has occurred.

Program information greater than or equal to '004' results in abnormal termination when the programming system is left.

Example

The example shows a DO procedure for compiling and statically linking a program. Following a compiler error the program should not be linked. The job variable MONITOR is defined for program monitoring.

```
/PROC A
/DCLJV MONITOR
/SYSFILE SYSDTA=(SYSCMD)
/EXEC $USERID.PASCAL-XT, MONJV=MONITOR
//COMPILE (PLAM.EXAMPLE,ERROR.PROG), (*STD,*STD), *STD
//END
/EXEC $TSOSLNK
PROGRAM TEST
INCLUDE EXAMPL1, PLAM.EXAMPLE
RESOLVE        , PLAM.EXAMPLE
END
/STEP
/GETJV MONITOR
/ENDP
```

Executing this procedure:

```
/DCLJV MONITOR
/SYSFILE SYSDTA=(SYSCMD)
/EXEC $USERID.PASCAL-XT, MONJV=MONITOR
%  BLS0500 PROGRAM 'PASCALXT', VERSION '22A00' OF ...  LOADED<
%  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG.1990. ALL RIGHT
S RESERVED
//COMPILE (PLAM.EXAMPLE,FEHLER.PROG), (*STD,*STD), *STD
   >>> 4 COMPILATION ERRORS DETECTED (WARNINGS: 1;  NOTES: 3)
%  CMD0230 ERROR IN PRECEDING STATEMENT: ALL STATEMENTS WILL BE IGNORED UNTIL '//STEP' IS RECOGNIZED
//END
   END OF THE PASCAL SESSION  -  USED TIME = 0.298 SECONDS
%  EXC0732 ABNORMAL PROGRAM TERMINATION. ERROR CODE 'NRT0101' /HELP-MSG NRT0101
%  CMD0206 TERMJ: COMMANDS WILL BE IGNORED UNTIL /STEP OR /LOGOFF OR /ABEND IS RECOGNIZED
/STEP
/GETJV MONITOR
%$A 2004
```

2.5 Modes of operation of the programming system

Statements and data intended for the operating system are read from system file SYSDTA. Output is directed to system file SYSOUT.

The programming system can be used in both interactive and batch modes. With the exception of the spin-off mechanism, behavior is the same in both modes. The spin-off mechanism is activated if a statement is syntactically incorrect or if an error occurs in the execution of a statement. In that case, all statements up to the next STEP or END statement are skipped.

Interactive mode

The spin-off mechanism has no effect if statements of the programming system are read from the terminal. If SYSDTA is assigned to a file by means of the SYSFILE command, the spin-off mechanism is activated if an error occurs. In a DO procedure, SYSDTA is assigned to a file and an illegal programming system statement therefore results in activation of the spin-off mechanism. An illegal statement in a statement file (see CALL-STATEMENT-FILE) likewise activates the spin-off mechanism.

Batch mode

Illegal statements cause the spin-off mechanism to be activated.

2.6 Statements to the programming system

2.6.1 Definitions and notes

The following metasymbols and notational conventions are used to describe the statements submitted to the programming system:

`UPPERCASE LETTERS`
: Uppercase letters designate constants and keywords which must be entered in the form given (but not in uppercase). They may be abbreviated from right to left so long as the result is unambiguous.

ABBREVIATION
: Uppercase letters in bold type designate permissible unique abbreviations of constants and keywords.

`value`
: Lowercase letters designate metavariables that are replaced by actual values when input (see "metavariables" below).

`( )`
: Parentheses belong to the operand and must be specified with it.

NO (underscored)
: An underscored value denotes a default value which is automatically used when no value is specified.

{NO | YES} (NO and YES stacked inside braces)
: Braces enclose alternative values. If none of the values is designated as the default value, one of the values must be specified.

`NO | YES`
: A vertical bar between operand values likewise denotes alternative values. If none of the values is designated as the default value, one of the values must be specified.

`[ ]`
: Square brackets enclose optional specifications which may be omitted. The square brackets themselves should not be entered.

Keywords

The values of operands in a statement may be keywords (specified in uppercase in the description). A keyword may be prefixed by an asterisk (*). If an asterisk is given in the syntax description, its specification is *mandatory*.

Positional and keyword operands

Each operand may be entered either as a keyword operand or as a positional operand.

A keyword operand consists of the operand name, an equals sign and the operand value.

```
//statement operand=value
```

A positional operand consists of a number of commas indicating the position of the operand within the syntax description of the statement, and the operand value. If, for example, a value is to be entered for the third operand in a statement, commas must be entered for the two unspecified operands preceding it.

```
//statement ,,value
```

The sequence of keyword operands is optional. Once a keyword operand has been specified, no more positional operands may be specified after it for that statement.

Abbreviation rules

The names of statements, operands and keywords may be abbreviated from right to left, so long as the result is unambiguous.

Abbreviation is permitted

- within a name
- within any partial name separated by a hyphen.

Unique abbreviations for constants and keywords are highlighted in the description in bold type.

Alias names

An alias name for a statement or an operand is a second name under which the statement or operand can also be entered. Alias names cannot be abbreviated. They are used in order to permit specification of often-used statements or operands by means of a single letter or a short string.

An alias name for a statement or an operand is specified following the name of the statement or operand, separated by a vertical bar.

Metavariables

A few metavariables appear repeatedly in the syntax description. To avoid repetition, their definitions are given here.

pascal-name	Designates a package name or program name. Underscores in the identifier must be specified as hyphens.
name	Designates a file name, or a program name or package name.
filename	Designates a file name in accordance with BS2000 conventions [7].
element	Designates a PLAM library element. Only element names with the same syntax as in LMS are accepted.
vers	Designates the version of a PLAM element. The identifier may be 1 to 24 characters in length; it has the same syntax as in LMS.
type	Designates the type of a PLAM element. The identifier may be 1 to 8 characters in length; it has the same syntax as in LMS.
tool-name	Designates the name of a tool. It has the same syntax as a file name.

Description of library elements

In some operands, values can be specified as PLAM elements consisting of library name, element name, type and version. These specifications are combined to form a structure that starts with the keyword *LIBRARY. In the syntax definitions of these operands this keyword is omitted for the sake of clarity as its input is not required by SDF. In the menus, however, the keyword is output by SDF.

Example

This example shows equivalent specifications of the SOURCE operand in the COMPILE statement.

```
/EXEC $PASCAL-XT
%  BLS0500 PROGRAM 'PASCALXT', VERSION '22A00' OF ...  LOADED<
%  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG.1990. ALL RIGHT
S RESERVED
//C *LIBRARY(SOURCE-LIBRARY=PLAM.TOOL,SOURCE-ELEMENT=PAS.LMSCALL), *DUMMY

   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//C *LIBRARY(PLAM.TOOL, PAS.LMSCALL), *DUMMY
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//C (PLAM.TOOL, PAS.LMSCALL), *DUMMY
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//C SOURCE-LIBRARY=PLAM.TOOL,SOURCE-ELEMENT=PAS.LMSCALL, L=*DUMMY  ———— (01)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//END
   END OF THE PASCAL SESSION  -  USED TIME = 1.586 SECONDS
/
```

(01) The SOURCE operand consists of a structure that contains the two SOURCE-LIBRARY and SOURCE-ELEMENT operands. As these two operand names are unique throughout the COMPILE statement, SDF can assign them uniquely to the SOURCE operand. Specification of structural parentheses may thus be omitted. Input of keyword L (LISTING) is mandatory because no positional parameters must be specified following input of a keyword parameter.

Operand value *UNCHANGED

In guided dialog (menu), the operand value *UNCHANGED is also displayed for the COMPILE, MODIFY-COMPILE, EDIT and MODIFY-EDIT statements; this value is not included in the syntax descriptions in the sections concerned. This value means that the current default value is assumed unchanged if an operand is omitted in the input. The value may, however, be entered by the user, with the effect that the set default value is not changed.
If a default value does not yet exist for an operand, an error message will be issued. This may happen, for example, if only the element name has been specified for the compilation from a library and this library name had not been defined beforehand with MODIFY-COMPILE. The operand value for the library name is *UNCHANGED and thus represents an undefined value.

Example

```
//C (,TEST.PROG),*DUMMY
```

means:

```
//C (*UNCHANGED, TEST.PROG(*HIGHEST-EXISTING)), *DUMMY
```

and results in an error message if no library had been defined beforehand by means of MODIFY-COMPILE, e.g.

```
//MODIFY-COMPILE (PLAM.TEST,)
//C (,TEST.PROG),*DUMMY
```

means:

```
//C (PLAM.TEST, TEST.PROG(*HIGHEST-EXISTING)), *DUMMY
```

2.6.2 ADD-TOOL

Load program into main memory.

```
ADD-TOOL

    TOOL          =  (LIBRARY = filename | *OMF
                     ,ELEMENT = element)

  [,ALIAS-NAME    =  *ELEMENT-NAME | tool-name ]
```

Function

ADD-TOOL loads the program specified in the TOOL operand into main memory. It may then be executed as often as desired using the RUN statement (see 2.3).

The spin-off mechanism is activated if a tool cannot be loaded or a tool having the specified name already exists.

TOOL = (filename,element)
"element" designates the starter module of the program which is loaded from the object module library "filename" or the temporary EAM object module file (*OMF).

ALIAS-NAME
Defines a new name which may be used to refer to the tool. The name must be different from the names of tools that have already been loaded.

= *ELEMENT-NAME
The tool may be called only by means of the element name "element" specified in the TOOL operand.

= tool-name
An optional name for the tool.

Note

Loading of tools from a PLAM library is possible with effect from BS2000 version 8.

2.6.3 CALL-STATEMENT-FILE

Execute statements from a (statement) file.

```
CALL-STATEMENT-FILE

                   ⎧filename                                                 ⎫
                   ⎪*EDT (WORKFILE = *STD | 0..9)                            ⎪
    STMT-FILE  =   ⎨                                                         ⎬
                   ⎪(LIBRARY = filename,                                     ⎪
                   ⎪,ELEMENT = element(VERSION = *HIGHEST-EXISTING | vers    ⎪
                   ⎩                  ,TYP = J | typ ))                      ⎭

   [,PROTOCOL   =  NO | YES ]
```

In DO procedures and batch jobs, *STD is still accepted as a version specification.

Function

CALL-STATEMENT-FILE carries out several statements which are stored in a file. This statement file may be a SAM or ISAM file or a library element. The CALL statement opens the file, sequentially reads the statements from the file and executes them immediately.

The statement file need *not* contain any special opening or terminating statements. Each statement must appear in a separate line.

CALL statements may be nested to any depth. After leaving the statement file, processing continues with the statement immediately following the last CALL. The number of such nested calls is limited only by available main memory.

If the statement file contains an END statement, the programming system terminates immediately after it has been executed.

An error in the analysis or execution of a statement activates the spin-off mechanism; all subsequent statements in the statement file are skipped until a STEP or END statement is recognized.

STMT-FILE Designates the file or library element from which the statements are read.

= filename

Name of the statement file.

= *EDT(WORKFILE = *STD | 0..9)
: The statements are read from the work area (WORKFILE) of the EDT editor. The work areas 0 through 9 can be specified. By default, reading is performed from the current work area (*STD).

= (filename,element(version,type))
: The statements are read from the library element "element" with version "version" of type "type" of the PLAM library "filename".

The "version" und "type" specifications are optional.
Default value for "vers" = *HIGHEST-EXISTING
for "type " = "J".

The version specification *HIGHEST-EXISTING causes the statements from the highest existing version of the specified library element to be read.

As of Pascal-XT V2.2A, the version specification *STD is no longer permissible in dialog. *STD is still accepted in DO procedures and batch jobs.

The version specification *INCREMENT is not permissible.

PROTOCOL
: The statements read from the file may simultaneously be logged to the system file SYSOUT. On output, each statement is given the prefix "(%STMT)". Invalid statements are always output to SYSOUT.

= NO
: Logging is suppressed.

= YES
: Statements are logged.

2.6.4 COMPILE-UNIT

Call Pascal-XT compiler.

```
COMPILE-UNIT | C

                    ⎧*EDT (WORKFILE = *STD | 0..9)                              ⎫
                    ⎪name (KIND = FILE | SPEC | BODY | PROG)                    ⎪
  [ SOURCE     =    ⎨                                                           ⎬]
                    ⎪(SOURCE-LIBRARY = filename                                 ⎪
                    ⎪                                      ⎧*HIGHEST-EXISTING⎫  ⎪
                    ⎩,SOURCE-ELEMENT = element(VERSION =   ⎨                 ⎬))⎭
                                                           ⎩vers             ⎭

                    ⎧*SYSOUT                                 ⎫
                    ⎪*SYSLST                                 ⎪
                    ⎪*EDT (WORKFILE = *STD | 0..9)           ⎪
  [,LISTING    =    ⎨                                        ⎬]
                    ⎪*DUMMY                                  ⎪
                    ⎪(LIST-LIBRARY = *STD | filename         ⎪
                    ⎩,LIST-ELEMENT = *STD | element)         ⎭

  [,MODULE-LIBRARY =  *OMF (ERASE = YES | NO) | filename | *STD ]
  [,ASSEMBLER      =  *BY-SOURCE | ON | OFF ]
  [,CHECK          =  *BY-SOURCE | ON | OFF ]
  [,DEBUG          =  *BY-SOURCE | ON | OFF | RESTRICTED ]
  [,GENERATE       =  *BY-SOURCE | ON | OFF ]
  [,INITIALIZE     =  *BY-SOURCE | ON | OFF ]
  [,LIST | L       =  *BY-SOURCE | ON | OFF ]
  [,MAP            =  *BY-SOURCE | ON | OFF ]
  [,OPTIMIZE       =  *BY-SOURCE | ON | OFF ]
  [,STANDARD       =  *BY-SOURCE | ON | L0 | OFF ]
  [,XREF           =  *BY-SOURCE | ON | OFF ]
  [,LINES-PER-PAGE =  *STD | 11..2147483639 ]
  [,MESSAGE-LEVEL  =  NOTES | WARNINGS | ERRORS ]
```

*STD is still accepted as a version specification for the SOURCE operand in DO procedures and batch jobs.

Function

COMPILE-UNIT calls the Pascal-XT compiler. The source specified in the SOURCE operand is compiled. All listings generated by the compiler are output to the file specified in the LISTING operand. Following successful compilation of a main program or of a package body, the object modules are generated and stored in the object module library specified in the MODULE-LIBRARY operand. The compiler session can be controlled by means of compiler options.

Instead of the file name, it is also possible to specify the Pascal identifier of a package or main program in the SOURCE operand. The distinction between file name and Pascal identifier is controlled by the KIND operand. However, this method assumes that a project directory has been defined and that the program, or the package, has been entered in it. For a compilation unit (specification, body or main program) to be entered in the project directory, it must first have been compiled successfully from a file (other than EDT).

For the compilation of a package or a main program that imports packages, the compiler requires a project directory defined with the DEFINE statement. Through this project directory, the compiler finds the specifications of imported packages required for interface checking. Following successful compilation, the compilation unit is marked as valid in the project directory. This information is necessary for recompilations to be identified correctly. The project directory functions are discussed at length in chapter 3.

The compiler can also read in source programs from an EDT work area. In this case the project directory will not be updated because, in general, the source will not be available later on. Following successful compilation of a package specification from an EDT work area, an additional message is issued saying that no entry has been made in the project directory.

The default values of operands can be changed in a session using the MODIFY-COMPILE statement. These changes apply until the next change, or until program termination.

The spin-off mechanism is activated after the following errors:

(a) Errors in the syntax of the COMPILE statement

(b) Errors have been found during compilation

(c) A package specification has been compiled from an EDT work area

(d) Compilation required the project directory, but it has not been created

(e) The SOURCE operand contained the name of a package or program for which no entry exists in the project directory

(f) The same file has been specified in SOURCE and LISTING

If a job variable is specified when calling the programming system, the program information of the termination code (see section 2.4) describes the severest error the spin-off mechanism has activated.

SOURCE Specifies the name of the file that contains the source program, or the name of a package or main program that is to be compiled.

= *EDT (WORKFILE = *STD | 0..9)

The source program resides in a work area (0..9) of the EDT. By default, the current work area (*STD) is addressed.

= filename

Name of the source file.

= name (KIND = FILE | SPEC | BODY | PROG)

"name" designates a file, a package or a main program. The KIND operand defines how "name" is to be interpreted:

= FILE "name" is the name of a file
= SPEC "name" is the Pascal identifier of a package whose specification is to be compiled
= BODY "name" is the Pascal identifier of a package whose body is to be compiled
= PROG "name" is the Pascal identifier of a main program that is to be compiled

Specification of SPEC, BODY, or PROG assumes that a project directory has been defined and that there is an entry for the package specification, package body or the main program, otherwise an error message will be issued. The Pascal source is read from the file in which the project directory is specified. Underscores in the Pascal identifier must be specified in "name" as hyphens ("-").

= (filename, element(vers))

The source program is read from the library element "element" with the version "vers" of the PLAM library "filename". The library element *must* be of type "S" (for source). By default, the element with the highest-existing version (*HIGHEST-EXISTING) is read.

As of Pascal-XT V2.2A, the version specification *STD is no longer permissible in dialog. *STD is still accepted in DO procedures and batch jobs.

LISTING

Specifies the name of the file to which the compiler outputs the generated listings. The number of lines per page can be controlled with the LINES-PER-PAGE option. The compilation is aborted with a message if the SOURCE and LISTING operands designate the same file.

= *SYSOUT

Output to system file SYSOUT

= *SYSLST

Output to system file SYSLST

= *EDT (WORKFILE = *STD | 0..9)

Output to the specified work area of the EDT. By default (*STD), this is the current area. The work area is cleared beforehand. If the SOURCE operand also specifies an EDT work area, the two work areas must be different.

= *DUMMY

Output to system file *DUMMY. Thus all compiler listings are lost.

= filename

Name of the output file (SAM file). An existing file will be overwritten.

= (filename, element)

The compiler writes the generated listing to the element "element" of PLAM library "filename". This list element is of type "P" (for print file), and this cannot be changed by the user.
Any existing list element of the same name, type and version is overwritten.

If a library element was specified in the SOURCE operand, the following applies to the list element:

- If the value *STD is specified for LIST-LIBRARY, the list element is written to the same library as was specified for SOURCE-LIBRARY.
- If the value *STD is specified for LIST-ELEMENT, the list element (of type "P") is given the same element name as the source element (of type "S").

The version given to the list element is governed by whether or not the source program itself is a library element.

- If it is, the list element is given the same version as the source element. If the source element version has the default value (as of Pascal-XT V2.2A = *HIGHEST-EXISTING; previously = *STD), the list element no longer receives the highest-possible version, but instead is given the same version as the source element.
- If not, the list element receives the version *UPPER-LIMIT (= highest-possible version).

MODULE-LIBRARY

Specifies the module library in which, following successful compilation of a program or package body, the generated object modules will be stored. The CSECT names of the generated modules are generated from the name of the compilation unit (see section 4.4).

= *OMF (ERASE = YES | NO)

The generated object modules are stored in the temporary object module file (*OMF) of the user task. The ERASE operand is used to specify whether *OMF is to be erased before compilation:
ERASE = YES *OMF is erased beforehand
ERASE = NO *OMF is not erased beforehand (default value)

= *STD

This entry is only permissible if a library element has been specified in the SOURCE operand. The generated object modules are stored in the library containing the source element.
Each object module generated is stored as an individual library element of type "R" (for relocatable). The elements are given the same names as the object modules (CSECT names). The elements are given the same version number as the source elements. Existing elements of the same name, type and version are overwritten.

= filename

Is the name of a PLAM library. For the rest, the same conventions apply as for entry *STD.

ASSEMBLER, CHECK, DEBUG, GENERATE, INITIALIZE, LIST, MAP, OPTIMIZE, STANDARD, XREF

Compiler options for controlling compilation. These can also be specified in the Pascal-XT source program. Their meaning is given in section 4.5.

= *BY-SOURCE

stands for the option value specified in the source program or, if the relevant option is not specified there, the default value.

= ON
The compiler option is activated.

With the option STANDARD, ON causes the compiler to accept only those language constructs which conform to the Pascal standard ISO 7185 Level 1.

= OFF
The compiler option is deactivated.

With the option STANDARD, OFF causes the compiler to accept any language constructs within the scope of Pascal-XT V2.1.

= L0
With the option STANDARD, L0 causes the compiler to accept only those language constructs which conform to the Pascal standard ISO 7185 Level 0.

LINES-PER-PAGE
Compiler option which can only be specified as an operand of the COMPILE-UNIT statement. It defines the number of lines per page in the compiler listing. The number of lines must lie within the specified range of 11.. 2147483639. The default value (*STD) is 63.

MESSAGE-LEVEL
Compiler option which can only be specified as an operand of the COMPILE-UNIT statement. It defines which messages are to be issued in the compiler listing. There are three types of message: notes, warnings and errors (see 4.6.3).

= NOTES
Default value. Messages are issued in the compiler listing as errors (syntax or semantic errors), warnings (potential runtime errors) or notes.

= WARNINGS
Messages are issued as errors and warnings.

= ERRORS
Messages are issued as errors only.

Note

In the program development cycle EDIT-COMPILE-RUN for a main program it must be ensured that the temporary EAM object module (*OMF) is deleted before every compilation so that RUN causes the last program compiled to be executed.

Examples

Call using default parameters or the parameters set by means of the MODIFY-COMPILE statement:

```
//C
```

Specification of a file name:

`//C BEISPIEL.PROG`	File name with default setting of KIND
`//C BEISPIEL.PROG(F)`	File name with changed setting of KIND=FILE

Specification of a package name or program name:

`//C A(S)`	Specification of package A (*S*pecification)
`//C A(B)`	Body of package A (*B*ody)
`//C MAIN(P)`	Main program MAIN (*P*rogram)

Specification of a library element:

`//C (PLAMLIB, ELEM)`	Element ELEM from library PLAMLIB
`//MC (PLAMLIB,)`	Preset the library PLAMLIB
`//C (,ELEM)`	Element ELEM from the preset library PLAMLIB
`//C S-E=ELEM`	Corresponds to previous line
`//C (BSPLIB, BSP(2.0))`	Version 2.0 of element BSP from library BSPLIB

Specification of an EDT work area:

`//C *E`	Current work area
`//C *E(5)`	Work area 5

Various options for the output of compiler listings (for SOURCE, the default value or another presetting applies):

`//C ,*D`	Listing to *DUMMY (listing is immaterial)
`//C ,*SYSOUT`	Listing to screen
`//C ,*SYSLST`	Listing to printer
`//C ,*E`	Listing to current EDT work area
`//C ,*E(7)`	Listing to EDT work area 7
`//C ,LST.BSP`	Listing to file LST.BSP
`//C ,(LIB, LST)`	Listing to element LST of library LIB
`//MC (SOURCES,A)`	Define library for SOURCE operand
`//C ,(*S,LST.A)`	Listing to element LST.A of library SOURCES
`//C ,(*S,*S)`	Listing to element A (type P) in library SOURCES
`//C ,(LSTLIB,*S)`	Listing to element A (type P) in library LSTLIB

The various options for specifying the object module library (for SOURCE and LISTING the presettings are assumed):

`//C ,,(Y)`	Object modules to *OMF which is deleted beforehand
`//C ,,MODLIB`	Object modules to PLAM library MODLIB
`//C ,,*S`	Object modules (type R) to source library. This requires a library to have been specified in the SOURCE operand.

2.6.5 DEFINE-PROJECT-FILE

Define project directory.

```
DEFINE-PROJECT-FILE

     DIRECTORY  =  filename
```

Function

DEFINE-PROJECT-FILE defines a project directory which the Pascal-XT compiler requires for accessing package specifications. The definition remains in effect until the programming system is left, or until a new project directory is defined using the DEFINE statement. If no project directory with the name given exists yet, a new project directory is generated.

The defined project directory remains open throughout the entire session. It can be used by several users simultaneously.

If the project directory is swapped in the course of a session, the programming system outputs a list of the current assignments involving packages and EDT work areas (see EDIT-UNIT) and asks whether the directory is really to be swapped. If "N" or "n" is entered in response the command is terminated with a message. If "Y" or "y" is entered the directory is swapped and the assignments stored in the programming system are deleted, but the contents of the EDT work areas remain unchanged. If no assignments have been specified the user is not asked if the directory is to be swapped. This procedure ensures that the consistency of the project directory is maintained.

The spin-off mechanism is activated if an invalid project directory is defined.

DIRECTORY = filename
Name of the project directory.

2.6.6 EDIT-UNIT

Call the editor EDT.

```
EDIT-UNIT | E

                 ⎧*EDT                                                     ⎫
                 ⎪name(KIND = FILE | SPEC | BODY | PROG)                   ⎪
                 ⎪                                                         ⎪
                 ⎨                                                         ⎬]
[ UNIT     =     ⎪(LIBRARY = filename,                                     ⎪
                 ⎪,ELEMENT = element(VERSION = *HIGHEST-EXISTING | vers    ⎪
                 ⎩                   ,TYP = S | type ))                    ⎭

[,WORKFILE = *STD | 0..9]

[,QUERY    = YES | NO]
```

*STD is still accepted as a version specification for the UNIT operand in DO procedures and batch jobs.

Function

EDT calls the EDT editor. The UNIT operand specifies whether only EDT is to be called or whether, prior to the call, a file is to be loaded in the work area defined by the WORKFILE operand. The QUERY operand is used to specify the security to be undertaken on leaving EDT.

The work area defined when leaving EDT is designated as the current work area (*STD). If in the following paragraphs EDT is called without specification of a work area, a branch will be made to this work area. However, if a work area has been explicitly defined with the MODIFY-EDIT statement, this work area will be branched into on calling EDT.

Instead of a file name a Pascal identifier of a package or main program may be specified in the UNIT operand. The distinction between file name and Pascal identifier is controlled by the KIND operand. This method assumes the presence of a project directory that contains an entry for the identifier.

For each EDT work area, the programming system stores the name of the file whose contents have been loaded. Thus, work areas may be swapped at will in the certainty that, on leaving EDT, the contents of the current work area will be stored in the associated file. This method only works if the EDT commands READ and @READ are *not* used, as otherwise the new contents of the work area will be rewritten.

The QUERY operand is used to specify whether the programming system, on returning from EDT, will automatically rewrite the contents of the EDT work area to the file, or whether this should be queried at the terminal. The work areas of EDT are retained.

Compilation units which have been changed must be recompiled, usually together with other compilation units from the same program. The SHOW-ATTRIBUTES statement (see SHOW-ATTRIBUTES) indicates which compilation units need to be recompiled. However, necessary recompilations can only be displayed correctly if any changes to package specifications, package bodies or main programs are performed exclusively under the control of the programming system. The programming system will only recognize and store modifications in compilation units under the following conditions:

- EDT is called with the UNIT operand as follows:

  ```
  UNIT = name (KIND = SPEC | BODY | PROG)
  ```

- EDT is quitted with "h" and the EDT work area is *not* rewritten to the file with an EDT command.

A specification, body or a main program should be compiled immediately after a modification so that any modifications in the package references (WITH clauses) can be recorded in the project directory.

The default values of operands can be changed during a session with the aid of the MODIFY-EDIT statement. These changes remain effective until the next change or until termination of the programming system.

The output resulting from various error conditions is described below.

The spin-off mechanism is activated if the following errors occur:

(a) Errors in the syntax of the EDIT statement

(b) The required package/program cannot be loaded because no project directory has been defined

(c) The UNIT operand contains the name of a package or program for which no entry exists in the project directory

(d) The file does not exist or cannot be opened

UNIT — This operand specifies whether merely EDT is to be called or whether a file is to be loaded in the work area prior to the call.

= *EDT — A branch is made to the current work area of the EDT. The contents of the work area are not changed.

= name (KIND = FILE | SPEC | BODY | PROG)

"name" is the name of a file, package or main program that is loaded into the defined EDT work area. This area is cleared prior to loading.
The KIND operand specifies how "name" is to be interpreted:
= FILE as the name of a BS2000 file
= SPEC as the Pascal identifier of a package specification
= BODY as the Pascal identifier of a package body
= PROG as the Pascal identifier of a main program.
The SPEC, BODY or PROG specification assumes that a project directory has been defined and that there is an entry for the specified compilation unit. Otherwise an error message will be issued (see below). The Pascal source is read from the file specified in the project directory. The name of this file is superimposed on the EDT command line. Underscores in the Pascal identifier must be specified as hyphens ("-") in "name".

= (filename, element (vers, type))

The current EDT work area is cleared and library element "element" with specified version "vers" and type "type" is loaded from PLAM library "filename" into the EDT work area. Upon quitting EDT, the version of the library element which was being read is always overwritten.

The "vers" and "type" specifications are optional.
Default value for "vers" = *HIGHEST-EXISTING,
for "type" = "S".

The effect of version specification *HIGHEST-EXISTING is as follows:

- If the specified library element exists, the highest version of the library element is read and, upon quitting EDT, overwritten. Hence, as of Pascal-XT V2.2A, a new version is no longer automatically created when rewriting, as was previously the case with version specification *STD.
- If the specified library element does not exist, an element is created with version "001".

The version specification *STD is no longer permissible in dialog. *STD can still be used in DO procedures and batch jobs, but its meaning has changed:

- If the specified library element exists, the highest version of the library element is read and, upon quitting EDT, overwritten.
- If the specified library element does not exist, an element with version *UPPER-LIMIT (highest-possible version) is created.

The version specification *INCREMENT is not permissible.

WORKFILE — This operand is used to define the EDT work area.

= *STD — Designates the current work area.

= 0..9 — Number of the EDT work area.

QUERY — On leaving EDT, a query may follow to decide whether the content of the work area has to be rewritten to the file or to the library element from which it had been loaded. By default (YES), the query follows. The query and the rewriting do not follow if
- the EDT work area is empty, or
- no work area has been allocated to a file in the programming system (e.g., after a work area switch in in EDT).

= YES — A query will follow. On entering "Y" or "y" rewriting is peformed; on entering "N" or "n" no writing is performed

= NO — No query will follow and the contents of the work file are rewritten to the file.

Error messages

If any errors should occur when a file is loaded into EDT, either the programming system issues a message (with the prefix ">>>") or a message with an error code is issued in the EDT message line (immediately above the command line), depending on the type of error. The error codes, err-no, are explained in section 10.4.

(a) File does not exist. A branch is made to the EDT and

```
FILE "name" DOES NOT EXIST
```

is displayed on the message line. If the EDT work area is not empty on leaving EDT, the file "name" is generated if QUERY=NO and the contents of the work area are stored in it. If QUERY=YES, the system queries whether the file has to be generated.

(b) File exists but cannot be opened in read mode. Message

```
>>> OPEN ERROR ON SPECIFIED FILE (err-no)
```

is issued and no branch is made to EDT.

(c) The specified library element does not exist. Control branches to EDT where it issues this message:

```
ELEMENT "element (version,type)" DOES NOT EXIST
```

On leaving EDT, processing continues as described under (a).

(d) The specified library cannot be opened. Control branches to EDT where it issues this message:

```
OPEN ERROR ON LIBRARY "filename"
```

On leaving EDT, processing continues as described under (a) if the library did not exist; processing is aborted if the library is locked.

(e) The name has not been entered in the project directory. Message

```
>>> UNIT NOT FOUND IN PROJECT DIRECTORY
```

is issued and control is not passed to EDT.

(f) The file specified in the project directory cannot be opened. Message

```
>>> OPEN ERROR ON FILE FOUND IN PROJECT DIRECTORY (err-no)
```

is issued and control is not passed to EDT.

Examples

Call which uses parameters preset by default or specified by means of the MODIFY-EDIT statement:

```
//E
```

Editing an EDT work area:

`//E *E`	Current work area
`//E *E,5`	Work area 5
`//E ,5`	Work area 5

Editing a file:

`//E A.PROG`	File A.PROG
`//E A.PROG(F)`	File A.PROG with option KIND=FILE

Specification of a package name or program name:

`//E BSP(S)`	Specification of package BSP (*S*pecification)
`//E BSP(B)`	Body of package BSP (*B*ody)
`//E BEISPIEL(P)`	Main program BEISPIEL (*P*rogram)

Editing a library element:

`//E (TOOLLIB, LMS.PROG)`	Library TOOLLIB and element LMS.PROG
`//ME (TOOLLIB, )`	Define library TOOLLIB
`//E (,LMS.PROG)`	LMS.PROG from the defined library TOOLLIB
`//E (LIB, TOOLS.S (V1.0))`	Version 1.0 of element TOOL.S from library LIB
`//E (TOOLLIB,LMS (,R))`	Element LMS having the highest version (*HIGHEST-EXISTING) and type R

Deactivate any query before overwriting:

`//E ,,N`	or
`//ME ,,N`	definition of a new default value

2.6.7 END

Terminate programming system.

```
END
```

Function

END terminates the Pascal-XT programming system. All files still open are closed. A job variable specified for program monitoring when the programming system was called is set.

If an error occurred during the analysis or execution of a statement, the operating system issues the following message on leaving the programming system:

```
ABNORMAL PROGRAM TERMINATION
```

In batch mode, control is subsequently passed to the next job step.

2.6.8 MODIFY-COMPILE

Temporarily change the operand default values of the COMPILE statement.

```
MODIFY-COMPILE | MC

   [ SOURCE      = { *EDT (WORKFILE = *STD | 0..9)                                   }
                   { name (KIND = FILE | SPEC | BODY | PROG)                         }
                   {                                                                 }]
                   { (SOURCE-LIBRARY = filename                                      }
                   {                                          {*HIGHEST-EXISTING}    }
                   { ,SOURCE-ELEMENT = element(VERSION =      {                 }))  }
                   {                                          {vers             }    }

                   { *SYSOUT                                        }
                   { *SYSLST                                        }
                   { *EDT (WORKFILE = *STD | 0..9)                  }
   [,LISTING     = {                                                }]
                   { *DUMMY                                         }
                   { (LIST-LIBRARY = *STD | filename                }
                   { ,LIST-ELEMENT = *STD | element)                }

   [,MODULE-LIBRARY =  *OMF (ERASE = YES | NO) | filename | *STD ]
   [,ASSEMBLER      =  *BY-SOURCE | ON | OFF ]
   [,CHECK          =  *BY-SOURCE | ON | OFF ]
   [,DEBUG          =  *BY-SOURCE | ON | OFF | RESTRICTED ]
   [,GENERATE       =  *BY-SOURCE | ON | OFF ]
   [,INITIALIZE     =  *BY-SOURCE | ON | OFF ]
   [,LIST | L       =  *BY-SOURCE | ON | OFF ]
   [,MAP            =  *BY-SOURCE | ON | OFF ]
   [,OPTIMIZE       =  *BY-SOURCE | ON | OFF ]
   [,STANDARD       =  *BY-SOURCE | ON | L0 | OFF ]
   [,XREF           =  *BY-SOURCE | ON | OFF ]
   [,LINES-PER-PAGE =  *STD | 11..2147483639 ]
   [,MESSAGE-LEVEL  =  NOTES | WARNINGS | ERRORS ]
```

The specification *STD is still accepted as the SOURCE operand in DO procedures and batch jobs.

Function

The MODIFY-COMPILE statement has the same format as the COMPILE statement. The descriptions of the operands may be found under that statement.

The MODIFY-COMPILE statement is used to change the default operand values of the COMPILE statement. This change remains in effect until the next change, or until the programming system is left. The menus of the COMPILE and MODIFY-COMPILE statements always show the current default values.

In the case of library elements, library and element names may be defined independently of one another. When only one of the operands is changed, the other retains the value it had. In the SOURCE-ELEMENT operand the version must be defined together with the element name.

The spin-off mechanism is activated if the input for the MODIFY-COMPILE statement is syntactically incorrect.

Note

The operand value *UNCHANGED, shown in the menu, is described in section 2.6.1.

2.6.9 MODIFY-EDIT

Change operand default values in the EDIT statement.

```
MODIFY-EDIT | ME

                 ⎧*EDT                                                     ⎫
                 ⎪name(KIND = FILE | SPEC | BODY | PROG)                   ⎪
[ UNIT      =    ⎨                                                         ⎬]
                 ⎪(LIBRARY = filename,                                     ⎪
                 ⎪,ELEMENT = element(VERSION = *HIGHEST-EXISTING | vers    ⎪
                 ⎩                   ,TYP = S | typ ))                     ⎭

[,WORKFILE = *STD | 0..9]

[,QUERY    = YES | NO]
```

*STD is still accepted as the UNIT operand in DO procedures and batch jobs.

Function

The MODIFY-EDIT statement has the same format as the EDIT-UNIT statement. The descriptions of the operands may be found under that statement.

This statement permits the operand default values in the EDIT statement to be changed. The change remains in effect until the next change, or until the programming system is left. The menus of the EDIT and MODIFY-EDIT statements always show the current values.

In the case of library elements, library and element names may be defined independently of one another. When only one of the operands is changed, the other retains the value it had. Version and type in the ELEMENT operand can be defined only together with the element name.

Note

The operand value *UNCHANGED, shown in the menu, is described in section 2.6.1.

2.6.10 REMOVE-DIRECTORY-ENTRY

Delete entry in the current project directory.

```
REMOVE-DIRECTORY-ENTRY

     UNIT =  pascal-name
```

Function

REMOVE deletes all entries for the identifier specified in the UNIT operand from the current project directory. The associated source files are not affected by this function.

Removal of an entry makes all compilation units that import the removed package invalid. They are marked accordingly in the project directory.

The spin-off mechanism is activated if

(a) No project directory has been defined

(b) The name of the package or main program does not exist in the project directory

UNIT = pascal-name

Name of the package or main program whose entry is to be deleted.

2.6.11 RUN-PROGRAM

Execute tool or program with or without the interactive Pascal debugging aid.

```
RUN-PROGRAM | R

                        ⎧*LAST-COMPILED-PROG                 ⎫
                        ⎪tool-name                           ⎪
  [ PROGRAM     =       ⎨                                    ⎬]
                        ⎪(LIBRARY = *OMF | filename          ⎪
                        ⎩,ELEMENT = element)                 ⎭

  [,PARAMETER   =   *NONE | string ]

  [,DEBUG       =   NO | YES ]
```

Function

RUN executes a Pascal-XT program or a tool without having to leave the programming system. Following termination of the program the programming system continues operation. Current BS20000 files can be associated with program parameters prior to program execution (see chapter 5). The DEBUG operand is used to specify whether the program is to be tested with PATH, the Pascal debugging aid (see chapter 9).

With effect from Pascal-XT V2.2A, no Pascal-XT programs with interfaces to other languages can be executed with RUN, as this is incompatible with ILCS (see 7.1).

The program started with RUN remains loaded until the next compilation. Thus a program can be executed several times in succession without reloading.

If an exception (runtime error) occurs that is not handled by the program itself, the dynamic call chain (see section 10.2) is output, the program terminates and control is returned to the programming system.

A program is not executed if a tool having the same name as the program has already been loaded. An error message is issued and the spin-off mechanism is activated. If the program to be executed requires a package (or several packages) already loaded and used by a tool that has been loaded itself, execution of that program may give rise to error conditions on account of the multiple use of the package.

The spin-off mechanism is activated if the program cannot be loaded or the program terminated with an error.

PROGRAM Designates the Pascal program to be executed. It is loaded with DLL if this has not been done already. External references are resolved in accordance with the autolink mechanism of DLL (see also section 6.3).

= *LAST-COMPILED-PROG
The last program compiled is loaded from the library specified in the COMPILE statement and executed.

= tool-name
Name of the tool to be executed. The tool must have been loaded previously with the aid of the ADD-TOOL statement.

= (*OMF,element)
The program designated with "element" is loaded from the temporary object module file (*OMF) and executed. "element" is the name of starter module of the program. *OMF must be be deleted with ERASE * prior to each compilation if a program is to be compiled more than once and only the last program compiled is to be executed.

= (filename,element)
From the PLAM library "file name", the program designated with "element" is loaded and executed. "element" is the name of the starter module of the program.

Note

DLL can be loaded from PLAM libraries with effect from BS2000 version 8.

PARAMETER Describes the assignment of BS2000 files to external files (program parameters) of the program.

= *NONE
No physical files are specified. If the program was already executed immediately before this execution with files specified, those files are taken as default values in subsequent calls.

= string
Assignments of BS2000 files to external Pascal files are entered in a string enclosed in single quotes ’...’. Each individual file assignment is specified in the form

```
pascal-file = physical-file
```

Blanks are skipped. Multiple assignments within the string are to be separated by commas:

```
'pas-file1 = file1, pas-file2 = file2, ...'
```

The correct number of parameters and the validity of the Pascal file identifiers cannot be checked. An error is not detected until runtime. The programming system stores these file assignments for as long as the program is loaded. Following a compilation or following execution of another program the assignments are no longer in effect. For permanently loaded tools the assignments remain stored until the programming system is left. If in a subsequent call of the RUN statement other physical files are specified, they replace the previously stored values. If no file assignments are specified, those previously made are assumed, if possible.
As "physical file" a SAM or ISAM file may be specified. For ISAM files the remarks made in section 5.2.2 should be noted.
Physical files must not be assigned to the predefined files INPUT and OUTPUT. These are always assigned to the system files SYSDTA and SYSOUT respectively. Therefore, no assignments are recorded for them.

DEBUG Specifies whether the program is to run under the control of the PATH debugging aid.

= NO execution without the debugging aid

= YES execution with the debugging aid

Note

Programs requiring large amounts of main memory should be statically linked and executed outside of the programming system because any additional memory required by the programming system and compiler is available there.

Examples

`//R`	Execute the program compiled last
`//R LMS`	Execute the tool named LMS (must have been loaded beforehand by means of the ADD-TOOL statement).
`//R (,TEST)`	Execute program TEST from the *OMF (Note: any required packages cannot be loaded dynamically from *OMF by DLL. They must be included in an object module library defined by means of SYSFILE TASKLIB).
`//R (MODLIB, BEISPIE)`	Execute program BEISPIE (name of the starter module) from the MODLIB object module library
`//R d=y`	Execute the program compiled last under control of the PATH debugging aid
`//R ,'SRC=QUELLE,DEST=ZIEL'`	Execute the program compiled last, assigning BS2000 files QUELLE and ZIEL to Pascal files SRC and DEST respectively

2.6.12 SHOW-ATTRIBUTES

Provide information on the programming system and the current project directory.

```
SHOW-ATTRIBUTES | S

                  ⎧LOADED-PROGRAM                                               ⎫
                  ⎪TOOLS  (TOOL = *ALL | tool-name)                             ⎪
                  ⎪                                                             ⎪
                  ⎪PROJECT-FILE                                                 ⎪
                  ⎪      (PACKAGES    = * | pascal-name                         ⎪
                  ⎪       ,KIND       = ALL | SPEC | BODY | PROG                ⎪
   [ WHAT    =    ⎨                                                             ⎬]
                  ⎪       ,REFERENCES = NONE | ALL | DIRECT | INDIRECT          ⎪
                  ⎪       ,USED-BY    = NONE | ALL | DIRECT | INDIRECT)         ⎪
                  ⎪                                                             ⎪
                  ⎪RECOMPILATIONS                                               ⎪
                  ⎪      (PACKAGES    = *NECESSARY | *ALL | pascal-name         ⎪
                  ⎩       ,KIND       = ALL | SPEC | BODY | PROG)               ⎭

                  ⎧*SYSOUT                                                      ⎫
                  ⎪*SYSLST                                                      ⎪
                  ⎪*EDT(WORKFILE = *STD | 0..9)                                 ⎪
                  ⎪filename                                                     ⎪
   [,OUTFILE =    ⎨                                                             ⎬]
                  ⎪(LIBRARY=filename,                                           ⎪
                  ⎪                             ⎧*UPPER-LIMIT⎫                  ⎪
                  ⎪,ELEMENT=element(VERSION=    ⎨            ⎬,TYP=type))       ⎪
                  ⎩                             ⎩vers        ⎭                  ⎭
```

*STD is still accepted as version specification for the OUTFILE operand in DO procedures and batch jobs.

Function

SHOW-ATTRIBUTES provides information regarding the program executed last, available tools, project directory and the compilation units to be compiled. By default, the information is output to system file SYSOUT; it can, however, be directed to any other file. The output format of the SHOW statement is described below.

The specifications WHAT=*LOADED-PROGRAM and TOOLS cause the program executed last and the tools available, respectively, as well as any assignments of files to program parameters to be output (see also RUN statement).

The specification WHAT=PROJECT causes the information stored in the current project directory to be output via the compilation units. By default, the package or program name and the file name in which the unit is stored are output. Additionally,

(a) all directly and indirectly imported package specifications

(b) all compilation units that import the specified package directly or indirectly

can be output.

The information under (b) is of particular importance as all compilation units affected by a change in specification are shown.

Specification of RECOMPILATIONS causes COMPILE statements for recompilation of compilation units to be generated. Recompilation may be necessary for the following reasons:

(a) A compilation unit has been edited

(b) The specification of an imported package has been changed or recompiled (without being changed)

(c) Errors have been generated during the compilation of a compilation unit.

(d) The GENERATE=OFF option has been specified for the compilation of a package body or a new main program.

Unless specified otherwise, the COMPILE statements are generated only for those compilation units that must be recompiled in order that all compilation units associated with a program are present in a consistent state. COMPILE statements for the compilation of all packages can also be generated.

The COMPILE statements are arranged in such a way that the compilations are performed in the correct order. This assumes that compilation units have been changed by means of the EDIT statement exclusively. Following changes in the WITH clause (list of imported packages) the compilation unit should be compiled so that the new package references in the project directory can be updated. If the compilation is not peformed, the package references valid prior to the change are assumed for establishing the sequence of compilations. This may result in the compilations being performed in the wrong order.

The spin-off mechanism is triggered by the following errors, but (b) through (c) only if WHAT=PROJECT or WHAT=RECOMPILATIONS):

(a) Errors in the syntax of the SHOW statement

(b) The project directory has not been defined

(c) The name of a package or program has been defined for which no entry in the project directory exists

WHAT Specifies the desired information.

= LOADED PROGRAM
Specifies the name of the program executed last with RUN. If file assignments were specified in the RUN statement, they are also output.

= TOOLS (*ALL | tool-name)
By default (*ALL), the name of all tools are output. For each tool the assignments of physical files to program parameters are also given if the tool has already been executed (see RUN statement). When "tool-name" is specified, only the information concerning that tool is output.

= PROJECT-FILE (PACKAGES=.., KIND=.., REFERENCES=.., USED-BY=..)
This specification causes information from the project directory to be output. The extent of the output is controlled by additional operands. By default the names of all compilation units (PACKAGE NAME), the kind of compilation units (KIND) and the file names (LOCATION) under which they are stored, are output. Packages to be compiled for the first time are characterized by an "!" preceding the kind of compilation unit (KIND).

PACKAGES = * | [*] pascal-name [*]

Defines the names of the packages for which information is to be output. By default ("*"), all packages will be considered. Specifying "pascal-name" causes the information regarding this package or main program to be output.
It is also possible to specify the wildcard character "*", which stands for any (or an empty) string. "*pascal-name" provides all entries ending in the string specified in "pascal-name"; "pascal-name*", all entries starting with the string and "*pascal-name*", all entries containing this string.

KIND = ALL | SPEC | BODY | PROG

This operand is used to restrict the output to specifications (SPEC), or bodies (BODY), or main programs (PROG) of the names specified in PACKAGE. By default (=ALL), information about all compilation units is output.

REFERENCES = NONE | ALL | DIRECT | INDIRECT

For each package specified by PACKAGE and KIND, all packages imported directly or indirectly can be output.
By default (= NONE), no package references are output.
If ALL is specified, all packages imported directly and indirectly are output.
If DIRECT is specified, only packages imported directly are output.
If INDIRECT is specified, only packages imported indirectly are output. All of these are packages imported directly or indirectly by directly imported package specifications.
If a compilation unit does not import any packages, "--" is output instead. If the compilation unit had been modified but not yet compiled, "- undefined -" is output.

USED-BY = NONE | ALL | DIRECT | INDIRECT

For each package specified by PACKAGE and KIND, all packages can be output that import this package directly or indirectly. Hence these are packages that are to be recompiled in case of modification of the specified package. This therefore corresponds to the reversed relation of REFERENCES.
By default (= NONE), this information is not output.
If ALL is specified, all packages are output that import this package directly or indirectly.
If DIRECT is specified, all packages are output that import this package directly, i.e. that have this package specified in their WITH clause.
If INDIRECT is specified, all packages are output that import the specified package directly or indirectly.

= RECOMPILATIONS (PACKAGES=.., KIND=..)

This operand controls the generation of COMPILE statements for compilation units. The PACKAGES and KIND operands are used to select the compilation units for which COMPILE statements are to be generated.
In the generated COMPILE statements only the SOURCE operands (name of compilation unit and the KIND) are specified. The statements, insofar as they are output to a file, can be brought into effect by means of the CALL statement (for the unspecified COMPILE operands the current values of the COMPILE statement are assumed).
The order of the statements represents a valid compilation sequence (see notes above).

PACKAGES = *NECESSARY | *ALL | pascal-name

By default (= *NECESSARY), COMPILE statements for modified compilation units and units depending on them are output.
If *ALL is specified, COMPILE statements for all compilation units are generated; if "pascal-name" is specified, only the compilation unit mentioned and the compilation units depending on it.

KIND = ALL | SPEC | BODY | PROG

By default (= ALL), all compilation units are checked to find out whether they have to be recompiled. The KIND specification is used to restrict this check to specifications (SPEC), or bodies (BODY), or main programs (PROG) of the names specified in PACKAGES.

OUTFILE Output is sent to the specified file.

= *SYSOUT
Output to system file SYSOUT

= *SYSLST
Output to system file SYSLST

= *EDT (WORKFILE = *STD | 0..9)
Output to the specified work area of EDT. By default (*STD), this is the current area.

= filename
Name of the output file

= (filename, element (vers, type))
Output is sent to library element "element" of PLAM library "filename". By default, the name of the library element is "P" (for print file). If the generated element is to be executed by means of the CALL statement, if for example it contains COMPILE statements, then type "J" (for job control) must be specified explicitly. By default, the element version is *UPPER-LIMIT (highest-possible version). If an element with the same name, version and type already exists, it is overwritten.

As of Pascal-XT V2.2A, the version specification *STD is no longer permissible in dialog. *STD is still accepted in DO procedures and batch jobs.

Output format of the SHOW statement for the project directory

The example from chapter 3 illustrates the use of the output format.

```
//D TEST.DIRECTORY
//S P(,,ALL,ALL) ────────────────────────────────────────────────────── (01)


     CONTENTS OF THE PROJECT DIRECTORY FILE   (TEST.DIRECTORY)

PACKAGE NAME    KIND  LOCATION

A               (SPEC) ($USERID.PLAM.SPEC,A.SPEC(*UPPER-LIMIT,S)) ──────(02)
    REFERENCES ─────────────────────────────────────────────────────── (03)
      direct:   - - ───────────────────────────────────────────────── (04)
      indirect: - -
    USED BY ────────────────────────────────────────────────────────── (05)
      direct:   A (OWN BODY), B (SPEC) ─────────────────────────────── (06)
      indirect: B (BODY), C (SPEC), C (BODY)

A               (BODY) $USERID.A.BODY
    REFERENCES
      direct:   A (OWN SPEC)
      indirect: - -

B              !(SPEC) ($USERID.PLAM.SPEC,B.SPEC(*UPPER-LIMIT,S)) ────── (07)
    REFERENCES
      direct:   - undefined - ─────────────────────────────────────── (08)
      indirect: - undefined -

B              !(BODY) ($USERID.PLAM.BODY,B.BODY(*UPPER-LIMIT,S))
    REFERENCES
      direct:   B (OWN SPEC)
      indirect: A

C              !(SPEC) ($USERID.PLAM.SPEC,C.SPEC(*UPPER-LIMIT,S))
    REFERENCES
      direct:   B
      indirect: A
    USED BY
      direct:   C (OWN BODY)
      indirect: - -

C              !(BODY) ($USERID.PLAM.BODY,C.BODY(*UPPER-LIMIT,S))
    REFERENCES
      direct:   C (OWN SPEC), D
      indirect: A, B

D               (SPEC) ($USERID.PLAM.SPEC,D.SPEC(*UPPER-LIMIT,S))
    REFERENCES
      direct:   - -
      indirect: - -
    USED BY
      direct:   C (BODY), D (OWN BODY)
      indirect: - -
```

```
D               (BODY) ($USERID.PLAM.BODY,D.BODY(*UPPER-LIMIT,S))
    REFERENCES
      direct:   D (OWN SPEC)
      indirect: - -
```

(01) The SHOW statement is used to output the contents of the project directory, including the relations between packages. The format of the statement when written in full is:

```
SHOW-ATTRIBUTES WHAT = PROJECT-FILE (REFERENCES = ALL, USED-BY = ALL)
```

(02) For each entry in the project directory the package name, or program name, is output followed by information on whether this is a package specification (SPEC), a package body (BODY) or a main program (PROG) and, finally, by the name of the associated source file. For library elements, the library name, the element name, the version and the type of element are output. If the source resides on a file, the associated BS20000 file name is output.

(03) The packages imported by this package or main program are listed under the heading REFERENCES. Packages that have been specified directly in the WITH clause and, where package bodies are concerned, also any user-own specification (OWN SPEC), are considered directly imported. Indirectly imported are all those packages that are imported by the specifications of the imported packages (even if this involves several stages).

(04) Two successive minus signs ("--") mean that no packages are imported, or that this package is not imported by other compilation units.

(05) Compilation units that import this package are listed under the heading USED-BY. The information given for REFERENCES applies equally to direct and indirect. This output is only produced for specifications as package bodies and main programs cannot be imported by other packages.

(06) If an entry refers to a user-own specification or a user-own body, it is prefixed by the word OWN.

(07) All compilation units that have to be recompiled are identified by an exclamation mark ("!"). These include all modified compilation units and the compilation units depending on them.

(08) After a compilation unit has been edited "- undefined -" is output as the system does not know whether the WITH clause of the compilation unit has been changed.

2.6.13 STEP

Define restart point.

```
STEP
```

Function

STEP defines the point at which execution is to restart if an error is encountered in a statement. This statement can only be specified in a statement file (see CALL statement), a DO procedure or an ENTER job.

2.6.14 SYSTEM-COMMAND

Switch to BS2000 command mode or execute a BS2000 system command.

```
SYSTEM-COMMAND

   [COMMAND  =  bs2000-cmd]
```

Function

SYSTEM-COMMAND switches to BS2000 command mode (same effect as the K2 key), or carries out a BS2000 command without terminating the programming system.

If an error occurs during the execution of the command "bs2000-cmd", the spin-off mechanism in the programming system is activated. An error in command execution after processing has switched to BS2000 command mode has no effect on the spin-off mechanism of the programming system.

COMMAND
: Defines whether processing is to switch to BS2000 command mode or only one BS2000 command is to be carried out. If the operand is not specified, the programming system is interrupted and any BS2000 commands may be entered. The operand name COMMAND must not be entered as otherwise it may be interpreted as part of the BS2000 command. Following input of the RESUME command the programming system continues.

= bs2000-cmd
: Designates a BS2000 command, specified in the customary format.

Note

Commands such as EXEC unload the programming system. Calling such commands should be avoided because the programming system cannot be terminated properly.

2.7 Examples

Developing a main program

This sample session shows how a program is developed that outputs a BS2000 text file to the screen. Subsequently, this program is loaded as a tool in the programming environment.

```
/EXEC $userid.PASCAL-XT
%  BLS0500 PROGRAM 'PASCALXT', VERSION '22A00' OF ...  LOADED<
%  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG.1990. ALL RIGHT
S RESERVED
//e list.pas ──────────────────────────────────────────── (01)
```

```
   1.00 ..........................................................................
   2.00 ..........................................................................
   3.00 ..........................................................................
   4.00 ..........................................................................
   5.00 ..........................................................................
   6.00 ..........................................................................
   7.00 ..........................................................................
   8.00 ..........................................................................
   9.00 ..........................................................................
  10.00 ..........................................................................
  11.00 ..........................................................................
  12.00 ..........................................................................
  13.00 ..........................................................................
  14.00 ..........................................................................
  15.00 ..........................................................................
  16.00 ..........................................................................
  17.00 ..........................................................................
  18.00 ..........................................................................
  19.00 ..........................................................................
  20.00 ..........................................................................
  21.00 ..........................................................................
  22.00 ..........................................................................
                 FILE "$userid.LIST.PAS" DOES NOT EXIST                          ── (02)
............................................................0000.00:001(0)
```

(01) Calling EDT specifying the name of the file to be edited. The format of the statement when written in full is:

```
EDIT-UNIT UNIT = LIST.PAS (KIND = FILE)
```

(02) In the EDT status line, a message is issued that the file list.pas does not yet exist.

```
   1.00   program list (output, f);
   2.00   var
   3.00      f : text;
   4.00      line  : string;
   5.00   begin
   6.00      reset (f);
   7.00      while not eof (f) do
   8.00         begin
   9.00         readln (f, line)
  10.00         writeln (line);
  11.00         end;
  12.00   end.
  13.00 ........................................................................
  14.00 ........................................................................
  15.00 ........................................................................
  16.00 ........................................................................
  17.00 ........................................................................
  18.00 ........................................................................
  19.00 ........................................................................
  20.00 ........................................................................
  21.00 ........................................................................
  22.00 ........................................................................
  23.00 ........................................................................
h..........................................................0001.00:001(0)  —— (03)
```

```
   >>> (OVER)WRITE "$userid.LIST.PAS" (y/n) ? ————————————————————————— (04)
*y
   >>> "$userid.LIST.PAS" (OVER)WRITTEN —————————————————————————————— (05)
//mc *e(0),*e(5),(y),ch=on,in=on,op=on ——————————————————————————————— (06)
//c ———————————————————————————————————————————————————————————————————— (07)
   >>> 1 COMPILATION ERROR DETECTED (WARNINGS:  0;  NOTES:  0)
//e ,5 ————————————————————————————————————————————————————————————————— (08)
```

(03) After the source text has been entered, EDT is left by means of HALT. Subsequently, the program is stored by the Pascal-XT programming system, so no WRITE command is required in EDT.

(04) The programming system asks whether the contents of the EDT work area are to be stored in file list.pas. This query can be suppressed if the QUERY=NO operand is specified in the EDIT statement when this is called. In this case, rewriting is always performed. Since file list.pas does not yet exist at this very moment, it is created right now.

(05) This message confirms successful writing to the file. It also appears if QUERY=NO.

(06) The MODIFY-COMPILE statement enables the desired presettings to be made for the COMPILE statement. Its format when written in full is:

```
MODIFY-COMPILE SOURCE = *EDT (WORKFILE=0),
               LISTING = *EDT (WORKFILE=5),
               MODULE-LIBRARY = *OMF (ERASE = YES),
               CHECK = ON, INITIALIZE = ON, OPTIMIZE = ON
```

In case of a subsequent COMPILE statement the compiler expects the source to be in EDT work area 0, writes the listing to EDT work area 5 and the generated object program to the * area which is cleared prior to every compilation.

(07) Calling the compiler using the preset parameters.

(08) As the compilation was unsuccessful, an attempt is now being made to find the error in the listing. To do this, work area 5 of EDT is entered. The format of this statement when written in full is:

```
EDIT-UNIT UNIT = *EDT, WORKFILE = 5
```

The entry *EDT in the UNIT operand leaves the specified work area unchanged when EDT is called, so nothing is read in.

```
  1.00   A*** SOURCE LISTING ***    BS2000 PASCAL-XT COMPILER  V2.2A00
  2.00
  2.00
  3.00
  4.00    GLOBAL OPTIONS FOR THIS COMPILATION
  5.00
  6.00    CHECK       =    ON            BY COMMAND
  7.00    INITIALIZE =    ON            BY COMMAND
  8.00    OPTIMIZE    =    ON            BY COMMAND
  9.00    DEBUG       =    OFF           BY OPTIMIZE OPTION
 10.00    GENERATE    =    ON            BY DEFAULT
 11.00    MAP         =    OFF           BY DEFAULT
 12.00    STANDARD    =    OFF           BY DEFAULT
 13.00    XREF        =    OFF           BY DEFAULT
 14.00
 15.00
 16.00    CURRENT COMPILATION UNIT (SOURCE FILE)
 17.00
 18.00                   *EDT(0)
 19.00
 20.00          1      program list (output, f);
 21.00          2      var
 22.00          3         f : text;

+.................................................................0001.00:001(5)
```

```
 23.00          4         line : string;
 24.00          5      begin
 25.00          6         reset (f);
 26.00          7         while not eof (f) do
 27.00          8            begin
 28.00          9            readln (f, line)                              —— (09)
 29.00         10            writeln (line);
 30.00    ------------------1
 31.00    >>> 1: ERROR    218: ";" INSERTED
 32.00
 33.00         11            end;
 34.00         12      end.
 35.00
 36.00
 37.00    ***********************************************************
 38.00    *                   COMPILATION SUMMARY                   *
 39.00    ***********************************************************
 40.00    *  ERRORS DETECTED         :        1                     *
 41.00    *  WARNINGS                :        0                     *
 42.00    *  NOTES                   :        0                     *
 43.00    *  SIZE OF CODE MODULE     :        0 BYTES               *
 44.00    *  SIZE OF DATA MODULE     :        0 BYTES               *
 45.00    *  COMPILATION TIME        :    0.174 SEC                 *
 46.00    ***********************************************************
 47.00
0.................................................................0024.00:001(5)   —— (10)
```

(09) A semicolon was missed out in line 9 when the program was input.

(10) To correct this, control is passed to work area 0 of EDT.

```
   1.00   program list (output, f);
   2.00   var
   3.00      f : text;
   4.00      line  : string;
   5.00   begin
   6.00      reset (f);
   7.00      while not eof (f) do
   8.00         begin
   9.00         readln (f, line);
  10.00         writeln (line);
  11.00         end;
  12.00   end.
  13.00 .........................................................................
  14.00 .........................................................................
  15.00 .........................................................................
  16.00 .........................................................................
  17.00 .........................................................................
  18.00 .........................................................................
  19.00 .........................................................................
  20.00 .........................................................................
  21.00 .........................................................................
  22.00 .........................................................................
  23.00..........................................................................
h.................................................................0001.00:001(0)
```

```
   >>> (OVER)WRITE "$userid.LIST.PAS" (y/n) ? ---------------------------- (11)
*y
   >>> "$userid.LIST.PAS" (OVER)WRITTEN
//c -------------------------------------------------------------------- (12)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//sy file list.pas,link=f ---------------------------------------------- (13)
//r -------------------------------------------------------------------- (14)
program list (output, f);
var
   f : text;
   line  : string;
begin
   reset (f);
   while not eof (f) do
      begin
      readln (f, line);
      writeln (line);
      end;
end.
```

(11) EDT is left by means of HALT. As the programming system keeps a record of the contents of the EDT work area, it now queries whether file list.pas is to be rewritten.

(12) Compilation successful.

(13) Prior to program execution a BS2000 file is assigned to Pascal file F by means of the BS2000 FILE command. In this case, the source of the program, file list.pas, is used as an input file.

(14) The RUN statement is used to load program list from the * area and start it. Its format when written in full is:

```
RUN-PROGRAM PROGRAM = *LAST-COMPILED-PROGRAM
```

```
//s ─────────────────────────────────────────────────── (15)

                    LAST LOADED PROGRAM

     list
//sy rel f ──────────────────────────────────────────── (16)
//r ,'f=list.pas' ───────────────────────────────────── (17)
program list (output, f);
var
   f : text;
   line  : string;
begin
   reset (f);
   while not eof (f) do
      begin
      readln (f, line);
      writeln (line);
      end;
end.
```

(15) THE SHOW statement provides information about the last program loaded. The format of this statement when written in full is:

```
SHOW-ATTRIBUTES WHAT = LOADED-PROGRAM
```

(16) The BS2000 RELEASE command releases the link between BS2000 file and Pascal file F.

(17) The assignment of Pascal file F to a BS2000 file can also be made with the RUN statement. Its format when written in full is:

```
RUN-PROGRAM PROGRAM = *LAST-COMPILED-PROGRAM,
            PARAMETER = 'f = list.pas'
```

This assignment is retained until it is overwritten by a new entry in the RUN statement, until another program is loaded or the programming system is left.

```
//s ─────────────────────────────────────────────────────────── (18)

                    LAST LOADED PROGRAM

     list
          F  <- - - LIST.PAS
//s t ──────────────────────────────────────────────────────────── (19)
   >>> NO TOOL(S) AVAILABLE
//a (*omf,list),l ─────────────────────────────────────────────── (20)
//s t ──────────────────────────────────────────────────────────── (21)

     TOOLS OF THE PROGRAMMING ENVIRONMENT

     L
//r l,'f=list.pas' ────────────────────────────────────────────── (22)
program list (output, f);
var
   f : text;
   line  : string;
begin
   reset (f);
   while not eof (f) do
      begin
      readln (f, line);
      writeln (line);
      end;
end.
```

(18) The SHOW statement outputs the program name and the BS2000 file assigned to Pascal file F.

(19) As the program is to be loaded as a tool in the programming system environment, information about any tools loaded already is first requested by way of the SHOW command. Issuing alias names twice is thus prevented. The format of this statement when written in full is:

```
SHOW-ATTRIBUTES WHAT = TOOLS (TOOL = *ALL)
```

(20) The ADD statement loads program "list" as a tool in the programming environment under the name l. The format of this statement when written in full is:

```
ADD-TOOL (LIBRARY = *OMF, ELEMENT = LIST), ALIAS-NAME = L
```

(21) Now the SHOW statement shows that a tool having name l is present.

(22) Before the tool is executed, the link between Pascal file F and a BS2000 file must be established, either by way of a FILE command or, as is done here, with the aid of the RUN command. The format of this statement when written in full is:

```
RUN-PROGRAM PROGRAM = L, PARAMETER = 'f = list.pas'
```

This link is retained until it is overwritten by a new entry in the RUN statement, or the programming system is left.

```
//s t ─────────────────────────────────────────────────────────────────── (23)

     TOOLS OF THE PROGRAMMING ENVIRONMENT

     L
          F  <- - - LIST.PAS
//r l ─────────────────────────────────────────────────────────────────── (24)
program list (output, f);
var
   f : text;
   line  : string;
begin
   reset (f);
   while not eof (f) do
      begin
      readln (f, line);
      writeln (line);
      end;
end.
//end ─────────────────────────────────────────────────────────────────── (25)
   END OF THE PASCAL SESSION  -  USED TIME = 2.918 SECONDS
%  E732 ABNORMAL PROGRAM TERMINATION APTT101 ─────────────────────────── (26)
```

(23) The SHOW statement outputs the name of the tool as well as the file allocations made in the RUN statement.

(24) As the file allocation has already been made, the specification of parameters can be omitted when tool l is called once more.

(25) The Pascal-XT programming system terminates. All modifications of operand presettings, e.g. by the MODIFY-COMPILE statement, see (06), will be lost and will not be restored when the programming system is restarted.

(26) If an illegal statement is encountered in the programming system, or an error occurred during compilation, the BS2000 spin-off mechanism is activated when the programming system terminates. As a result, message `ABNORMAL PROGRAM TERMINATION` is issued in interactive mode, or a branch is made to the next STEP or ENDP or to LOGOFF in DO procedures and ENTER jobs.

Working with PLAM libraries and the project directory

This sample session shows how a program is developed which represents an aid to cleaning up user IDs. It outputs a listing of all cataloged files and subsequently asks whether each file has to be deleted or not.

The program consists of the main program ERAQ and the packages FSTAT and DIALOG. Furthermore, the two predefined packages BS2000CALLS and DMSIO are used.

In particular, the example illustrates project directory handling and program development with the aid of PLAM libraries.

```
/EXEC $userid.PASCAL-XT
%  BLS0500 PROGRAM 'PASCALXT', VERSION '22A00' OF ... LOADED<
%  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG.1990. ALL RIGHT
S RESERVED
//E (ERAQ.PLAM,FSTAT.SPEC) ——————————————————————————————————————————— (01)
```

```
   1.00 ..........................................................................
   2.00 ..........................................................................
   3.00 ..........................................................................
   4.00 ..........................................................................
   5.00 ..........................................................................
   6.00 ..........................................................................
   7.00 ..........................................................................
   8.00 ..........................................................................
   9.00 ..........................................................................
  10.00 ..........................................................................
  11.00 ..........................................................................
  12.00 ..........................................................................
  13.00 ..........................................................................
  14.00 ..........................................................................
  15.00 ..........................................................................
  16.00 ..........................................................................
  17.00 ..........................................................................
  18.00 ..........................................................................
  19.00 ..........................................................................
  20.00 ..........................................................................
  21.00 ..........................................................................
  22.00 ..........................................................................
                    OPEN ERROR ON LIBRARY "$userid.ERAQ.PLAM"                        —— (02)
..................................................................0000.00:001(0)
```

(01) Calling EDT, specifying library name and name of the element to be edited. Neither the library nor the element exist at this point in time. The format of this statement when written in full is:

```
EDIT-UNIT UNIT = *LIBRARY (LIBRARY = ERAQ.PLAM,
                           ELEMENT = FSTAT.SPEC)
```

(02) As library eraq.plam does not yet exist, a message that it cannot be opened is issued in the EDT status line.

```
   1.00 package fstat;
   2.00 type
   3.00    fntype = string[54];
   4.00
   5.00 procedure makefstat (filename: fntype);
   6.00
   7.00 end.
   8.00 ........................................................................
   9.00 ........................................................................
  10.00 ........................................................................
  11.00 ........................................................................
  12.00 ........................................................................
  13.00 ........................................................................
  14.00 ........................................................................
  15.00 ........................................................................
  16.00 ........................................................................
  17.00 ........................................................................
  18.00 ........................................................................
  19.00 ........................................................................
  20.00 ........................................................................
  21.00 ........................................................................
  22.00 ........................................................................
  23.00.........................................................................
h.........................................................0001.00:001(0)
```

```
   >>> (OVER)WRITE "($userid.ERAQ.PLAM,FSTAT.SPEC(*UPPER-LIMIT,S))" (y/n) ? (03)
*y
   >>> "($userid.ERAQ.PLAM,FSTAT.SPEC(*UPPER-LIMIT,S))" (OVER)WRITTEN
//ME (ERAQ.PLAM) ———————————————————————————————————————————————————— (04)
//E (,FSTAT.BODY),1 ————————————————————————————————————————————————— (05)
```

(03) Once the source text has been input, EDT is left by means of HALT. The programming system asks whether the contents of the work area have to be stored in library element fstat.spec of library eraq.plam. This query can be suppressed by specifying QUERY=NO in the EDIT statement when the call is made. In this case, rewriting is always performed, without any query being made. As library eraq.plam does not yet exist at this point in time, it is now created as a PLAM library and element fstat.spec of type S is subsequently added to it.

(04) To make things easier the MODIFY-EDIT statement is used to preset the name of library eraq.plam. The format of this statement when written in full is:

```
MODIFY-EDIT UNIT = *LIBRARY (LIBRARY = ERAQ.PLAM)
```

(05) From now on it suffices to specify the element name in the EDT statement. The body of the package must be entered in work area 1 so that the specification is still present in work area 0 in case it is needed. The format of this statement when written in full is:

```
EDIT-UNIT UNIT = *LIBRARY (, ELEMENT = FSTAT.BODY), WORKFILE = 1
```

```
   1.00 ..................................................................
   2.00 ..................................................................
   3.00 ..................................................................
   4.00 ..................................................................
   5.00 ..................................................................
   6.00 ..................................................................
   7.00 ..................................................................
   8.00 ..................................................................
   9.00 ..................................................................
  10.00 ..................................................................
  11.00 ..................................................................
  12.00 ..................................................................
  13.00 ..................................................................
  14.00 ..................................................................
  15.00 ..................................................................
  16.00 ..................................................................
  17.00 ..................................................................
  18.00 ..................................................................
  19.00 ..................................................................
  20.00 ..................................................................
  21.00 ..................................................................
  22.00 ..................................................................
                 ELEMENT "FSTAT.BODY (*UPPER-LIMIT,S)" DOES NOT EXIST        —— (06)
.............................................................0000.00:001(1)
```

```
   1.00 with BS2000CALLS;
   2.00 package body fstat;
   3.00
   4.00 procedure makefstat (filename: fntype);
   5.00 var
   6.00    b    : boolean;
   7.00 begin
   8.00    BS2000CALLS.cmd (concat('/SYSFILE SYSLST=',filename), b);
   9.00    BS2000CALLS.cmd ('/OPTION MSG=FHL', b);
  10.00    BS2000CALLS.cmd ('/FSTAT', b);
  11.00    BS2000CALLS.cmd ('/OPTION MSG=F', b);
  12.00    BS2000CALLS.cmd ('/SYSFILE SYSLST=(PRIMARY)', b);
  13.00 end;
  14.00
  15.00 begin
  16.00 end.
  17.00 ..................................................................
  18.00 ..................................................................
  19.00 ..................................................................
  20.00 ..................................................................
  21.00 ..................................................................
  22.00 ..................................................................
  23.00...................................................................
h............................................................0001.00:001(1)
```

```
   >>> (OVER)WRITE "($userid.ERAQ.PLAM,FSTAT.BODY(*UPPER-LIMIT,S))" (y/n) ?
*y
   >>> "($userid.ERAQ.PLAM,FSTAT.BODY(*UPPER-LIMIT,S))" (OVER)WRITTEN
```

(06) In the EDT status line, a message is issued that element fstat.body does not yet exist.

```
//E (,DIALOG.SPEC),2 ———————————————————————————————————————————— (07)
```

```
    1.00 ........................................................................
    2.00 ........................................................................
    3.00 ........................................................................
    4.00 ........................................................................
    5.00 ........................................................................
    6.00 ........................................................................
    7.00 ........................................................................
    8.00 ........................................................................
    9.00 ........................................................................
   10.00 ........................................................................
   11.00 ........................................................................
   12.00 ........................................................................
   13.00 ........................................................................
   14.00 ........................................................................
   15.00 ........................................................................
   16.00 ........................................................................
   17.00 ........................................................................
   18.00 ........................................................................
   19.00 ........................................................................
   20.00 ........................................................................
   21.00 ........................................................................
   22.00 ........................................................................
                  ELEMENT "DIALOG.SPEC (*UPPER-LIMIT,S)" DOES NOT EXIST
..................................................................0000.00:001(2)
```

```
    1.00 package dialog;
    2.00 type
    3.00    fntype = string[54];
    4.00
    5.00 procedure makedialog (filename: fntype);
    6.00
    7.00 end.
    8.00 ........................................................................
    9.00 ........................................................................
   10.00 ........................................................................
   11.00 ........................................................................
   12.00 ........................................................................
   13.00 ........................................................................
   14.00 ........................................................................
   15.00 ........................................................................
   16.00 ........................................................................
   17.00 ........................................................................
   18.00 ........................................................................
   19.00 ........................................................................
   20.00 ........................................................................
   21.00 ........................................................................
   22.00 ........................................................................
   23.00.........................................................................
h.................................................................0001.00:001(2)
```

```
   >>> (OVER)WRITE "($userid.ERAQ.PLAM,DIALOG.SPEC(*UPPER-LIMIT,S))" (y/n) ?
*y
   >>> "($userid.ERAQ.PLAM,DIALOG.SPEC(*UPPER-LIMIT,S))" (OVER)WRITTEN
```

(07) The specification of package "dialog" is entered in EDT work area 2.

//E (,DIALOG.BODY),3 ———— (08)

```
   1.00 ..........................................................................
   2.00 ..........................................................................
   3.00 ..........................................................................
   4.00 ..........................................................................
   5.00 ..........................................................................
   6.00 ..........................................................................
   7.00 ..........................................................................
   8.00 ..........................................................................
   9.00 ..........................................................................
  10.00 ..........................................................................
  11.00 ..........................................................................
  12.00 ..........................................................................
  13.00 ..........................................................................
  14.00 ..........................................................................
  15.00 ..........................................................................
  16.00 ..........................................................................
  17.00 ..........................................................................
  18.00 ..........................................................................
  19.00 ..........................................................................
  20.00 ..........................................................................
  21.00 ..........................................................................
  22.00 ..........................................................................
                  ELEMENT "DIALOG.BODY (*UPPER-LIMIT,S)" DOES NOT EXIST
...................................................................0000.00:001(3)
```

```
   1.00 with BS2000CALLS, DMSIO;
   2.00 package body dialog (input, output, tmp);
   3.00 var
   4.00    tmp  : text;
   5.00
   6.00 procedure makedialog (filename: fntype);
   7.00 var
   8.00    c    : char;
   9.00    b    : boolean;
  10.00    line : string;
  11.00 begin
  12.00    Assignfile (tmp, filename);
  13.00    reset (tmp);
  14.00    while not eoln (tmp) do
  15.00       begin
  16.00       readln (tmp, line);
  17.00       delete (line, 1, 27);
  18.00       writeln ('Is the file ''', line,
  19.00                ''' to be deleted?(Y/N/E):');
  20.00       readln;
  21.00       read (c);
  22.00       if c in ['Y','y','j','J'] then
  23.00           begin
+..................................................................0001.00:001(3)
```

(08) The body of package "dialog" is entered in EDT work area 3.

```
  24.00           BS2000CALLS.cmd (concat ('/ERASE ',line), b);
  25.00           if not b then
  26.00              begin
  27.00              writeln ('file ''', line, ''' deleted!');
  28.00              writeln;
  29.00              end
  30.00           end
  31.00       else if (c = 'E') or (c = 'e') then
  32.00           exit;
  33.00       end;
  34.00    DMSIO.close (tmp ) ;
  35.00    BS2000CALLS.cmd (concat('/ERASE', filename), b )
  36.00 end;
  37.00
  38.00 begin
  39.00 end.
  40.00 ........................................................................
  41.00 ........................................................................
  42.00 ........................................................................
  43.00 ........................................................................
  44.00 ........................................................................
  45.00 ........................................................................
  46.00 ........................................................................
h.................................................................0024.00:001(3)
```

```
   >>> (OVER)WRITE "($userid.ERAQ.PLAM,DIALOG.BODY(*UPPER-LIMIT,S))" (y/n) ?
*y
   >>> "($userid.ERAQ.PLAM,DIALOG.BODY(*UPPER-LIMIT,S))" (OVER)WRITTEN
//E (,ERAQ.PROG),4 ———————————————————————————————————————————————————— (09)
```

```
   1.00 ........................................................................
   2.00 ........................................................................
   3.00 ........................................................................
   4.00 ........................................................................
   5.00 ........................................................................
   6.00 ........................................................................
   7.00 ........................................................................
   8.00 ........................................................................
   9.00 ........................................................................
  10.00 ........................................................................
  11.00 ........................................................................
  12.00 ........................................................................
  13.00 ........................................................................
  14.00 ........................................................................
  15.00 ........................................................................
  16.00 ........................................................................
  17.00 ........................................................................
  18.00 ........................................................................
  19.00 ........................................................................
  20.00 ........................................................................
  21.00 ........................................................................
  22.00 ........................................................................
                   ELEMENT "ERAQ.PROG (*UPPER-LIMIT,S)" DOES NOT EXIST
.................................................................0000.00:001(4)
```

(09) Main program eraq.prog is entered in EDT work area 4.

```
   1.00 with FSTAT, DIALOG;
   2.00 program eraq (input, output);
   3.00 const
   4.00    tempfile = 'TMP.SYSLST';
   5.00
   6.00 begin
   7.00    FSTAT.makefstat (tempfile);
   8.00    DIALOG.makedialog (tempfile);
   9.00 end.
  10.00 ........................................................................
  11.00 ........................................................................
  12.00 ........................................................................
  13.00 ........................................................................
  14.00 ........................................................................
  15.00 ........................................................................
  16.00 ........................................................................
  17.00 ........................................................................
  18.00 ........................................................................
  19.00 ........................................................................
  20.00 ........................................................................
  21.00 ........................................................................
  22.00 ........................................................................
  23.00.........................................................................
h..........................................................0001.00:001(4)
```

```
   >>> (OVER)WRITE "($userid.ERAQ.PLAM,ERAQ.PROG(*UPPER-LIMIT,S))" (y/n) ?
*y
   >>> "($userid.ERAQ.PLAM,ERAQ.PROG(*UPPER-LIMIT,S))" (OVER)WRITTEN
//D ERAQ.DIR ──────────────────────────────────────────────────────── (10)
   >>> PROJECT DIRECTORY FILE CREATED ─────────────────────────────── (11)
//C ($PASSUP-XT,BS2000CALLS.SPEC),*D ───────────────────────────────── (12)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//C ($PASSUP-XT,DMSIO.SPEC),*D
   >>> COMPILATION SUCCESSFUL
```

(10) After all sources have been input, the project directory must now be defined prior to the first compilation. The format of this statement written in full is:

```
DEFINE-PROJECT-FILE DIRECTORY = ERAQ.DIR
```

(11) As project directory eraq.dir does not exist yet, it is created by the preceding DEFINE-PROJECT-FILE statement, opened for processing and this message is issued. If the project directory exists already, it is only opened and no message is issued.

(12) Before a specification can be used during the compilation of another specification, a package body or a main program, this specification must be entered in the project directory. This is effected by compilation of this specification. Here, the specifications of the predefined packages BS2000CALLS and DMSIO are compiled first. Since the listings of these two compilations are immaterial, the listing is written to BS2000 file *DUMMY. The format of this statement when written in full is:

```
COMPILE-UNIT SOURCE = *LIBRARY (SOURCE-LIBRARY = $PASSUP-XT,
                                SOURCE-ELEMENT = DMSIO.SPEC),
             LISTING = *DUMMY
```

```
//MC (ERAQ.PLAM),(*STD,*STD),*STD,CH=ON,IN=ON,OP=ON ──────────────────── (13)
//C (,FSTAT.SPEC) ──────────────────────────────────────────────────────── (14)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//C (,FSTAT.BODY)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//C (,DIALOG.SPEC)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//C (,DIALOG.BODY)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//C (,ERAQ.PROG)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//R ────────────────────────────────────────────────────────────────────── (15)
0000003: C:$userid.AAAA
0000003: C:$userid.AAAA2
0000003: C:$userid.ERAQ.DIR
0000048: C:$userid.ERAQ.PLAM
Is the file 'C:$userid.AAAA' to be deleted?(Y/N/E): ──────────────────── (16)
n
Is the file 'C:$userid.AAAA2' to be deleted?(Y/N/E):
e
//E DIALOG(B) ──────────────────────────────────────────────────────────── (17)
```

(13) The MODIFY-COMPILE statement modifies the default values of the operands for the following compilations: source library = ERAQ.PLAM, listing library = source library, listing element = P element with the same name as source element, module library = source library and the options Check, Initialize and Optimize set to ON. The format of this statement when written in full is:

```
MODIFY-COMPILE SOURCE = *LIBRARY (SOURCE-LIBRARY = ERAQ.PLAM),
               LISTING = *LIBRARY (LIST-LIBRARY = *STD,
                                   LIST-ELEMENT = *STD),
               MODULE = *STD,
               CHECK = ON, INITIALIZE = ON, OPTIMIZE = ON
```

Thanks to these settings, only a PLAM library is required for program development. In this library, the sources, listings and object modules are kept.

(14) At compile time only the element name has to be specified in the COMPILE statement. The format of this statement when written in full is:

```
COMPILE-UNIT SOURCE = *LIBRARY (SOURCE-ELEMENT = FSTAT.SPEC)
```

(15) The RUN statement is now used to start the last program compiled taken from the library to which it had been written. Hence, the main program must always be the last to be compiled. The DLL loads the necessary packages from the same library (autolink mechanism) from which the main program was loaded. It is not possible to start a program with packages from within the * area because the autolink mechanism of DLL does not work there. So here program eraq from eraq.plam is started and the external references to packages fstat and dialog are resolved from library eraq.plam.

(16) The program is invalid. It deletes the first character of the file name (here a blank).

(17) After the first successful compilation, all compilation units were entered in the project directory. It is now possible to work with the package or program names when editing or compiling. The compilation unit DIALOG is now edited with KIND = BODY. The format of this statement when written in full is:

```
EDIT-UNIT UNIT = DIALOG (KIND = BODY)
```

```
   1.00 with BS2000CALLS, DMSIO;
   2.00 package body dialog (input, output, tmp);
   3.00 var
   4.00    tmp  : text;
   5.00
   6.00 procedure makedialog (filename: fntype);
   7.00 var
   8.00    c    : char;
   9.00    b    : boolean;
  10.00    line: string;
  11.00 begin
  12.00    Assignfile (tmp, filename);
  13.00    reset (tmp);
  14.00    while not eoln (tmp) do
  15.00       begin
  16.00       readln (tmp,  line);
  17.00       delete (line, 1, 27);                                         —— (18)
  18.00       writeln ('Is the file ''', line,
  19.00                ''' to be deleted?(Y/N/E):');
  20.00       readln;
  21.00       read (c);
  22.00       if c in ['Y','y','j','J'] then
                    ($userid.ERAQ.PLAM,DIALOG.BODY(*UPPER-LIMIT,S))
.................................................................0001.00:001(0)
```

```
   1.00 with BS2000CALLS, DMSIO;
   2.00 package body dialog (input, output, tmp);
   3.00 var
   4.00    tmp  : text;
   5.00
   6.00 procedure makedialog (filename: fntype);
   7.00 var
   8.00    c    : char;
   9.00    b    : boolean;
  10.00    line: string;
  11.00 begin
  12.00    Assignfile (tmp, filename);
  13.00    reset (tmp);
  14.00    while not eoln (tmp) do
  15.00       begin
  16.00       readln (tmp, line);
  17.00       delete (line, 1, 26);                                         —— (19)
  18.00       writeln ('Is the file ''', line,
  19.00                ''' to be deleted?(Y/N/E):');
  20.00       readln;
  21.00       read (c);
  22.00       if c in ['Y','y','j','J'] then
  23.00           begin
h................................................................0001.00:001(0)
```

```
   >>> (OVER)WRITE PACKAGE BODY "DIALOG" (y/n) ? ———————————————————————— (20)
*y
   >>> PACKAGE BODY "DIALOG" (OVER)WRITTEN
```

(18) The program error occurred in line 17. The length specification in string procedure "delete" is exceeded by one.

(19) Error correction.

(20) The unit name is now used for the query as to whether the package is to be overwritten, and no longer the library and element names.

```
//S R ——————————————————————————————————————————————————————————————— (21)
COMPILE DIALOG (BODY) —————————————————————————————————————————————— (22)
//S P ——————————————————————————————————————————————————————————————— (23)

     CONTENTS OF THE PROJECT DIRECTORY FILE   (ERAQ.DIR)

PACKAGE NAME    KIND  LOCATION
BS2000CALLS     (SPEC) ($PASSUP-XT,BS2000CALLS.SPEC(*UPPER-LIMIT,S))
DIALOG          (SPEC) ($userid.ERAQ.PLAM,DIALOG.SPEC(*UPPER-LIMIT,S))
DIALOG         !(BODY) ($userid.ERAQ.PLAM,DIALOG.BODY(*UPPER-LIMIT,S)) ——(24)
DMSIO           (SPEC) ($PASSUP-XT,DMSIO.SPEC(*UPPER-LIMIT,S))
ERAQ            (PROG) ($userid.ERAQ.PLAM,ERAQ.PROG(*UPPER-LIMIT,S))
FSTAT           (SPEC) ($userid.ERAQ.PLAM,FSTAT.SPEC(*UPPER-LIMIT,S))
FSTAT           (BODY) ($userid.ERAQ.PLAM,FSTAT.BODY(*UPPER-LIMIT,S))

//S P(K=BODY,REF=ALL) ———————————————————————————————————————————————— (25)

     CONTENTS OF THE PROJECT DIRECTORY FILE   (ERAQ.DIR)

PACKAGE NAME    KIND  LOCATION

DIALOG         !(BODY) ($userid.ERAQ.PLAM,DIALOG.BODY(*UPPER-LIMIT,S))
    REFERENCES
      direct:   - undefined -   —————————————————————————————————————— (26)
      indirect: - undefined -

FSTAT           (BODY) ($userid.ERAQ.PLAM,FSTAT.BODY(*UPPER-LIMIT,S))
    REFERENCES
      direct:   BS2000CALLS, FSTAT (OWN SPEC) ——————————————————————— (27)
      indirect: - -
```

(21) The SHOW statement is used to output the necessary recompilations. The format of this statement when written in full is:

```
SHOW-ATTRIBUTES WHAT = RECOMPILATIONS (PACKAGES = *NECESSARY,
                                       KIND = ALL)
```

(22) SHOW RECOMPILATIONS gives you the COMPILE statements for the required compilation units.

(23) The SHOW statement is used to output the contents of the project directory. The format of this statement when written in full is:

```
SHOW-ATTRIBUTES WHAT = PROJECT-FILE
```

(24) The packages to be recompiled are marked with an exclamation mark in the output.

(25) The SHOW statement outputs the specifications imported by the package bodies. The format of this statement when written in full is:

```
SHOW-ATTRIBUTES WHAT = PROJECT-FILE (KIND = BODY,
                                     REFERENCES = ALL)
```

(26) The body of dialog is marked as invalid. So no references can be given for this package. The WITH clauses of the package might have been changed.

(27) The body of fstat uses the specification of BS2000CALLS and, of course, the user-own specification. In this example, there are no indirectly imported packages as the imported specifications, on their part, do not import any other packages.

```
//S P (USE=ALL) ──────────────────────────────────────────────────────── (28)

     CONTENTS OF THE PROJECT DIRECTORY FILE   (ERAQ.DIR)

PACKAGE NAME    KIND  LOCATION

BS2000CALLS    (SPEC) ($PASSUP-XT,BS2000CALLS.SPEC(*UPPER-LIMIT,S))
    USED BY
      direct:   DIALOG (BODY), FSTAT (BODY) ──────────────────────────── (29)
      indirect: - -

DIALOG         (SPEC) ($userid.ERAQ.PLAM,DIALOG.SPEC(*UPPER-LIMIT,S))
    USED BY
      direct:   DIALOG (OWN BODY), ERAQ (PROG)
      indirect: - -

DIALOG        !(BODY) ($userid.ERAQ.PLAM,DIALOG.BODY(*UPPER-LIMIT,S)) ── (30)

DMSIO          (SPEC) ($PASSUP-XT,DMSIO.SPEC(*UPPER-LIMIT,S))
    USED BY
      direct:   DIALOG (BODY)
      indirect: - -

ERAQ           (PROG) ($userid.ERAQ.PLAM,ERAQ.PROG(*UPPER-LIMIT,S))

FSTAT          (SPEC) ($userid.ERAQ.PLAM,FSTAT.SPEC(*UPPER-LIMIT,S))
    USED BY
      direct:   ERAQ (PROG), FSTAT (OWN BODY)
      indirect: - -

FSTAT          (BODY) ($userid.ERAQ.PLAM,FSTAT.BODY(*STD,S))
```

(28) The SHOW statement now outputs the reverse references, i.e. by which packages a package is imported. The format of this statement when written in full is:

```
SHOW-ATTRIBUTES WHAT = PROJECT-FILE (USED-BY = ALL)
```

(29) The specification of BS2000CALLS is used by the bodies of packages DIALOG and FSTAT.

(30) Package bodies cannot be imported from other packages.

```
//S R,*E(8) ————————————————————————————————————————————— (31)
//CA *E(8) —————————————————————————————————————————————— (32)
(%STMT) COMPILE DIALOG (BODY)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//S R ——————————————————————————————————————————————————— (33)
   >>> NO RECOMPILATIONS NECESSARY
//R (ERAQ.PLAM,ERAQ) ———————————————————————————————————— (34)
0000003 :C:$userid.AAAA
0000003 :C:$userid.AAAA2
0000003 :C:$userid.ERAQ.DIR
0000048 :C:$userid.ERAQ.PLAM
Is the file ':C:$userid.AAAA' to be deleted?(Y/N/E):
y
File ':C:$userid.AAAA' deleted!
Is the file ':C:$userid.AAAA2' to be deleted?(Y/N/E):
y
File ':C:$userid.AAAA2' deleted!
Is the file ':C:$userid.ERAQ.DIR' to be deleted?(Y/N/E):
n
Is the file ':C:$userid.ERAQ.PLAM' to be deleted?(Y/N/E):
e
//END
   END OF THE PASCAL SESSION  -  USED TIME = 31.091 SECONDS
```

(31) The SHOW statement once more outputs the necessary recompilations. This time, output is sent to EDT work area 8 so that the generated COMPILE statements can be executed with a CALL-STATEMENT-FILE statement. The format of this statement when written in full is:

```
SHOW-ATTRIBUTES WHAT = RECOMPILATIONS (PACKAGES = *NECESSARY,
                                       KIND = ALL),
                OUTFILE = *EDT (WORKFILE = 8)
```

(32) The statements in EDT work area 8 are executed by means of the CALL-STATEMENT-FILE statement. The presettings established by the MODIFY-COMPILE statement, see (13), are still effective. The format of this statement when written in full is:

```
CALL-STATEMENT-FILE STMT-FILE = *EDT (WORKFILE = 8)
```

(33) The SHOW statement reports that no recompilations are necessary.

(34) As the last compilation was not a main program, the library and the name of the program to be started must be specified in the RUN statement because otherwise the RUN statement will be rejected with this message:

```
>>> LAST COMPILATION UNIT IS NOT EXECUTABLE
```

The format of this statement when written in full is:

```
RUN-PROGRAM PROGRAM = (LIBRARY = ERAQ.PLAM, ELEMENT = ERAQ)
```

3 Project directory

The project directory supports the package concept, which is a key feature of the Pascal-XT language. The package concept permits large, complex programs to be divided into smaller units (packages) that can be processed, compiled and managed separately. Each package consists of a specification and a body.

Compilation units are package specifications, package bodies and main programs. The name of a compilation unit is always the Pascal identifier which is placed after the keyword "program", "package" or "body".

The program name of a main program must be different from the package names of all the packages associated with the program.

3.1 Tasks of the project directory

Assigning program names to file names

In the project directory, the package names and program name of the main program are assigned to the names of files which contain the compilation units of the program. Such assignment is required for a number of functions.

- In order to check the interfaces between packages, the Pascal-XT compiler rereads the specifications of the imported packages. Since the WITH clauses of a compilation unit contain only the names of the imported packages, the compiler must be informed about the names of the files in which the specifications are stored. The compiler takes this information from the assignments of specification package names to file names in the project directory.
- With the frequently used statements for the programming system EDIT and COMPILE, the name of a main program or package to be processed may be specified instead of a file name. The assignments in the project directory are also accessed in this case.

The assignments of package and program names to file names are entered by the compiler in the project directory (see 3.2).

Defining relationships between compilation units

A Pascal-XT program usually consists of a main program and a number of packages. The relationships between the compilation units are statically defined in the WITH clauses of the compilation units. Thus, a program can be represented by a graph whose nodes are the compilation units and in which the main program represents a root. Knowledge of the relationships between the compilation units is of particular importance when a compilation unit is to be changed. The relationships clearly show whether the changes affect any other compilation units, which must then be recompiled.

The relationships between the compilation units can be gathered from the project directory and be output with the aid of the SHOW statement (see sections 2.6.12 and 3.2).

Storing status information

Status information is stored in the project directory for each compilation unit. It includes date and time of the latest compilation and a change mark that is set following a modification by means of the EDIT statement. The SHOW statement accesses this status information to identify which compilation units need to be recompiled.

3.2 Defining and processing the project directory

Defining the project directory

The project directory is defined by means of the DEFINE statement (see section 2.6.5). It is created if it does not yet exist and it remains open until another project directory is defined or the programming system is left.

Modifying the project directory

The assignments of package/program names to the file names that contain the source programs, are entered in the project directory by the compiler only if one has been defined and compilation is successful. Existing assignments are overwritten. Following successful compilation of a package specification the programming system issues an error message if the project directory cannot be updated.

Following a modification a compilation unit is marked as modified, if the modification is effected under programming system control (see EDIT statement), to enable the user to ascertain whether any recompilations are necessary.

Following every compilation of a compilation unit the compilation date is recorded in the project directory to be able to identify any necessary recompilations. For one thing, the compilation date of a compilation unit must be earlier than that of the specifications of all imported packages. A compilation of a specification, even if no modifications were performed, always results in recompilation of the dependent packages.

Deleting entries in the project directory

The information for a superfluous main program or package can be deleted from the current directory file by means of the REMOVE-DIRECTORY-ENTRY statement. All entries for the names specified when the statement was called are deleted. Although the entries are deleted the files of the source program are retained.

Compilation units that import a deleted package are marked as invalid. They must be recompiled.

Output of status information

The SHOW statement can be used to output various items of information regarding the project directory by specifying the PROJECT or RECOMPILATIONS operand. By default, it outputs all package or main program names and the file names in which the source programs are contained. In addition, the following package references may be listed:

(a) For a compilation unit all packages that are directly and indirectly imported by X can be output. The directly imported packages are specified in the WITH clause of X. These, in turn, can import packages which are thus indirectly imported by X. The output furthermore discriminates between packages imported by the specification and packages imported by the body.

(b) For each package X all compilation units can be established that depend on this package X, i.e. which imported package X directly or indirectly. These relations are required to determine the necessary recompilations. Following a modification of the specification of package X, all compilation units that depend on X have to be recompiled. This ensures the consistency of a program and restricts the number of compilations to those compilation units that were affected by the modification.

Following modifications or recompilations of compilation units all packages can be identified that have to be recompiled. The SHOW RECOMPILATIONS statement generates COMPILE statements so that the compilations are performed in the right order.

3.3 Hints on working with the project directory

The project directory is shareable, i.e. several users can access it simultaneously in read and write mode. If several users simultaneously access the same record of the project directory in write mode, these write access operations are executed in succession.

In project development, the following approach has proven useful:

- Create for each project a separate project directory, in which all package specifications belonging to the project are entered.
- Maintain package specifications separately from the package bodies.
- Protect package specifications against unauthorized modification: only permit read access or set a write password.
- "Freeze" the interfaces (package specifications) after design. Make any necessary changes only at specific times and inform all individuals involved.
- Following changes to specifications, recompile all compilation units that directly or indirectly import the changed specifications.
- Make all object modules of tested package bodies accessible in object module libraries.
- When modifying a package body a developer should work on a separate copy of the source. For testing, the object modules of the modified package are linked together with the other object modules belonging to the program from the central object module libraries.

Example: Working with the project directory

In this example the four packages A, B, C and D illustrate the use of the project directory. The specification of B imports package A, the specification of C imports package B and the body of C imports package D.

In Fig. 3-1 the relations between the packages are indicated by arrows, an arrow always pointing in the direction of the imported specification. In order to distinguish between different compilation units, the package names are followed by the suffix ".SPEC" (for specification) and ".BODY".

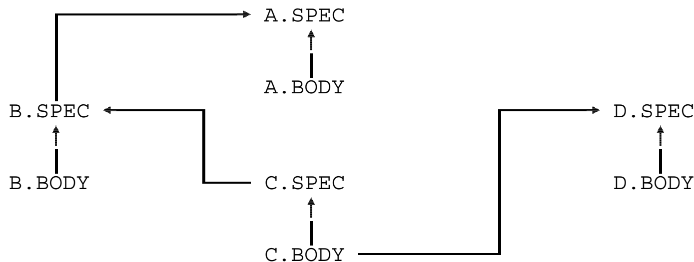

Fig. 3-1 Relations between packages

The source files of the package specifications are assumed to be stored in PLAM library PLAM.SPEC; the package bodies, in PLAM.BODY.

```
/EXEC $USERID.PASCAL-XT
%  BLS0500 PROGRAM 'PASCALXT', VERSION '22A00' OF ... LOADED<
%  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG.1990. ALL RIGHT
S RESERVED
//D TEST.DIRECTORY ——————————————————————————————————————————————— (01)
   >>> PROJECT DIRECTORY FILE CREATED
//MC (PLAM.SPEC,),*D,CH=ON,IN=ON,OP=ON
//C (,A.SPEC)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//C (,B.SPEC)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//C (,C.SPEC)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//C (,D.SPEC)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//MC (PLAM.BODY)
//C (,A.BODY)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//C (,B.BODY)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//C (,C.BODY)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//C (,D.BODY)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
```

(01) In order to generate a project directory for this programming system, two steps are necessary:
- first define a project directory with the DEFINE statement
- then compile all packages successfully with the COMPILE statement.

```
//S P ──────────────────────────────────────────────────────────── (02)

     CONTENTS OF THE PROJECT DIRECTORY FILE   (TEST.DIRECTORY)

PACKAGE NAME    KIND  LOCATION
A              (SPEC) ($USERID.PLAM.SPEC,A.SPEC(*UPPER-LIMIT,S))
A              (BODY) ($USERID.PLAM.BODY,A.BODY(*UPPER-LIMIT,S))
B              (SPEC) ($USERID.PLAM.SPEC,B.SPEC(*UPPER-LIMIT,S))
B              (BODY) ($USERID.PLAM.BODY,B.BODY(*UPPER-LIMIT,S))
C              (SPEC) ($USERID.PLAM.SPEC,C.SPEC(*UPPER-LIMIT,S))
C              (BODY) ($USERID.PLAM.BODY,C.BODY(*UPPER-LIMIT,S))
D              (SPEC) ($USERID.PLAM.SPEC,D.SPEC(*UPPER-LIMIT,S))
D              (BODY) ($USERID.PLAM.BODY,D.BODY(*UPPER-LIMIT,S))
//S P (,,A,A) ──────────────────────────────────────────────────── (03)

     CONTENTS OF THE PROJECT DIRECTORY FILE   (TEST.DIRECTORY)

PACKAGE NAME    KIND  LOCATION

A              (SPEC) ($USERID.PLAM.SPEC,A.SPEC(*UPPER-LIMIT,S))
    REFERENCES
      direct:   - -
      indirect: - -
    USED BY
      direct:   A (OWN BODY), B (SPEC)
      indirect: B (BODY), C (SPEC), C (BODY)

A              (BODY) ($USERID.PLAM.BODY,A.BODY(*UPPER-LIMIT,S))
    REFERENCES
      direct:   A (OWN SPEC)
      indirect: - -

B              (SPEC) ($USERID.PLAM.SPEC,B.SPEC(*UPPER-LIMIT,S))
    REFERENCES
      direct:   A
      indirect: - -
    USED BY
      direct:   B (OWN BODY), C (SPEC)
      indirect: C (BODY)

B              (BODY) ($USERID.PLAM.BODY,B.BODY(*UPPER-LIMIT,S))
    REFERENCES
      direct:   B (OWN SPEC)
      indirect: A
```

```
C               (SPEC) ($USERID.PLAM.SPEC,C.SPEC(*UPPER-LIMIT,S))
    REFERENCES
      direct:   B
      indirect: A
    USED BY
      direct:   C (OWN BODY)
      indirect: - -

C               (BODY) ($USERID.PLAM.BODY,C.BODY(*UPPER-LIMIT,S))
    REFERENCES
      direct:   C (OWN SPEC), D
      indirect: A, B

D               (SPEC) ($USERID.PLAM.SPEC,D.SPEC(*UPPER-LIMIT,S))
    REFERENCES
      direct:   - -
      indirect: - -
    USED BY
      direct:   C (BODY), D (OWN BODY)
      indirect: - -

D               (BODY) ($USERID.PLAM.BODY,D.BODY(*UPPER-LIMIT,S))
    REFERENCES
      direct:   D (OWN SPEC)
      indirect: - -
```

(02) Now, the SHOW statement can be used to output the contents of the project directory. Standard output consists of the package name, an entry showing whether it is a package specification (SPEC), a package body (BODY) or a main program (PROG), and the file name or, in the case of library elements, the name of the element, version and type of element. This statement when written in full has this format:

```
SHOW-ATTRIBUTES WHAT = PROJECT-FILE
```

(03) In addition to standard output, the SHOW statement now lists the relations between the compilation units. Under the heading REFERENCES the imported packages are listed. Directly imported packages are those that have been specified directly in the WITH clause; in the case of package bodies this also includes packages of the user's own specification. Indirectly imported are those packages that are imported by the specifications of the directly imported packages (even if imported through several stages). Under the USED-BY relation, those compilation units are listed that import this package. For "direct" and "indirect" the above definitions equally apply. The format of this statement when written in full is:

```
SHOW-ATTRIBUTES WHAT = PROJECT-FILE (USED-BY = ALL,
                                     REFERENCES = ALL)
```

```
//E A(S) ——————————————————————————————————————————————————————————— (04)
   >>> (OVER)WRITE PACKAGE SPEC "A" (y/n) ?
*y
   >>> PACKAGE SPEC "A" (OVER)WRITTEN
//S P —————————————————————————————————————————————————————————————— (05)

     CONTENTS OF THE PROJECT DIRECTORY FILE   (TEST.DIRECTORY)

PACKAGE NAME    KIND  LOCATION
A             !(SPEC) ($USERID.PLAM.SPEC,A.SPEC(*UPPER-LIMIT,S))
A             !(BODY) ($USERID.PLAM.BODY,A.BODY(*UPPER-LIMIT,S))
B             !(SPEC) ($USERID.PLAM.SPEC,B.SPEC(*UPPER-LIMIT,S))
B             !(BODY) ($USERID.PLAM.BODY,B.BODY(*UPPER-LIMIT,S))
C             !(SPEC) ($USERID.PLAM.SPEC,C.SPEC(*UPPER-LIMIT,S))
C             !(BODY) ($USERID.PLAM.BODY,C.BODY(*UPPER-LIMIT,S))
D              (SPEC) ($USERID.PLAM.SPEC,D.SPEC(*UPPER-LIMIT,S))
D              (BODY) ($USERID.PLAM.BODY,D.BODY(*UPPER-LIMIT,S))

//S R —————————————————————————————————————————————————————————————— (06)
COMPILE A (SPEC)
COMPILE A (BODY)
COMPILE B (SPEC)
COMPILE B (BODY)
COMPILE C (SPEC)
COMPILE C (BODY)
```

(04) The editor is used to modify and rewrite specification A. In the project directory the specification of package A is marked as changed.

(05) All those compilation units that have to be recompiled because of a change of specification A are marked with an exclamation mark in the output of the project directory. These are those compilation units that import package A directly or indirectly.

(06) The SHOW statement is used to generate the required COMPILE statements for these marked compilation units. The format of the statement when written in full is:

```
SHOW-ATTRIBUTES WHAT = RECOMPILATIONS
```

```
//S R,*E(8) ──────────────────────────────────────────────────────── (07)
//CA *E(8) ───────────────────────────────────────────────────────── (08)
(%STMT) COMPILE A (SPEC)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
(%STMT) COMPILE A (BODY)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
(%STMT) COMPILE B (SPEC)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
(%STMT) COMPILE B (BODY)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
(%STMT) COMPILE C (SPEC)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
(%STMT) COMPILE C (BODY)
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;  NOTES:  0)
//S P ────────────────────────────────────────────────────────────── (09)

     CONTENTS OF THE PROJECT DIRECTORY FILE   (TEST.DIRECTORY)

PACKAGE NAME    KIND  LOCATION
A              (SPEC) ($USERID.PLAM.SPEC,A.SPEC(*UPPER-LIMIT,S))
A              (BODY) ($USERID.PLAM.BODY,A.BODY(*UPPER-LIMIT,S))
B              (SPEC) ($USERID.PLAM.SPEC,B.SPEC(*UPPER-LIMIT,S))
B              (BODY) ($USERID.PLAM.BODY,B.BODY(*UPPER-LIMIT,S))
C              (SPEC) ($USERID.PLAM.SPEC,C.SPEC(*UPPER-LIMIT,S))
C              (BODY) ($USERID.PLAM.BODY,C.BODY(*UPPER-LIMIT,S))
D              (SPEC) ($USERID.PLAM.SPEC,D.SPEC(*UPPER-LIMIT,S))
D              (BODY) ($USERID.PLAM.BODY,D.BODY(*UPPER-LIMIT,S))

//REM B ──────────────────────────────────────────────────────────── (10)
```

(07) For the generated COMPILE statements to be executed with the aid of a CALL-STATEMENT-FILE statement, they are output to EDT work area 8. The format of this statement when written in full is:

```
SHOW-ATTRIBUTES WHAT = RECOMPILATIONS,
             OUTFILE = *EDT (WORKFILE = 8)
```

(08) The statements in EDT work area 8 are executed with the aid of the CALL-STATEMENT-FILE statement. The format of this statement when written in full is:

```
CALL-STATEMENT-FILE STMT-FILE = *EDT (WORKFILE = 8)
```

(09) No other compilations are necessary as there no marked compilation units any more.

(10) The REMOVE statement is used to delete all entries for package name B; this affects both the entry for the specification and the entry for the body. The format of this statement when written in full is:

```
REMOVE-DIRECTORY-ENTRY UNIT = B
```

```
//S P ──────────────────────────────────────────────────────────────────── (11)

     CONTENTS OF THE PROJECT DIRECTORY FILE   (TEST.DIRECTORY)

PACKAGE NAME    KIND  LOCATION
A              (SPEC) ($USERID.PLAM.SPEC,A.SPEC(*UPPER-LIMIT,S))
A              (BODY) ($USERID.PLAM.BODY,A.BODY(*UPPER-LIMIT,S))
C             !(SPEC) ($USERID.PLAM.SPEC,C.SPEC(*UPPER-LIMIT,S))
C             !(BODY) ($USERID.PLAM.BODY,C.BODY(*UPPER-LIMIT,S))
D              (SPEC) ($USERID.PLAM.SPEC,D.SPEC(*UPPER-LIMIT,S))
D              (BODY) ($USERID.PLAM.BODY,D.BODY(*UPPER-LIMIT,S))

//END
   END OF THE PASCAL SESSION  -  USED TIME = 2.965 SECONDS
%  EXC0732 ABNORMAL PROGRAM TERMINATION. ERROR CODE 'NRT0101':
   /HELP NRT0101,INF=D
```

(11) The specification and the body of package C must be modified and recompiled because they have directly or indirectly imported specification B, which no longer exists.

4 Pascal-XT compiler

4.1 Using the project directory

The compiler requires the project directory in order to find specifications of imported packages, to check the interfaces and to store the package and file names for specifications. Additional project directory functions (see chapter 3) merely serve to facilitate package usage.

When must a project directory be defined?

If a compilation unit that imports packages is to be compiled, the user must define a project directory. The compiler requires the project directory in order to find the specifications of the imported packages. If no project directory has been defined, the package identifiers in the WITH clause appear in the compiler listing with the error:

```
PACKAGE SPECIFICATION NOT FOUND
```

What entries are generated in the project directory?

When a compilation unit has been successfully compiled, the compiler stores the package or program name, the source file name and other status information in the project directory. If a package specification is to be compiled and no project directory has been defined, the following message is issued:

```
NO PROJECT DIRECTORY IS DEFINED
```

When the compilation unit is read from an EDT work area, no entry is generated in the project directory.

In what sequence are the compilation units compiled?

Package specifications can only be used by other compilation units after they have been compiled successfully (this is in line with the Pascal rule saying that identifiers can only be used after they have been declared). The WITH clauses set up a static relationship between the packages, through which a compilation sequence for the package specifications is automatically defined. Note that the specifications of the imported packages must always be compiled first. No compilation sequence is stipulated for package bodies.

4.2 Implementation-defined attributes

This section defines the implementation-defined attributes listed in the Pascal-XT Language Reference Manual ([1], appendix A.6) and describes the new, high-precision mathematical routines available as of V2.2A. The numbering is the same as in the Language Reference Manual.

1) Values of predefined real constants:

```
Long_Minreal  =   5.397605346934027E-79
Long_Maxreal  =   7.237005577332262E+75
Short_Minreal =   5.397605E-79
Short_Maxreal =   7.237005E+75
```

2) Presettings of the predefined constants:

```
Maxint  = Long_Maxint   ( = 2^31 - 1)
Minint  = Long_Minint   ( = - 2^31)
Minreal = Long_Minreal
Maxreal = Long_Maxreal
```

3) Predefined types:

```
Integer  =  Long_Integer
Real     =  Long_Real
```

4) Values of type Short_Real are represented as 32-bit floating-point numbers, values of type Long_Real as 64-bit floating-point numbers as described in the Assembler Instructions manual [12]. The absolute value range W for both real types is:

```
-Maxreal <= W <= Maxreal
```

5) The precision for real operations and real functions is defined by the representation [12] and rounding, in case of intermediate results. The number of significant decimal positions is for values of type

```
Long_Real:   about 16
Short_Real:  about 6
```

Exponent underflow is not recognized, exponent overflow with arithmetic operations is generated as a Numeric_Error and, when reading in a real number, as a Read_Error.

High-precision mathematical routines:

Pascal-XT provides the following arithmetical functions (see the Pascal-XT Language Reference Manual [1], section 15.4):

```
Arctan(x)                Ln(x)
Cos(x)                   Sin(x)
Exp(x)                   Sqrt(x)
```

As of version 2.2A they use high-precision mathematical routines which perform their calculations with so-called ulp-precision algorithms (ulp = *u*nit of *l*ast *p*lace). Maximum precision is guaranteed with these routines, i.e. there is no

representable floating-point number between the calculated function result and the exact result. Despite this considerable increase in precision, the new routines offer the same high level of performance as the old ones.

As these new routines may deliver different function values to those obtained previously, unexpected program behavior can occur if, for example, programs created previously use the function values in comparisons.

6) The values of type Char are the elements of the EBCDIC character set SN77315 (see [5]). The ordinals of the character values are also specified in the table.

7) Type string without type parameter has a maximum length of 254.

8) The size of a storage unit is one byte (8 bits).

9) The restrictions for the offset and bit range specifications in a field identifier are described in section 4.3.

10) The base type of a set can contain between 2041 and 2048 elements at most (see section 4.3).

11) For non-qualified set constructors the ordinal values of the elements must lie in the range 0 .. 2047.

12) As of Pascal-XT V2.2A, the "C" directive is also supported. However, it is recommended that only the "EXTERNAL" directive should be used for non-Pascal subprograms (see section 7.2).

13) Not all Pascal-XT parameter types are permitted for subprograms with a directive. The relevant restrictions are described in sections 7.2 to 7.4.

14) The file operations executed for Rewrite, Put, Reset and Get are described in section 5.4.

15) The description of an external file in the predefined procedure Assignfile and the effect of this procedure are dealt with in section 5.3.3.

16) The use of Reset or Rewrite on the predefined text files Input or Output results in an Open_Error with system error code 1607.

17) Default output length:

```
Integer values: 12
Real values   : 22
Boolean values:  5.
```

The boolean value TRUE is output right-justified in the output field.

18) If Real values are output in floating-point representation,

- ’E’ is output as the exponent character and
- a sign and two decimal digits are output for the exponent.

19) Boolean values are output in upper case (TRUE and FALSE).

20) The predefined procedure Page only generates a page control character (character ’A’ output in the first column) in a text file if the attribute SPACE=E was specified (see section 5.3.3) in the file assignment with the predefined procedure Assignfile. For the BS2000 system file SYSLST the same procedure is followed as conventionally for SPACE=E. Calling Page several times in succession results in only one page throw being effected. Calling Page has no effect if no other text is output after Page. Page has no effect if system file SYSOUT has been assigned to the Pascal file.

21) The restrictions for entry subprograms are given in section 7.3.

22) The assignment of external files to program parameters is discussed in section 5.3.

23) The preset compiler options are described in section 4.5. When the COMPILE statement is called the LINES-PER-PAGE option (see section 2.6.4) can be specified as well.

Compiler limitations and system dependencies

- In a source program the line length is limited to 254 characters. Where longer lines are encountered an error is reported.
- Identifiers can have a maximum length of 254 characters.
- A maximum of 50 levels of record nesting is permitted.
- A maximum of 50 WITH statements may be nested.

4.3 Representation of objects in main memory

Memory requirements and alignment for objects of simple types and pointer types are shown in Table 4-1. For structured types specification of these items is somewhat more complex and can therefore be discussed only briefly.

The predefined functions SIZEOF and ALIGNOF supply information about memory requirements and the alignment of values for a type.

Data type T	Value range	Storage req'd SIZEOF (T) (in bytes)	Alignment ALIGNOF (T) (in bytes)
BOOLEAN	FALSE..TRUE	1	1
CHAR	CHR(0)..CHR(255)	1	1
SHORT_INTEGER	SHORT_MININT..SHORT_MAXINT	2	2
LONG_INTEGER	LONG_MININT..LONG_MAXINT	4	4
INTEGER	LONG_MININT..LONG_MAXINT	4	4
Enumerated	Up to 256 elements	1	1
Enumerated	Up to 32768 elements	2	2
Enumerated	Beyond	4	4
Subrange	0 .. 255	1	1
Subrange	SHORT_MININT..SHORT_MAXINT	2	2
Subrange	LONG_MININT..LONG_MAXINT	4	4
LONG_REAL		8	8
SHORT_REAL		4	4
REAL		8	8
Pointer		4	4

Table 4-1 Memory requirements and alignment of simple types and pointer types

Packed representation of structured types

No distinction is generally made between packed and unpacked representation where objects of structured types are concerned. Record types represent an exception to this rule (see below).

Set types

The maximum number of elements of a set is restricted to 2048. If the lower set bound of the base type of the set is not equal to zero, the maximum number may, at worst, be restricted to 2041.
The storage required for a set (in bytes) is approximately the number of elements divided by 8.
Alignment is at a byte boundary.

String types

The storage required equals the maximum number of characters plus one half word for the length field.
Alignment is always on half word boundary, even within packed record types.
The storage required for the default string length is 256 bytes.

Array types

The storage required is calculated from the sum of the element sizes and any gaps that may appear.
Alignment is the same as the alignment of the element type.

Record types

The storage required is calculated from the storage required for the fields of the fixed part, the fields of the greatest variant of the variant part and any gaps that may appear (depending on the alignment requirements). The field with the greatest alignment determines the alignment of the entire RECORD type.

For packed record types scalar and pointer type fields are aligned on byte boundaries. Fields regarded as belonging to a structured type are aligned in accordance with the alignment requirements of the appropriate type (which in turn can be influenced by the 'packed' specification).

Variants of a variant part are allocated the same storage location for the sake of storage optimization. Where a variant contains a field of a FILE type, all variants are allocated in succession. As a result, fast creation on the heap by means of the New procedure is no longer possible.

If a RECORD type contains fields of a FILE type, then no specifications regarding representation in storage can be given in the RECORD type.

If no specifications regarding representation in storage are given for fields, the compiler may allocate the fields in a different order from that specified in the text in order to make better use of available storage.

For offset specifications in a field the following additional restrictions apply:

- The offset of a field of a string type must always be a multiple of 2 (a string type is a packed type).
- The offset of a field of type Long_Real (and, hence, also Real) must always be a multiple of 8.
- The offset of a field of type Short_Real must always be a multiple of 4.
- In an unpacked RECORD type the offset of a field must be an integer multiple of the alignment of its type.
- In a packed RECORD type any offset may be specified for fields of an ordinal type or pointer type. The offset of a field of a structured type must again be a multiple of the alignment of this type.
- Storage areas for fields in different variants of a variant part may overlap.

For bit range specifications in a field the following additional restrictions apply:

- Bit ranges can only be specified for fields of an ordinal type.
- The values of the bit range boundaries must lie in the range 0..31.

Note

If variables of a RECORD type are used as interfaces with external programs, storage representations should be specified for all components. The compiler then guarantees adherence to the desired representation provided that it does not contravene any of the above requirements.

File types

The storage required is calculated from the size of the component type and the size of a fixed management block. For text files the line buffer is created on the heap. Alignment is at a doubleword boundary.

4.4 Generated object modules

The Pascal-XT compiler generates object modules for a compilation unit if the following conditions are satisfied:

- The compilation unit is a main program or a package body.
- The compilation was successful.
- The compiler option "Generate" was activated.

In accordance with the specification in the MODULE operand of the COMPILE statement, the modules generated are stored in the temporary object module file *OMF or in a PLAM library. The following modules are generated:

Code module
: The code module contains the constants and the object code for the compilation unit. It is reentrant and shareable.

Data module
: The data module contains the variables of the compilation unit. This module is not shareable.

Test tables module
: Contains information for the PATH debugging aid. This module is reentrant and shareable.

Starter module
: This module is generated only for a Pascal main program. It contains the start address of the program and is not shareable.

XS capability

The compiler generated object modules and the runtime system modules support the XS capability. AMODE=ANY and RMODE=ANY is true for every object module. A Pascal-XT program can therefore run anywhere in the 31-bit address space (see also chapter 6).

Naming the object modules

The names of the generated object modules are generated as follows:

characters 1-7	Taken from the program or package name. In the case of the starter module, shorter names are padded with blanks; with the code, data and test table modules they are padded with number signs ("#"). Underscore characters ("_") are converted to number signs ("#").
character 8	Used to distinguish the modules. The meanings are: "C" for code modules "D" for data modules "T" for test table modules " " for starter modules

Naming the procedures

The names of external procedures and entry procedures are outwardly visible. They are a maximum of 8 characters in length. Longer names are truncated, shorter names are padded with blanks. Underscore characters ("_") are converted to number signs ("#").

Note

In other languages underscores in names are generally not permitted. Therefore, names of external and entry procedures should not contain any underscores in the first 8 characters.

4.5 Compiler options

The compiler options (control statements for the compiler) defined in the Pascal-XT Language Reference Manual [1] apply to all Pascal-XT implementations. They can all be specified as operands in the source program and, with the exception of PAGE and TITLE, also in the COMPILE statement. PAGE and TITLE can only be specified in the source program.

In a Pascal-XT source program, compiler options are given as pseudo-comments as in the example below.

Example

```
{$Check = On, Initialize = On, List = Off}
```

There are also the LINES-PER-PAGE and MESSAGE-LEVEL options, which can only be specified in the COMPILE statement (see 2.6.4). LINES-PER-PAGE defines the number of lines per page in the compiler listing; MESSAGE-LEVEL controls the nature and scope of compiler messages.

If an option is specified as an operand in the COMPILE statement, that setting applies to the entire compilation unit and overrides the default setting or the setting specified in the source program. Hence, the setting valid for an option is determined by where it was specified. The following priorities apply to the input of options:

```
option in COMPILE statement    |
option in source program       |    decreasing priority
default setting                v
```

The following table shows all options with their permissible values in alphabetical order. The default values are underlined.

Option	Permissible values	Meaning	May be specified in source	May be specified in COMPILE stmt.
Assembler	On Off	Output of an object code listing in assembly language format	x	x
Check	On Off	Generate code for runtime checks	x	x
Debug	On Restricted Off	Generate test information for the symbolic debugging aid PATH	x	x
Generate	On Off	Generate object code	x	x
Initialize	On Off	Storage areas for variables are filled with hexadecimal value 88	x	x
Lines-per-page	11 through 2147483639 63	Define number of lines per page in the compiler listing		x
List	On Off	Activate/deactivate compiler listing	x	x
Map	On Off	Generate address tables	x	x
Message-level	Errors Warnings Notes	Define nature and scope of compiler messages in the compiler listing		x
Optimize	On Off	Generate optimized object code	x	x
Page		Generate a page throw	x	
Standard	On L0 Off	On: Standard Pascal level 1 accepted L0: Standard Pascal level 0 accepted Off: Pascal-XT accepted	x x	x x x
Title	String '' ___	Specification of a title in the header of the compiler listing	x	
Xref	On Off	Generate a cross-reference listing	x	x

Table 4-2 Pascal-XT compiler options under BS2000

4.6 Listings generated by the compiler

During compilation, the Pascal-XT compiler generates a series of listings containing information on the structure of the source program, the generated object modules and the flow of compilation. Examples of the different listings are given in appendix A.2.

All listings have a uniform heading consisting of

- Identification of the particular listing
- Specification of the operating system and of the compiler
- Version and date of the Pascal-XT compiler
- Date and time of the start of the compilation
- Consecutive page numbering

The heading is output at the beginning of each page. As of version 2.2A, the date in the compiler listing header will show the year as four digits.

4.6.1 Controlling listing output

All listings generated by the compiler are output to an output file. This file is defined by means of the LISTING operand in the COMPILE statement (see section 2.6.4).

The generation of the various listings is governed by the compiler options Assembler, List, Map and Xref. The meaning of the options is described in section 4.5.

The number of lines per page in the compiler listing defaults to 63 but can be altered using the LINES-PER-PAGE option.

4.6.2 Source listing

The source listing is divided into the sections prolog, source program and compilation summary. The output of this listing is governed by the List option. By default, a source listing is generated (List=On). If List=Off only the prolog and the compilation summary are output.

Prolog

The prolog summarizes the global information for the compilation unit.

GLOBAL OPTIONS FOR THIS COMPILATION
Gives the values of the options that apply globally for the entire compilation unit. The Debug option deactivates the Optimize option if both are specified either in the source or in the COMPILE statement. Following the options information is given as to where they were set.

BY COMMAND	in the COMPILE statement
IN SOURCE	within the source program
BY DEFAULT	default value (see section 4.5)
BY DEBUG OPTION	the Optimize option was deactivated by the Debug option
BY OPTIMIZE OPTION	the Debug option was deactivated by the Optimize option (through specification in the COMPILE statement)

LIST OF RECOMPILED PACKAGE SPECIFICATIONS (SOURCE FILES)
The files are specified from which the directly and indirectly imported package specifications are read. Indirectly imported specifications do not appear in the context specification of the compilation unit, but are imported by at least one of the specifications specified there.
This listing is output only if package specifications are imported in the compilation unit.

CURRENT COMPILATION UNIT (SOURCE FILE)
Specifies the file from which the compilation unit is read.

Source program

Following the prolog the source program is output, with continuous line numbering, starting with 1. The line numbers in the compiler listing match the line numbering in the source program, even if within the source listing output is temporarily deactivated.
If a comment in the source program extends over several lines, all continuation lines of the comment are marked with the character "C" appearing immediately after the line number.

Errors, warnings and notes in the source program listing give rise to additional message lines but these do not affect the line numbering. The format of the message lines is described in section 4.6.3.

Compilation Summary

In the compilation summary the results of the compilation are summarized. The number of messages suppressed with the compiler option MESSAGE-LEVEL is also given.

ERRORS DETECTED
: Gives the number of syntax and semantic errors.

WARNINGS
: Gives the number of warnings.

NOTES
: Gives the number of notes.

SIZE OF CODE MODULE
: Storage required for the generated code module, in bytes. The value 0 is output if Generate=Off, if compilation errors were reported, or if the compilation unit was a package specification.

SIZE OF DATA MODULE
: Storage required for the generated data module. The value 0 is output if Generate=Off, if compilation errors were reported, or if the compilation unit was a package specification.

COMPILATION TIME
: Gives the CPU time required for the compilation and the generation of the various listings. If specifications are referenced in the compilation, the time to reread those specifications is likewise added to the compile time.

4.6.3 Errors, warnings and notes

In the course of the compilation process the compiler may output messages indicating errors, warnings or notes. These messages appear in the compiler listing.

Errors designate syntax and semantic errors in the compilation unit.

Warnings are output at those points where the compiler recognizes that an error will occur when the program is run. For example, the compiler recognizes that the statement

```
x := long_maxint + 1
```

will result in an overflow condition when the addition is effected and therefore issues a WARNING at this point. Despite the presence of WARNINGS an executable object program is generated.

Notes report errors in pseudo-comments and identifiers which have been declared but not used. Despite the presence of NOTES an executable object program is generated. Errors in pseudo-comments result in compiler options not having the desired effect. Identifiers which are declared but not used are reported in accordance with the following rules:

- All identifiers declared in the program but not used are marked. This applies to identifiers for constants, types, variables, functions, procedures and formal parameters. Excluded from this list are field identifiers (components of record types) and constant identifiers of enumerated types.
- The identifiers in the WITH and USE clauses in the main compilation unit or the associated package specification are reported if they are not used in the program text. This does not apply when the main compilation unit is a package specification.
- Function identifiers are marked if the function is not called in the program text. The assignment to a function identifier within the function block is not regarded as use of the function identifier.
- The identifier of an enumerated type is considered as used if one of the enumerated constants of this type is used.
- No NOTES are output with respect to identifiers declared in a package specification or with respect to formal parameters in external subprograms.

The total number of errors, warnings and notes together with the number of suppressed messages (compiler option MESSAGE-LEVEL, see 2.6.4) is output in the COMPILATION SUMMARY and written to the system file SYSOUT.

Where the List option is switched off (List=Off), only the prolog, the message lines together with the associated source program lines, and the compilation summary are output.

The columns of the source program line to which the messages refer are marked by various digits in a separate line and output immediately after the source program line. To highlight it in the source listing, this line is filled with "-" characters. Subsequently, one or more self-explanatory error message texts are output for each digit.

4.6.4 Assembler listing

In the assembler listing the code generated is output in assembly language format. Controlled by the Assembler option, the listing is output after the compiler listing. The assembler listing can be generated for the entire compilation unit or for individual subprograms. In the latter case, the option must be activated within the subprogram block and deactivated in the following lines. For procedures and functions declared "inline", no assembly listing is generated, since for them code is generated only at the calling point.

The format of the assembler listing corresponds essentially to the format as generated by the assembler.

Field	Meaning
LOCTN	Relative address in the module (hexadecimal, in bytes)
OBJECT CODE	Object code in hexadecimal form
ADDR1 ADDR2	Addresses used in the instruction. The addresses are calculated from the contents of the base register and the displacements. They give the relative address in the module, i.e. correspond to the entries in the LOCTN column.
STMNT	Consecutive numbering in the listing
ASSEMBLY CODE	Assembly code generated

In addition, comment lines are inserted into the assembly code. These give the number of the source line for which the subsequent instructions were generated.

4.6.5 Cross-reference listing

The cross-reference listing contains all user-defined identifiers used in the compilation unit, along with all predefined identifiers used in the compilation unit, in alphabetical order. Identifiers written in the same way but from different scopes appear in the cross-reference listing once for each defining point. For each identifier there is a description line, followed by a list of line numbers in which that identifier appears.

Generation of the cross-reference listing is governed by the Xref option. The listing is output following the compiler listing.

The description line consists of four fields. Interpretation of the values of fields 2 and 3 depends on field 1. The fields mean:

field 1	class of the identifier
field 2	type of the identifier
field 3	class-specific attribute
field 4	defining point of the identifier

Classes of identifiers

CONSTANT	constant identifier
FIELD	field identifier in a record type
FUNCTION	function identifier
FUNC-PARAM	identifier of a formal function
PACKAGE	package identifier
PROCEDURE	procedure identifier
PROC-PARAM	identifier of a formal procedure
TYPE	type identifier
VAL-PARAM	identifier of a formal value parameter
VAR-PARAM	identifier of a formal variable parameter
VARIABLE	variable identifier

Type of the identifier

This column specifies the type of the identifier. For identifiers of the classes PACKAGE and PROCEDURE this specification is omitted. For identifiers of the classes FUNCTION and FUNC-PARAM the type specification designates the type of the function result.

Class-specific attribute

This field contains a value whose interpretation depends on the class of the identifier.

Class	Meaning of field 3
CONSTANT	For ordinal constants their value is output, otherwise this is omitted
FIELD	Specifies the relative offset from the start of the record, in bytes
FUNCTION PROCEDURE	} Specifies the line number of the first statement in the block of the function or procedure
TYPE	Storage required by the type, in bytes
other	Omitted

Defining point of the identifier

This column specifies the number of the line in which the identifier has been declared. If the identifier has been declared in a foreign package (or the current package specification), then in addition to the line number the name of the package is specified. If a predefined identifier is involved, "PREDEFINED" is output.

4.6.6 Map listing

The map listing provides information on the contents of the generated object modules of the compilation unit. Output of this listing is controlled by the Map option.

Addresses of procedures and functions

For the procedures and functions of the compilation unit, the following is output (from left to right):

entry address	contains the starting address of the procedure (or function)
start address	of the procedure code, relative to the beginning of the code module
name	of the procedure or function

Global constants

All structured constants of the compilation unit are output. The items mean (from left to right):

relative address	of the constant in the code module. The information is given in decimal and hexadecimal form.
type	of the constant
identifier	for declared constants the constant identifier, for non-declared constants the empty string is output.
value	of the constant. With sets only the values of the smallest and largest set elements are given.

Global variables

The variables declared as global in the compilation unit are output. The addresses given (in decimal and hexadecimal form) are relative addresses in the generated data module.

5 Files

5.1 Pascal files

5.1.1 External Pascal files

External Pascal files are file variables which, by specifying their variable names in the program parameter list, are signed on as external, program-independent files. These file variables must be assigned to BS2000 files when the program is run.

An exception is represented by the predefined text files Input and Output to which, by default, system files SYSDTA and SYSOUT respectively are assigned.

Section 5.2 describes BS2000 files that can be assigned to the external files. Section 5.3 describes how the assignments may be made.

5.1.2 Local Pascal files

Local Pascal files are file variables or components of variables that do not appear as program parameters. The life span of a local file corresponds to the life span of the block in which the file (file variable) has been declared. After a block is left, its local files can no longer be accessed. Local files in dynamic variables exist as long as pointers to the variables exist or the dynamic variable is destroyed with the predefined procedure Dispose. When a procedure (function) with a local file is called recursively, a new local file is created for each incarnation of the call.

5.2 Supported BS2000 files and libraries

The Pascal-XT system supports access to the following BS2000 files and libraries:

- Standard files
- Sequential (SAM) and indexed sequential (ISAM) files
- PLAM library elements
- Temporary files

Section 5.3 deals with the various options for assigning BS2000 files to Pascal files.

Open modes for files

In addition to the predefined subprograms Reset and Rewrite the procedures Extend and Replace are supported in the predefined package DMSIO (A.5). Sections 5.2.1 through 5.2.4 deal with the admissible procedures (open modes) that apply.

Reset
: The file or library element is opened for reading. If the file, element or library does not exist, a runtime error OPEN_ERROR is generated.

Rewrite
: The file or library element is opened for writing. Any existing element will be overwritten. If the element does not exist it will be created.

Extend
: This procedure is defined in the predefined package DMSIO (A.5). It has the same effect as Rewrite, except that the contents of any existing file will not be lost.

Replace
: This procedure is defined in the predefined package DMSIO (A.5); it opens a file for reading and writing.

Permitted open modes for the different files

Tables 5-1 and 5-2 both show possible combinations of files and open modes. "ok" identifies a valid combination, while for invalid combinations the error message reported and the associated system error code (system_code) are shown.

If the Pascal file variable f belongs to type "Text", then the following rules apply:

	Reset(f)	Rewrite(f)	Replace(f)	Extend(f)
*SYSDTA	ok	Open_Error (1605)	Open_Error (1605)	Open_Error (1605)
*SYSLST *SYSOUT	Open_Error (1605)	ok	Open_Error (1605)	ok
*DUMMY *EDT SAM file ISAM file	ok	ok	Open_Error (1605)	ok
PLAM element EAM file	ok	ok	Open_Error (1605)	Open_Error (1605)

Table 5-1: Permitted open modes for text files

If the Pascal file variable f belongs to type "FILE OF t", then the following rules apply:

	Reset(f)	Rewrite(f)	Replace(f)	Extend(f)
*DUMMY SAM file ISAM file	ok	ok	ok	ok
PLAM element EAM file	ok	ok	Open_Error (1605)	Open_Error (1605)

Table 5-2: Permitted open modes for non-text files

5.2.1 Standard files

Standard files include BS2000 system files and the work areas of the editor EDT. To distinguish them from permissible file names in the data management system they must be prefixed by "*". A standard file can only be assigned to a Pascal text file.

*SYSDTA	corresponds to system file SYSDTA
*SYSOUT	corresponds to system file SYSOUT
*SYSLST	corresponds to system file SYSLST
*DUMMY	corresponds to system file *DUMMY. Data written to *DUMMY is lost; when *DUMMY is read, EOF is returned immediately.
*EDT(number)	designates a work area of the editor EDT. "number" specifies the number of the work area and must lie in the range 0...9. The specification of "(number)" is optional. By default, the current work area is assumed.

Open modes

- Reset
- Rewrite
- Extend

5.2.2 SAM and ISAM files

The Pascal-XT system can process all SAM files supported by BS2000. Sequential files may be both text files and non-text files.

When a Pascal file is opened for writing (Rewrite) the assigned BS2000 file is created as a sequential file with variable record length (FCBTYPE=SAM, RECFORM=V) as a standard procedure. For all other file attributes the BS2000 default values are assumed (see [6]). Any file attributes deviating from these values must be specified by means of the FILE command.

When a file is opened for reading (Reset) and for the open modes Extend and Replace (see appendix A.5) the file attributes specified in the catalog entry are taken over by the Pascal-XT runtime system. Any incompatibilities between the specifications in the catalog entry and the values preset by the program will be reported by the Pascal-XT runtime system (see the tables in section 5.3.2).

The Pascal-XT language itself does not support any indexed sequential (ISAM) files. However, a predefined package, DMSIO (see appendix A.5), is available which permits the processing of ISAM files. The ISAM file attributes must be defined with the FILE command (see section 5.3.2) prior to its generation. If an ISAM file has been assigned to a Pascal text file, no keys will be generated during writing and any keys that are encountered during reading will be ignored. The key of a record can be obtained by using the "movekey" procedure in the DMSIO package (see appendix A.5).

Keyed and keyless file formats

With effect from BS2000 V10 the so-called keyless disk peripheral is supported. From this version it is necessary to distinguish between

- the previous keyed format (also known as the K format) and
- the newly introduced keyless format (also known as the NK format)

The file format is controlled through the BLKCTRL operand in the FILE command. The PAM keys contained in the K format are moved to the data area in the NK format and thus reduce the area available for the data records. This has the effect of changing the minimum and optimum block size for a specific record size. The user must take this into account when specifying the block size in a FILE command. If he does not specify the block size, then the runtime system determines the minimum block size on the basis of the PAM key format when creating a new file.

Open modes

- Reset
- Rewrite
- Extend
- Replace

5.2.3 PLAM library elements

The Pascal-XT system permits the use of program libraries processed with the PLAM library access method. Such libraries, PLAM libraries for short, may contain elements of various types. For each element of a given type several versions may exist. A detailed description of program libraries can be found in the LMS manual [3].

PLAM library elements can be assigned to Pascal files only via the Assignfile (see section 5.3.3) standard procedure.

Open modes

- Reset
- Rewrite

Element designation

The specification of a library element is analogous to that in statements in the programming system. The syntax is:

plam-element = (library-name , element)

element = element-name [([version][,type])]

library-name
: Must be a valid file name in the data management system.

element-name
: Designates the name of the element in the library. The name may be up to 64 characters in length.

version
: Designates the element version. It may be up to 24 characters in length. The version identifiers *INCREMENT, *UPPER-LIMIT und *HIGHEST-EXISTING have a special position.

*INCREMENT in write mode, causes the version number of the library element to increase automatically. This specification is permissible as of PLAM version 2.0.

*UPPER-LIMIT in read and write modes, stands for the highest-possible version. This specification corresponds to the version designated to date as the @ version in LMS.

*HIGHEST-EXISTING
stands for the highest existing version. In read mode, this version is read. In write mode, if the element is present the highest version is overwritten, otherwise a default version is specified.

	The version specification *STD is still possible in Pascal-XT V2.2A. In read mode, this stands for the highest existing version (viewed lexicographically), so corresponds to the specification *HIGHEST-EXISTING. In write mode, it stands for the highest-possible version, so corresponds to the specification *UPPER-LIMIT. Default value: *HIGHEST-EXISTING
type	Designates the element type. It may be up to 8 characters in length. Elements of type C (binary files) are not supported. Default value: S (for Source)

Additional conventions for the identification of elements are not defined by PLAM. However, it is recommended that the conventions laid down in LMS [3] be adhered to.

5.2.4 Temporary files

Temporary files are implemented by EAM files. Following termination of the user program all EAM files are deleted.

Open modes

- Reset
- Rewrite

5.3 Assigning BS2000 files to Pascal files

While a program is being executed the Pascal files (FILE variables) declared in the program must be assigned operating system files. Sections 5.3.1 through 5.3.4 describe the different options for effecting such assignments.

5.3.1 Default assignments

Input and Output

Unless specified otherwise, the predefined text files Input and Output are assigned to the SYSDTA and SYSOUT system files, respectively. The predefined procedure Assignfile (see section 5.3.3), however, can be used to assign to them any files that are described in section 5.2 (but: SYSOUT cannot be assigned to Input, SYSDTA cannot be assigned to Output).

External Pascal files

A BS2000 file must be assigned to an external Pascal file at runtime prior to opening the file (Reset, Rewrite, Extend, Replace). By default the runtime system assumes the Pascal name of the external Pascal file as the link name, and takes the file name specified for the link name from the Task File Table. An OPEN_ERROR is generated if no entry is present in the table.

Local Pascal files

Local Pascal files are normally created as temporary EAM files.

5.3.2 Assignment with the FILE command

The BS2000 FILE command is provided to define the attributes of a file as well as creating an entry in the file catalog and in the Task File Table. The TFT permits the Pascal-XT runtime system to establish a link to an external Pascal file if a link name is specified in the FILE command that corresponds with the link name formed from the variable name of the external file (see below).

Conventions for the formation of link names

The link name is formed from the Pascal name of an external file in accordance with the following rules:

- If the Pascal name is longer than 8 characters, it is truncated to 8 characters.
- Underscores ('_') in the Pascal name are replaced in the link name by number signs ('#').

Uniqueness of program parameters

Link names may be up to 8 characters long. This is the reason why variable names of all external files associated with a program must be distinct in the first 8 characters. Any violation of this rule cannot be detected by the compiler and may therefore result in an undefined program runtime condition.

What are the possible assignments?

The Pascal-XT runtime system makes the assignment to a BS2000 file conventionally via the link mechanism. This permits only SAM and ISAM files to be assigned to external Pascal files (see section 5.1.1). The assignment is made via the link name which is derived from the identifier of the Pascal file. The assignment of SYSDTA and SYSOUT to the predefined text files Input and Output respectively cannot be changed by the FILE command (see also Assignfile in section 5.3.3).

When must the FILE command be executed?

The FILE command must be executed prior to opening the Pascal file (Reset, Rewrite, Extend, Replace). The appropriate call may be issued prior to program execution or in the program itself via the CMD procedure of the predefined package BS2000CALLS (see appendix A.4).

When a Pascal file is opened for writing, the assigned BS2000 file is normally created as a SAM file with the BS2000 defined file attributes.
A change in file attributes can only be achieved with the FILE command. In particular when ISAM files are processed, the attributes must be defined with the FILE command (see predefined package DMSIO in appendix A.5). Tables 5-3 and 5-4 show what may happen when files are opened, depending on whether a FILE command was issued or not.

Interaction with the Assignfile procedure

File assignments can be made by means of the predefined Assignfile procedure together with the FILE command. This is necessary in particular when file attributes must be changed, which is not possible with the Assignfile procedure.

Compatibility between catalog entry and FILE command

When opening an existing file with Reset or Replace the runtime system requires the attributes of the catalog entry to be compatible with those of the FILE command.

- Checks are carried out for the RECFORM, FCBTYPE, KEYPOS and KEYLEN attributes.
- For the RECSIZE attribute the size of the runtime system’s internal buffer is also taken into account. For non-text files this is equal to the size of the element type of the FILE variables and for text files it is equal to the value of MAXLINELENGTH (the default value is 254, see also section 5.3.3). Where RECFORM=V these values increase by 4 and where SPACE=E they increase by 1.
- If there is no FILE command or the RECSIZE operand is not specified, then the runtime system compares the size of its internal buffer with the RECSIZE entry in the catalog and reports an Open_Error condition where incompatibilities are detected.
- If the RECSIZE operand is specified in the FILE command, the runtime system compares the size of its internal buffer with this entry and reports an Open_Error condition where incompatibilities are detected.
- The runtime system leaves to the operating system the comparison of the RECSIZE entry from the catalog with that from the FILE command.
- The operating system (DMS) regards RECSIZE=0 as being compatible with other RECSIZE values and processes the file correctly.

The compatible combinations of RECSIZE entries are described in the DMS manual [14].

<table>
<tr><th rowspan="2">Open with</th><th colspan="2">Physical file</th></tr>
<tr><th>exists</th><th>does not exist</th></tr>
<tr><td>Reset</td><td>The file is opened with its current attributes. If the file attributes are not compatible with the attributes of the Pascal file, a runtime error occurs (READ_ERROR) when reading.</td><td>OPEN_ERROR (System_Code 1604)</td></tr>
<tr><td>Rewrite
Extend</td><td colspan="2">The file is created as a SAM file with variable record length.</td></tr>
<tr><td>Replace</td><td colspan="2">File exists: same procedure as for Reset
File does not exist: same procedure as for Rewrite</td></tr>
</table>

Table 5-3 Opening a file without specification of a FILE command

<table>
<tr><th rowspan="2">Open with</th><th colspan="2">Physical file</th></tr>
<tr><th>exist</th><th>does not exist</th></tr>
<tr><td>Reset</td><td>An OPEN_ERROR is generated if the file attributes in the catalog entry are not compatible with the attributes in the FILE command. If the file attributes are not specified in the FILE command and the attributes are not compatible, a runtime error will occur only when reading is attempted.</td><td>OPEN_ERROR (System_Code 1604)</td></tr>
<tr><td>Rewrite
Extend</td><td colspan="2">The file is created with the attributes specified in the FILE command.</td></tr>
<tr><td>Replace</td><td colspan="2">File exists: same procedure as for Reset
File does not exist: same procedure as for Rewrite</td></tr>
</table>

Table 5-4 Opening a file with specification of a FILE command

5.3.3 Assignment with the predefined procedure assignfile

The predefined procedure Assignfile can be used within a program in order to assign any of the files described in in section 5.2 to an external or Pascal file. Some restrictions, however, apply for ISAM files (see below). The procedure supports two parameters:

Assignfile (f, description)

"f" is a variable of any Pascal file type. "description" is a string (string expression) describing the BS2000 file to be assigned to the Pascal file. The description can contain a file description, an attribute description or both (see definitions in a later paragraph). The string can contain lower- and upper-case letters; blanks are ignored.

Calling Assignfile merely causes the syntax of "description" and the storing of specifications in the runtime system to be checked. The semantic characteristics (validity of attributes, etc) are only checked when the file is opened (see also section 5.3.2).

Except for Input and Output, which are opened implicitly, assignment of a BS2000 file to a Pascal file is only made when the file is opened (Reset, Rewrite, Extend, Replace). If a previous assignment is still in force, the BS2000 file assigned up to that point is closed. Assignfile on the predefined text file Input or Output then causes the text file to be closed and reopened, i.e. the assignments defined in "description" will come into effect immediately.

The string within the "description" parameter must satisfy the following syntax requirements for file. Any blanks between the lexical units are ignored.

```
file                =  file-description              |
                        [","] attribute-description  |
                        file-description "," attribute-description.

file-description    =  standard-file |
                       file-name     |
                       plam-element  |
                       temp-file.

attribute-description = attribute { "," attribute }.

attribute             = "SPACE" "=" "E"           |
                        "LINK" "=" link-name      |
                        "MAXLINELENGTH" "="  integr.
```

Note

In the following examples the file variable "f" may stand for any external or local Pascal file. For the "description" parameter, only string literals are specified. Obviously, any type of string expression may occur.

- **File description**

In the file description the name of the desired BS2000 file is specified. Any previous assignments and attribute descriptions are deleted.

standard-file

The permissible standard files are described in section 5.2.1. The Pascal file must be a text file.

Examples

```
Assignfile (f, '*DUMMY');
Assignfile (f, '*SYSOUT');
Assignfile (f, '*EDT');
Assignfile (f, '*EDT(5)');
```

file-name

"file-name" stands for the name of a BS2000 file. When the Pascal file is opened for writing (Rewrite) a SAM file is created with the BS2000 default attributes and given this name. If the file to be generated is an ISAM file, the file parameters must be defined beforehand by means of the FILE command (see sections 5.2.2 and 5.3.2). When the Pascal file is opened for reading "file-name" may stand for a SAM or ISAM file. The use of keys in ISAM text files is discussed in section 5.2.2.

Example

```
Assignfile (f, '$USERID.BEISPIEL.PROG')
```

File $USERID.BEISPIEL.PROG is assigned to Pascal file f.

plam-element

A PLAM element is defined by library name, element name, type and version. An exact description is given in section 5.2.3. If type and version are not specified, the runtime system assumes the default value "S" (for source); in read and write modes, it assumes the highest-existing and highest-possible versions respectively. The library can also be addressed via a link name (see LINK attribute).

Examples

```
Assignfile (f, '(TOOLS, LMS.PROG (10A,S))')
```

Element LMS.PROG with version 10A and type S from PLAM library TOOLS is assigned to Pascal file f.

```
Assignfile (f, '(TOOLS, LMS.PROG), LINK=F')
```

Library element LMS.PROG with the highest version (*STD) and type S is assigned to Pascal file f. The library is searched for using link name F in the Task File Table. If link name F is not found, library TOOLS is used.

```
Assignfile (f, '(,LMS.PROG(,P)), LINK=LIB')
```

Library element LMS.PROG with the highest version (*STD) and type P is assigned to Pascal file f. The library is addressed via the link name LIB. Prior to opening the library element, a FILE command with the file name of the PLAM library and the link name LIB must be issued. Otherwise an Open_Error with system error code 1603 will be generated.

temp-file

A temporary file (EAM file) can be assigned to every Pascal file by specifying a null string. Temporary files can no longer be accessed once the program is terminated.

Example

```
Assignfile (f, '')
```

- **Attribute description**

The attribute description may specify additional file attributes such as changing the buffer sizes for text files or addressing the file via link names. The attribute descriptions may be specified together with a file name in the same Assignfile call or separately in a subsequent call. If the attributes are specified in an individual Assignfile call, the optional comma preceding the attribute list has an essential function.

(a) If the comma is not specified, the attributes of the current description, if any, are added.

(b) If the attribute list begins with a comma, the existing description, if any, will be deleted.

Local files have a description implicitly; it specifies the assignment to the temporary EAM files. Should a non-EAM file be assigned to a local file, the assignment must be made by specifying either a file name or a link attribute with preceding comma.

SPACE = E

This attribute, in a text file, reserves the first column for feed control characters. The user cannot access this character explicitly. For the predefined Page procedure to accomplish the required effect, this attribute must be specified when a physical file is assigned. The following effects are achieved on input/output:

Output: Column 1 of each line is reserved for the control character and filled with a blank, i.e. the line output by a program is lengthened by one character.
When the predefined procedure Page is called, the current line, possibly not yet closed by Writeln, is output, and a feed control character ("A") is output to column 1 of the next line. Calling Page has no effect if the SPACE attribute has not been specified.
For system files SYSLST and SYSOUT, SPACE=E is set by default.

Input: Column 1 of each line is skipped and not passed to the processing program.

Example

```
Assignfile (listing, 'LST.MAIN, SPACE=E')
```

File LST.MAIN is assigned to Pascal file "listing". The first column of the file is reserved for the feed control character.

LINK = link-name

This attribute is necessary to assign a BS2000 file to a Pascal file via the BS2000 link mechanism. Prior to opening (Reset, Rewrite,...) of the Pascal file a FILE command must be issued specifying the desired file name and the link name specified in Assignfile. When the Pascal file is opened, the runtime system then retrieves the file name via the Task File Table and accepts further FILE command parameters, if any. If in Assignfile a file is assigned to a Pascal file and additionally the LINK attribute is specified (in several calls or in the same Assignfile call), the file name specified in Assignfile is assumed as the default value. It is used only if the runtime system does not find, in the Task File Table, a file name for the link name used.

When the Pascal file is opened an Open_Error of system error code 1603 is generated if neither a file name in Assignfile nor a FILE command for the link name (with specification of a file name) has been specified.

For "link-name" (up to 8 characters) any name or the name derived from the Pascal file identifier (see section 5.3.2) can be specified.

Link name for external Pascal files

The link mechanism permits file attributes of the assigned BS2000 file to be preset by means of the appropriate parameters in the FILE command. For existing files, these parameters must match the entries in the catalog (see chapter 5 and appendix A.5).

Link names for local Pascal files

For local Pascal files (i.e. file variables not listed in program parameter lists) the leading comma **must** precede the attribute list.

Examples

```
Assignfile (f, 'TOOLS.SPEC, LINK=F')
```

A file is assigned to Pascal file f via link name F. If no FILE command has been issued for this link name, file TOOLS.SPEC is used.

```
Assignfile (f, 'LINK=F');
BS2000CALLS.cmd ('/FILE TEST, LINK=F, KEYPOS=3, KEYLEN=10', error);
Reset (f);
...
Assignfile (f, 'LST.BSP');
```

ISAM file TEST with specified attributes is assigned to Pascal file f via link name F. When Assignfile is called for the second time a file will be assigned to Pascal file f thus deleting the previous assignment and attributes.

MAXLINELENGTH = integer

In the case of text files the internal line buffer of the runtime system has a default length of 254 characters. When longer lines are encountered in reading a file or writing to a file a FILE_ERROR results. The attribute MAXLINELENGTH can now be used to alter the size of the line buffer (e.g. to output the entire contents of a screen with a single call to Writeln). "integer" must be a positive integer value and defines the desired number of characters per line. Where the value is equal to or less than 0 a FILE_ERROR is reported. When specifying the length the additional character stipulated if SPACE=E need not be taken into account. If the attribute is not specified when Assignfile is called, then the default value applies regardless whether or not a value has previously been defined.

The attribute is not effective until the Pascal file is opened. An exception is formed by the predefined text files Input and Output which the user cannot open explicitly. They are closed implicitly by the Assignfile call and subsequently reopened with the new attributes.

Examples

```
Assignfile (output, '*SYSOUT, MAXLINELENGTH=4000')
```

After Assignfile has been called, lines with a maximum length of 4000 characters can be output to the predefined text file Output. The assignment of Output to the terminal, or any file selected before, remains unaffected.

```
Assignfile (f, 'MAXLINELENGTH=512, SPACE=E')
```

After file f has been opened (Reset, Rewrite), lines having a maximum length of 512 characters can be output. The additional characters required for form feed control (SPACE=E) need not be taken into account.

Example

The LIST program permits the contents of a file to be output to a terminal. In the Assignfile call both the file name requested interactively and a link name are specified. The name entered interactively is used only if no appropriate FILE command has been issued.

```
/EXEC $PASCAL-XT
%  BLS0500 PROGRAM 'PASCALXT', VERSION '22A00' OF ... LOADED<
%  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG.1990. ALL RIGHT
S RESERVED
//sy er *
//c (plam.manual,list.prog),*dummy
   >>> COMPILATION SUCCESSFUL (WARNINGS:  0;   NOTES:  0)
//run
Enter name of the file
(plam.manual, list.prog)  ─────────────────────────────────────────────  (01)
(*$TITLE = 'Output of a file to SYSOUT'*)

program LIST (input, output, source);

var
      source    : text;
      str       : string;
      filename  : string;

begin
   writeln (output, 'Enter name of the file');
   readln  (input);
   read    (input, filename);
   Assignfile (source, concat (filename, ',LINK=SOURCE'));
   reset   (source);
   while not eof (source) do begin
      readln  (source, str);
      writeln (output, str);
      end;
end (* LIST *).
//sy file testfile,link=source  ───────────────────────────────────────  (02)
//run
Enter name of the file
(plam.manual, list.prog)  ─────────────────────────────────────────────  (03)
   ***   This file "testfile" contains this line only   ***
//
```

(01) Program LIST asks the name of the file to be output. The name of the library element is entered that contains the source program of LIST. The link mechanism has no effect as no FILE command has been issued yet.

(02) For the SOURCE link name a FILE command is issued specifying the file name "testfile".

(03) A file name entry has no effect now. The file name specified in the FILE command is used.

5.3.4 Assignment in the RUN statement

When a program is started in the programming environment by means of the RUN statement (see section 2.6.11), file assignments may be specified in the PARAMETER operand as a string in the form:

```
PARAMETER = 'pascal-file = BS2000-filename'
```

Multiple assignments must be separated by commas. For "pascal-file", only the identifier of an external Pascal file is valid. For "BS2000-filename" only SAM and ISAM files can be specified. Details concerning file attributes are given in section 5.3.2. The linking of the BS2000 file to the Pascal file is made with the FILE command using the name of the Pascal file as the link name, as described in section 5.3.2. Following execution of the program, the link name is released. However, the programming environment records the assignment and resets it when the RUN statement is called next time. No BS2000 files can be assigned to the predefined text files Input and Output. They are assigned to system files SYSDTA and SYSOUT, respectively.

Example 1

The BS2000 file "testfile" is to be assigned to a program with the external file SOURCE. The two assignments illustrate the different entry options for the RUN statement.

```
//run parameter = 'source=testfile'
//r ,'source=testfile'
```

Example 2

The BS2000 files "file1" and "file2" are to be assigned at execution time to a program with the external files SOURCE and DESTINATION.

```
//r ,'source = file1, destination = file2'
```

5.4 File operations

If the Pascal file specified in the predefined subprograms Rewrite, Reset, Get, Put, Readln and Writeln as the first parameter, is assigned to a BS2000 file (SAM or ISAM), library or system file, these subprograms will have a specific effect through the Data Management System (DMS) or the PLAM library access system. Here are the details:

Rewrite: The BS2000 file may first have to be closed with the DMS CLOSE operation and then opened with the DMS OPEN operation in the open mode OUTPUT.
For PLAM library elements, first the library is opened with the PLAM PMATCH operation in the attach mode INOUT and then the element, with the PLAM PMOPEN operation in the open mode WRITE. If the file or the PLAM library or library element cannot be opened, an Open_Error will be generated.
System files will not be opened because they are considered to be permanently open.
For a text file an internal line buffer of type string [n] is created, n being the maximum length set in the latest Assignfile call or the default length of 254.

Reset: Analogous to Rewrite, the BS2000 file will first be closed. Before closing text files opened with Rewrite or Replace, the internal line buffer must be transferred to the file if it is not empty (implicit Writeln). Transfer of the line buffer in case of Close depends on the next Rewrite/Reset open operation. Subsequently, the BS2000 file is opened with the DMS OPEN operation, in the open mode IN.
For PLAM library elements, first the library is opened with the PLAM PMATCH operation in the attach mode INPUT, after which the element is opened with the PLAM PMOPEN operation in the OPEN mode INPUT. If the file or library cannot be opened, an Open_Error will occur.
System files will not be opened because they are considered to be permanently open.
For non-text files the first record of the BS2000 file is read with the DMS GET or PLAM PMGETA operation and transferred to the buffer variable of the Pascal file.
For a text file, a line buffer of type string [n] is created, where n is the maximum line length set in the latest Assignfile call or the default length of 254.
The first record of the BS2000 file is read with the DMS GET, PLAM PMGETA or DMS RDATA operation and transferred to this line buffer. The first character of the line buffer is copied into the buffer variable of the text file and Eoln is set to False. Prefetching the first record by means of RDATA is only valid for SYSDTA, no prefetch of the first record

being allowed with standard input. If the first record is empty a blank is entered for it in the buffer variable and Eoln is set to True. If on SAM and ISAM files the DMS GET operation could not read a record because the file is empty, Eof is set to True. The same applies if on PLAM library elements the PLAM PMGETA operation did not succeed in reading another record of record type 1.
Command entry /EOF on system file SYSDTA is interpreted as end of file.

Get: On non-text files the next record of the BS2000 file is read with the DMS GET or PLAM PMGETA operation and transferred to the buffer variable of the Pascal file. On ISAM files opened with Replace or on ISAM files with SHARED-UPDATE=YES the record read is locked (LOCK). The information concerning LOCK also applies to text files. If Eoln is true on text files at the time immediately before calling Get Eoln, the next record of the BS2000 file is read with the DMS GET, PLAM PMGETA or DMS RDATA operation and transferred to the internal line buffer, the first character of the line buffer is entered in the buffer variable of the text file and Eoln is set to False. If the record read is empty, a blank is set for it in the buffer variable and Eoln is set to True. Otherwise, the next character of the line buffer is copied into buffer variable. If there are no other characters in the line buffer, a character is entered in the buffer variable instead and Eoln is set to True. If the DMS GET operation failed to read another record on SAM and ISAM files because an end-of-file condition occurred, Eof is set to True. The same applies if, on PLAM library elements, no other record of record type 1 can be read with the PLAM PMGETA operation.
The command entry /EOF on system file SYSDTA is interpreted as end of file.

Put: On non-text files the next record of a BS2000 file with the contents of the buffer variables of the Pascal file is written by means of a DMS operation, the DMS PUT operation being used for ISAM files opened with Rewrite, the DMS STORE operation for ISAM files opened with Replace. For SAM files, the buffer variable is only entered in the internal system buffer of the BS2000 file because locate mode is used here. If the system buffer cannot accommodate the buffer variable, the system buffer is emptied with the DMS RELEASE operation.
Libraries with non-text elements have the buffer variable of the Pascal file transferred into the library using the PLAM PMPUTA operation.
For text files the contents of the buffer variable of the text file are written as the next character to the internal line buffer. If an overflow condition occurs in the internal line buffer, an error will be generated.

Readln: Allowed for text files only. The next record of the BS2000 file is read with the DMS GET or PLAM GETA operation and transferred to the internal line buffer. If the DMS GET operation could not read another record on SAM and ISAM files because the end of file was reached, Eof is set to True. The same applies if no other record of record type 1 can be read on PLAM library elements of type S. Otherwise, the first character of this record is copied into the buffer variable of the text file. If an empty record has been read, Eoln is set to True instead and a blank is entered in the buffer variable.

Writeln: Allowed for text files only. The next record of the BS2000 file, together with the contents of the internal line buffer, is written using a DMS operation (analogous to Put for non-text files) or the PLAM PMPUTA operation. For SYSLST, the the line buffer is transferred with the DMS WRLST operation and for SYSOUT the line buffer is transferred with the DMS WROUT operation to the proper system file. Note that the writing of a blank line to SYSOUT does not generate a line break on the part of DMS.

Close: Is only effective if the BS2000 file is open. The BS2000 file is closed with the DMS CLOSE operation. On PLAM library elements, first the element is closed, using the PLAM PMCLOS operation and then the library, using the PLAM PMDTCH operation. On text files opened with Rewrite or Replace the internal line buffer is transferred to the file before it is closed, unless this buffer is empty (implicit Writeln). Close is called when the block is left in which a Pascal file has been declared.

Assignfile: The actual Assignfile actions are not performed until a subsequent Reset, Rewrite or Replace. Only in case of change in file assignment of temporary file to BS2000 file, library or system file will the temporary file be deleted immediately. Additionally, for standard input an implicit Reset, for standard output, an implicit Rewrite is performed.

6 Linking and executing object programs

6.1 General

Creating an executable program involves the following steps:

- The program sources (main program, specifications and bodies of imported packages) need to be compiled in the correct sequence (see 4.1).
- The object modules generated by the compiler and the runtime system modules need to be linked to a program.
- The program needs to be loaded and started.

Following correct compilation, the Pascal-XT compiler generates the following object modules for the main program and package bodies:

- code module
- data module
- test tables module

and for the main program also a

- starter module.

No object modules are generated for package specifications.

The name of an object module is generated from the name of the compilation unit (program or package name) and an additional character which identifies the type of module generated (see section 4.4 for details).

Following compilation, modules are normally written to the temporary EAM object module file (*OMF), but they can also be stored in a PLAM library (see section 2.6.4, COMPILE-UNIT statement, MODULE-LIBRARY operand).

An executable Pascal program consists of a main program, a number of packages and the Pascal-XT runtime system. Packages cannot be executed without a main program. Linking of the object modules to a program can be either static (using the linkage editor TSOSLNK) or dynamic (using the Dynamic Linking Loader). TSOSLNK and DLL are described in detail in [4].

Linking a program using the PATH debugging aid is described in section 9.4.

References between object modules

Each code module of a package or main program contains external references to the code modules of the packages which are directly and indirectly imported and to the code modules of the required runtime system modules. Parallel to this, each data module has external references to the data modules of the imported packages and runtime system modules. If a package includes an entry procedure, the data module will have an external reference to its own code module. No external references from code modules to data modules exist. The starter module of a main program contains external references to the code and data modules of the main program. The external references to external (non-Pascal) subprograms are contained in the data modules.

Example 1

Fig. 6-1 shows the external references in a Pascal-XT program without entry procedures and foreign subprograms. The main program BEISPIEL requires the package AUSGABE, which in turn uses the package TOOLS. The main program has a reference to TOOLS, since it is indirectly imported. No external references to the runtime system are given.

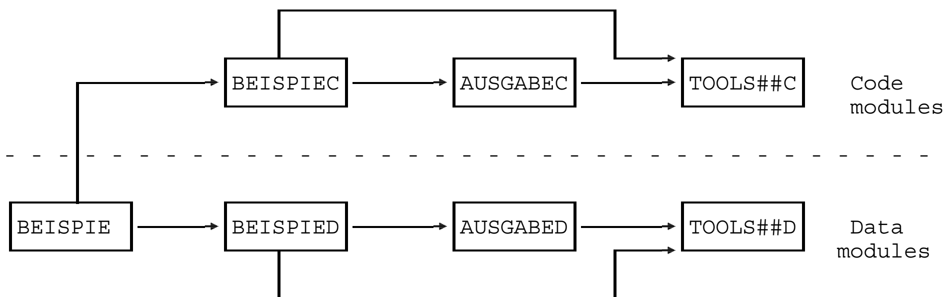

Fig. 6-1 External references between object modules

Example 2

The package TEST uses the package AUSGABE, which in turn calls the external (other language) module EXTUP. As an entry procedure is defined in TEST, the data module contains an external reference to its own code module. The external reference to the foreign subprogram EXTUP is contained in the data module of AUSGABE. No external references to the runtime system are given.

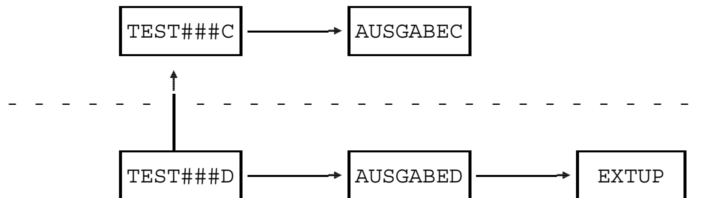

Fig. 6-2 External references for entry procedures and external subprograms

Shareable programs

As a result of the strict separation of code modules and data modules, Pascal-XT programs (main program, packages and runtime system) are shareable. Thus the code modules of programs or of the Pascal-XT runtime system can be loaded into shared code or the common memory pool (see section 6.2.2).

Pascal-XT runtime system and ILCS (Inter-Language Communication Services)

The following sections assume that the Pascal-XT runtime system is located in the object module library $PASLIB-XT. This object module library also contains the modules required by ILCS (see 7.1), which are automatically linked at the same time as the Pascal-XT runtime system (see section 7.1 and the ASSEMBH manual [15]).

Compatibility of runtime system and compiler versions

Pascal-XT objects generated by an earlier compiler version can be linked with the latest Pascal-XT runtime system. Objects generated by the latest compiler, however, cannot be linked with an earlier runtime system.
A Pascal-XT program with ILCS capability requires version 2.2A of the runtime system.

6.2 Static linking

The static linkage editor TSOSLNK enables the object modules of a program to be linked to form a loadable phase (load module), prelinked modules, or segments.

When a program is linked, all required packages must be linked in, even if they are not accessed when the program is executed. All packages are initialized at program start time. If one of the packages is missing, the result is program abortion.

6.2.1 Linking to form a phase

The object modules of the main program and the imported packages are interlinked with the runtime system to form a phase (load module). Using the INCLUDE statement, the starter module of the main program containing the start address of the program must be loaded. To resolve the external references to the imported packages and to the modules of the runtime system, RESOLVE statements must be specified for the object module libraries containing these modules. The linkage editor is not allowed to report unresolved external references (see above). The generated phase can be executed on all BS2000 operating system versions from V7.5 on.

Static linking on XS processors

On XS processors, the loading point can be defined by specifying LOADPT=*XS in the upper address space (above 16 Mb). In this case, however, it is necessary to ensure that the external subprograms are also XS-compatible (see section 7.2).

The simplest form of static linking is provided by issuing the following link commands.

```
/EXEC $TSOSLNK
COMMENT   *** Linking a program to form a load module ***
PROGRAM   prog  ──────────────────────────────────────────────── (01)
INCLUDE   progname , modlib  ──────────────────────────────────── (02)
RESOLVE            , modlib  ──────────────────────────────────── (03)
RESOLVE            , $PASLIB-XT  ─────────────────────────────── (04)
END
```

(01) The PROGRAM statement specifies the name of the linked program (load module). The program can subsequently be started under this name.

(02) "progname" is the name of the main program (name of the starter module) which is read from the object module library "modlib". The linkage editor determines the start address of the program from this module.

(03) The RESOLVE statement specifies the object module library from which TSOSLNK is to resolve the external references. If not all of the required modules are present in the library, additional libraries must be specified.

(04) External references to modules of the runtime system are resolved from the object module library $PASLIB-XT.

Example 1

The object modules of the sample program from appendix A.2 are included in the object module library LIB.BEISPIEL. The program is first linked into a load module with the name BSP.

```
/EXEC $TSOSLNK
%  P500 TSOSLNK/190/85-07-10 LOADED
PROGRAM BSP
INCLUDE BEISPIE, LIB.BEISPIEL
RESOLVE        , LIB.BEISPIEL
RESOLVE        , $PASLIB-XT
END
%  T500 PROG BOUND
%  T503 PROG FILE WRITTEN: BSP
%  T504 NUMBER PAM PAGES USED:     41
```

The program is then started and executed under the name of the load module BSP.

```
/EXEC BSP
%  P500 BSP/000/88-01-21 LOADED
32767
Hexadecimal value = #00007FFF
0
Hexadecimal value = #00000000
/
```

Example 2

The same program is then linked in such a way that it executes in the upper address space (above 16 MB). To do so, LOADPT=*XS must be specified in the PROGRAM statement. The generated phase can then only execute on XS processors.

```
/EXEC $TSOSLNK
%  BLS0500 PROGRAM 'TSOSLNK', VERSION 'V21.0C08' OF '87-09-28' LOADED.
PROGRAM BSP,LOADPT=*XS
INCLUDE BEISPIE, LIB.BEISPIEL
RESOLVE        , LIB.BEISPIEL
RESOLVE        , $PASLIB-XT
END
%  LNK0500 PROG BOUND
%  LNK0062 PHASE CAN BE LOADED ON XS SYSTEM ONLY
%  LNK0503 PROG FILE WRITTEN: BSP
%  LNK0504 NUMBER PAM PAGES USED:     41

/EXEC BSP
%  BLS0500 PROGRAM 'BSP', VERSION ' ' OF '88-01-21' LOADED.
777
Hexadecimal value = #00000309
0
Hexadecimal value = #00000000
/
```

6.2.2 Prelinking to form prelinked modules

TSOSLNK enables several object modules to be prelinked to form prelinked modules. These can be processed by TSOSLNK in further linking procedures or, if they represent an executable program, they can be loaded by DLL and executed.

Prelinking is used, for example, to prelink a program to a module (no phase) and store it in an object module library. The code and data modules of a program can also be separately prelinked to form prelinked modules. The various possibilities are described in the following sections.

For prelinking, the external references and entries must remain apparent. This is controlled via the LINK-SYMBOLS statement of TSOSLNK (see [4]). The exact procedure is described in the examples below.

The Pascal-XT runtime system should not be linked into a prelinked module. With static linking there are no negative effects if this rule is broken, since if a number of prelinked modules are each statically linked with a runtime system, the program behaves as if only one runtime system were linked in.
However, in the case of dynamic link loading (DLL) or dynamic loading (LINK macro) of a prelinked module with its own linked runtime system, DLL would terminate the loading and issue an error message.

6.2.2.1 Prelinking to form a single prelinked module

The code and data modules of a program are prelinked to form a prelinked module without the runtime system. The external references to the runtime system are not resolved until the program is loaded. The runtime system modules may be in shared code or reside in the common memory pool, or they can be dynamically loaded from a library (e.g. $PASLIB-XT).

```
/EXEC $TSOSLNK      "Beginning with Version 18"
COMMENT   * Linking a main program without runtime system *
MODULE    prog ──────────────────────────────────────────────────────── (01)
INCLUDE   progname , modlib   ──────────────────────────────────────────── (02)
RESOLVE            , modlib   ──────────────────────────────────────────── (03)
BIND   ─────────────────────────────────────────────────────────────────── (04)
```

(01) The modules read are linked to form the prelinked module "prog" and stored in the temporary object module file (*OMF).

(02) The starter module "prog-name" of the main program is read from the "modlib" object module file. From this module the linkage editor determines the start address of the program.

(03) The RESOLVE statement specifies the object module library from which TSOSLNK is to resolve the external references. If not all required modules are specified in the library, further libraries must be specified.

(04) Input to TSOSLNK is terminated with the BIND statement, i.e. the linking procedure is executed even though there are still external references to the runtime system which have not been resolved.

Example

The object modules of the sample program from appendix A.2 are included in the object module library LIB.BEISPIEL. The object modules of the program are prelinked without the runtime system to form a prelinked module. The runtime system modules are to be dynamically loaded from $PASLIB-XT when the program is executed. To do so, the Tasklib must be set for $PASLIB-XT.

```
/ERASE *
/EXEC $TSOSLNK
%  P500 TSOSLNK/190/85-07-10 LOADED
MODULE  BSPMOD
INCLUDE BEISPIE, LIB.BEISPIEL
RESOLVE        , LIB.BEISPIEL
BIND
 UNRESOLVED EXTRNS:
 IP@#OP2D  IP@#ER2D  IP@#RT2D  IP@#OP2C  IP@#ER2C  IP@#RT2C  IP@#OU2C
 IP@#iN2C  IP@#TX2C  IP@#RE2C  IP@#OU2D  IP@#iN2D  IP@#TX2D  IP@#RE2D
%  T056 MODULE BOUND WITH UNRESOLVED EXTERNS
%  T505 MODULE BSPMOD WRITTEN TO EAM OMF

/EXEC $LMS
%  P500 LMS/11B/85-08-02 LOADED
LMS (BS2000) VERSION  V1.1B05
CTL=(RDR)                                                          PRT=(CON)
LIB LIB.GROSSMODUL,OUT,OML,ANY
ADDR *OMF
END
****  E N D   O F   R U N  ****  LMS (BS2000) VERSION  V1.1B05

/SYSFILE TASKLIB=$PASLIB-XT
/EXEC (BSPMOD, LIB.GROSSMODUL)
%  P001 DLL VER 761
%  P500 BSPMOD/001/88-01-21 LOADED
255
Hexadecimal value = #000000FF
0
Hexadecimal value = #00000000
/
```

6.2.2.2 Prelinking code and data modules separately

The code and data modules of a program are separately prelinked to form prelinked modules. In this case the runtime system is not included. This type of prelinking permits, for example, the code module to be loaded into shared code. In this case, it should be noted that the code modules of the runtime system must also be in the shared code.

The starter module of a main program is part of the data modules. When prelinking the data modules it must be the first module that is read by means of the INCLUDE statement, since it contains the program’s start address. Data modules must contain external references to code modules (in the starter module and in the case of entry procedures). When prelinking data modules, it should be noted that these references are not resolved. This can be effected using the EXCLUDE statement or by reading in the required data modules using the INCLUDE statement.
When prelinking the code modules it should be noted that the required entry names (CSECT name of the code module of the main program and the names of the entry procedures) must remain apparent. This must be guaranteed using the LINK-SYMBOLS statement of TSOSLNK (see [4]).

```
/EXEC $TSOSLNK    "Beginning with Version 18"
COMMENT   *** Linking the code modules ***
MODULE    code  ------------------------------------------------------ (01)
INCLUDE   progcode , modlib   ---------------------------------------- (02)
RESOLVE            , modlib   ---------------------------------------- (03)
LINK-SYMBOLS *KEEP  -------------------------------------------------- (04)
BIND  ---------------------------------------------------------------- (05)

/EXEC $TSOSLNK    "Beginning with Version 18"
COMMENT   *** Linking the starter module and the data modules ***
MODULE    prog  ------------------------------------------------------ (06)
INCLUDE   progname       , modlib   ---------------------------------- (07)
RESOLVE                  , modlib   ---------------------------------- (08)
EXCLUDE   (progcode,...), modlib   ----------------------------------- (09)
BIND  ---------------------------------------------------------------- (05)
```

(01) The code modules read in are linked to form the prelinked module "code" and stored in the temporary object module file (*OMF).

(02) "progcode" is the name of the main program code module read from the object module library "modlib".

(03) The external references to the remaining code modules are to be resolved from the object module library "modlib".

(04) The external names remain apparent.

(05) The linking procedure is performed despite unresolved external references.

(06) The starter module of the main program as well as the data modules are linked to form the prelinked module "prog" and stored in the temporary object module library (*OMF).

(07) The starter module "progname" is read from the object module library "modlib". From this module, the linkage editor determines the start address of this program.

(08) All external references to the remaining data modules are resolved from object module library "modlib".

(09) The external reference to the main program code module "progcode" must not be resolved. If the application program contains entry procedures, the names of the code modules containing these procedures must likewise be specified in this list. The EXCLUDE statement can be omitted if the code modules are kept in a separate object module library.

Example

The object modules of the sample program from appendix A.2 are contained in the object module library LIB.BEISPIEL. The code and data modules of the program are separately prelinked without the modules of the runtime system. The prelinked modules BSPDAT and BSPCOD are generated and stored in object module library LIB.GROSSMODUL. The EXEC command is used to execute the program. The data module BSPDAT and the object module library are to be specified as parameters; the code module BSPCOD is dynamically loaded by DLL automatically. To resolve the external references to the runtime system modules, the Tasklib must be set to the $PASLIB-XT library prior to program execution.

```
/ERASE *
/EXEC $TSOSLNK
%  P500 TSOSLNK/190/85-07-10 LOADED
MODULE BSPCODE
INCLUDE BEISPIEC, LIB.BEISPIEL
RESOLVE         , LIB.BEISPIEL
LINK-SYMBOLS *KEEP
BIND
 UNRESOLVED EXTRNS:
 IP@#OP2C  IP@#ER2C  IP@#RT2C  IP@#OU2C  IP@#iN2C  IP@#TX2C  IP@#RE2C
%  T056 MODULE BOUND WITH UNRESOLVED EXTERNS
%  T505 MODULE BSPCODE WRITTEN TO EAM OMF

/EXEC $TSOSLNK
%  P500 TSOSLNK/190/85-07-10 LOADED
MODULE BSPDATA
INCLUDE BEISPIE , LIB.BEISPIEL
RESOLVE         , LIB.BEISPIEL
EXCLUDE BEISPIEC, LIB.BEISPIEL
BIND
 UNRESOLVED EXTRNS:
 IP@#OP2D  IP@#ER2D  IP@#RT2D  IP@#OU2D  IP@#iN2D  IP@#TX2D  IP@#RE2D
 IP@#RT2C  BEISPIEC
%  T056 MODULE BOUND WITH UNRESOLVED EXTERNS
%  T505 MODULE BSPDATA WRITTEN TO EAM OMF

/EXEC $LMS
%  P500 LMS/11B/85-08-02 LOADED
LMS (BS2000) VERSION  V1.1B05
LIB LIB.GROSSMODUL,OUT,OML,ANY
ADDR *OMF
END
****  E N D   O F   R U N  ****  LMS (BS2000) VERSION  V1.1B05

/SYSFILE TASKLIB=$PASLIB-XT
/EXEC (BSPDATA, LIB.GROSSMODUL)
%  P001 DLL VER 761
%  P500 BSPDATA/001/88-01-21 LOADED
16383
Hexadecimal value = #00003FFF
0
Hexadecimal value = #00000000
/
```

6.2.2.3 Prelinking the runtime system

To conserve main memory space it is recommended that the Pascal-XT runtime system exists once only, even if there is more than one application. This is achieved by loading the code modules of the runtime system into the shared code or common memory pool (see [6]). The data modules are private for each user and must be linked to his or her program.

To prelink the code modules, all modules from the runtime system library PASLIB-XT whose names begin with "IP@" and whose eighth character is a "C" (for code) are read in. Correspondingly, all data modules whose names begin with "IP@" and whose eighth character is a "D" (for data) are read in. The module names must remain apparent (see the LINK-SYMBOLS statement of TSOSLNK).

The object modules of the predefined packages (see appendix A.3) are contained in PASLIB-XT. When required, they can be linked to form the prelinked modules of the runtime system.

The module #TEST must not be linked in. It contains a switch function and would result in execution of the program under the control of the debugging aid (see chapter 9). Likewise the modules #PATH##C and #PATH##D of the PATH debugging aid should only be linked in when required.

6.2.3 Segmented linking

A Pascal program consisting of one main program and several packages can be linked in segments (see [4]). Linking using an overlay structure is only possible under certain conditions:

(a) The modules of the main program and of all shared packages must be linked into the basic segment (root). These also include the modules of the runtime system. No references to packages in other segments may be given.

(b) Packages in a segment different from the root segment may only import packages from their own segment and/or from the root segment.

(c) A procedure in a segment can only be called from the basic segment via the "external entry mechanism". Therefore a procedure with the directive EXTERNAL must be defined in the basic segment and have the same name (procedure identifier) and the same formal parameter list as the calling entry procedure in the segment.
Restrictions on parameter passing must be taken into account as a condition of this calling mechanism (see chapter 7).

(d) The first time an entry procedure in a segment is called and upon the first call after a segment change, all packages belonging to the segment will be initialized.

Notes

Segmented programs which have been linked cannot be tested with the PATH debugging aid.

On XS processors programs can only be linked in segments in the lower address space (below 16 Mb).

Example

The simplified example below shows an overlay structure with two segments. For reasons of clarity, the basic segment consists merely of the main program TEST and each segment consists of one package (OVL1 or OVL2). However, each segment could also consist of any number of packages.
Messages which make it easier to understand the execution of the program are output in the main program and the packages.

```
/EXEC $USERID.PASCAL-XT
%  BLS0500 PROGRAM 'PASCALXT', VERSION '22A00' OF ... LOADED<
%  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG.1990. ALL RIGHT
S RESERVED
//D DIRECTORY
//MC (PLAM.EXAMPLE,), *SYSOUT, *STD
//C (, OVL1.SPEC)
*** SOURCE LISTING ***    BS2000 PASCAL-XT COMPILER  V2.2A00...

GLOBAL OPTIONS FOR THIS COMPILATION

DEBUG      =    OFF          BY DEFAULT
GENERATE   =    ON           BY DEFAULT
MAP        =    OFF          BY DEFAULT
STANDARD   =    OFF          BY DEFAULT
XREF       =    OFF          BY DEFAULT

CURRENT COMPILATION UNIT (SOURCE FILE)

               ($USERID.PLAM.EXAMPLE,OVL1.SPEC(*STD,S))

     1      package OVL1;
     2
     3         entry procedure proz1;
     4         entry procedure proz11 ( var i : integer );
     5
     6      end { package OVL1 }.
```

```
*********************************************************
*                    COMPILATION SUMMARY                *
*********************************************************
*  ERRORS DETECTED          :        0                  *
*  WARNINGS                 :        0                  *
*  NOTES                    :        0                  *
*  SIZE OF CODE MODULE      :        0 BYTES            *
*  SIZE OF DATA MODULE      :        0 BYTES            *
*  COMPILATION TIME         :    0.228 SEC              *
*********************************************************
   >>> COMPILATION SUCCESSFUL (WARNINGS: 0;  NOTES: 0)
//C (, OVL2.SPEC)
*** SOURCE LISTING ***    BS2000 PASCAL-XT COMPILER  V2.2A00

GLOBAL OPTIONS FOR THIS COMPILATION

DEBUG      =    OFF          BY DEFAULT
GENERATE   =    ON           BY DEFAULT
MAP        =    OFF          BY DEFAULT
STANDARD   =    OFF          BY DEFAULT
XREF       =    OFF          BY DEFAULT

CURRENT COMPILATION UNIT (SOURCE FILE)

               ($USERID.PLAM.EXAMPLE,OVL2.SPEC(*STD,S))

     1      package OVL2;
     2
     3         entry procedure proz2 ;
     4
     5      end { package OVL1 }.


*********************************************************
*                    COMPILATION SUMMARY                *
*********************************************************
*  ERRORS DETECTED          :        0                  *
*  WARNINGS                 :        0                  *
*  NOTES                    :        0                  *
*  SIZE OF CODE MODULE      :        0 BYTES            *
*  SIZE OF DATA MODULE      :        0 BYTES            *
*  COMPILATION TIME         :    0.068 SEC              *
*********************************************************
   >>> COMPILATION SUCCESSFUL (WARNINGS: 0;  NOTES: 0)
//C (, OVL1.BODY)
*** SOURCE LISTING ***    BS2000 PASCAL-XT COMPILER  V2.2A00

GLOBAL OPTIONS FOR THIS COMPILATION

DEBUG      =    OFF          BY DEFAULT
GENERATE   =    ON           BY DEFAULT
MAP        =    OFF          BY DEFAULT
STANDARD   =    OFF          BY DEFAULT
XREF       =    OFF          BY DEFAULT
```

```
LIST OF RECOMPILED PACKAGE SPECIFICATIONS (SOURCE FILES)

               ($USERID.PLAM.EXAMPLE,OVL1.SPEC(*STD,S))

CURRENT COMPILATION UNIT (SOURCE FILE)

               ($USERID.PLAM.EXAMPLE,OVL1.BODY(*STD,S))

     1      package body OVL1 ( output ) ;
     2
     3      procedure proz1;
     4      begin
     5        writeln ('- execution of OVL1.proz1');
     6      end;
     7
     8      procedure proz11 ( var i : integer ) ;
     9      begin
    10        writeln ('- execution of OVL1.proz11; i = ', i:1);
    11      end;
    12
    13      begin
    14        writeln ('- initialization of package OVL1');
    15      end { package body OVL1 }.


***********************************************************
*                   COMPILATION SUMMARY                   *
***********************************************************
*  ERRORS DETECTED         :        0                     *
*  WARNINGS                :        0                     *
*  NOTES                   :        0                     *
*  SIZE OF CODE MODULE     :      948 BYTES               *
*  SIZE OF DATA MODULE     :      236 BYTES               *
*  COMPILATION TIME        :    0.242 SEC                 *
***********************************************************
   >>> COMPILATION SUCCESSFUL (WARNINGS: 0;  NOTES: 0)
//C (, OVL2.BODY)
*** SOURCE LISTING ***    BS2000 PASCAL-XT COMPILER  V2.2A00

GLOBAL OPTIONS FOR THIS COMPILATION

DEBUG      =    OFF          BY DEFAULT
GENERATE   =    ON           BY DEFAULT
MAP        =    OFF          BY DEFAULT
STANDARD   =    OFF          BY DEFAULT
XREF       =    OFF          BY DEFAULT
```

```
LIST OF RECOMPILED PACKAGE SPECIFICATIONS (SOURCE FILES)

               ($USERID.PLAM.EXAMPLE,OVL2.SPEC(*STD,S))

CURRENT COMPILATION UNIT (SOURCE FILE)

               ($USERID.PLAM.EXAMPLE,OVL2.BODY(*STD,S))

     1      package body OVL2 ( output );
     2
     3      procedure proz2;
     4      begin
     5        writeln ('- execution of OVL2.proz2');
     6      end;
     7
     8      begin
     9        writeln ('- initialization of package OVL2');
    10      end { package body OVL2 }.


***********************************************************
*                   COMPILATION SUMMARY                   *
***********************************************************
*  ERRORS DETECTED         :        0                     *
*  WARNINGS                :        0                     *
*  NOTES                   :        0                     *
*  SIZE OF CODE MODULE     :      676 BYTES               *
*  SIZE OF DATA MODULE     :      168 BYTES               *
*  COMPILATION TIME        :    0.216 SEC                 *
***********************************************************
   >>> COMPILATION SUCCESSFUL (WARNINGS: 0;  NOTES: 0)
//C (, OVLTEST.PROG)
*** SOURCE LISTING ***    BS2000 PASCAL-XT COMPILER  V2.2A00

GLOBAL OPTIONS FOR THIS COMPILATION

DEBUG       =    OFF          BY DEFAULT
GENERATE    =    ON           BY DEFAULT
MAP         =    OFF          BY DEFAULT
STANDARD    =    OFF          BY DEFAULT
XREF        =    OFF          BY DEFAULT

CURRENT COMPILATION UNIT (SOURCE FILE)

               ($USERID.PLAM.EXAMPLE,OVLTEST.PROG(*STD,S))

     1      program OVLTEST ( input , output );
     2
     3      var k : integer;
     4
     5      procedure proz1;                      external;
     6      procedure proz11 ( var i : integer ); external;
     7      procedure proz2;                      external;
     8
```

```
     9      begin
    10         writeln ('- start of main program');
    11         proz1;                { call OVL1.proz1  }
    12         k := 1024;
    13         proz11 ( k );         ⎡ call OVL1.proz11 ⎤
    14         proz2;                ⎨ call OVL2.proz2  ⎬
    15         proz1;                ⎣ call OVL1.proz1  ⎦
    16         writeln ('- end of main program');
    17      end { program OVLTEST }.

***********************************************************
*                   COMPILATION SUMMARY                   *
***********************************************************
*  ERRORS DETECTED         :        0                     *
*  WARNINGS                :        0                     *
*  NOTES                   :        0                     *
*  SIZE OF CODE MODULE     :      740 BYTES               *
*  SIZE OF DATA MODULE     :      116 BYTES               *
*  COMPILATION TIME        :    0.229 SEC                 *
***********************************************************
   >>> COMPILATION SUCCESSFUL (WARNINGS: 0;  NOTES: 0)
//END
   END OF THE PASCAL SESSION  -  USED TIME = 1.247 SECONDS
/EXEC $TSOSLNK
%  BLS0500 PROGRAM 'TSOSLNK', VERSION 'V21.0D10' OF '90-05-10' LOADED.
PROG OVLTEST, FILENAM=PH.OVL,CONTROL=Y
COMMENT ***      Root     ***
INCLUDE OVLTEST , PLAM.EXAMPLE
COMMENT ***   Segment 1   ***
OVERLAY K1,O1
TRAITS  OVL1###C, RMODE=24
TRAITS  OVL1###D, RMODE=24
INCLUDE OVL1###C, PLAM.EXAMPLE
INCLUDE OVL1###D, PLAM.EXAMPLE
COMMENT ***   Segment 2   ***
OVERLAY K1,O2
TRAITS  OVL2###C, RMODE=24
TRAITS  OVL2###D, RMODE=24
INCLUDE OVL2###C, PLAM.EXAMPLE
INCLUDE OVL2###D, PLAM.EXAMPLE
COMMENT *** Root segment ***
RESOLVE         , $USERID.PASLIB-XT
RESOLVE         , PLAM.EXAMPLE
END
PROG BOUND
PROGRAM FILE WRITTEN : PH.OVL
NUMBER PAM PAGES USED:     49
```

```
/EXEC PH.OVL
%  BLS0500 PROGRAM 'OVLTEST', VERSION ' ' OF '90-11-27' LOADED.
- start of main program
- initialization of package OVL1
- execution of OVL1.proz1
- execution of OVL1.proz11; i = 1024
- initialization of package OVL2
- execution of OVL2.proz2
- initialization of package OVL1
- execution of OVL1.proz1
- end of main program
```

6.3 Dynamic linking

The Dynamic Linking Loader (DLL) loads the object modules into main memory, links them, and starts the program.

External references, e.g. to imported packages, are resolved by the autolink mechanism of DLL (see [4]).

Note

The autolink mechanism does not affect the temporary object module file (*OMF) of the task.

Dynamic linking and loading within the programming system

Linking and loading are performed implicitly within the programming system when the RUN-PROGRAM statement (see section 2.6.11) is used. With this statement, a program can be linked and loaded from the temporary object module file (*OMF) of the user task or from a specified object module library. The program uses the runtime system of the programming system.

Dynamic linking and loading outside the programming system

Dynamic linking and loading are performed implicitly when the EXEC or LOAD command is used. Loading from the temporary object module file (*OMF) is not possible; it must take place from an object module library.

```
/EXEC (prog,modlib)
```

The program "prog" to be executed is loaded from the object module library "modlib". "prog" identifies the starter module of the main program or the name of the prelinked module into which the program was prelinked. The object modules of imported packages can be linked from `shared` code or from object module libraries that can be "accessed" via the autolink mechanism of DLL.

Dynamic linking and loading on XS processors

Pascal-XT programs and the Pascal-XT runtime system are XS-compatible, i.e. they can execute anywhere in the 31-bit address space.

If a Pascal-XT program contains external subprograms (e.g. assembly language sub-programs), these must likewise be XS-compatible if the program is to be executed in the upper address space (above 16 Mb). If this is not the case, a program is usually terminated as undefined.

Example

When the program is called, the PROG-MOD parameter informs DLL that the program can be executed in the upper address space.

```
/SYSFILE TASKLIB=$PASLIB-XT
/EXEC (BEISPIE,LIB.BEISPIEL),PROG-MOD=ANY
%  BLS0001 DLL VER 917
%  BLS0517 MODULE 'BEISPIE' LOADED
55
Hexadecimal value = #00000037
0
Hexadecimal value = #00000000
/
```

6.4 Program termination code

A Pascal-XT program which has executed is terminated in accordance with the conventions for program termination under BS2000. A description of these conventions can be found in section 2.4.

The termination code can have the following values and meanings:

’0’ The program run was without error and terminated normally.

’2’ There was an error in program execution. The spin-off mechanism is activated.

’3’ The program has been terminated due to an error in the runtime system. The spin-off mechanism is activated.

The program information bytes normally contain 3 blanks, but may be given program-specific values in the program within the framework of the conventions. The program information bytes in the program are set using the procedure SET_RETURN_CODE from the predefined package BS2000CALLS. To call the program the termination code must be specified.

6.5 License protection for the Pascal-XT runtime system

As of version 2.2A, any program implemented in Pascal-XT with a linked runtime system is checked for execute permission when it starts. Execute permission has been granted if the $TSOS user ID includes a Pascal-XT license module with a version greater than or equal to the version of the runtime system linked in the program. If the license module has the wrong version or is missing, the program is terminated and the following message is issued:

```
PASCAL-XT: MISSING LICENSE KEY FOR RUNTIME SYSTEM.
```

The license module is a component of both the Pascal-XT programming system and the Pascal-XT runtime system. If you wish to use software implemented in Pascal-XT but do not have a Pascal-XT programming system, you can use the Pascal-XT runtime system.

7 Language interfaces

A program generally consists of a number of program sections which may be implemented in different programming languages.

A Pascal-XT main program or a Pascal-XT entry procedure together with any subprograms called within it is known as a Pascal-XT program section. Calls between Pascal-XT program sections and program sections in other languages are effected via the interfaces described in this section.

The boundaries of Pascal-XT program sections are important both in terms of error handling (see 10.2) and in terms of the behavior of the debugging aid PATH (see 9.1.4.3 and 9.1.4.4).

Pascal-XT supports language interfacing using a simple approach:

- Calling a non-Pascal subprogram

  The subprogram must be declared in the Pascal-XT program as a function or procedure using the directive EXTERNAL and must be called as a Pascal-XT procedure or function.

- Calling Pascal-XT procedures from non-Pascal programs

  The Pascal-XT procedure must be declared in a package specification and identified as an entry procedure with the keyword ENTRY (see 7.3). Pascal-XT functions may *not* be called from within programs written in other languages. Entry procedures may also be called from within Pascal-XT programs.

As of Pascal-XT V2.2A, subprograms in other languages are invoked in accordance with the ILCS conventions (see 7.1); the former "standard subprogram interface" still applies to earlier versions. With language interfacing, there are no restrictions on the sequence in which the different program sections are called. The naming conventions (see 4.4) should be observed.

In addition, it is possible to invoke assembly language subprograms using a high-speed mechanism (see 7.4) using the INTERNAL directive.

The same file variable cannot be accessed across language boundaries since Pascal-XT cannot exchange information relating to the status of files with external subprograms.

7.1 ILCS program communication interface

The software product ILCS (Inter-Language Communication Services) standardizes and simplifies essential communication functions across different languages both between the programs of a run unit and between run unit and operating system.

ILCS is a combination of software and interface conventions:
It contains runtime routines that are combined in a PLAM library; also, it guarantees a communication interface that complies with the "standard linkage conventions in BS2000". This means that every object module generated by a compiler with ILCS capability is ready for linking with other programs written in the same or a different language in accordance with the standard linkage conventions.

The library of ILCS runtime routines is shipped with all ILCS-capable compilers - as an additional runtime system, as it were.

Specifically, ILCS offers the following functions:

- multilateral convention for linking programs in different languages
- standardized guidelines for event handling
- storage management (stack and heap storage)
- program mask handling
- processing of non-local branches

In this section, only the ILCS function ’program linkage’ used by the Pascal-XT compiler is described, together with the underlying ILCS data structures.

Note

Programs that have been compiled with ILCS-capable compilers are required to be linked into a program system by means of ILCS. If a program system includes programs that do not behave according to ILCS conventions, they may have to be restructured in order to be ILCS-compliant. Otherwise there is a risk of incompatibility - at least as far as the linkage of programs written in different languages is concerned.

7.1.1 ILCS register conventions

Register contents at program call

The following table presents an overview of the register contents as supplied by the calling program before passing control to the called program.

Register number	Contents
0	Number of parameters
1	Start address of the parameter address list
2 - 12	Program data
13	Start address of the save area of the calling program
14	Address of the return point to the calling program
15	Address of the entry point in the called program
PM	Program mask: value of PCD field "program mask"

Register contents on return to the calling program

The following table presents an overview of the register contents as supplied by the called program when returning control to the called program.

Register number	Contents
0 - 1	Return values of integer functions or undefined
2 - 14	Same as when the program was called
15	undefined
PM	Program mask: value of PCD field "program mask"

7.1.2 ILCS data structures

Save area

The calling program passes the address of a save area in which the called program can store the current register states. The called program sets up a new save area and chains the two save areas together.
The save area has the following structure:

Byte	Contents

1-4	1st byte:
	   1st bit: activity bit (1: program active, 0: program inactive)
	   2nd-7th bits: reserved
	   8th bit = generally 0
	2nd byte: version = X’01’
	3rd and 4th bytes: X’FEFF’

5-8	contains the start address of the save area of the calling program; in the **first** calling program the content of this field is -1
9-12	contains the start address of the next (chained) save area (if any)
13-16	contents of register 14
17-20	contents of register 15
21-24	contents of register 0
25-28	contents of register 1
29-32	contents of register 2
.	
.	
69-72	contents of register 12
73-76	reserved for FOR1
77-80	address of the PCD
81-84	address of the EHL (Event Handler List): if no EHL is defined, the field contains a value of -1
85-128	reserved

Prosys Common Data Area (PCD)

The PCD is a common data area that is used by all programming languages. It is 4096 bytes long. The first part contains the data areas used by ILCS, including the "program mask" field (byte 148), which is initialized to the value X'0C'. The second part of the PCD contains the programming language areas, each 128 bytes long, which are available to the runtime systems of the different languages.

7.1.3 Initializing the Pascal-XT runtime system

Pascal-XT deviates from the ILCS conventions in that it initializes its runtime system itself. When the Pascal-XT main program or the first Pascal-XT entry subprogram is executed, the Pascal-XT runtime system initializes itself and then ILCS. Every ILCS-compliant external subprogram which is called from within any Pascal-XT program section can therefore assume that it will find an initialized ILCS.

7.1.4 Program mask handling by ILCS

The program mask for program execution is set during the initialization to the value of the PCD "program mask" field (preset to X'0C'). If modified during the program run, it must be reset to the value of the PCD field "program mask" before the next program call or before control is returned.

7.1.5 Parameter passing in ILCS program systems

There are significant differences in the semantics of the data types of the programming languages that can be linked using ILCS. The table below lists the data types that can be passed as parameters without causing problems because they have the same data representation in the individual languages. If other data types are used as parameters, a precise knowledge of the relevant data storage is necessary in order to ensure correct program execution.

Compiler	Data types			
	Binary word	Floating-point word	Floating-point doubleword	String
COBOL85	PIC S9(i) COMP SYNCHRONIZED 5<=i<=9	COMP-1	COMP-2	USAGE DISPLAY
FOR1	INTEGER*4	REAL*4	REAL*8	CHARACTER*i
Pascal-XT	long_integer	short_real	long_real	packed array [<range>]of char
PLI1	BIN FIXED(31) ALIGNED	BIN FLOAT(21) DEC FLOAT(6)	BIN FLOAT(53) DEC FLOAT(16)	CHAR(i)
C	long	float	double	char <var> [<size>]
Columbus-Assembler	F	E	D	C

Pascal-XT also supports the parameter types "short_integer" and "char", which are not provided by ILCS. In COBOL85 the corresponding types are called "PIC S9(4) USAGE COMP" and "PIC X".

The data must always be stored properly aligned, i.e. 32-bit integers in binary representation on a word boundary, floating-point numbers on a word or doubleword boundary, strings on a byte boundary. The length of strings is constant and known to the called program.

Parameters are passed "by reference", i.e. the address of the parameter is passed and not its value. The calling program sets up a list of the addresses that have been passed. The number of parameters is passed in register 0, the address of the list in register 1 (see "ILCS register conventions").

Passing function return values

Return values of integer functions are passed in registers 0 and 1, and return values of floating-point functions in floating-point register 0, starting with the floating point.
It is possible to pass return values with other data types in registers 0 and 1, but this is not defined by ILCS. Their representation is left to the individual programming languages.

7.1.6 Linking ILCS program systems

The following basic rule applies:
The user should link in such a way that the ILCS module IT0INITS is contained only once in the program system. This module is located in the library $PASLIB-XT, together with the Pascal-XT runtime system. See chapter 6 for further details.

7.2 Interfacing non-Pascal subprograms

Procedures written in other languages ("external" or "non-Pascal" subprograms must be identified as such by means of a directive in the procedure declaration within the Pascal program (package).

Declaration of non-Pascal subprograms in the Pascal program

For a subprogram written in a language other than Pascal, the procedure or function header must be declared in the Pascal-XT program as follows:

```
PROCEDURE    procedure-identifier (formal-parameter-list); directive,    or
FUNCTION     function-identifier (formal-parameter-list): type; directive.
```

One of the following values must be specified for "directive":

`EXTERNAL`	for all other programming languages, necessary in all cases for XS processors (see "Special conditions for XS processors"); As of Pascal-XT V2.2A, it is recommended that only this directive should be used. However, the following directives are still supported.
`FORTRAN`	for Fortran subprograms
`COBOL`	for COBOL subprograms
`C`	for C subprograms

The subprogram declaration is permitted in a package specification, package body or main program. There can be no subprogram block identification in addition to the subprogram declaration.

Result types of external functions

The result type for functions can be ordinal, real or a pointer. Ordinal values and pointer values must be returned in register R0, while real values must be returned in register F0. External functions are possible only with the EXTERNAL and FORTRAN directives.

Formal parameters

Variable and value parameters are permitted as formal parameters. For value parameters, copies are created prior to the call and their addresses are passed.

Interpretation of directives

The different directives result in minor differences in interpretation. The following table shows what the Pascal-XT system places in register 0 and how it sets the most-significant bit in the last parameter, depending on the directive.

Directive	Register 0 contains	"first bit"
EXTERNAL	the number of parameters	is reset "0"
FORTRAN	the number of parameters	is set "1"
COBOL	the number of parameters	is set "1"

With procedures having no parameters, register 0 and register 1 contain the value 0.

Special conditions for XS processors

The EXTERNAL directive must always be specified for XS-compatible external subprograms which are called via the standard subprogram interface, regardless of the language the subprograms are written in.

Actions prior to calling a non-Pascal subprogram

When calling an external subprogram the following actions are taken:

- The registers are set as described in 7.1
- The Pascal-XT error handling facility remains activated, so that errors occurring in the external subprogram can be propagated to the Pascal-XT program section. For information on the behavior of external subprograms, please refer to the manual for the relevant language.
- Pascal-XT INTR-handling is deactivated internally, but not for ILCS. The INTR event, which is initiated by [K2] /INTR, is therefore only interpreted as a Break_Error and propagated if it occurs while a Pascal-XT program section is executing.
- The subprogram is called according to the ILCS convention, as described in 7.1.

Actions in the called external subprogram

The actions necessary within the called subprogram will be described in the manuals for the language concerned. Possible differences in the storage representation of parameters must be noted.
The called subprogram is responsible for the saving of the required registers in the save area.

Action following return from the external subprogram

Pascal-XT INTR handling is reactivated internally.

Up to V2.1:
The Pascal-XT program assumes that the external subprogram has executed correctly if it receives control again afterwards and no errors are passed on to it. If there is to be a return code, it must be returned via an additional parameter.

As of V2.2A:
Errors can be propagated across language boundaries, i.e. an error in an external subprogram no longer necessarily causes the program to abort (see chapter 10, Runtime errors and error handling). For information on the error propagation facilities offered by a subprogram programming language, please refer to the appropriate Language Reference Manual.

Note

The names of all external subprograms used within a program (main program and packages) must be unique (up to BS2000 V11: this means that they must be differentiated in the first 8 characters). This is not checked at the time of compilation.

Examples

1. Language interfacing between Pascal-XT and Cobol

A Pascal-XT main program calls a Cobol subprogram and passes a record variable with two components as a variable parameter. In the Cobol subprogram these components are displayed on screen and their values are then changed. After the return to the main program, the two components, their values now globally changed, are redisplayed.

Source code of Pascal-XT main program "PASCMAIN":

```
PROGRAM PASCMAIN (INPUT,OUTPUT);
TYPE
   RA   =   1..8;
   STR8 =   ARRAY [RA] OF CHAR;
   SATZ =   RECORD
            A  :  INTEGER;
            B  :  STR8;
            END;
CONST
   AGGR =   STR8('T','E','S','T',' ':4); ------------------------------- (01)
VAR
   RC   :   SATZ;
   I    :   INTEGER;
PROCEDURE COBUPROG (VAR PAR:SATZ); EXTERNAL; --------------------------- (02)

BEGIN   { PASCMAIN }
```

```
    RC.A := 1111;
    RC.B := AGGR;
    COBUPROG (RC);                                                    (03)
    WITH RC DO BEGIN
       WRITELN ('A:',A);
       WRITE ('B:');
       FOR I := FIRST(RA) TO LAST(RA) DO
          WRITE (B[I]);
       WRITELN;
       END;
END.
```

Source code of Cobol subprogram "COBUPROG":

```
       ID DIVISION.
       PROGRAM-ID. COBUPROG. ------------------------------------------ (04)
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SPECIAL-NAMES.
           TERMINAL IS SCREEN.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       77 TEILDR    PIC 9(8). ----------------------------------------- (05)
       LINKAGE SECTION.
       01  SATZ.
           02  TEILB   PIC S9(8) COMP. -------------------------------- (06)
           02  TEILA   PIC X(8). -------------------------------------- (07)
       PROCEDURE DIVISION USING SATZ.
       ANF.
           MOVE TEILB TO TEILDR. ------------------------------------- (08)
           DISPLAY "TEILB: " TEILDR UPON SCREEN. --------------------- (09)
           DISPLAY "TEILA: " TEILA UPON SCREEN.
           MOVE "XXXXZZZZ" TO TEILA. --------------------------------- (10)
           ADD 1 TO TEILB. ------------------------------------------- (11)
       BACK.
           EXIT PROGRAM.
```

Call to executable phase "PASCOB" and runtime listing:

```
/EXEC PASCOB ------------------------------------------------------------ (12)
%  BLS0500 PROGRAM 'PASCOB', VERSION ' ' OF '...' LOADED
TEILB: 00001111 -------------------------------------------------------- (13)
TEILA: TEST
A:        1112 --------------------------------------------------------- (14)
B:XXXXZZZZ
```

Explanation:

(01) Definition of constant AGGR of type STR8: it is set via an aggregate (see [1], section 9.5) with the character string "TEST " as value.

(02) Declaration of the external Cobol procedure COBUPROG.

(03) Call of subprogram COBUPROG: the variable RC of type record is passed. The two components of RC contain the values 1111 and "TEST ".

(04) Name of the Cobol subprogram: COBUPROG

(05) Declaration of variable TEILDR of type PIC 9(8).

(06) The passed variable of type SATZ ("record") has two components: The first is of type PIC S9(8), which in PASCAL-XT corresponds to type long_integer. Its value is assigned to variable TEILB. TEILB then contains the value 1111.

(07) The second is of type PIC X(8), which in PASCAL-XT corresponds to type array [8] of char. Its value is assigned to variable TEILA. TEILA then contains as its value the character string "TEXT   ".

(08) The value of variable TEILB is assigned to variable TEILDR. TEILDR then contains the value 1111.

(09) Name and value of the variables TEILB and TEILA are displayed. See (13).

(10) Variable TEILA is assigned the character string "XXXXZZZZ" as its value.

(11) The value of variable TEILB is incremented by 1.

(12) Startup of executable phase PASCOB.

(13) Output from the Cobol subprogram: name and value of variables TEILB and TEILA. See (09).

(14) Output upon return to main program: names and values of A and B.

2. Language interfacing between Pascal-XT and Assembler

A Pascal-XT main program calls an Assembler subprogram and passes two value parameters, which are added, and a variable parameter, in which the result of the addition is stored. After the return to the main program, the result is displayed on screen.

Source code of Pascal-XT main program "PASASS":

```
PROGRAM PASASS (OUTPUT);
PROCEDURE ASSUP (A,B : INTEGER; VAR ERG : INTEGER); EXTERNAL; ———————— (01)
VAR
   ERG   :  INTEGER;
BEGIN
   ASSUP (5,7,ERG); —————————————————————————————————————————————————— (02)
   WRITELN (ERG);
END.
```

Source code of Assembler subprogram "ASSUP":

```
ASSUP    START ─────────────────────────────────────────────────────── (03)
ASSUP    AMODE  ANY
ASSUP    RMODE  ANY
PCD#AREA DSECT
         IT0PCD PCD# ───────────────────────────────────────────────── (04)
ASSUP    CSECT
ASS#VOR  STM    14,12,12(13) ───────────────────────────────────────── (05)
         BALR   5,0 ─────────────────────────────────────────────────── (06)
         USING  *,5
         ST     13,SAVPSA
         L      6,SAVPCD-SAVAI(,13)
         ST     6,SAVPCD
         LA     13,SAVAI
         ST     13,PCD#ASA-PCD#EYEC(,6)
         LM     2,4,0(1) ────────────────────────────────────────────── (07)
         L      7,0(2) ──────────────────────────────────────────────── (08)
         L      8,0(3) ──────────────────────────────────────────────── (09)
         AR     7,8 ─────────────────────────────────────────────────── (10)
         ST     7,0(4) ──────────────────────────────────────────────── (11)
ASS#NACH L      6,SAVPCD ────────────────────────────────────────────── (12)
         L      13,SAVPSA
         ST     13,PCD#ASA-PCD#EYEC(,6)
         LM     14,12,12(13)
ASS#RET  BR     14 ──────────────────────────────────────────────────── (13)
         DS     0F
SA       IT0VSA SA ──────────────────────────────────────────────────── (14)
         END
```

Call to executable phase "PASASSEM" and runtime listing:

```
 /EXEC PASASSEM ─────────────────────────────────────────────────────── (15)
 %  BLS0500 PROGRAM 'PASASSEM', VERSION ' ' OF '...' LOADED
          12 ────────────────────────────────────────────────────────── (16)
```

Explanation:

(01) Declaration of external procedure ASSUP.

(02) Call of procedure ASSUP; the value parameters 5 and 7 and variable parameter ERG are passed at the same time. Variable ERG is to store the result of the addition of 5 and 7 that is to be performed by the Assembler subprogram.

(03) Name of the Assember subprogram: ASSUP

(04) IT0PCD is an ILCS macro which generates the fields of the PCD area (Prosys Common Data Area).

(05) The registers and SA (Save Area) of the caller are saved.

(06) The base address registers are loaded.

(07) Register 1 is loaded with the address of the parameter list.

(08,09) The passed parameters are loaded.

(10) The passed value parameters are added.

(11) The passed variable parameter is supplied with the result of the addition.

(12) The register status prior to the Assembler subprogram call is restored.

(13) Return to caller, i.e. to the Pascal-XT main program.

(14) ILCS macro IT0VSA generates the ILCS-conforming save area.

(15) Startup of executable phase PASASSEM.

(16) Value of variable ERG displayed after return to main program.

```
)L85 V01.1A02   COBOL-85 COMPILATION              SOURCE LISTING...
  (*$TITLE = 'Call a COBOL subprogram'*)
```

7.3 Invocation by programs in other languages

Procedures labeled with the keyword ENTRY in a package specification can be called by any program (even from within Pascal-XT programs). Recursive calls of entry procedures are permitted.

Requirements of the Pascal procedure

- The procedure must be declared in a package specification. The corresponding procedure identification must appear in the package body. Pascal functions cannot be called by external programs.
- The procedure must be labeled as an entry procedure by means of the keyword ENTRY:

  ENTRY PROCEDURE procedure-identifier (formal-parameter-list);

  The procedure identification in the package body must not contain this keyword.
- As formal parameters only variable (VAR) parameters are permitted.

Parameter passing

The number of parameters accepted is defined by the declaration of the Pascal-XT procedure. Entries in register 0 regarding the number of parameters and/or the "first bit" in the last parameter of the parameter address list are ignored.

Floating-point registers are not saved in the Pascal program.

Actions in the external program before calling

To call a Pascal-XT entry subprogram there is no need for the external caller to support ILCS since the Pascal-XT runtime system is self-initializing and causes the initialization of ILCS. The only requirement is for the caller’s save area to be chained backward with that of its caller. If this is not the case, the user must provide a suitable adapter module.

Note for assembly language programmers:
If a Pascal-XT prelinked module containing Pascal-XT entry procedures is loaded dynamically, IT0ININ needs to be called only if mathematical functions are used.

Actions in the called Pascal-XT entry procedure

- The registers of the caller are saved in the assigned save area.
- The license check is performed for the Pascal-XT runtime system (see 6.5).
- ILCS is initialized if this has not already happened.
- The Pascal-XT runtime system is initialized if this has not already happened, and the user routine STXIT for the INTR event is activated for ILCS. Stack and heap are set up.
- The save area of the Pascal-XT entry procedure is set up in accordance with ILCS conventions and chained with the save area of the caller.
- The Pascal-XT error handling facility is activated for ILCS if this has not already happened.

  The Pascal-XT INTR handling facility is activated at each invocation. If the INTR event (initiated by [K2]/INTR) occurs while a Pascal-XT program section is executing, it is interpreted as a Break_Error and propagated.
- The package containing the entry procedure, and all packages imported by it are initialized, if this has not already happened. This also causes any packages referenced by the package to be initialized, where this has not already been done.
- The entry procedure is called.

Actions after termination of an entry procedure

- Local files in entry procedures, and in procedures and functions called within them, are automatically closed.
- The stack is logically released. Physically, however, a memory area is retained for subsequent calls.
- The Pascal-XT INTR handling facility is deactivated internally, but not for ILCS.
- The save area of the Pascal-XT entry procedure is unchained and released. This automatically deactivates Pascal-XT error handling under ILCS.
- Orderly return to the caller is effected, providing no error has occurred.

Note

On return from an entry procedure, the Pascal-XT runtime system continues to exist in the same status.

The heap is not released. However, the user can explicitly control the release of the heap using "mark" and "release" or functions from the predefined package HEAPSUPPORT (see appendix A.8).

The external files defined and opened in the packages are not closed. If the heap is not released, the files located there are not closed either.

Error handling

If an error occurs within the Pascal-XT program section and is not handled in that program section, it is propagated to the caller (see chapter 10, Runtime errors and error handling). In this event, the Pascal-XT program section is maintained for diagnostic purposes in the same state as when the error occurred.

If the error is subsequently handled in the calling program, the Pascal-XT program section is terminated normally via ILCS by a subprogram termination routine.

If the error is not handled by the caller and the program is aborted, the Pascal-XT program section is not terminated normally. Files are not closed.

Example

Language interfacing between Cobol and Pascal-XT

A Cobol main program calls a Pascal-XT entry procedure and passes to it two variable parameters. The entry procedure displays the transferred variables on screen and then globally modifies their values. After the return to the Cobol main program, the now modified variables are output.

Source code of the Cobol main program "COBPAS":

```
IDENTIFICATION DIVISION.
PROGRAM-ID.      COBPAS. ---------------------------------------- (01)
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS SCREEN.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  PARAMS.
    10  VAR1      PIC S9(8) COMP.
    10  VAR2      PIC X.
PROCEDURE DIVISION.
CALLPAS.
    MOVE 777 TO VAR1.
    MOVE "A" TO VAR2.
    CALL "PASUPRO" USING VAR1, VAR2. ---------------------------- (02)
ENDE.
```

```
        DISPLAY VAR1 UPON SCREEN. ──────────────────────────────── (03)
        DISPLAY VAR2 UPON SCREEN.
        STOP RUN.
```

Source code of the Pascal-XT entry procedure "PASUPRO" in the package "PAS":

Package specification:

```
PACKAGE PAS;
ENTRY PROCEDURE PASUPRO (VAR F1 : LONG_INTEGER; VAR F2 : CHAR); ──────── (04)
END.
```

Package body:

```
PACKAGE BODY PAS (OUTPUT);
PROCEDURE PASUPRO (VAR F1 : LONG_INTEGER; VAR F2 : CHAR);
BEGIN
   WRITELN ('F1: ',F1); ──────────────────────────────────────────── (05)
   WRITELN ('F2: ',F2);
   F1 := 111; ─────────────────────────────────────────────────────── (06)
   F2 := 'Z';
END;
BEGIN
END.
```

Call to the executable phase "COBOLPAS" and runtime listing:

```
 /EXEC COBOLPAS ──────────────────────────────────────────────────── (07)
 %  BLS0500 PROGRAM 'COBOLPAS', VERSION ' ' OF '...' LOADED
 F1:          777 ─────────────────────────────────────────────────── (08)
 F2: A
 00000111 ────────────────────────────────────────────────────────── (09)
 Z
```

Explanation:

(01) Name of the Cobol main program: COBPAS

(02) Call of the Pascal-XT entry procedure PASUPRO; at the same time the variables VAR1 and VAR2 are passed with the values 777 and 'A' respectively.

(03) After the return from the Pascal-XT entry procedure PASUPRO, the values of the VAR1 variables and VAR2 are displayed. See (09).

(04) Declaration of the Pascal-XT entry procedure PASUPRO in the specification of the PAS package.

(05) Names and values of the passed variables F1 and F2 are displayed. See (08).

(06) The values of the passed variables are globally modified.

(07) Startup of the executable phase COBOLPAS.

(08) Output from the Pascal-XT entry procedure.

(09) Output from the Cobol main program following the return from the Pascal-XT entry procedure.

7.4 Internal interface

Assembly language subprograms can be called via the internal interface by specifying the directive INTERNAL in the procedure or function declaration, e.g.:

```
PROCEDURE procedure-identifier (formal-parameter-list); internal

or

FUNCTION function-identifier (formal-parameter-list): type; internal;
```

The purpose of the subprograms called via the internal interface is to implement those functions which cannot be formulated in Pascal-XT. These functions include, for example, the use of special machine instructions or the invocation of operating system functions (SVCs). This interface is not intended for calling more extensive subprograms, since the call is not to be understood as a procedure call but more as the insertion of several machine instructions in place of the call. Consequently, the call can be performed more efficiently, with the benefit of considerably reduced runtimes as compared to those for the standard subprogram interface (EXTERNAL and other directives, etc., see also section 7.1).

Restrictions for parameter types

For formal parameters, all types are permitted except for conformant array schemata, files, structured types with components from one file type, parameter procedures and parameter functions.

Register conventions

R1 through R4	These registers are provided for passing parameters and returning the result of a function (see the sections on "Parameter passing" and "Functions").
F0, F2	These two floating-point registers are used for passing real values or returning the result of a function from a real type.
R9	This register can be used as a work register. Prior to return, however, it must have the same value as during the call. However, it is strongly recommended that this register is not changed, as this could lead to an error in the event of an exception.
R10, R11	These two registers must never be changed, not even temporarily (see below).
R14	Return address.
R15	Address of the called subprogram.

The other registers can be used as work registers. Upon return, only registers R9, R10 and R11 must have their original values.

Registers R10 and R11 must not be changed, otherwise exceptions which occur will not be handled correctly and proper return to the caller will no longer be possible.

Parameter passing

All parameters are passed in registers. Registers R1 through R4 and the floating-point registers F0 and F2 are used for this purpose. The number of parameters which can be passed is influenced by the type of subprogram and the types of the formal parameters. In the case of functions, a further parameter is implicitly required to transfer the result of the function (see "Functions" below). In the case of the value parameter transfer of a string (of variable length), two parameters are always passed (see "Passing of value parameters" below).

Assignment of parameters to registers takes place in accordance with the following guidelines:

- The formal parameters are assigned in the order in which they are written to the registers.
- Assignment of the registers begins with R1 or, for real values, with F0.
- A register is required for the result of a function when the function is a subprogram. If the function result is of the real type, then the function result is returned in F0 and only a parameter of a real type can be passed in register F2. In all other cases the function result is passed in register R1 and the only registers still available for parameter transfer are registers R2 through R4.
- For VAR parameter transfer the next free register Ri (where "i" = 1..4) is used (see below).
- For value transfer of a real value, the next free floating-point register is used, otherwise the next free register Ri. For variable-length strings registers Ri and Ri+1 are taken (see below).

If the registers are insufficient for transferring the parameters, an error message is issued as early as compilation time. If more parameters are to be passed, they must be combined in one record (if possible with specifications regarding the memory representation).

Passing of variable parameters

For variable parameters, the address of the actual parameter is always passed in a general-purpose register (Ri). Depending on the justification requirements of the parameter type, the address may be a byte, halfword, word or doubleword address (see section 4.3).

For variable-length strings, the address of the 2-byte length field is passed. This is directly followed by the character values.

Passing of value parameters

For value parameters, the value or the address of the actual parameter is passed, depending on the type of formal parameter. When the address is transferred, the actual parameter must never be modified, otherwise the semantic rules for the program will be violated.

(a) For ordinal types (integer types, char, Boolean, enumerated and subrange types), the value is passed. Values are held right-justified in the register.

(b) For real types, the value is transferred in a floating-point register.

(c) For a variable string type, two parameters are always transferred. In the first parameter (Ri) the address of the first character in the string is passed; in the second parameter (Ri+1), the current length of the string is passed.

(d) For all other types, the address of the actual parameter is passed.

Representation of the objects in memory

Knowledge of how objects are represented in memory is required for processing the parameters which have been passed; this is described in section 4.3.

Functions

Functions can be understood as procedures containing an additional parameter in which the result of the function is returned. This parameter is always passed first. Transfer of the function result varies according to the particular result type:

Ordinal type
: The function result must be stored in register R1.

Real type
: The function result must be stored in floating-point register F0.

Structured type
: The caller has already allocated the area for the function result and, upon invocation, transfers in register 1 the address of this area, in which the result has to be stored.

Exception handling

The occurrence of an exception (error) during execution of the called subprogram is handled as if it had occurred in the caller. From there the next competent "exception handler" is sought. To ensure that error handling functions correctly, registers R9, R10 and R11 must never be changed within the called subprogram. If even one of these registers is changed, a runtime error with unpredictable consequences is the result.

Separation of code and data

The subprogram called via the internal interface differs from Pascal subprograms in that it does not have a work area for local data. The work area is required for such operations as the saving of registers or the copying of a structured value parameter. There are several ways of creating this area:

(a) The caller transfers a further variable parameter, e.g. a field (array) which the receiver of the call can use as a work area. Thus the subprogram is shareable.

(b) A data area is statically defined in the subprogram. Thus the subprogram is not shareable.

(c) The subprogram dynamically creates its own separate data area (e.g. using REQM), and is thus shareable.

Linking in the subprogram

The external references to external subprograms are contained in the data module of the package or main program in which the subprogram has been declared. When code and data modules are linked separately, the following must be taken into account:

(a) When the program is **shareable:**

The subprogram must be explicitly linked to the code modules. When the data modules are linked, resolution of the external references to the subprogram must be explicitly ruled out.
No local work area may be declared within the subprogram.

(b) When the program is **not shareable:**

The subprogram is linked to the data modules. A local work area may be defined within this subprogram.

Example 1

In this example the function UPPER is invoked via the internal interface; the function replaces all lowercase letters in a string by uppercase letters. The main program transfers a string to the function in the form of a value parameter, and the converted string is returned as the function result. The program is loaded by DLL and started. The Tasklib must be set to the library $PASLIB-XT to resolve the external references to the runtime system modules.

```
/EXEC $ASSEMB
%  BLS0500 PROGRAM ’ASSEMB’, VERSION ’...’ OF ’...’ LOADED.
V30.0A20 OF SIEMENS BS 2000 ASSEMBLER READY

SIEMENS F-ASSEMBLER LISTING                                                   10:02:47  90-11-27   PAGE 0001
          SYMBOL   TYPE ID  ADDR  LENGTH                              EXTERNAL SYMBOL DICTIONARY

          LETTER    SD 0001 00000  00124
          UPPER     LD 0001 00000
SIEMENS F-ASSEMBLER LISTING                                                   10:02:47  90-11-27   PAGE 0002
FLAG LOCTN OBJECT CODE    ADDR1  ADDR2  STMNT M  SOURCE STATEMENT

    000000                                  1    LETTER   CSECT

    000000                                  2             ENTRY UPPER
                                            3    *
                                            4    *        This function is called via the internal interface.
                                            5    *        All lowercase letters are converted to uppercase.
                                            6    *        Conversion is only within the current length of the string.
                                            7    *        With this function the type of string is unimportant.
                                            8    *
                                            9    *
                                           10    *        PROCEDURE upper ( s: String ): String;  Internal;
                                           11    *
                                           12    *        Parameter:
                                           13    *           R1:  Address of the function result
                                           14    *           R2:  Address of the 1st character of the string
                                           15    *           R3:  Current length of the string
                                           16    *
                          000001           17    R1       EQU   1
                          000002           18    R2       EQU   2
                          000003           19    R3       EQU   3
                          00000E           20    RR       EQU   14
                          00000F           21    RX       EQU   15
                                           22    *
    000000                                 23             USING UPPER,RX
    000000 40 30 1000                      24    UPPER    STH   R3,0(0,R1)          Copy current length
    000004 5B 30 F120     000120           25             S     R3,=F’1’            and decrease by 1
    000008 07 4E                           26             BRM   RR                  Return when length negative
    00000A 44 30 F014     000014           27             EX    R3,MOVE             Copy string
    00000E 44 30 F01A     00001A           28             EX    R3,TRANS            Transform string
    000012 07 FE                           29             BR    RR

    000014 D2 00 10022000                  30    MOVE     MVC   2(0,R1),0(R2)
    00001A DC 00 1002F020        000020    31    TRANS    TR    2(0,R1),TABLE
                                           32    *
                                           33    *

                                           34    *        Converts ’a’..’z’ to ’A’..’Z’
                                           35    *
                                           36    *               0 1 2 3 4 5 6 7 8 9 A B C D E F
    000020 0001020304050607                37    TABLE    DC    X’000102030405060708090A0B0C0D0E0F’    00 - 0F
    000030 1011121314151617                38             DC    X’101112131415161718191A1B1C1D1E1F’    10 - 1F
    000040 2021222324252627                39             DC    X’202122232425262728292A2B2C2D2E2F’    20 - 2F
    000050 3031323334353637                40             DC    X’303132333435363738393A3B3C3D3E3F’    30 - 3F
    000060 4041424344454647                41             DC    X’404142434445464748494A4B4C4D4E4F’    40 - 4F
    000070 5051525354555657                42             DC    X’505152535455565758595A5B5C5D5E5F’    50 - 5F
    000080 6061626364656667                43             DC    X’606162636465666768696A6B6C6D6E6F’    60 - 6F
```

```
    000090 7071727374757677              44            DC    X'707172737475767778797A7B7C7D7E7F'    70 - 7F
    0000A0 80C1C2C3C4C5C6C7              45            DC    X'80C1C2C3C4C5C6C7C8C98A8B8C8D8E8F'    80 - 8F
    0000B0 90D1D2D3D4D5D6D7              46            DC    X'90D1D2D3D4D5D6D7D8D99A9B9C9D9E9F'    90 - 9F
    0000C0 A0A1E2E3E4E5E6E7              47            DC    X'A0A1E2E3E4E5E6E7E8E9AAABACADAEAF'    A0 - AF
    0000D0 B0B1B2B3B4B5B6B7              48            DC    X'B0B1B2B3B4B5B6B7B8B9BABBBCBDBEBF'    B0 - BF
    0000E0 C0C1C2C3C4C5C6C7              49            DC    X'C0C1C2C3C4C5C6C7C8C9CACBCCCDCECF'    C0 - CF
    0000F0 D0D1D2D3D4D5D6D7              50            DC    X'D0D1D2D3D4D5D6D7D8D9DADBDCDDDEDF'    D0 - DF
SIEMENS F-ASSEMBLER LISTING                                                             10:02:47  90-11-27   PAGE 0003
FLAG LOCTN OBJECT CODE    ADDR1  ADDR2  STMNT M  SOURCE STATEMENT
    000100 E0E1E2E3E4E5E6E7              51            DC    X'E0E1E2E3E4E5E6E7E8E9EAEBECEDEEEF'    E0 - EF
    000110 F0F1F2F3F4F5F6F7              52            DC    X'F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF'    F0 - FF

    000120 00000001                      54                  =F'1'
                                         55            END
FLAGS IN 00000 STATEMENTS, 000 PRIVILEGED FLAGS, 000 MNOTES
HIGHEST ERROR-WEIGHT : -
THIS PROGRAM WAS ASSEMBLED BY THE SIEMENS ASSEMBLER (F) V30.0A20   CORR LEVEL: (2:0000010110:0000000000)
SOURCE LIBRARY  :       :V:$USERID.PLAM.EXAMPLE
SOURCE PROGRAM  :       UPPER.ASS
SOURCE VERS/DATE:         @/901127
MODULE  LIBRARY   :     :V:$USERID.PLAM.EXAMPLE
LIBRARY  ELEMENT  :     LETTER    VER-
ASSEMBLY TIME :         0.4235  SEC.
```

```
/EXEC $USERID.PASCAL-XT
%  BLS0500 PROGRAM 'PASCALXT', VERSION '22A00' OF ...  LOADED<
%  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG.1990. ALL RIGHT
S RESERVED
//C (PLAM.EXAMPLE, TRANSFORM.PROG), *SYSOUT, *STD
*** SOURCE LISTING ***    BS2000 PASCAL-XT COMPILER  V2.2A00...

GLOBAL OPTIONS FOR THIS COMPILATION

DEBUG       =    OFF          BY DEFAULT
GENERATE    =    ON           BY DEFAULT
MAP         =    OFF          BY DEFAULT
STANDARD    =    OFF          BY DEFAULT
XREF        =    OFF          BY DEFAULT

CURRENT COMPILATION UNIT (SOURCE FILE)

               ($USERID.PLAM.EXAMPLE,TRANSFORM.PROG(*UPPER-LIMIT,S))

     1      program TRANSFORM (input, output);
     2
     3      var
     4            s : string;
     5
     6         function upper (s: string): string; internal;
     7
     8
     9      begin
    10         writeln (output, 'Please enter text');
    11         readln  (input);
    12         read    (input, s);
    13         writeln (output, upper (s));
    14      end.
```

```
**********************************************************
*                   COMPILATION SUMMARY                  *
**********************************************************
*  ERRORS DETECTED         :        0                    *
*  WARNINGS                :        0                    *
*  NOTES                   :        0                    *
*  SIZE OF CODE MODULE     :      600 BYTES              *
*  SIZE OF DATA MODULE     :      356 BYTES              *
*  COMPILATION TIME        :    0.379 SEC                *
**********************************************************
   >>> COMPILATION SUCCESSFUL (WARNINGS: 0;  NOTES: 0)
//END
   END OF THE PASCAL SESSION  -  USED TIME = 0.450 SECONDS
```

Execution of the program

```
/SYSFILE TASKLIB=$USERID.PASLIB-XT
/EXEC (TRANSFO, PLAM.EXAMPLE)
%  BLS0001 *** DBL VERSION 067 RUNNING ***
%  BLS0517 MODULE 'TRANSFO' LOADED
Please enter text
123aAbBccCdD$=Q
123AABBCCCDD$=Q
```

Example 2

In this example the subprogram LINK for dynamically loading a program is called via the internal interface. The main program LOADER readies the definition for the LINK macro (see also the DLINK macro [6]) and transfers it in the form of a parameter. In the USERBLK parameter, the subprogram LINK returns the list of modules loaded by DLL.
As a demonstration, the program TRANSFORM from example 1 is loaded from the library PLAM.EXAMPLE and the list of loaded modules is output.

```
/EXEC $USERID.PASCAL-XT
%  BLS0500 PROGRAM 'PASCALXT', VERSION '22A00' OF ...  LOADED<
%  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG.1990.
ALL RIGHTS RESERVED
//C (PLAM.EXAMPLE, LINK.PROG), *SYSOUT, *STD
*** SOURCE LISTING ***    BS2000 PASCAL-XT COMPILER  V2.2A00...

GLOBAL OPTIONS FOR THIS COMPILATION

DEBUG       =    OFF          BY DEFAULT
GENERATE    =    ON           BY DEFAULT
MAP         =    OFF          BY DEFAULT
STANDARD    =    OFF          BY DEFAULT
XREF        =    OFF          BY DEFAULT

CURRENT COMPILATION UNIT (SOURCE FILE)

               ($USERID.PLAM.EXAMPLE,LINK.PROG(*UPPER-LIMIT,S))

     1      program LOADER (input, output);
     2
     3
     4      {***     General definitions    ***}
     5
     6      type
     7         symbol    = packed array [1..8]  of char;
     8         filename  = packed array [1..54] of char;
     9
    10
    11      {***     Structure of table for external references list    ***}
    12
    13      type
    14         maxsym  = 1 .. 100;
    15         symtab  = record
    16                    len   : short_integer;        { Record length field }
    17                    res1  : short_integer;        { Reserved            }
    18                    sym   : array [maxsym] of symbol;
    19                   end;
    20
    21      const
    22         end_of_table = symbol (#'00':8);
    23
    24      {***     List of operands for the LINK macro    ***}
    25
    26      const
```

```
27        krzinhbt = #01;
28        krztsklb = #02;
29        krzlibnm = #04;
30        krzuserb = #40;
31
32        defaults = krzinhbt + krztsklb + krzlibnm + krzuserb;
33
34     type
35        byte    = 0 .. 255;
36        linkop  = record
37                    krzcontr  : byte;
38                    krzentry  : symbol;
39                    krzlibry  : filename;
40                    krzform   : byte;
41                    case boolean of
42                      true:  (krzpage : short_integer);
43                      false: (krzmpid : long_integer);
44                    end;
45
46
47     var
48        userblk   : symtab;
49        linkdescr : linkop;
50        result    : integer;
51
52     {***        Return codes          ***}
53
54     const
55        error  =  #FFFFFFFF;   { Invalid operands or address          }
56        ok     =  0;           { Module loaded, address in register 1 }
57
58
59     procedure link (     descr   : linkop;
60                          tab     : symtab;
61                      var retcode : integer ); internal;
62
63
64     function loaded: boolean;
65      begin
66         if result = error then begin
67            loaded  := false;
68            writeln (output, 'Operand error or invalid address');
69          end
70         else
71            loaded := (result mod 256) = ok;
72      end { loaded };
73
74
75     procedure print_list;
76      var
77          i : maxsym;
78      begin
79         writeln (output, 'List of the modules loaded by DLL:');
80         for i := first (maxsym) to last (maxsym) do
81            if userblk.sym [i] <> end_of_table then
82               writeln (output, '   ', userblk.sym [i]);
83      end { print_list };
84
```

```
        85
        86
        87      begin  { LOADER }
        88         userblk.len := sizeof (symtab);
        89         with linkdescr do begin
        90            krzcontr := defaults;
        91            krzentry := 'TRANSFO ';
        92            krzlibry := 'PLAM.EXAMPLE                                             ';
        93            end;
        94         link (linkdescr, userblk, result);
        95         if loaded then
        96            print_list;
        97      end.


************************************************************
*                  COMPILATION SUMMARY                     *
************************************************************
*  ERRORS DETECTED         :        0                      *
*  WARNINGS                :        0                      *
*  NOTES                   :        0                      *
*  SIZE OF CODE MODULE     :     1156 BYTES                *
*  SIZE OF DATA MODULE     :      980 BYTES                *
*  COMPILATION TIME        :    0.532 SEC                  *
************************************************************
   >>> COMPILATION SUCCESSFUL (WARNINGS: 0;  NOTES: 0)
//END
   END OF THE PASCAL SESSION  -  USED TIME = 0.602 SECONDS


/EXEC $ASSEMB
%  BLS0500 PROGRAM 'ASSEMB', VERSION '300' OF '89-11-03' LOADED.
V30.0A20 OF SIEMENS BS 2000 ASSEMBLER READY

SIEMENS F-ASSEMBLER LISTING                                                 10:02:16  90-11-27   PAGE 0001
          SYMBOL   TYPE ID  ADDR  LENGTH                              EXTERNAL SYMBOL DICTIONARY

          LNKCALL   SD 0001 00000  00008
          LINK      LD 0001 00000
SIEMENS F-ASSEMBLER LISTING                                                 10:02:16  90-11-27   PAGE 0002
FLAG LOCTN OBJECT CODE    ADDR1  ADDR2  STMNT M  SOURCE STATEMENT

    000000                                 1    LNKCALL  CSECT

    000000                                 2             ENTRY LINK
                                           3    *
                                           4    *        This subprogram is called via the internal interface.
                                           5    *        It calls the LINK macro.
                                           6    *        Registers R0, R1 and R15 are required for the LINK macro.
                                           7    *
                                           8    *        PROCEDURE LINK (     operands : { a record type  };
                                           9    *                             userblk  : { a record type  };
                                          10    *                         var result   : Integer );
                                          11    *        Parameter:
                                          12    *           R1:  Address of the operand list
                                          13    *           R2:  Address of the user block
                                          14    *           R3:  Address of the result
                                          15    *
                      000000              16    R0       EQU   0
                      000001              17    PAR1     EQU   1                1st parameter
                      000002              18    PAR2     EQU   2                2nd parameter
                      000003              19    PAR3     EQU   3                3rd parameter
                      00000E              20    RR       EQU   14
                      00000F              21    RX       EQU   15           ,
                                          22    *
```

```
    000000 18 02                            23    LINK     LR    R0,PAR2          Load USERBLK address
                                            24             LINK  MF=(E,(1))       Call the LINK macro
                                            25  1          #INTF REFTYPE=REQUEST,INTNAME=LINK,INTCOMP=1       00019000
    000002 0A 6E                            26  1          SVC   110                                         00064000
    000004 18 3F                            27             LR    PAR3,RX          Result in the 3rd parameter
    000006 07 FE                            28             BR    RR

                                            29             END
FLAGS IN 00000 STATEMENTS, 000 PRIVILEGED FLAGS, 000 MNOTES
HIGHEST ERROR-WEIGHT : -
THIS PROGRAM WAS ASSEMBLED BY THE SIEMENS ASSEMBLER (F) V30.0A20   CORR LEVEL: (2:0000010110:0000000000)
SYSTEM MACROLIBRARY :  :D:$TSOS.MACROLIB
SOURCE LIBRARY  :      :V:$USERID.PLAM.EXAMPLE
SOURCE PROGRAM  :      LINK.ASS
SOURCE VERS/DATE:        @/901127
MODULE  LIBRARY   :    :V:$USERID.PLAM.EXAMPLE
LIBRARY  ELEMENT  :    LNKCALL   VER-
ASSEMBLY TIME :        0.8187  SEC.
```

Execution of the program

```
/SYSFILE TASKLIB=$USERID.PASLIB-XT
/EXEC (LOADER, PLAM.EXAMPLE)
%  BLS0001 *** DBL VERSION 067 RUNNING ***
%  BLS0517 MODULE 'LOADER' LOADED
List of modules loaded by DLL:
   TRANSFO
   TRANSFOC
   TRANSFOD
   LETTER
```

8 UTM linkage

As of UTM V3.0, UTM program units can also be written in Pascal-XT. This chapter gives a brief outline of the special points and marginal conditions that have to be observed when UTM program units are implemented in Pascal-XT. For more detailed descriptions the following UTM manuals should be referred to:

- "Planning and Design" [9]
- "Generating and Administering Applications" [10]
- "Programming Applications in Pascal-XT" [11]

The COMP operand must be set as follows in the program statement for the generation procedure KDCROOT:
Pascal-XT V2.1 or earlier COMP = PASCAL
Pascal-XT V2.2 or higher COMP = ILCS

UTM program units are called as subprograms by the UTM linkage program. Communication between the called program unit and the other components of the UTM application is effected exclusively by calling the external procedure KDCS in the KDCROOT linkage program. KDCS is an abbreviation of the German equivalent for "compatible data communication interface".

Pascal-XT V2.2A makes exclusive use of ILCS (Inter-Language Communication Services, see 7.1) as an interface to UTM; earlier Pascal-XT versions use IUTMHLL. Pascal-XT V2.2A therefore requires UTM V3.2. Earlier Pascal-XT programs can also run under UTM V3.2, as this version supports both the IUTMHLL and the ILCS interface:

	UTM V3.1	UTM V3.2
Up to Pascal-XT V2.1	IUTMHLL	IUTMHLL
As of Pascal-XT V2.2	——	ILCS

Table 8-1: Possible Pascal-XT / UTM combinations

The UTM linkage program KDCROOT calls the Pascal-XT program unit in accordance with ILCS convention. UTM program units written in Pascal-XT should therefore satisfy the requirements outlined in chapter 7 ("Language interfaces"):

- UTM program units must be declared as entry procedures in a package specification.

- The external procedure KDCS must be declared by means of the COBOL directive.

Language interfaces under UTM

As of Pascal-XT V2.2A, external subprograms which are not UTM program units may be called from a UTM program unit written in Pascal-XT. If these external subprograms use ILCS, they can also use the ILCS error handling facility.

The reverse also applies: Pascal-XT entry routines may be called from external UTM program units which use ILCS. In this case, the Pascal-XT entry routines are not separate UTM program units.

Data types and constants

The data structures and constants used in parameters when calling the external procedure KDCS are available to the user in the form of predefined packages:

- KCKBL contains
  - the definition of the KDCS operation code
  - the type declaration for the heading of the communication area
  - the type declaration for the parameter field
- KCINL contains
  - the type declaration for the return information of the INFO call
- KCDFL contains
  - the constant definitions for the screen output functions
- FIELD_ATTRIBUTE_PACKAGE contains
  - the type and constant declarations for the field attributes when the terminal or terminals operate in format mode
- KCMSL contains
  - the constant and type declarations for the UTM messages.

Note

Though the package bodies of the above-mentioned packages are all "empty", they must be compiled and linked when the program units are being linked, since they are accessed when the Pascal-XT environment is initialized. (As the Pascal-XT compiler reads only the specifications of the referenced packages, it does not "know" whether a package body is empty.)

Basic structure of UTM program units

The number and structure of procedure parameters are defined to a large extent by UTM (see the manual "Programming Applications in Pascal-XT" [11]). The package specifications for UTM program units must therefore conform to the following pattern:

```
with KCKBL;
from KCKBL use kckb, kcpal;
...
package UTM_TP;
...
type
   ...
   t_progb = ...;
   ...

   t_kckb  = record          (* Communication area *)
      kbhead: kckb;          (* ..- Heading *)
      progb : t_progb;       (* Program area *)
      end;
   ...

   t_nb = ...;
   ...

   t_spab = record           (* Standard primary work area *)
      parm: kcpal;           (* Parameter field *)
      nb  : t_nb;            (* Message area *)
      end;
   ...

   procedure KDCS (var p: kcpal; var n: t_nb); cobol;

   (* Note: A component type of the message area can also be used for the
            second parameter of the KDCS procedure if the message area has
            been defined as a record type *)

   ...

   entry procedure program unit_1 (var kb: t_kckb; var spab: t_spab);
   ...
   entry procedure program unit_n (var kb: t_kckb; var spab: t_spab);

   (* Note: Program units using the same parameter layout can be
            combined into one package *)
end (* UTM_TP *).
```

As regards the package body of UTM program units the following points should be noted:

- Except for the user exits START, SHUT and VORGANG, in which no KDCS calls are allowed, it is not possible effectively to use the current parameters of the entry procedure before the first KDCS call (with operation code INIT).
- Except for the user exits START, SHUT and VORGANG, communication with the UTM linkage program must be terminated using a KDSC call with the operation code PEND.
  Subsequently UTM does not return to the entry procedure.
  This has the following consequences:
  - The entry procedure should not be terminated normally, otherwise a PEND error will occur.
  - The **return** statement should not be used in the outer block of such an entry procedure, otherwise a PEND error will occur.
  - Local files declared in the outer block of such an entry procedure are not closed after the PEND call and can therefore only be used in inner blocks.

Requesting memory with "NEW"

As of Pascal-XT V2.2A, UTM program units written in Pascal-XT can request memory dynamically with "New" (see [1], 15.2). The user is responsible for subsequently releasing the memory again using "dispose", "release" or "Heapsupport.Release_Heap" before the KDCS call PEND. The Pascal-XT runtime system does not release dynamically requested memory either at the end of the UTM program unit or at the end of the total UTM application. Even if memory requested with "New" is not released, it cannot be accessed by a subsequent program because with UTM this program could be running under another task. Memory requested with "New" is not available for any other task.

External files

When external files are used in UTM program units it should be borne in mind that these are not closed at end of program (like those in a Pascal-XT main program). The user is responsible for the opening and closing of files. This can be done effectively in an entry procedure for the user exit SHUT. Local files from subprograms (procedures and functions) are closed by the Pascal system.

Note

If program units of a UTM application are to be implemented in various packages and if the same external file is used in more than one package, it is only possible for it to be declared in one package in the program parameter list and as a file variable. In other packages it must be imported by means of a context specification (WITH/USE list).

Error handling as of Pascal-XT V2.2A

When a UTM program unit written in Pascal-XT is called, the Pascal-XT runtime system signs on to the ILCS error handling facility. If an error occurs while the Pascal-XT program unit is executing or if an error is propagated from an external subprogam, the runtime system searches the program unit for an exception handler. If an exception handler is defined, the Pascal-XT program unit is continued from that point.
If no exception handler is defined, the Pascal-XT runtime system propagates the error to the caller (UTM). UTM outputs a user-dump and terminates the process with a KDCS call PEND /Error.

Error handling up to Pascal-XT V2.2A

The UTM linkage program KDCROOT incorporates its own error handling facility. This ensures that, if an error occurs (STXIT events), all UTM and data base transactions that are not closed will be reset (UTM-S only) and provides for the possibility of terminating the application in an orderly manner.

If a UTM application contains at least one Pascal-XT program unit, a Pascal-XT-specific linkage module is linked in to perform the following actions:

- Calling a Pascal-XT program unit dynamically for the first time causes the Pascal-XT runtime system to be informed that STXIT handling is performed by UTM; Pascal-XT STXIT handling is not activated.
- When an STXIT event occurs during the active phase of a Pascal-XT program unit (in Pascal-XT code or during the execution of a KDCS call), the error handling facility of the Pascal-XT runtime system is invoked (if PROCHK, ERROR and TIMER events occur). If the error in the Pascal-XT program unit is not handled by an exception handler, UTM error handling is continued after the cause of error and the dynamic call chain have been output by the Pascal-XT runtime system.

– The Pascal-XT runtime system reports to UTM the first line of the dynamic call chain if during the execution of a Pascal-XT program unit a runtime error occurs that is not handled by an exception handler. UTM then outputs this line together with the UTM dump.

Note

This behavior enables the user to handle any error conditions (by using the standard procedure RAISE) reported by UTM in the return field of the communication area.

Linking the application

No special action is required when linking a UTM application. The information given in chapter 6, "Linking and executing object programs", applies.

The predefined package UTM_ADAPTER is no longer required as of Pascal-XT V2.2A, but it is still available so that existing UTM program units written in Pascal-XT do not have to be recompiled.

9 Debugging aid PATH

The debugging aid PATH is specific to Pascal-XT and is a component of every Pascal-XT implementation. The user interface is the same for every implementation except for some minor differences dictated by the different operating systems.

Sections 9.1 through 9.3 describe the general characteristics of the debugging aid PATH, i.e. not those specific to BS2000. Sections 9.4 and 9.5 describe the way in which PATH is used under BS2000.

9.1 Features and characteristics of PATH

PATH is an interactive Pascal debugging aid which permits the symbolic testing of Pascal-XT in interactive mode or in prepared test runs. In order to test a Pascal-XT compilation unit by means of PATH, the option DEBUG=ON or DEBUG=RESTRICTED must be set when the compilation unit is compiled with the Pascal-XT compiler so that a test table can be generated.

Writing of multiple Pascal-XT statements on one line of the source program is not recommended (see section 9.1.4.2).

The most important functions of PATH include:

- halting of Pascal programs
- display and modification of program data
- conditional execution of commands
- display of the dynamic call chain (call history)

Program data is addressed symbolically (by using the names defined in the source program) and in accordance with its type.

In particular, PATH reproduces the types from Pascal-XT (e.g. VAR COLOR: (RED, YELLOW, GREEN) in its output; "COLOR:=RED;" is output as COLOR=RED).

In addition, PATH takes into account the block-oriented structure of Pascal and, during testing, reflects the scope of identifiers defined by the nesting of procedures. In the same way, PATH also permits testing of recursive procedures and functions of a program.

When language interfaces are used for a mix of programs, Pascal-XT packages which are called by non-Pascal-XT programs via the ENTRY interface can also be tested using PATH.

To use PATH, no knowledge of the object code generated by the Pascal-XT compiler is necessary.

As long as no debugging aid commands have been issued, PATH has no effect on the execution time of the program (apart from the time required for the initialization of PATH).

PATH executes interactively, has a syntax similar to Pascal, is free-format and recursive, and permits structured debugging aid commands (compound commands). There are abbreviations for the most commonly used commands (highlighted in the format description). To the extent that they are not Pascal keywords, the command names and abbreviations may also be referred to as program identifiers.

The compiler’s conventions with respect to uppercase and lowercase characters, alternate symbols, hexadecimal specification of integer, character and string literals, and underscore characters in identifiers apply analogously to the debugging aid. Like the Pascal-XT compiler, PATH detects syntactic and semantic errors. If an invalid command is entered, it is output up to the point at which the error was detected. The rest of the input line is not analyzed and is replaced in the output by three periods. The invalid command and any new commands in which it is contained must be reentered.

If errors occur in the execution of debugging aid commands, this is reported and the command is executed only in part or not executed at all.

All debugging aid output is given a prefix to make it easier to distinguish it from program output. In the BS2000 version of PATH and the examples used in the present manual "%%" is used as the prefix.

9.1.1 Command summary

PATH commands are classed as either testpoint commands or action commands. They should be terminated with a semicolon. Their syntax formats and precise meanings are described in 9.2.

TESTPOINT COMMANDS:

AT	Sets a testpoint and stores a debugging aid command which is executed each time the testpoint is reached. AT always needs to be specified with another PATH command.
REMOVE	Deletes the specified testpoints (set with AT).
SLEEP	Deactivates the commands set (with AT) for the specified testpoints.
AWAKE	Activates the commands deactivated (with SLEEP) for the specified testpoints.
GETCMD	Interrupts the test run and requests input of further commands.
RESUME	Continues at the point interrupted by GETCMD.

ACTION COMMANDS:

DISPLAY	Outputs data to the PATH output medium.
ASSIGN	Assigns values to variables of the program.
IF	Specifies conditional execution of debugging aid commands.
BEGIN ... END	Unites the debugging aid commands appearing between BEGIN and END into one (compound) command.
SYSTEM	Switches to system mode or executes the system command enclosed in apostrophes.
EDIT	Calls an editor.
SHOW	Outputs status information (SHOW UNITS, SHOW CALLS, SHOW WHERE).
DUMP	Outputs all variables of the dynamic chain (the variables on the stack) and the global variables of all packages to the PATH output medium.
KILL	Aborts execution of the program under test.
SWITCH	Reassigns PATH input and output files.

9.1.2 Definition of terms

The terms explained here are frequently used in the following sections.

- **Program under test**

  The Pascal-XT program to be tested, including all associated packages.

- **Compilation unit**

  Main program or package (= specification and body together).

- **Current compilation unit**

  Compilation unit in which the current testpoint (see 9.1.4) is located.

- **Global scope (visibility)**

  This means that only the names of all compilation units as well as predefined names (e.g.: Maxint) are visible. In this case, names declared within compilation units can be referenced only when prefixed by a block qualification (see section 9.1.3).

- **Potential testpoint**

  Point at which a testpoint can be set (e.g. at the beginning of most statements). Which beginnings of statements represent potential testpoints, and whether other points (e.g. ends of procedures) represent potential testpoints as well, is machine-dependent.

- **Block**

  Main program, package, procedure or function.

9.1.3 Syntax elements

The following syntactic units frequently appear in PATH commands. The use of these syntactic units is described in section 9.2.

- **Block qualification**

  Sequence of names which is terminated with a period. The individual names are separated from one another by a period. The first name may be that of a compilation unit or of a subprogram visible from the current testpoint. Each additional name designates a procedure or function that is declared immediately within the block as qualified up to that point (see section 9.1.2).

  Examples

  ```
  procx.procy.funcz.
  unity.procx.funcx.
  unity.
  ```

- **Incarnation qualification**

  Block qualification followed by an incarnation number and a period. An incarnation number consists of a % character immediately followed by an integer without a sign. When a procedure is called recursively, different incarnations of the subprogram arise; with %i, a specific incarnation can be referenced; %1 designates the "most current" incarnation.

  Examples

  ```
  procx.%2.
  procx.funcy.%3.
  unity.procx.%4.
  ```

- **Testpoint specification**

  List of potential testpoints (see section 9.1.2). It consists of line numbers and/or line number ranges from the compiler listing separated by commas. In the simplest case, this list consists of one line number.

  It is also possible to write %ALL as the line number range in order to specify all potential testpoints of a compilation unit or subprogram.

  Examples

  ```
  25
  30..40
  25, 30, 35..45, 50
  %ALL
  ```

  Each line number or line number range may be prefixed by a block qualification.

If the line number range is not %ALL, then the block qualification may designate a compilation unit but not a subprogram.

Before %ALL, however, not only are block qualifications permissible that designate a compilation unit (which causes all potential testpoints of this compilation unit including all subprograms declared in it to be specified) but also those that designate a subprogram (which causes all potential testpoints of this subprogram but not those in subprograms contained in it to be specified).

Examples

```
unity.30
unitx.35, unity.42, unitz.28..35
unitx.%ALL, unity.100..108, unitz.%ALL
unitx.procy.funcz.%ALL
procx.%ALL
```

It is also possible to combine line numbers or line range numbers of a compilation unit by enclosing in parentheses the list of individual line numbers or line number ranges for the compilation unit.

Example

```
unity.123, 128..134, unitz.(15, 25, 35..40), 78
```

In this example, potential testpoints are specified in line 123 of the compilation unit "unity", in lines 78 and 128 through 134 of the current compilation unit, and in lines 15, 25 and 35 through 40 of the compilation unit "unitz".

In addition to this type of testpoint specification (i.e. an enumeration of line numbers or line number ranges), there is also the testpoint specification %ALL_UNITS, which defines all potential testpoints of all testable compilation units of the program under test.

At the end of the testpoint specification (i.e. at the end of the entire list) ": ENTER" may be added. The effect of this is the same as for the potential testpoints specified in the list, with the difference that only those testpoints are selected which are the first potential testpoints of any block (see section 9.1.2).

Examples

```
%ALL_UNITS: ENTER
unitx.%ALL, unity.30..100: ENTER
```

The meanings of the above forms of testpoint specification are described in detail in section 9.1.4.2.

Furthermore, it is possible to specify the keyword EXCEPTION as a testpoint specification. This potential testpoint is called "exception testpoint" (see section 9.1.4.4).

Note

The keyword EXCEPTION must not occur together with line numbers, line number ranges, %ALL, %ALL_UNITS, block qualifications or :ENTER in the same testpoint specification.

- **Factor**

Integer, real, character or string literal, a constant or variable declared in the program under test, or a qualified set constructor. Structure constants and variables may be indexed ([ ]), qualified (.) or dereferenced (@). An index in turn may itself be a factor. Furthermore a subrange of an array or of a string may be referred to by specification of a slice ([..]) whose limits are themselves factors. A constant or variable may be prefixed by a block qualification.

Examples

```
unity.procz.i
unitx.a[4, b[i], 2..k]
p[j]@.x@.y[l]
a[1..10]
char_set_type ([’A’, ’X’..’Z’])
int_set_type ([5..i, k, v[1]..p.x, 100])
```

In the first two examples, a block qualification is specified by "unity.procz" and "unitx".

It is possible to output one-dimensional array and string slices e.g. A [7..10]. Assignment and comparison of array slices, on the other hand, are not permitted. It is likewise not permitted to slice slices even further or to index them.

Example

```
VAR A: ARRAY [1..10, 1..20] OF STRING[50];
```

Permitted:

```
DISPLAY (A, A[5], A[6, 15, 43], A[7, 8, 30..40], A[I, J..16], A[2..8]);
```

Not permitted:

```
DISPLAY (A[3..4, 15], A[5..9, 10..20], A[I, 12..17][10..30]);
```

All of the above also applies to variables referred to via an incarnation qualification.

9.1.3.1 Debugging aid comments and options

For the purpose of documenting test runs, both in debugging aid commands or in Pascal-XT programs, comments in the form (*comments*) or {comments} may be inserted.

Note

In contrast to Pascal-XT, in PATH commands a comment must not exceed the limits of a line.

Furthermore it is also possible to specify debugging aid options in the form of pseudo-comments. At present only one option is implemented; in later versions others might also be added.

`{$R-}` Only debugging aid commands recognized (at analysis time) as containing errors are output to the PATH output medium, in the form of requests for repeat input.

`{$R+}` All debugging aid commands entered are repeated on the PATH output medium (default value).

9.1.4 Testpoints

PATH actions take place exclusively at "testpoints". There are the following types of testpoints:

Testpoint before program start (preliminary testpoint)
User-set testpoint
Postmortem testpoint
Exception testpoint
Entry testpoint

9.1.4.1 Testpoints before program start

Before the packages belonging to a program are initialized and the main program is executed, a GETCMD command is executed at an implicit testpoint. This permits the user to enter PATH commands (in particular, to set testpoints in the different compilation units). Since a preliminary testpoint is not located in any particular package, global scope (visibility) applies (see section 9.1.2).

Messages at the testpoint before program start:

```
%% testpoint before program start
%% scope seen from outside all compilation units ———————————————— (01)
* —————————————————————————————————————————————————————————————— (02)
```

(01) Global scope.

(02) Command input prompt.

9.1.4.2 User-set testpoints

The testpoint commands AT, SLEEP, AWAKE and REMOVE as well as SHOW WHERE are used to specify, by means of a testpoint specification (see section 9.1.3) the points (potential testpoints) in the program to be tested at which testpoints are to be set, deleted or displayed.

In the following, only the setting of testpoints (AT command) is discussed; testpoint specifications in the other commands mentioned have analogous meanings.

In the simplest case (setting a single testpoint in the current compilation unit, see section 9.1.2), the testpoint specification consists solely of a line number, which can be taken from the compiler listing.

The testpoint is then set in that line at the first potential testpoint (see section 9.1.2).

If the specified line contains no potential testpoints, an error is reported.

Example

```
1    {$DEBUG} PROGRAM STATEMENTS;
2    VAR I: INTEGER;
3    BEGIN
4       I := 1;
5       I := 2; I := 3; I
6       := 4;
7    END.
```

AT 1 DO ...; results in an error message;
AT 2 DO ...; results in an error message;
AT 3 DO ...; results in an error message;

AT 4 DO ...; sets a testpoint before `I` := 1;
AT 5 DO ...; sets a testpoint before `I` := 2;

AT 6 DO ...; results in an error message;
AT 7 DO ...; sets a testpoint before END.

The statements I := 3; and I := 4; in line 5 cannot be directly (individually) referenced by the debugging aid.

Therefore, in programs that are to be tested it is advisable to use one line for each statement.

It is also possible with one AT command to set testpoints at several consecutive potential testpoints, by specifying a line number range in the form "line number .. line number" or in the form "%ALL".

Thus, in the above example, as the result of the command

```
AT 5..6 DO ...;
```

a testpoint is set before each of the statements I := 2; I := 3; and I := 4; (i.e. 3 testpoints).

With the command

```
AT %ALL DO ...;
```

a testpoint is set before the statements I := 1; I := 2; I := 3; I := 4 and before END (i.e. 5 testpoints).

Thus it is also possible to set testpoints before statements that do not begin on a separate line (e.g. I :=3;). As stated, however, such statements cannot be referenced individually, but instead only within a line number range.

In this example, with the command

```
AT %ALL: ENTER DO ...;
```

a testpoint is set only in line 4 (before the statement I :=1;).

By specifying a testpoint specification as a series of line numbers and/or line number ranges separated by commas, it is possible with one AT command to set testpoints at several, not necessarily consecutive statements.

Example

```
AT 550, 576..612, 434, 639 DO ...;
```

All the formats described thus far allow testpoints to be set only in the current compilation unit. However, the syntax of testpoint specifications also permits potential testpoints in other compilation units of the program under test to be referenced.

Example

```
AT 125, unitx.30, unity.(120, 130..135), 150, unitz.%ALL DO ...;
```

sets testpoints at the first potential testpoint in lines 125 and 150 of the current compilation unit, in line 30 of the compilation unit "unitx", in line 120 and at consecutive potential testpoints in lines 130 through 135 of the compilation unit "unity", as well as at all potential testpoints in the compilation unit "unitz".

If ": ENTER" is added at the end of the above testpoint specification, i.e.

```
AT 125, unitx.30, unity.(120, 130..135), 150, unitz.%ALL: ENTER DO ...;
```

then of the potential testpoints specified, only those that are the first potential testpoints of a block (see section 9.1.2) are selected.

Example

```
AT unitz.(87, 316..318, 772..775, 961) DO ...;
```

The parentheses in this example are necessary, since the command

```
AT unitz.87, 316..318, 772..775, 961 DO ...;
```

would set testpoints at the first potential testpoint in line 87 of the compilation unit "unitz", as well as in lines 316 through 318, 772 through 775, and 961 of the current compilation unit.

The testpoint specification %ALL_UNITS specifies all potential testpoints of all compilation units of the program under test that were compiled with the option DEBUG=ON or DEBUG=RESTRICTED and whose test tables are available.
In particular, %ALL and %ALL_UNITS are suitable for deleting (REMOVE command), deactivating (SLEEP command) or activating (AWAKE command) all user-set testpoints of a compilation unit or of all compilation units of the program under test.

9.1.4.3 Postmortem testpoints

If, when testing a program with PATH (see section 7), a runtime error occurs in a Pascal-XT program section or is passed to it from an external subprogram, and if the runtime error is not handled in that program section, the following steps are taken:

(1) If the program section tested is the main program, the Pascal-XT runtime system displays the cause of the error together with the dynamic call chain (see 10.2.4). Step (2) is then performed.

(2) The debugging aid behaves in each case as if a user-set testpoint with an input prompt (GETCMD command) were set at one of the following points:
- at the statement in which the error occurs.
- at the point at which the external subprogram which propagated the runtime error is called.

PATH then displays the input prompt * and the user can enter PATH commands, in particular DUMP, SHOW CALLS and DISPLAY, to identify the cause of the error.

The PATH command KILL causes the program to abort.

The effect of the PATH command RESUME differs according to the nature of the Pascal-XT program section tested.

In a main program the program under test is terminated if no new debugging run (see 9.5 "Restart) is requested.

In another program section, the Pascal-XT runtime system propagates the error to the caller. Error propagation makes it possible for other Pascal-XT program sections to be informed of the error. The debugging aid then offers a postmortem testpoint to any Pascal-XT program section which propagates the error without handling it.

Example

```
 1      {$DEBUG=ON} PROGRAM DIVIDE
 2      VAR
 3         dividend:  Integer;
 4         divisor:   Integer;
 5         result:    REAL;
 6
 7      BEGIN
 8         ⎡ Allocation of a value to the      ⎤
 9         ⎨ variables dividend and divisor is ⎬
10         ⎣ achieved from the debugging aid   ⎦
11         result := dividend / divisor;
12      END.
```

After program start:

```
%% testpoint before program start
%% scope seen from outside all compilation units
*at divide.11 do getcmd;
*resume
%% program continued
%% testpoint at line DIVIDE.11
*assign dividend := 5;
*assign divisor := 0; {provokes division by 0 in program under test}
*resume;
%% program continued
 NUMERIC_ERROR ( 104) RAISED FROM  DIVIDE.DIVIDE  AT 000E5F3A
%% testpoint because of unhandled Numeric_Error at line DIVIDE.11
*dump;
%% global variables of DIVIDE:
%%    dividend = 5
%%    divisor = 0
%%    result = 0.000000000000000E+00
*resume;
%% program aborted
```

9.1.4.4 Exception testpoints

The testpoint commands AT, SLEEP, AWAKE, REMOVE and SHOW WHERE permit a so-called exception testpoint to be set, deactivated, (re)activated, deleted and displayed by specifying the keyword EXCEPTION.

When an exception testpoint is set and a runtime error occurs in a Pascal-XT program section (see section 7) or is passed to it from an external subprogram, the debugging aid issues a message at the following points:

- the statement in which the error occurs
- the position in all Pascal-XT program sections at which the external subprogram passing on the error is called.

Irrespective of whether an exception handler has been defined for the error, an exception testpoint is activated and the PATH commands specified for it are executed. This permits the user to identify the cause of the error even if the error is dealt with by an exception handler. Only then is an exception handler sought, provided that the program under test is not terminated by the PATH command KILL. If an exception handler has been defined, the program under test will continue from that point. If no exception handler has been defined, a postmortem testpoint is activated.

Example

```
 1     {$DEBUG=ON} PROGRAM POWER;
 2     VAR
 3        result: Real;
 4
 5     FUNCTION power_of_e (x: Integer): Real;
 6     BEGIN
 7        power_of_e := Exp (x);
 8     EXCEPTION
 9        IF Error_Number = Numeric_Error THEN
10           power_of_e := Maxreal
11        ELSE { propagate other errors }
12           Raise (Error_Number);
13     END;
14
15     BEGIN
16        result := power_of_e (maxint);
17     END.
```

After program start:

```
%% testpoint before program start
%% scope seen from outside all compilation units
*at exception do display ('Exception occurred'); ———————————————————————— (01)
*at power.16 do getcmd;
*resume;
%% program continued
%% testpoint at line POWER.16
*show where exception; —————————————————————————————————————————————————— (02)
%% testpoint at exception ———————————————————————————————————————————————— (03)
*at exception do begin dump; getcmd; end; ——————————————————————————————— (04)
*resume;
%% program continued
%% testpoint because of Numeric_Error at line POWER.7 (power_of_e) ——————— (05)
%% 'Exception occurred' —————————————————————————————————————————————————— (06)
%% parameters and local variables of power_of_e: ————————————————————————— (07)
%%    x = 2147483647
%% global variables of POWER;
%%    result = 0.000000000000000E+00
*resume; ———————————————————————————————————————————————————————————————— (08)
%% program continued
```

(01) If an exception occurs in the program under test, the testpoint message must be issued (05) and the DISPLAY command executed (06).

(02) Has the exception testpoint been set?

(03) Indicates that the exception testpoint has been set.

(04) The command indicated in (01) and to be executed if an error occurs, is expanded to include a DUMP and a GETCMD command.

(05) Testpoint message issued at exception testpoint.

(06) Output of the DISPLAY command defined in (01).

(07) Output of the DUMP command defined in (04).

(08) Execution of the GETCMD command defined in (04) causes the test run to be interrupted; it is resumed by entering the RESUME command.

9.1.4.5 Entry testpoints

If a Pascal-XT entry procedure is called from a program segment written in another language and the Pascal-XT package that contains the procedure has not yet been made known to the debugging aid, then, before the package is initialized, a GETCMD command is executed at an implicit testpoint (entry testpoint). This enables the user to enter PATH commands, and, in particular, to define testpoints both in this package and in all the packages referenced via WITH clauses.

"Global scope" applies at the entry testpoint (see section 9.1.2) and the following messages appear:

```
testpoint before entry call
scope seen from outside all compilation units
```

On subsequent calls to the same entry procedure or to another entry procedure in a package already known to the debugging aid, no further entry testpoint is created since the user has already had the opportunity to set testpoints in the package and the many testpoints that would result if there were frequent calls to entry procedures would place a burden on processing capacity.

Example

The assembly-language main program calls the Pascal-XT entry procedure 'divide' which performs a division calculation.

```
Assembly language main program:

 1     CALLDIV  CSECT
 2     *
 3              EXTRN DIVIDE
 4     *
 5     R0       EQU   0
 6     R1       EQU   1
 7     R10      EQU   10
 8     R13      EQU   13              Address of the save area
 9     R14      EQU   14              Return address
10     R15      EQU   15              Entry address
11     *
12              BALR  R10,R0
13              USING *,R10
14              LA    R1,PARLIST      Load address of parameter list
15              LA    R13,SAVEAREA    Load address of save area
16              L     R15,=A(DIVIDE)  Load address of Pascal procedure
17              BALR  R14,R15         Call Pascal entry procedure
18              TERM
19     *
20     PARLIST  DC    A(PAR1)
21              DC    A(PAR2)
22              DC    A(PAR3)
23     PAR1     DC    F'10'           Dividend
24     PAR2     DC    F'5'            Divisor
25     PAR3     DS    D               Result
26     SAVEAREA DS    18A             Save area for Pascal
27              END
```

Pascal-XT package specification:

```
 1     PACKAGE DIVI;
 2     ENTRY PROCEDURE divide (VAR x, y: Integer; VAR z: Real);
 3     END.
```

Pascal-XT package body:

```
 1     {$Debug=On}
 2     WITH Errors;
 3     PACKAGE BODY DIVI;
 4     PROCEDURE divide (VAR x, y: Integer; VAR z: Real);
 5     BEGIN
 6        z := x / y;
 7     EXCEPTION
 8        IF Error_Number = Numeric_Error THEN
 9           z := Minreal
10        ELSE { propagate other errors }
11           Errors.ReRaise;
12     END;
13
14     BEGIN
15     END.
```

After program start:

```
%% testpoint before entry call ──────────────────────────────────────────── (01)
%% scope seen from outside all compilation units
*show units; ────────────────────────────────────────────────────────────── (02)
%% compilation-units:
%% DIVI, compiled 90-01-12 10:17:37, complete testtable ?
*at divi.12 do getcmd; ──────────────────────────────────────────────────── (03)
*resume; ────────────────────────────────────────────────────────────────── (04)
%% program continued
%% testpoint at line DIVI.12 (divide) ────────────────────────────────────── (05)
*display (%param);
%% parameters of divide:
%%    x = 10
%%    y = 5
%%    z =  2.000000000000000E+00
*resume; ────────────────────────────────────────────────────────────────── (06)
%% program continued
%% program terminated
```

(01) Output of the testpoint message at the entry testpoint.

(02) Output of a list of all the compilation units currently known to the debugging aid.

(03) Definition of a testpoint in the Pascal-XT procedure.

(04) Exit from the entry testpoint.

(05) The testpoint set in (03) is reached.

(06) Exit from the user-set testpoint.

Notes

– The entry testpoint is thus not offered when starting the main program written in a language other than Pascal, but only when the first call is made to a Pascal-XT procedure.

– This call makes known to the debugging aid the package that contains the entry procedure and the packages that the procedure directly and indirectly references. There may, however, be other packages involved. The debugging aid does not know of them until entry procedures contained in them are called. When this occurs, an entry testpoint is offered again and this enables the user also to set testpoints in further packages and so on. It is possible to establish at any time the packages currently known to the debugging aid by using the PATH command SHOW UNITS ("SH U" in its short form).

– Since an entry testpoint is not offered when calling an entry procedure in a package already known to the debugging aid, the user now has no further opportunity in this entry procedure to set testpoints. This means that he ought to take the chance to set testpoints in this package at the entry testpoint offered earlier when the package is being made known to the debugging aid.

9.1.5 Scope of identifiers

The scope of identifiers for a testpoint (both when a testpoint is reached and within an AT command) reflects the static nesting of procedures in the Pascal-XT source program.

Also the predefined constant identifiers in Pascal-XT (e.g. Maxreal, Index_Error,...) may be referred to in PATH (DISPLAY, ASSIGN, IF).

Also the scope expanded in the source program by means of WITH statements is reflected by the debugging aid.

Identifiers hidden in inner procedures by other identifiers having the same names can be addressed by prefixing a block qualification (see section 9.1.3).

Identifiers that are not visible in the source program, because they are declared neither in the current subprogram nor in a surrounding one, but which are declared in a subprogram incarnation in the dynamic call chain, can be addressed by means of prefixing an incarnation qualification (see section 9.1.3).

Example of scope

```
 1    {$DEBUG}
 2    PROGRAM scope:
 3    VAR i: integer;
 4        rec: RECORD
 5           i: integer;
 6           END;
 7    PROCEDURE proc;
 8    VAR i: integer;
 9    BEGIN
10        i:= 1;
11    END;
12
13    BEGIN
14       WITH rec DO BEGIN
15          i := 2;
16          END;
17       i := 3;
18       proc;
19    END.
```

Line numbers	Meaning
10	Used to refer to a testpoint in the procedure "proc". If at this testpoint the variable "i" is referred to, reference is being made to the variable "i" local to the procedure (= proc.i = scope.proc.i, corresponding to the scope in the source program). "scope.1", on the other hand, refers to the hidden global variable "i" declared in the main program ("scope").

17,18	Testpoints in the main program are referred to; "i" refers to the main program variable "i" (= "scope.1", corresponding to the scope in the source program).
15	Since modification of the scope by means of WITH statements in a source program is likewise reflected in the debugging aid, "i" at this point (as in the source program) refers to the RECORD component i, rather than to the main program variable "i". The latter can be referred to by means of "scope.i".

If several testpoints are set in different (sub)programs by means of one AT command, only those names visible from the closest common surrounding (subprogram) can be referenced in the inner command of that AT command.

If an AT command is used to set several testpoints in different compilation units, only "global scope" in the inner command of that AT command applies (see section 9.1.2).

However, the scope limitations described in the previous two paragraphs apply only within the inner command (deferred action) of such an AT command. If, on the other hand, due to a GETCMD command contained in the AT command a branch is made to command input mode at one of the testpoints set, all names visible from that testpoint may of course be referred to.

Identifiers that are spelled the same as PATH commands or PATH command abbreviations (e.g. SHOW, SH, CALLS, C, DISPLAY, D, ...) may also be referred to.

Example

```
1  {$DEBUG} PROGRAM xyz;
2  CONST remove = True;
3  VAR display: INTEGER;
. . .

*display (display, remove);
%% display = 347
%% remove = True
*assign display := 1;
```

9.1.6 Access to identifier types

Program variables and program constants of all types can be symbolically referenced and, corresponding to their types, output, assigned and compared. The rules regarding type compatibility correspond to those in Pascal-XT.

Entire RECORD, ARRAY and SET variables can be referenced (also in the DISPLAY command). When records with variants are output, the components of the fixed part are output along with the currently active variant, to the extent that a tag field exists or the tag type matches the tag type of the higher-ranking variant. For variant parts without tag fields (... CASE typename OF ...) only the components of the fixed part (if present) are output; the components of the variants can then be referenced only individually.

9.1.7 Generation of test tables

To test a Pascal-XT compilation unit using PATH, the DEBUG=ON or DEBUG=RESTRICTED option must be set when the compilation unit is compiled with the Pascal-XT compiler, in order to generate a test table.

A "complete test table" is generated if DEBUG is set in a package body or in a main program. In the case of DEBUG=ON, any PATH command may be used at testpoints in the compilation unit; in the case of DEBUG=RESTRICTED, assignments (ASSIGN command) are not permitted.

A "partial test table" is generated if DEBUG is set in a package specification but not in the corresponding package body. This makes it possible to refer to all variables, constants and types declared in the package specification.
Variables whose types are private pointer types cannot be dereferenced.

A "minimum test table" is generated if DEBUG=OFF is in effect both for the package specification and for the corresponding package body. This permits only the output of the dynamic call chain (in particular in the event of errors).

Note

When the GENERATE option is deactivated, DEBUG is also implicitly deactivated. By activating the DEBUG option, the option OPTIMIZE is implicitly deactivated.

9.2 PATH commands

When a GETCMD command is executed at a testpoint, a command sequence is read in from the PATH input medium. The individual commands of the sequence are each terminated with a semicolon. Commands are executed as soon as they are ended with a semicolon and determined to be syntactically and semantically correct.

When a user-set testpoint is reached (see section 9.1.4.2) the "deferred actions" defined for that testpoint with one or more AT commands are carried out (interpreted). Additional commands can be entered only when a GETCMD command is executed.

The semicolon may be omitted at the end of an input line if at that point in the command all that is syntactically still permissible in any case is a semicolon.

A semicolon at the end of a line must therefore **not** be omitted if the command could be continued in the next line, i.e. following an

- IF command without an ELSE clause, since the next line could of course contain the ELSE part
- ASSIGN command, since the expression on the right-hand side could of course still be indexed, qualified or dereferenced in the next line
- REMOVE, SLEEP, AWAKE or SHOW WHERE command, since the next line could of course still contain parts of the testpoint specification
- EDIT, SYSTEM or SWITCH command without a string, since this could of course still appear in the next line.

In cases of error, the invalid command, including the point at which the error was detected, is output to the PATH output medium, along with a corresponding error message, and a new command is awaited.

There are testpoint commands and action commands.

The SHOW command is described under the action commands, although one of its formats, namely SHOW WHERE, has the characteristics of a testpoint command.

A few commands and other PATH keywords may be abbreviated:

```
ASSIGN        A        SHOW CALLS          SH C
BEGIN         B        SHOW UNITS          SH U
DISPLAY       D        SHOW WHERE          SH W
EDIT          ED
GETCMD        G        SWITCH INPUT        SW I
RESUME        R        SWITCH OUTPUT       SW O
SYSTEM        SY       SWITCH LIST         SW L
```

9.2.1 Testpoint commands

Five of the testpoint commands are used for setting (AT command), deleting (REMOVE command), deactivating (SLEEP command), activating (AWAKE command) and displaying (SHOW WHERE command) user-set testpoints (see section 9.1.4.2).

These commands may contain a testpoint specification (see sections 9.1.3, 9.1.4.2 and 9.1.4.4); they refer to potential testpoints (see section 9.1.2).

The two remaining testpoint commands are used for activating (GETCMD command) and deactivating (RESUME command) command input mode.

9.2.1.1 AT command

Format 1:

```
AT testpoint-specification DO command
```

This command sets testpoints at the potential testpoints (see section 9.1.2) defined by the testpoint specification (see sections 9.1.3, 9.1.4.2 and 9.1.4.4). When execution of the program reaches one of the set testpoints, it is interrupted and the command appearing after the DO (e.g. GETCMD, a compound command or also an AT command) is then carried out ("deferred action").
Following execution of the command, the program run is continued. An input prompt is output at the testpoint only if the deferred action of that testpoint is a GETCMD command or contains one (with RESUME, processing continues at the point interrupted by GETCMD).

An AT command at an existing testpoint supplements the commands to be carried out at that testpoint (adds to them); the commands are carried out in the order in which they are issued.

Example

```
1  {$DEBUG}
2  PROGRAM LOOP;
3  VAR i, k: INTEGER;
4  BEGIN
5     FOR i := 1 TO 5 DO BEGIN
6        k := i;
7        k := k;
8        END;
9  END.
```

Starting the test program:

Prior to program execution, the input of debugging aid commands is requested at an implicit testpoint ("testpoint before program start").

```
%% testpoint before program start
%% scope seen from outside all compilation units ———————— (01)
*at loop.6..7 do display (i, k); ——————————————————————— (02)
*resume; ——————————————————————————————————————————————— (03)
%% k = 0
%% program continued
%% testpoint at line LOOP.7
%% i = 1
%% k = 1
%% program continued
%% testpoint at line LOOP.6
%% i = 2
%% k = 1
%% program continued
%% testpoint at line LOOP.7
%% i = 2
%% k = 2
%% program continued
%% testpoint at line LOOP.6
%% i = 3
%% k = 2
%% program continued
%% testpoint at line LOOP.7
%% i = 3
%% k = 3
%% program continued
%% testpoint at line LOOP.6
%% i = 4
%% k = 3
%% program continued
%% testpoint at line LOOP.7
%% i = 4
%% k = 4
%% program continued
%% testpoint at line LOOP.6
%% i = 5
%% k = 4
%% program continued
%% testpoint at line LOOP.7
%% i = 5
%% k = 5
```

(01) Global scope (see section 9.1.2).

(02) In lines 6 and 7, the values of i and k are output.

(03) With RESUME, the testpoint before program start is left.

Format 2:

```
AT DO command
```

The AT command without a testpoint specification can be used to place a further deferred action at the testpoint at which the AT command is carried out. This additional deferred action is then carried out the next time the testpoint is reached. If this testpoint is the testpoint before program start (see section 9.1.4.1), the command will be rejected with an error message (see section 9.3).
However, if the AT command without a testpoint specification is carried out at the postmortem or exception testpoint (see sections 9.1.4.3 and 9.1.4.4), it will not affect this testpoint but the potential testpoint (see section 9.1.2) at this position.

9.2.1.2 GETCMD command

```
GETCMD
```

A GETCMD command interrupts the test run and switches to input mode, i.e. PATH commands are awaited.

Any number of debugging commands may be entered from the PATH input medium.

Note

A GETCMD command is permitted only within an AT command (i.e. in a deferred action).

9.2.1.3 RESUME command

RESUME

The command input made possible by means of a GETCMD command is ended.

The test run is continued at the point at which it was interrupted by the GETCMD command, i.e. any further commands following a GETCMD command are processed (these may also be further GETCMD commands).

When a RESUME command is executed, the debugging aid input buffer is cleared; this means that any further commands appearing on the same line will no longer be processed at the next GETCMD command.

A RESUME command within an AT command (i.e. in a deferred action) makes no sense and therefore is not permitted.

If all commands defined for the testpoint (the entire deferred action) have been processed, the program run is continued.

Example of the interaction between AT, GETCMD and RESUME

```
%% testpoint before program start
%% scope seen from outside all compilation units
*at loop.6 do getcmd; resume;
%% program continued
%% testpoint at line LOOP.6
*display (i); resume;
%% i = 1
%% program continued
%% testpoint at line LOOP.6
*display (k); resume;
%% k = 1
%% program continued
*
```

9.2.1.4 REMOVE command

Format 1:

```
REMOVE testpoint-specification
```

With regard to the meaning of "testpoint-specification" see sections 9.1.3, 9.1.4.2 and 9.1.4.4.

The testpoints specified by "testpoint-specification", which were previously set by means of one (or more) AT command(s), are deleted again.

When these potential testpoints are reached during a subsequent run of the program under test, the program will no longer be interrupted, and the deferred actions will no longer be carried out.

If not a single testpoint is found to be set (and deleted) at the potential testpoints specified by the testpoint specification, an error message is issued (see section 9.3).

However, if the command

```
REMOVE 63, 135, 716;
```

is executed and then at least one testpoint is found and deleted, the command is considered successful and no error message is issued (regardless of whether testpoints are found in the other lines as well).

Example

```
%% testpoint before program start
%% scope seen from outside all compilation units
*at loop.6 do getcmd; at loop.7 do getcmd; resume;
%% program continued
%% testpoint at line LOOP.6
*display (i); resume;
%% i = 1
%% program continued
%% testpoint at line LOOP.7
*remove 6; ------------------------------------------------------------ (01)
*resume;
%% program continued
%% testpoint at line LOOP.7
*
```

(01) The program is no longer interrupted at line 6.

Format 2:

```
REMOVE
```

The REMOVE command without a testpoint specification causes the testpoint (at which the REMOVE command is executed) to be deleted. If this testpoint is the testpoint before program start (see section 9.1.4.1), the command will be rejected with an error message (see section 9.3).
If the command is executed at the postmortem or exception testpoint (see sections 9.1.4.3 and 9.1.4.4), it will not affect this testpoint. However, if at the position where the error occurred a user-set testpoint is (incidentally) located (see section 9.1.4.2), this testpoint will be deleted. If no testpoint has been set at this position, the command will be rejected with an error message (see section 9.3).

9.2.1.5 SLEEP command

Format 1:

```
SLEEP testpoint-specification
```

Regarding the meaning of "testpoint-specification" see sections 9.1.3, 9.1.4.2 and 9.1.4.4).

The testpoints specified by "testpoint-specification", set previously by means of the AT command(s), are deactivated for the time being; the commands defined for the testpoints (deferred actions) are no longer carried out when the testpoints are reached. In contrast to the REMOVE command, however, the commands still exist logically and can be reactivated by means of the AWAKE command.

If an AT command is triggered at a testpoint deactivated with the SLEEP command, the SLEEP command also affects the newly added command.

The SLEEP command is analyzed and executed analogously to the REMOVE command.

Format 2:

```
SLEEP
```

The SLEEP command without a testpoint specification causes the testpoint (at which the SLEEP command is executed) to be deactivated. If this testpoint is the testpoint before program start (see section 9.1.4.1), the command will be rejected with an error message (see section 9.3).
If the command is executed at the postmortem or exception testpoint (see sections 9.1.4.3 and 9.1.4.4), it will not affect this testpoint. However, if at the position where the error occurred a user-set testpoint is (incidentally) located (see section 9.1.4.2), this testpoint will be deactivated. If no testpoint has been set at this position, the command will be rejected with an error message (see section 9.3).

9.2.1.6 AWAKE command

Format 1:

```
AWAKE testpoint-specification
```

Regarding the meaning of "testpoint-specification" see sections 9.1.3, 9.1.4.2 and 9.1.4.4.

The testpoints specified by "testpoint-specification", previously deactivated by means of the SLEEP command(s), are again in effect; i.e. the commands in effect for those testpoints prior to the execution of the SLEEP command and any further commands set thereafter by means of AT commands will again be carried out the next time one of the testpoints is reached.

If no set testpoints are found (and reactivated) among the potential testpoints specified by "testpoint-specification", an error message is issued (see section 9.3).

Example

```
%% testpoint before program start
%% scope seen from outside all compilation units
*at loop.6 do display (i); at loop.7 do getcmd; resume;
%% program continued
%% testpoint at line LOOP.6
%% i = 1
%% program continued
%% testpoint at line LOOP.7
*display (i, k);
%% i = 1
%% k = 1
*resume;
%% program continued
%% testpoint at line LOOP.6
%% i = 2
%% program continued
%% testpoint at line LOOP.7
*sleep 6; resume;
%% program continued
%% testpoint at line LOOP.7
*at 6 do display ('Will be output only after AWAKE 6');
*resume;
%% program continued
%% testpoint at line LOOP.7
*awake 6; resume;
%% program continued
%% testpoint at line LOOP.6
%% i = 5
%% 'Will be output only after AWAKE 6');
%% program continued
%% testpoint at line LOOP.7
* . . .
```

Format 2:

```
AWAKE
```

An AWAKE command without a testpoint specification causes the testpoint (at which the AWAKE command is executed) to be activated. If this testpoint is the testpoint before program start (see section 9.1.4.1), the command will be rejected with an error message (see section 9.3).
If the command is executed at the postmortem or exception testpoint (see sections 9.1.4.3 and 9.1.4.4), it will not affect this testpoint. However, if at the position where the error occurred a user-set testpoint (deactivated by SLEEP) is (incidentally) located, this testpoint will be reactivated. If no testpoint has been set at this position, the command will be rejected with an error message (see section 9.3).

9.2.1.7 Example of the interaction between testpoint commands

```
*at loop.6 do getcmd;
*at loop.7 do display ('first AT at 7');
*at loop.9 do display ('end of program');
*resume; display (i); ——————————————————————————— (01)
%% program continued
%% testpoint at line LOOP.6
*resume;
%% program continued
%% testpoint at line LOOP.7
%% 'first AT at 7'
%% program continued
%% testpoint at line LOOP.6
*at 7 do begin display (i); getcmd; end; ———————— (02)
*resume;
%% program continued
%% testpoint at line LOOP.7
%% 'first AT at 7'
%% i = 2
*resume;
%% program continued
%% testpoint at line LOOP.6
*sleep 7;
*at 7 do display ('third AT at 7'); ————————————— (03)
*resume;
%% program continued
%% testpoint at line LOOP.6 ———————————————————————— (04)
*resume;
%% program continued
%% testpoint at line LOOP.6
*awake 7; resume;
%% program continued
%% testpoint at line LOOP.7
%% 'first AT at 7'
%% i = 5
*resume; ———————————————————————————————————————— (05)
%% 'first AT at 7'
%% program continued
%% testpoint at line LOOP.9
%% 'end of program'
*remove 7
* . . .
```

(01) After the RESUME command, the rest of the input line is cleared; i.e. the command "display (i)" is no longer carried out.

(02) Supplements the DISPLAY command for line 7 by the addition of the compound command.

(03) Becomes active only after AWAKE 7.

(04) Because of SLEEP 7, the program run is not interrupted in line 7.

(05) Return to the point interrupted by GETCMD; i.e. execution of the subsequent DISPLAY command, set with the third AT command.

9.2.2 Action commands

The action commands are used for outputting (DISPLAY and DUMP) and changing (ASSIGN) program data, for conditional execution of debugging aid commands (IF), for combining several debugging aid commands into one command (i.e. a compound command), for executing system commands (SYSTEM), for calling the editor (EDIT) and for displaying status information (SHOW).

9.2.2.1 DISPLAY command

```
DISPLAY (list)
```

The DISPLAY command causes the factors (see section 9.1.3) and "variable groups" specified in the list, separated by commas, to be output (appropriately for their types) to the PATH output medium.

There are 2 types of variable groups:

- %LOCAL comprises the variables (not parameters) of a block (see section 9.1.2). This block is defined by the block or incarnation qualification (see section 9.1.3) preceding %LOCAL.

  If no block or incarnation qualification is specified, the block is assumed to be the one containing the testpoint at which the DISPLAY command is executed.

  If the block is a package, %LOCAL refers to the variables declared directly within the package specification and package body, but not to any variables of other packages made visible by means of USE clauses.

  If the block contains no variables, an appropriate message is issued.

  Example

```
%% testpoint at line p.333
*display (%local);
%% variables of p:
%%    strg = ’ab’
%%    a_ch = ’a’
%%    z_ch = ’z’
%%    r
%%       .i = 1
%%       .j = 10
* ...
```

  The variables declared in the current block, i.e. in package p (specification and body) are output.

```
*display (a.b.c.%local);
%% variables of a.b.c:
%%    i = 5
%%    ch = 'x'
```

The local variables of the subprogram defined by the block qualification "a.b.c." are output.

```
*at 15, unity.(11, 30) do display (%local); resume
%% program continued
%% testpoint at line unitx.15 (proc12)
%% variables of proc12:
%%    b = True
%%    k = 10
%% program continued
%% testpoint at line unity.11 (proc21)
%% variables of proc21:
%%    i = 8
%% program continued
%% testpoint at line unity.30 (func22)
%% variables of func22: none
%% program continued
```

The local variables of the block containing the current (just reached) testpoint are output.

- %PARAM comprises the parameters of a block. If the block is not a subprogram, or is a subprogram without parameters, a corresponding message is issued.

  Example

```
*display  (a.b.%2.%param);
%% parameters of a.b.%2:
%%    x = 1
%%    b = False

*at unity. (46, 67) do display (%param); resume
%% program continued
%% testpoint at line unity.46 (proc22)
%% parameters of proc22:
%%    m = 5
%%    n = 5
%% program continued
%% testpoint at line unity.67 (proc23)
%% parameters of proc23: none
%% program continued
```

By appending :HEX, variables and constants can be output in hexadecimal form. When this is applied to structured values, the individual components or elements are output in hexadecimal form.

Example of the use of the DISPLAY command

```
 1    {$DEBUG} PROGRAM G (input, output);
 2    TYPE
 3       color_type = (black, brown, red, orange, yellow,
 4                     green, blue, violet, gray, white);
 5       colors = set of color_type;
 6    VAR
 7       i : integer;
 8       b : boolean:
 9       ch: string [3];
10       s : colors;
11       f : color_type;
12
13    PROCEDURE even (var v_b: boolean;
14                            w_i: integer);
15    begin
16       if (w_i mod 2) = 0
17       then v_b := true
18       else v_b := false;
19    end;
20
21    begin
22       s := [green, gray, yellow, blue];
23       f := white;
24       readln;
25       read (i);
26       even (b, i);
27       if b
28       then ch := 'yes';
29       else ch := 'no ';
30       writeln (ch);
31    end.
```

After the start of the test program:

```
%% testpoint before program start
%% scope seen from outside all compilation units
*at g.19, g.32 do getcmd; resume;
%% program continued
*9 ----------------------------------------------------------------- (01)
%% testpoint at line G.19 (even)
*display (v_b, w_i);
%% v_b = False
%% w_i = 9
*display (even.%local); --------------------------------------------- (02)
%% variables of even: none
*display (even.%param); --------------------------------------------- (03)
%% parameters of even:
%% v_b = False
%% w_i = 9
```

(01) Input for the program.

(02) Output of the local variables (if present) of the procedure "even".

(03) Output of all parameters of the procedure "even".

```
*dump; ———————————————————————————————————————————————————————— (04)
%% parameters and local variables of even:
%%    v_b = False
%%    w_i = 9
%% global variables of G:
%%    i = 9
%%    b = False
%%    ch = ''
%%    s = [yellow..blue,gray]
%%    f = white
*display (i, i:hex);
%% i = 9
%% i = #9
*resume;
%% program continued
no ———————————————————————————————————————————————————————————— (05)
%% testpoint at line G.31
*display (%local);
%% variables of G:
%%    i = 9
%%    b = False
%%    ch = 'no '
%%    s = [yellow..blue,gray]
%%    f = white
*display (ch[1], ch[1]:hex);
%% ch[1] = 'n'
%% ch[1] = #'95'
*display (ch[2], ch[3]);
%% ch[2] = 'o'
%% ch[3] = ' '
*display (ch, ch:hex);
%% ch = 'no '
%% ch = #'959640'
*display ('string');
%% 'string'
*display (2, 'A', 3.14, true);
%% 2
%% 'A'
%% 3.1400000000000E+00
%% True = True
*display (colors ([white, brown.. orange, red.. yellow])); ———————— (06)
%% colors ([white, brown.. orange, red.. yellow]) = [brown.. yellow, white]
```

(04) Output of all variables in the dynamic call chain and the global variables of all packages (see section 9.2.2.8).

(05) Output produced by the program.

(06) Output of a qualified set constructor.

9.2.2.2 ASSIGN command

```
ASSIGN variable := factor
```

The variable is assigned the value of the factor (see section 9.1.3). If this value exceeds the limit set by the type of the variable, the assignment is rejected with an error message.

The ASSIGN command is permissible only if the compilation unit in which the testpoint lies was compiled with the option DEBUG=ON, i.e. not with DEBUG=RESTRICTED (see section 9.1.7).

Example

```
%% testpoint before program start
%% scope seen from outside all compilation units
*at g.19 do getcmd; resume;
%% program continued
*12
%% testpoint at line G.19 (even)
*display (i, w_i);
%% i = 12
%% w_i = 12
*assign w_i := 5; ———————————————————————————————————————————— (01)
*display (i, w_i);
%% i = 12
%% w_i = 5
*display (b, v_b);
%% b = True
%% v_b = True
*assign v_b := false; ———————————————————————————————————————— (02)
*display (b, v_b);
%% b = False
%% v_b = False
*assign s := color ([gray, red.. green, yellow.. blue]); ————— (03)
*display (s);
%% s = [red.. blue, gray]
*assign s := colors ([]);
*display (s);
%% s = []
* ...
```

(01) The assignment to w_i **does not** change variable i, passed as a value parameter.

(02) The assignment to v_b **does** change variable b, passed as a var parameter.

(03) A qualified set constructor is assigned to the variable s.

9.2.2.3 IF command

Format 1:

```
IF condition THEN command
```

Format 2:

```
IF condition THEN command ELSE command
```

This command corresponds to the IF statement in Pascal-XT.

For "condition" the Boolean variables, Boolean constants and expressions of the type

```
factor    relational operator    factor
```

are permitted.

For "relational operator" the following operators are permitted:
"=", "<>", "<=", ">=", "<", ">" and "in".
For "command" in the THEN and in the optional ELSE clause, all debugging aid commands are permitted.

Example

```
%% testpoint before program start
%% scope seen from outside all compilation units
*at g.26 do if i < 0 then getcmd;
*at g.27 do if blue in s then begin
*               if b then display ('even')
*                    else display ('odd')
*               end
*            else display ('blue not in s');
*resume;
%% program continued
*-6
%% testpoint at line G.26
*assign i := 3;
*if f in colors ([yellow.. blue]) then display (f); resume;
%% program continued
%% testpoint at line G.27
%% 'odd'
%% program continued
```

9.2.2.4 Compound command

```
BEGIN sequence-of-commands END
```

This command corresponds to the compound statement in Pascal-XT.

It can be used to combine several commands in one AT or IF command or to postpone the execution of a sequence of commands until the terminating END.

If an error in analysis occurs in a command within a compound command, the entire compound command must be reentered.

Example

```
%% testpoint before program start
%% scope seen from outside all compilation units
*begin display (1); ------------------------------------------------- (01)
*      display (2); end;
%% 1
%% 2
*at g.22 do begin display (i); getcmd;
*                 display ('Hello !!!') end; resume;
%% program continued
%% testpoint at line G.22
%% i = 0
*resume; ------------------------------------------------------------ (02)
%% 'Hello !!!'
%% program continued
* . . .
```

(01) The DISPLAY command is not carried out until the compound command is concluded with END.

(02) The RESUME command refers to the GETCMD command in the compound command.

9.2.2.5 SYSTEM command

Format 1:

```
SYSTEM 'system-command'
```

The BS2000 command 'system-command' is executed and then a return is made to debugging aid mode. Both uppercase and lowercase letters can be specified within 'system-command'.

Example

```
%% testpoint at line G.22
*system 'fstat ,r';
% PUBLIC SPACE:       13212 PAGES FOR    206 FILES
* ...
```

Format 2:

```
SYSTEM
```

This has the same effect as the Break key (K2). After /RESUME is entered a return is made to debugging aid mode.

Example

```
%%testpoint at line G.22
*system;
BKPT PCOUNT 00722E
/FSTAT ,R
% PUBLIC SPACE:       13212 PAGES FOR    206 FILES
/RESUME
*
```

9.2.2.6 EDIT command

Format 1:

```
EDIT 'file-name'
```

The procedure area in use is cleared and an EDT FILE command issued using the specified file name. Then the file is loaded into the procedure area and a branch is made to EDT. After input of '@RET' or 'HALT', or after pressing key K1, a return is made to the debugging aid. Both uppercase and lowercase letters can be specified in 'file-name'.

Format 2:

```
EDIT
```

A branch is made to the last EDT procedure area used. The first time it is called this will be procedure area 0. EDT can, however, be exited from, as described above.

9.2.2.7 SHOW command

Format 1:

```
SHOW WHERE testpoint-specification
```

The SHOW WHERE command with testpoint specification (see sections 9.1.3, 9.1.4.2 and 9.1.4.4) shows at which of the potential testpoints specified (see section 9.1.2) testpoints are actually set.

If no set testpoints are found among the potential testpoints specified by "testpoint-specification", an error message is issued (see section 9.3).

Example

```
%% testpoint at line unitx.26
*show where 1..100, unity.(10..50, 63);
%% testpoints at:
%% line unitx.26
%% line unitx.35
%% line unitx.42
%% line unity.17
*show where unity.%all;
%% testpoints at:
%% line unity.17
%% line unity.98
*show where %all_units;
%% testpoints at:
%% line unitx.26
%% line unitx.35
%% line unitx.42
%% line unitx.378
%% line unity.17
%% line unity.98
%% line unitz.136
*show where exception;
%% testpoint at exception
*
```

Format 2:

SHOW **W**HERE

An "empty" SHOW WHERE command (without testpoint specification) forces the output of the current testpoint message prior to the next PATH input or output, or at the latest when the testpoint is left, even if it has already been output at this testpoint (see section 9.3).

This command is useful if the user has forgotten which testpoint is the current one, or (as a deferred action) for tracing the execution of a program.

Example

```
*show where;
%% testpoint at line unitx.26
*at 27..40 do show where; at 42 do getcmd;
*resume;
%% testpoint at line unitx.27; program continued
%% testpoint at line unitx.28; program continued
%% testpoint at line unitx.38; program continued
%% testpoint at line unitx.38; program continued
%% testpoint at line unitx.42
*
```

Format 3:

SHOW **U**NITS

The SHOW UNITS command provides a list of all compilation units (main program and referenced packages) of the program under test. For each compilation unit, the following can be taken from this list:

- its full name, if the appropriate test table is loaded, its abbreviated name, if the appropriate test table is not (yet) loaded.
- the setting of the DEBUG option in the specification and in the body, i.e. whether "minimum", "partial" or "complete" test tables (see section 9.1.7) were generated and whether assignments to program variables are permitted.
- information about the availability of the test tables.
- time compilation took place. As of version 2.2A, the date information output for old and new modules will include a four-digit year number.

Example

```
*show units
%% compilation units:
%% INTERPRETATION, compiled 1991-12-03 13:05:27, complete testtable
%% ANALYZATION (restricted), compiled 1991-11-31 10:13:36, complete testtable
%% NEGOTIAT, compiled 1991-12-04 10:24:57, partial testtable ?
%% XYZ, compiled 1991-12-04 10:21:26, minimal testtable ?
%% CONNECTI, compiled 1991-11-31 12:10:04, invalid testtable
%% AB, compiled 1991-12-04 10:17:23, unavailable testtable
```

The last four compilation units appear in this list with their abbreviated names. The first and second compilation units (package bodies) were compiled using the DEBUG option; "?" means that PATH has not yet accessed the test tables for these compilation units and has not yet tested the availability of those test tables.

Format 4:

SHOW **C**ALLS

The SHOW CALLS command provides a list of the elements of the dynamic call chain for the current testpoint, i.e. all subprogram incarnations, beginning with the one from which the subprogram incarnation containing the current testpoint was called, up to the main program, are output.

Example

```
%% testpoint at line PROG.237 (proc7)
*show calls
%% called from line PROG.100  (proc6)
%% called from line PROG.80   (proc5)
%% called from line PROG.77   (proc5)
%% called from line PROG.77   (proc5)
%% called from line PROG.253  (func2)
%% called from line PROG.103  (proc6)
%% called from line PROG.323
```

The term subprogram incarnation is necessary because as a result of direct or indirect recursive subprogram calls, one and the same subprogram may of course appear several times in the dynamic call chain.

The output of the sequence is the reverse of the call sequence in the program; thus the "most recent" incarnation appears first.

The variables of the individual subprogram incarnations may be referred to by means of incarnation qualifications (see section 9.1.3).

Thus, if a variable i is declared in a function F, for example, the i in the second incarnation of F can be referenced by means of F.%2.i, where F.%2. is the incarnation qualification.

Output of all local variables or parameters of a subprogram incarnation is also possible (see section 9.2.2.1).

Note

Prior to accessing data in the dynamic chain, it is expedient to output the list of the (sub)program incarnations contained in the dynamic chain, with the help of the SHOW CALLS command. In particular, when one testpoint has been left (by means of RESUME) and execution is proceeding to another one (or possibly even the same one again), the SHOW CALLS command should be used before data in the dynamic chain is accessed, since the dynamic chain may have changed since the last testpoint.

9.2.2.8 DUMP command

```
DUMP
```

This command outputs all variables in the dynamic call chain and all global variables of all packages in the PATH output medium.

Example

```
*dump
%% parameters and local variables of aaa:
%%    i = 17
%%    b = False
%% parameters and local variables of aa: none
%% parameters and local variables of a:
%%    ch = ’a’
%%    b = True
%%    x = L’2D1’
%% global variables of HAUPT:
%%    a = 200
%%    b = 100
%%    p = L’C2F11’
%% global variables of PACKA:
%%    j = 1
%%    rec_a
%%       .p = L’0’
%%       .q = L’A2F3’
%%       .size = 512373
%% global variables of PACKB: none
%% global variables of PACKC:
%%    c = 0
%%    d = 1
%%    tab
%%       [1] = ’one’
%%       [2] = ’two’
%%       [3] = ’three’
```

9.2.2.9 KILL command

```
KILL
```

The effect of KILL depends upon the Pascal-XT program section active at the time. In an entry procedure, the program under test is aborted. In a main program, the program under test is only aborted if no more testing is requested (see section 9.5 "Restart").

Example

```
*kill
%% program aborted
```

9.2.2.10 SWITCH command

Format 1:

```
SWITCH INPUT    'file-name'
SWITCH OUTPUT   'file-name'
SWITCH LIST     'file-name'
```

The PATH input, output or list medium is assigned to the specified file.

Format 2:

```
SWITCH INPUT
SWITCH OUTPUT
SWITCH LIST
```

An "empty" SWITCH command cancels the current file assignment and restores the last assignment that was valid.

Note

This command is not yet available.

9.3 Debugging aid messages

At analysis time, the command lines input by the PATH input medium are output to the PATH output medium (echoing of input). If they are faulty (syntactic or semantic errors), analysis stops at the first error and a corresponding error message is output. Echoing of error-free command lines can be suppressed by means of the {$R-} option (see section 9.1.3.1).

- Prior to each PATH input or output, the current testpoint message is output if it has not yet been output since the testpoint was reached, or if, following its (last) output, an (empty) SHOW WHERE command has been executed.

Example

```
%% testpoint at line unitx.72
*display (i);
%% i = 9
*show where;
%% testpoint at line unitx.72
* . . .
```

If at a (user-set) testpoint no PATH input or output takes place (but instead there is, for example, only an ASSIGN command) and no (empty) SHOW WHERE command is executed, no testpoint message is output.

Depending on the type of testpoint, the testpoint message will take one of the following forms:

Testpoint before program start:

```
testpoint before program start
```

User-set testpoint:

```
testpoint at line <unit-name>.<line-no>
```

Postmortem testpoint:

```
testpoint because of unhandled <error> at line <unit-name>.<line-no>
```

Exception testpoint:

```
testpoint because of <error> at line <unit-name>.<line-no>
```

Entry testpoint:

```
testpoint before entry call
```

When a testpoint is reached at which "global scope" is in effect (see section 9.1.2) the message

```
scope seen from outside all compilation units
```

is output, in particular at the testpoint before program start (see section 9.1.4.1) and (if the test table is not complete or is not available, see section 9.1.7) at the post-mortem testpoint (see 9.1.4.3) or at the exception testpoint (see section 9.1.4.4).

Examples

The following example shows the messages at the testpoint before program start:

```
%% testpoint before program start
%% scope seen from outside all compilation units
* . . .
```

The following example shows the messages at the postmortem testpoint of a program with minimum or partial test table:

```
POINTER_ERROR ( 92) RAISED FROM PROG AT 8C0013C0 (00099000).
FILE_INFO IN THE MOMENT OF RAISE : NONE.
%% testpoint because of Pointer_Error at line PROG.16
%% scope seen from outside all compilation units
* . . .
```

- When a testpoint is left a testpoint end message is output if the testpoint message has already been output (at least once) since the testpoint was reached. The testpoint end message "program continued" or "program aborted" indicates whether the program under test is being continued or was aborted due to execution of a KILL command.

  If an (empty) SHOW WHERE command is executed, but then, up to the point at which the testpoint is left, no PATH input or output (and thus also no testpoint message) occurs, or if a KILL command is executed and at the testpoint no testpoint message has been output yet, a combined testpoint and testpoint end message is output.

- In cases of errors during execution of a PATH command, one of the following error messages is output:

Error messages regarding test tables

`test-table not available`

The test table required to test a compilation unit is not available.
The cause is implementation-dependent.

`test-table invalid or incompatible`

The test table and the loaded object of this compilation unit do not stem from the same compilation, or the test table was generated with a compiler version that is not compatible with the PATH version being used.
Compile the compilation unit with a suitable compiler and, if necessary, relink.

`unit compiled without DEBUG-Option`

The compilation unit was compiled with DEBUG=OFF. For this reason, no testpoints can be set in it and variables that are declared in it (or whose types are declared) cannot be referenced.
Compile compilation unit with DEBUG=ON or DEBUG=RESTRICTED.

Errors in testpoint commands

`no such testpoint found`

The testpoint specification of a SLEEP, AWAKE, REMOVE or SHOW WHERE command does not contain a testpoint previously set by means of an AT command.

`testpoint cannot be set`

A testpoint specified in an AT command cannot be set. The cause is implementation-dependent (e.g. machine code write-protected).

`no testable unit`

An attempt was made to set testpoints in all testable compilation units by means of AT %ALL_UNITS... However, no compilation unit is testable, since the main program and all reference packages were compiled with DEBUG=OFF or the test tables are not available.
Compile the compilation units in which testing is to be performed with DEBUG=ON or DEBUG=RESTRICTED.

Errors in action commands

`no call chain`

A SHOW CALLS command could not be executed since the current testpoint is either the testpoint before program start, a postmortem testpoint, or a testpoint in the main program and, as a result, no dynamic call chain exists.

`no valid scope given`

Global scope is in effect.
The variable group (%LOCAL or %PARAM) must in this case be prefixed by a block qualification or incarnation qualification (see section 9.1.3).

`no such incarnation found`

The specified execution (incarnation) of the subprogram does not exist.
Display the dynamic call chain using the SHOW CALLS command and use the correct incarnation number if possible.

`invalid address`

An attempt was made to dereference a pointer containing an invalid address.

`invalid set`

In an action command a set type variable has been used and it contains invalid values. This is particularly likely to be the case if the variable has not (yet) been initialized. If an IF or ASSIGN command is involved, it is not executed. In the case of a DISPLAY or DUMP command the contents of the variables concerned (including the invalid values) are output.

`invalid string length`

The length of a string lies outside the permissible range. PATH therefore cannot access the string.

`index out of range`

The value of the specified index expression lies outside the range of values defined by the ARRAY index type.

`low bound > high bound`

The value of the slice lower limit is greater than the value of the slice upper limit.

`pointer is nil`

An attempt was made to dereference a pointer that has the value nil (null).

`out of range`

The value of a variable lies outside the range of values defined by its ordinal type.

`boolean variable is neither True nor False`

The variable specified in an IF command as a condition has an invalid value. Therefore, neither the THEN branch nor the ELSE branch is performed.

`value to be assigned is out of bounds`

An attempt was made to assign to a variable a value which is outside the range of values defined by its ordinal type. The variable was not changed.

`set-value to be assigned exceeds bounds of variable`

An attempt was made to assign to a set variable a value containing elements outside the range of values defined by the base type of the type of variable. The variable was not changed.

`command not executed`

The operating system command specified in the command string of a SYSTEM command was not carried out because it is faulty.

`illegal parameter`

The file specified in an EDIT command cannot be edited.

`set-element is out of set-bounds`

An element of the set qualified by the set constructor lies outside the range of values defined by its base type.

Further error messages beginning with "DP:" can occur in exceptional cases.

Either there is a fatal inconsistency in the program under test (e.g. updating of a package specification without recompilation of the package bodies involved) or, if this is not the case, the error is an internal error in the Pascal-XT system.

9.4 Linking with PATH

Before a program can be tested it must be linked with the PATH debugging aid. The required modules of the debugging aid are contained in object module library $PASLIB-XT.
Testing of programs in shared code is not possible.

Static linking

For static linking using TSOSLNK, linking of the module "#test" specifies that the program is to be tested with the PATH debugging aid. The debugging aid and the test table modules are loaded dynamically, provided that they have not already been loaded statically.
For further information on linking see section 6.2.

```
/EXEC $TSOSLNK
    PROGRAM  prog
    INCLUDE  progname          , modlib
    INCLUDE  #test             , $PASLIB-XT  ———————————————————————— (01)
    INCLUDE  (#path##c,#path##d), $PASLIB-XT ———————————————————————— (02)
    INCLUDE  test table        , modlib  ———————————————————————————— (03)
    RESOLVE                    , modlib
    RESOLVE                    , $PASLIB-XT
    END
```

(01) By linking in the module #test it is specified that the generated object program is to be executed under the control of the debugging aid PATH.

(02) The modules of the PATH debugging aid are linked in. If the INCLUDE statement is missing, the debugging aid is dynamically loaded when the program under test is started.

(03) The specified test table module of the program is explicitly linked in. If this INCLUDE statement is omitted, the test table module is then dynamically loaded when required. The same applies to test table modules of the packages linked to the program. See section 4.4 for naming conventions pertaining to the test table modules.

Dynamic linking

Programs linked dynamically can only be tested within the programming system. This is accomplished with the aid of the RUN-PROGRAM statement using the DEBUG=YES operand. The test table modules are loaded dynamically when required.

9.5 Testing with PATH

The program under test can be called and executed both on a BS2000 command level and from within the programming system.

Invocation on BS2000 command level

/EXEC test-program

Invocation from within the programming system

RUN-PROGRAM (modlib, progname), DEBUG=YES

The program under test ("progname") is loaded from the object module library "modlib" and started. If the library specification and/or the program name is omitted, the last program compiled is tested.

PATH checks the main program and all relevant packages to ensure that the respective code and data modules stem from the same compilation. If this is not the case, error messages of the type

code-module incompatible with data-module <unit-name>

are output and the debugging run is aborted.

Once the program has been loaded and started, PATH issues a message at the testpoint before program start and awaits input of further commands:

```
%% testpoint before program start
%% scope seen from outside all compilation units
*
```

Restarting the program under test

If the Pascal-XT main program tested is terminated (normally, by KILL or due to an unrecoverable runtime error) the user can terminate the program under test or restart it. The following message appears:

```
%% do you want to restart program with same testpoints ? (y/n)
```

Entry of "y" or "Y" restarts the program using the same testpoints and ILCS is neither terminated nor reinitialized.

Entry of "n" oder "N" causes a return to the work mode in effect before the Pascal-XT program was called (Pascal-XT programming system or BS2000 command mode).

No restart is possible for a main program implemented in another programming language.

Canceling PATH commands

PATH allows the user to cancel a currently active debugging aid command by pressing the K2 key (useful, for example, if the command is producing an unexpected amount of output). Entering the BS2000 command "/INTR" causes the debugging aid to cancel the PATH command and (after completing the output of any line already begun) to issue the following message:

"... command(s) cancelled by user"

It then requests new commands, i.e. all further commands related to this testpoint are ignored.

This behavior applies only when K2/INTR is entered at a testpoint; that is, at a point between the testpoint message and the testpoint end message (see section 9.3). However, if K2/INTR is input at some other point (when the program under test is not at a testpoint), then PATH is not interrupted, but the exception condition Break_Error is raised in the program under test. The program is then aborted if it does not contain an exception handler capable of handling this exception condition.

Package names

If the test tables have not been statically linked in and the program has been started by the main program and any packages linked in, the system will recognize only those names which have been abbreviated to eight characters. Unabbreviated names are not recognizable in their full length until the test tables have been loaded.

Dynamic loading of the test tables

For a program started by the RUN statement or a statically linked program (phase) in which the test tables have not been linked in, the test tables are loaded dynamically as required. For dynamic loading, the debugging aid proceeds according to the following strategy:

- For a program started with the RUN statement, the debugging aid attempts to load the test tables from the object module library from which the program was loaded.
- For statically linked programs, the debugging aid attempts to load the test tables from the file TASKLIB or $TASKLIB.

If they are not found there, the Dynamic Linking Loader (DLL) searches for them in a Tasklib created by the user with the SYSFILE command.

If no test table is found, the debugging aid requests input of a library name or of an asterisk "*" for the temporary EAM object module file, or it requests an empty string. The empty string is used to indicate the unavailability of the test table. The debugging aid then attempts to load further test tables from the specified library, requesting a new library each time it does not meet with success.

If the debugging aid requires the test table of a predefined package (e.g. BS2000CALLS) and cannot find it (e.g. ... `MOD "BS2000CT" NOT FOUND` ...), $userid.PASLIB-XT must be entered as the library name.

Validity of the test tables

The debugging aid checks all required test tables as to their validity, i.e. whether or not they were generated by the same compilation as the code and data modules. If the test tables are invalid, the debugging aid issues the following messages in the case of statically linked test tables:

```
statically linked test table is invalid, trying to link dynamically
```

For dynamically loaded test tables the message reads:

```
dynamically linked test table is invalid
```

and the system requests input of a library name (see above).

PATH input/output

Debugging aid outputs are prefixed by "**%%**" and are passed on to the system output file **SYSOUT**. Commands to the debugging aid are read from the system input file **SYSDTA**.

Output of pointer values

Values of pointer variables are output in the form "**L’address’**", with "address" being a hexadecimal value.

Handling "non-printable" characters in EBCDIC character sets

Non-printable characters (e.g. in a variable within a variable string type) are output as full stops (’.’). As before it is of course possible to output certain or all characters of the variables in hexadecimal form by appending :HEX.

Example

Assume that "strg" is a variable within the variable string type "String".

```
*DISPLAY (strg);
%% strg = '..Pascal.'
```

The first two characters and the last character of the character string are not printable (or possibly the printable character '.' could be used). If it is important to know which non-printable characters are contained in "strg", then it is possible to have the value of "strg" or parts of it output in hexadecimal form:

```
*DISPLAY (strg[1..2] :HEX);
%% strg [#1..#2] = #'1A2E'
*DISPLAY (strg[9] :HEX);
%% strg [#9] = #'80'
```

Runtime error response of the program under test

When the exception testpoint has been set, the debugging aid in a Pascal-XT program section issues a message either at the point where the error occurred or at the point where the external subprogram which passed on the error was called, and it executes the PATH command previously specified for the error condition.

If an exception handler is defined for the runtime error in the source program of the Pascal-XT program section tested, the program continues from that point on leaving the exception testpoint.

If no exception handler is defined, a postmortem testpoint is activated and the debugging aid issues a message, as described in section 9.1.4.3. The message appears either at the point in the Pascal-XT program section at which the runtime error occurred, or at the point at which the external subprogram which passed on the error was called.

The testpoint message contains the number of the source line in which the runtime error occurred. Afterwards debugging aid commands can be entered.

The PATH command KILL causes the program to terminate.

The effect of the PATH command RESUME depends on the type of Pascal-XT program section tested (see section 9.1.4.3).

Before the testpoint message is displayed the name of the object module library is requested if the test table has not been loaded.

Simultaneous use of the PATH debugging aid and the system debugging aid AID

Simultaneous use of the debugging aids PATH and AID does not lead to any conflicts as long as the Pascal program has been tested with PATH and the linked-in Assembler programs have been tested with AID. AID commands can also be issued at PATH testpoints using the SYSTEM command. However, it should be noted that when register contents are output by AID for example, the values of the program under test are not output, rather those of the debugging aid.

Potential testpoint

As well as most of the beginnings of statements, the end of a subprogram, main program or package body represents a potential testpoint (see section 9.1.2) In addition, the occurrence of an exception (see section 9.1.4.4) or a call to an entry subprogram in a compilation unit not known to the debugging aid (see section 9.1.4.5) represent potential testpoints.

10 Runtime errors and error handling

Runtime errors are errors and exceptions occurring during program execution. They can have various causes, for example there may be an addressing error or the user may produce an exception by calling the predefined procedure Raise (see [1], 15.11).

A runtime error is either intercepted by an exception handler (see [1], 14.2) in the program section in which it is detected, or passed on to the calling program section. Passing on a runtime error to the caller is also known as "error propagation".

As of Pascal-XT V2.2A, ILCS (see 7.1) also enables runtime errors to be propagated across language boundaries, i.e. when the caller and the subprogram called are implemented in different programming languages. Error propagation operates in both directions: from a Pascal-XT entry procedure to an external caller, or from an external subprogram to a Pascal-XT caller. However, as not all programming languages support error propagation, interfacing between different languages can lead to unexpected system behavior. For information on error handling mechanisms provided by other programming languages, please refer to the relevant language or compiler reference manual. The following section describes error handling for Pascal-XT V2.2A and indicates potential problems.

10.1 STXIT events and ILCS

The Pascal-XT exception handling facility is designed to permit the user to deal with potential runtime errors occurring during program execution. In BS2000 terminology, runtime errors detected by the operating system are called events and are subdivided into different STXIT event classes (see [6]).
As of Pascal-XT V2.2A, the following three event classes are no longer reported directly to the operating system by the runtime system, but instead are reported via the SEH (Standard Event Handler) or SSH (Standard STXIT Handler) of ILCS:

(a) Program check (PROCHK)

(b) Unrecoverable program error (ERROR)

These two event classes are reported to the operating system via the SEH of ILCS. If such an event occurs, the operating system passes the interrupt weight of the event to ILCS for use as the event code.

(c) Message to program (INTR)

This event class is reported to the operating system by the Pascal-XT runtime system during its initialization via the SSH of ILCS. Provided that a Pascal-XT program section is active, this event is activated by the key combination [K2]/INTR and is interpreted as a Pascal-XT Break_Error.

The Pascal-XT error handling facility and the debugging aid PATH function only if the SEH of ILCS has not been deactivated by an external program section or overlaid by the SSH in the same event classes.

10.2 Error handling and output in the event of an error

10.2.1 Pascal-XT handling of SEH events

As of Pascal-XT V2.2A, all runtime errors are reported to the Pascal-XT runtime system via ILCS, regardless of where they originate. ILCS assigns an event code which enables the runtime system to identify the error. ILCS bases the event code on a value (for STXIT events, the interrupt weight; for program errors, the Error_Number) which is passed to ILCS by the agency which first detects the error.

What are SEH events?

SEH events are runtime errors reported to the Pascal-XT runtime system by the SEH of ILCS. The runtime system uses the assigned event code to distinguish between SEH-STXIT events and SEH-NON-STXIT events.

SEH-STXIT events include the STXIT event classes PROCHK and ERROR; SEH-NON-STXIT events include runtime errors which occur in a Pascal-XT program section (e.g. due to Raise) or in an external program section written in another language.

How are SEH events handled?

The first time control passes to a Pascal-XT program section (main program or entry procedure) during program execution, the Pascal-XT runtime system activates its own event handling routine for SEH events within ILCS. This causes ILCS to report any occurrence of an SEH event to the runtime system. The event handling routine remains activated when an external subprogram is called from within the Pascal-XT program section. This makes it possible to deal with errors that are passed from the external subprogram to the calling Pascal-XT program section.

If an SEH event occurs in a Pascal-XT program section or is passed on from a subprogram, ILCS reports that event with its event code to the Pascal-XT runtime system, which then acts as follows:

(1) Determination of the Pascal-XT error number that corresponds to the SEH event.

The runtime system ascertains from the event code whether this is an SEH-STXIT event or an SEH-NON-STXIT event.

With SEH-STXIT events, the interrupt weight is used in setting the Pascal-XT system error code and, implicitly, the Pascal-XT error number. It does not matter whether the program section in which the STXIT event occurred is implemented in Pascal-XT or in another programming language.

With SEH-NON-STXIT events occurring in a Pascal-XT program section, the event code is used as the Pascal-XT error number. SEH-NON-STXIT events that occur in an external subprogram and are passed on to a Pascal-XT program section are interpreted as a SYSTEM_ERROR (error number -1) with the system error code 5002.

Errors that occur in mathematical routines and are passed on to a Pascal-XT program section are mapped to Numeric_Error with a special system error code.

(2) Search for an exception handler in the Pascal-XT program section.

If a procedure or function in the dynamic call chain of the Pascal-XT program section contains an exception handler, the program continues from the point at which it is defined. All program sections through which the error has been propagated are terminated by a subprogram termination routine.

If no exception handler is defined, subsequent response depends upon whether the program section examined is a Pascal-XT entry procedure or the main program.

If it is the main program, the runtime system displays the cause of the error and the dynamic call chain (see 10.2.4). The program is then terminated.

If it is an entry procedure, the error is passed to the caller. If the caller is a Pascal-XT program section or was itself called either directly or indirectly from within a Pascal-XT program section, then steps (1) and (2) are repeated.
If the caller is an external program section and the error is not handled there nor in any external program section above it, then the program is aborted.

If another runtime error ("Secondary_Error") occurs during steps (1) and (2), the whole program is aborted immediately.

10.2.2 The Pascal-XT Break_Error

The STXIT event INTR (activated by the key combination K2/INTR) is interpreted as a Pascal-XT Break_Error and passed to the caller only if a Pascal-XT program section is active. In this event, the user STXIT routine activated by the Pascal-XT runtime system for the INTR event switches to the interrupted Pascal-XT program section process and initiates an SEH-NON-STXIT event with the event code "Break_Error". If an exception handler has been defined for this event, the program continues from that point; if not, the error is passed to the caller.

If the INTR event occurs while an external program section is active, the SSH searches for other user STXIT routines for the INTR event. If such routines are defined, SSH reports the event to them; if not, the program continues without comment from the point at which it was interrupted.

10.2.3 Language interfacing between Pascal-XT and Assembler

As the Pascal-XT error handling facility and the debugging aid PATH only function correctly if the SEH is not deactivated by an external program section or overlaid by the SSH, the following points should be noted for language interfacing with Assembler:

If necessary, a user can report and handle STXIT events in Assembler subprograms (see the ASSEMBH manual [15]). If, however, Assembler program sections report their own STXIT routines to BS2000 for the event classes PROCHK and ERROR, note that Assembler programs leave STXIT routines in such a way that any errors are subsequently reported correctly to the Pascal-XT runtime system.

10.2.4 Output when a runtime error occurs

When a runtime error occurs, the Pascal-XT runtime system stores the error number and the names of the subprograms in the dynamic call chain.

The stored information is output automatically if a runtime error is not handled anywhere in the program and if the main program is implemented in Pascal-XT. If a runtime error occurs in a Pascal-XT entry procedure, the information is not output. However, the user can define an exception handler and output the dynamic call chain at that point to the system files SYSOUT and SYSLST with the predefined procedure Print_Error_Info from the ERRORS package (see appendix A.7).

If a runtime error is passed to the caller, please note the different effects of Raise (Error_Number) and Raise (0) (or ERRORS.reraise, see appendix A.7).
Raise (Error_Number) generates the same exception condition and the original point of failure, i.e. the subprogram call chain from there to Raise, is lost.
In contrast, Raise (0) passes the exception condition on to the caller without destroying the subprogram call chain (see also the examples at the end of this section).

The information output has the following structure:

```
            ┌── Exception
            │         ┌── System code
            │         │                          ┌── Dynamic call chain
            │         │                          │
%     READ_ERROR (1094) RAISED FROM quadr_eq.get_valu AT 000E3240
%                              FROM quadr_eq.solve    AT 000E32F0
%                              FROM quadr_eq.quadr_eq AT 000E3352
% FILE_INFO IN THE MOMENT OF RAISE : INPUT,'*SYSDTA,MAXLINELENGTH=254'.
                                             │
                     File allocation ────────┘
```

Where interfaces between different languages are used, other information is also output. More details of this are given in the section dealing with the structure of the dynamic call chain (see below).

Exception

The exception comprises an integer number supplied by the predefined function Error_Number [1]. If a predefined exception is involved, the runtime system outputs the identifier predefined in the syntax, instead of the error number. Identifiers and their associated error numbers are shown in the table below.

System code

In brackets immediately following the exception code a system code is provided to classify the exception more specifically. It is supplied by the System_Code function from the predefined package ERRORS (see appendix A.7). It is only defined for some of the predefined exceptions (refer to table); the meaning of this number is described in section 10.4. The runtime system outputs the error number in hexadecimal form if a BS2000 error code is involved.
The system code has the value 0 if the exception was triggered as the result of activating the Check option (see also section 10.4).

Predefined exception	ERROR_NUMBER	ERRORS.system_code
SYSTEM_ERROR	- 1	See section 10.4
NUMERIC_ERROR	- 2	See section 10.4
RANGE_ERROR	- 3	Undefined
SET_ERROR	- 4	Undefined
STRING_ERROR	- 5	See section 10.4
INDEX_ERROR	- 6	Undefined
POINTER_ERROR	- 7	See section 10.4
VARIANT_ERROR	- 8	Undefined
CASE ERROR	- 9	Undefined
FILE_ERROR	-10	See section 10.4
EOF_ERROR	-11	Undefined
OPEN_ERROR	-12	See section 10.4
READ_ERROR	-13	See section 10.4
MEMORY_ERROR	-14	See section 10.4
BREAK_ERROR	-15	Undefined 1)
ELAB_ERROR	-16	Undefined
	Otherwise	Undefined

1) A BREAK_ERROR is generated by pressing the K2 key (transition to BS2000 command mode) and subsequently entering the BS2000 command /INTR.

Dynamic call chain

The dynamic call chain reflects the sequence of subprogram calls at the time a runtime error occurs. The dynamic call chain output is structured as follows:

First and second line:
If the runtime error occurred in a Pascal-XT program section, the first line contains the name of the Pascal-XT procedure in which the error occurred.
If the error was passed to the Pascal-XT program section from an external subprogram, the first line contains the following text:

```
error number (system code) RAISED FROM OUTSIDE PASCAL ENVIRONMENT
```

In this case, the second line contains the name of the Pascal-XT procedure to which the error was first passed.

Subsequent lines:
The subsequent lines each contain the dynamic Pascal-XT predecessors as far back as the Pascal-XT main program. For each procedure specified the package name and procedure name are output, each with a maximum of 8 characters; for entry procedures, the keyword ENTRY is also output. If there is an external subprogram between a Pascal-XT entry procedure and a Pascal-XT caller, the following text is output:

```
FROM NON-PASCAL-ROUTINE(S)
```

Last line:
No more than the first 20 lines of the dynamic call chain are output. If there are more dynamic predecessors than this, it is indicated in the last line by three periods following the procedure name. If not, the last line contains either the name of the Pascal-XT main program or, if the main program is implemented in another programming language, the name of the last Pascal-XT entry procedure to pass on the error.

File assignment

If the runtime error occurs when a file is accessed, an additional line is output, with:
- The name of the Pascal file (8 characters, maximum), if no local file is involved.
- If the physical file was assigned by means of ASSIGNFILE, the external description from this call is output.
- In the case of non-temporary files, the name of the physical file is output.

If a runtime error occurs when testing a program (with PATH), following output of the runtime error message at the postmortem testpoint the number of the source line is output in which the runtime error occurred.

Secondary Error

In the case of significant errors the runtime system issues one of the following messages:

`SECONDARY ERROR ...` or `INTERNAL_ERROR...`

This means that an inconsistency has been identified in the system. In order to avoid never-ending recursions the runtime system issues the above message and aborts the program. The error or inconsistency may have been caused by the program itself (e.g. the effect of overwriting) or there may be an error in the system that has not yet been detected.

Examples

1. Difference between Raise (Error_Number) and Raise (0)

The different effects of Raise (Error_Number) and Raise (0) are shown by examples 14-4 and 14-5 in section 14.3 of the Pascal-XT Language Reference Manual (see [1]).

Processing for example 14-4

```
 1      PROGRAM quadr_equation (Input, Output);
 2
 3      PROCEDURE get_value (name: String; VAR value: Real);
 4      BEGIN
 5         Writeln ('Please enter the value for ', name, ':');
 6         Read (value);
 7      END;
 8
 9      PROCEDURE solve;
10      VAR
11        p, q, d: Real;
12      BEGIN
13         Writeln;
14         Writeln ('Quadratic equation  x**2 + p*x + q = 0');
15         get_value ('p', p);
16         get_value ('q', q);
17         d := Sqrt (Sqr (p) / 4 - q);
18         IF d = 0
19            THEN Writeln ('x = ', -p / 2)
20            ELSE Writeln ('x = ', -p / 2 + d, ' or ', -p / 2 - d);
21      EXCEPTION
22         IF Error_Number = Numeric_Error
23            THEN Writeln ('The equation has no solution')
24            ELSE Raise (Error_Number);
25      END;
26
27      BEGIN {PROGRAM quadr_equation}
28         WHILE True
29            DO solve;
30      EXCEPTION
31         IF Error_Number = Eof_Error
32            THEN Writeln ('Goodbye')
33            ELSE Raise (Error_Number);
34      END.

Quadratic equation  x**2 + p*x + q = 0
Please enter the value for "p":
6
Please enter the value for "q":
5
x = -1.000000000000000E+00 or -5.000000000000000E+00

Quadratic equation  x**2 + p*x + q = 0
Please enter the value for "p":
6
```

```
Please enter the value for "q":
9
x = -3.000000000000000E+00

Quadratic equation  x**2 + p*x + q = 0
Please enter the value for "p":
6
Please enter the value for "q":
10
The equation has no solution

Quadratic equation  x**2 + p*x + q = 0
Please enter the value for "p":
xxx
READ_ERROR (1094) RAISED FROM quadr_eq.quadr_eq AT 000E3660(000E5000).
FILE_INFO IN THE MOMENT OF RAISE : INPUT,'*SYSDTA,MAXLINELENGTH=254'.
```

As can be seen in the case of the Read_Error caused by the entry of "xxx" (and propagated in the program by the specification of "Raise (Error_Number)") the dynamic call chain is only output from the point of the last propagation (thus only from the main program "quadr_equation"). It is no longer possible to identify that the error has occurred in the procedure "get_value".

Processing for example 14-5

The program has been slightly modified when compared with that in the Language Reference Manual: Instead of the call "Raise (0)" the equivalent call "Errors.ReRaise" is used.

```
 1      WITH Errors;
 2      PROGRAM quadr_equation (Input, Output);
 3
 4      PROCEDURE get_value (name: String; VAR value: Real);
 5      BEGIN
 6         Writeln ('Please enter the value for ', name, ':');
 7         Read (value);
 8      END;
 9
10      PROCEDURE solve;
11      VAR
12        p, q, d: Real;
13      BEGIN
14         Writeln;
15         Writeln ('Quadratic equation  x**2 + p*x + q = 0');
16         get_value ('p', p);
17         get_value ('q', q);
18         d := Sqrt (Sqr (p) / 4 - q);
19         IF d = 0
20            THEN Writeln ('x = ', -p / 2)
21            ELSE Writeln ('x = ', -p / 2 + d, ' or ', -p / 2 - d);
22      EXCEPTION
23         IF Error_Number = Numeric_Error
24            THEN Writeln ('The equation has no solution')
25            ELSE Errors.ReRaise;
26      END;
27
28      BEGIN {PROGRAM quadr_equation}
29         WHILE True
30            DO solve;
31      EXCEPTION
32         IF Error_Number = Eof_Error
33            THEN Writeln ('Goodbye')
34            ELSE BEGIN
35               Writeln ('Program aborted, details on SYSLST');
36               Errors.ReRaise;
37               END;
35      END.
```

```
Quadratic equation  x**2 + p*x + q = 0
Please enter the value for "p":
xxx
Program aborted, details on SYSLST
%     READ_ERROR (1094) RAISED FROM quadr_eq.get_valu AT 000E3240
%                              FROM quadr_eq.solve    AT 000E32F0
%                              FROM quadr_eq.quadr_eq AT 000E3352
% FILE_INFO IN THE MOMENT OF RAISE : INPUT,'*SYSDTA,MAXLINELENGTH=254'.
```

With this solution the output of the dynamic call chain starts from the actual location of the error.

It is possible to obtain the same error report by calling the procedure "Errors.Print_Error_Info" in line 36 instead of "ReRaise" and the program would then terminate "normally".

2. Unhandled runtime error in Pascal-XT/Cobol language interfacing

This example is the same as example no. 1 in chapter 7, with one difference: A runtime error occurs in the Cobol subprogram (Numeric_Error due to division by 0) and is passed on to the caller (Pascal-XT main program). As the error is not handled there, it leads to abortion of the program.

Source code of the Pascal-XT main program "PASHAUPT":

```
PROGRAM PASHAUPT (INPUT,OUTPUT);
TYPE
   RA   =   1..8;
   STR8 =   ARRAY [RA] OF CHAR;
   SATZ =   RECORD
            A(0)  :  INTEGER;
            B(4)  :  STR8;
            END;
CONST
   AGGR =   STR8('T','E','S','T',' ':4);
VAR
   RC   :   SATZ;
   I    :   INTEGER;
PROCEDURE COBUPROG (VAR PAR:SATZ); EXTERNAL;

BEGIN   { PASHAUPT }
    RC.A := 1111;
    RC.B := AGGR;
    COBUPROG (RC);
    WITH RC DO BEGIN ──────────────────────────────────── (01)
       WRITELN ('A:',A);
       WRITE ('B:');
       FOR I := FIRST(RA) TO LAST(RA) DO
          WRITE (B[I]);
       WRITELN;
       END;
END.
```

Source code of the Cobol subprogram "COBUPROG":

```
       ID DIVISION.
       PROGRAM-ID. COBUPROG.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SPECIAL-NAMES.
           TERMINAL IS SCREEN.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       77 TEILDR    PIC 9(8).
       77 DIV       PIC 9. ──────────────────────────────── (02)
```

```
LINKAGE SECTION.
01  SATZ.
    02  TEILB   PIC S9(8) COMP.
    02  TEILA   PIC X(8).
PROCEDURE DIVISION USING SATZ.
ANF.
    MOVE TEILB TO TEILDR.
    DISPLAY "TEILB: " TEILDR UPON SCREEN.
    DISPLAY "TEILA: " TEILA UPON SCREEN.
    MOVE "XXXXZZZZ" TO TEILA.
    MOVE 0 TO DIV.                                                    (03)
    COMPUTE TEILDR = TEILDR / DIV.                                    (04)
BACK.
    EXIT PROGRAM.
```

Call of the executable phase "PASCOBOL" and runtime listing:

```
/EXEC PASCOBOL ──────────────────────────────────────────────────────────────── (05)
%  BLS0500 PROGRAM 'PASCOBOL', VERSION ' ' OF '...' LOADED
TEILB: 00001111 ─────────────────────────────────────────────────────────────── (06)
TEILA: TEST
9089 INTERRUPT-CODE= 68  AT PROGRAM COUNT= 000008F4 , PROGRAM-ID IS  COBUPROG
    NUMERIC_ERROR ( 104) RAISED FROM OUTSIDE PASCAL ENVIRONMENT───────────────── (07)
                                FROM PASHAUPT.PASHAUPT AT 0000027E ──────────── (08)
%  EXC0732 ABNORMAL PROGRAM TERMINATION. ERROR CODE 'NRT0101' /HELP-MSG NRT0101
%  CMD0205 ERROR IN PRECEDING COMMAND OR PROGRAM AND PROCEDURE STEP TERMINATION : COMMANDS ...
                     ... WILL BE IGNORED UNTIL /SET-JOB-STEP OR /LOGOFF OR /ABEND IS RECOGNIZED
```

Explanation:

(01) Execution of the Pascal-XT main program is interrupted at this point due to a runtime error in the Cobol subprogram which leads to abortion of the program.

(02) Declaration of the variable DIV of type PIC 9.

(03) The variable DIV is assigned a value of 0.

(04) The statement requests division of the value of TEILDR (1111) by the value of DIV (0); this raises a runtime error. See (07).

(05) Startup of the executable phase PASCOBOL.

(06) Output of the Cobol subprogram: Name and value of the variables TEILB and TEILA.

(07) Because a runtime error occurred in the Cobol subprogram and this was not handled by the main program, the program is aborted and the dynamic call chain is output. The first line indicates that the problem is a Numeric_Error and that the error has been propagated from within an external program section.

(08) This line indicates the point in the Pascal-XT program section at which the external subprogram that propagated the runtime error was called. Preceding the period is the name of the package, where applicable; the period is followed by the name of the entry procedure. If, as in this case, the Pascal-XT program section is a main program, the same name appears in front of and after the period.

3. Handled runtime error in Pascal-XT/Cobol language interfacing

This example is the same as example 2, with one difference: The runtime error passed on by the Cobol subprogram to the Pascal-XT main program is intercepted by an exception handler in the main program, and consequently does not cause an abnormal program termination. As the runtime error is an SEH-STXIT event (division by 0), the fact that it occurred in an external subprogram is immaterial: it is passed on as a Numeric_Error, exactly as if it had occurred in a Pascal-XT program section and had not been handled there.

Source code of the Pascal-XT main program "PASHAUPT":

```
WITH ERRORS;
PROGRAM PASHAUPT (INPUT,OUTPUT);
TYPE
   RA   =   1..8;
   STR8 =   ARRAY [RA] OF CHAR;
   SATZ =   RECORD
            A(0)  :  INTEGER;
            B(4)  :  STR8;
            END;
CONST
   AGGR =   STR8('T','E','S','T',' ':4);
VAR
   RC   :   SATZ;
   I    :   INTEGER;
PROCEDURE COBUPROG (VAR PAR:SATZ); EXTERNAL;

BEGIN  { PASHAUPT }
    RC.A := 1111;
    RC.B := AGGR;
    COBUPROG (RC);
    WITH RC DO BEGIN
       WRITELN ('A:',A);
       WRITE ('B:');
       FOR I := FIRST(RA) TO LAST(RA) DO
          WRITE (B[I]);
       WRITELN;
       END;
EXCEPTION
    IF ERROR_NUMBER = NUMERIC_ERROR ------------------------------------ (01)
       THEN WRITELN ('ERROR INTERCEPTED BY EXCEPTION HANDLER.')
       ELSE ERRORS.PRINT_ERROR_INFO;
END.
```

Source code of the Cobol subprogram "COBUPROG" (unchanged):

```
       ID DIVISION.
       PROGRAM-ID. COBUPROG.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SPECIAL-NAMES.
           TERMINAL IS SCREEN.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
```

```
77 TEILDR    PIC 9(8).
77 DIV       PIC 9.
LINKAGE SECTION.
01  SATZ.
    02  TEILB   PIC S9(8) COMP.
    02  TEILA   PIC X(8).
PROCEDURE DIVISION USING SATZ.
ANF.
    MOVE TEILB TO TEILDR.
    DISPLAY "TEILB: " TEILDR UPON SCREEN.
    DISPLAY "TEILA: " TEILA UPON SCREEN.
    MOVE "XXXXZZZZ" TO TEILA.
    MOVE 0 TO DIV.
    COMPUTE TEILDR = TEILDR / DIV.
BACK.
    EXIT PROGRAM.
```

Call of executable phase "PASCOB" and runtime listing:

```
/EXEC PASCOB ──────────────────────────────────────────────────────── (02)
%  BLS0500 PROGRAM 'PASCOB', VERSION ' ' OF '...' LOADED
TEILB: 00001111 ───────────────────────────────────────────────────── (03)
TEILA: TEST
ERROR INTERCEPTED BY EXCEPTION-HANDLER. ───────────────────────────── (04)
```

Explanation:

(01) The exception handler includes a test for whether a Numeric_Error has occurred or been passed on by a subprogram. If so, as is the case here, a message will be output. See (04). If not, the dynamic call chain will be output. For this purpose it is necessary to include the Errors package at the beginning of the program.

(02) Startup of the executable phase "PASCOB".

(03) Output from the Cobol subprogramm.

(04) Message issued by the exception handler from the Pascal-XT main program.

4. Unhandled runtime error in Pascal-XT/Fortran/Pascal-XT language interfacing

A Pascal-XT main program calls a Fortran subprogram; this, in turn, calls a Pascal-XT entry procedure. The Pascal-XT entry procedure calls the mathematical function SQRT for calculating the square root of a number, but passes a negative value. As negative values are illegal as parameters for this function, a runtime error (Numeric_Error) is raised and passed on to the caller, i.e. the Pascal-XT entry procedure. As the error is not handled there, it is passed on to the Fortran subprogram, and from there to the Pascal-XT main program. As the error is not handled in the Pascal-XT main program either, it leads to abortion of the program.

Source code of the Pascal-XT main program "PASFOPAS":

```
PROGRAM PASFOPAS (INPUT, OUTPUT);
VAR
   PAR1    :   LONG_REAL;
PROCEDURE FOPAS (VAR PAR1 : LONG_REAL); EXTERNAL; ———————————————————— (01)
BEGIN
   PAR1 := -88.1;
   FOPAS (PAR1); ———————————————————————————————————————————————————————— (02)
   WRITELN ('CALL TO FOPAS PROCEDURE');
   WRITELN ('PAR1: ',PAR1);
END.
```

Source code of the Fortran subprogram "FOPAS":

```
       SUBROUTINE FOPAS (VAR1)
       REAL*8  VAR1
       VAR1    = -1
       CALL PASPROZ (VAR1) ——————————————————————————————————————————— (03)
       WRITE(2,100) VAR1 ————————————————————————————————————————————— (04)
100    FORMAT (' VAR1: ',F6.2)
       RETURN
       END
```

Source code of the Pascal-XT entry procedure "PASPROZ" in the package "PASERR":

Package specification:

```
PACKAGE PASERR;
ENTRY PROCEDURE PASPROZ (VAR F1 : LONG_INTEGER); ————————————————————— (05)
END.
```

Package body:

```
PACKAGE BODY PASERR (OUTPUT);
PROCEDURE PASPROZ (VAR F1 : LONG_REAL);
VAR
   R  :  REAL;
BEGIN
   WRITELN ('F1: ',F1);
   R := SQRT(F1); ——————————————————————————————————————————————————————— (06)
END;
BEGIN
END.
```

Call of the executable phase "PAFOPAER" and runtime listing:

```
/EXEC PAFOPAER
% BLS0500 PROGRAM 'PAFOPAER', VERSION ' ' OF '...' LOADED
BS2000  F O R 1 : IMPROVED MATHEMATICAL ACCURACY
F1: -1.0E+00 ——————————————————————————————————————————————————————————————— (07)
    NUMERIC_ERROR (1004) RAISED FROM OUTSIDE PASCAL ENVIRONMENT——————————————— (08)
                            FROM    PASERR.PASPROZ   AT 00000C16 ENTRY
     FROM NON-PASCAL-ROUTINE(S) FROM  PASFOPAS.PASFOPAS  AT 000002A0
%  EXC0732 ABNORMAL PROGRAM TERMINATION. ERROR CODE 'NRT0101' /HELP-MSG NRT0101
%  CMD0205 ERROR IN PRECEDING COMMAND OR PROGRAM AND PROCEDURE STEP TERMINATION : COMMANDS ...
                    ... WILL BE IGNORED UNTIL /SET-JOB-STEP OR /LOGOFF OR /ABEND IS RECOGNIZED
```

Explanation:

(01) Declaration of the external procedure FOPAS.

(02) Call of the subprogram FOPAS: the variable PAR1 is passed with the value -88.1.

(03) Call of the Pascal-XT entry procedure PASPROZ: the variable VAR1 is passed with the value -1.

(04) Execution of the Fortran subprogram is interrupted at this point because the Pascal-XT entry procedure is not terminated in an orderly manner, but instead receives notification of a runtime error from the mathematical function SQRT; the procedure, in turn, passes the error on to its caller.

(05) Declaration of the Pascal-XT entry procedure PASPROZ in the specification of the PASERR package.

(06) Calling the square root function SQRT with a negative value raises a runtime error, which is passed on to the calling Pascal-XT entry procedure.

(07) Output of the variable F1 from the Pascal-XT entry procedure.

(08) Because a runtime error occurred in the mathematical routine SQRT and is not handled either in the calling Pascal-XT entry procedure or in the main program, the program is aborted and the dynamic call chain is output.
The first line of the dynamic call chain indicates that the error has been passed on by an external program section (in this case, by the mathematical routine SQRT) to a Pascal-XT program section.
The second line indicates the Pascal-XT program section (in this case, the Pascal-XT entry procedure PASPROZ) which first receives notification of the runtime error from the external program section.
The third line indicates that the runtime error has been passed on to one or more external program sections whose names are not given individually, and from there to the Pascal-XT main program PASFOPAS.

10.3 Detecting runtime errors

The following tables indicate all errors described in the Language Reference Manual and when they are detected. Section 10.4 gives the system error codes for each error class and provides additional information on error analysis.

NUMERIC_ERROR

Error	Error detection
- In an expression in the form x/y, y = 0.	Always detected
- In an expression in the form i DIV j, j = 0.	Always detected
- In an expression in the form i MOD j, j <= 0.	j=0 Always detected j<0 When Check=On
- The result of an arithmetic operation is not within the value range of the result type:	
- for operands of type Short_Integer or a subrange thereof, this is type Integer	Always detected
- for operands of type Long_Integer or of a subrange thereof, this is type Long_Integer	Always detected
- for operands of type Short_Real this is type Short_Real	Always detected
- for operands of type Long_Real this is type Long_Real	Always detected
- For Abs(x), the result of the function is not in the value range of the result type (Integer or Long_Integer or Short_Real or Long_Real).	Always detected
- For Sqr(x), the result of the function is not in the value range of the result type (Integer or Long_Integer or Short_Real or Long_Real).	Always detected
- The result of Exp(x) is not within the value range of the result type (Short_Real or Long_Real).	Always detected
- For Ln(x), x <= 0.	Always detected
- For Sqrt(x), x < 0.	Always detected
- The result of Trunc(x) or Short_Trunc(x) or Long_Trunc(x) is not in the value range of the result type (Integer or Short_Integer or Long_Integer).	When Check=On

Continued next page

Error (continued)	Error detection
- The result of Round(x) or Short_Round(x) or Long_Round(x) is not within the value range of the result type (Integer or Short_Integer or Long_Integer).	When Check=On
- In an expression in the form x**n - x is from an integer type and n < 0, or - x = 0 or x = 0.0 and n <= 0.	When Check=On
- In an assignment, the value of the expression (right side) of type Long_Real is not within the value range of the variable or of the function identifier (left side) of type Short_Real.	Cannot occur
- For the transfer of the value parameter, the value of the actual parameter of type Long_Real is not within the value range of the formal parameter of type Short_Real.	Cannot occur
- In an aggregate, the value of an aggregate element of type Long_Real is not within the value range of the associated aggregate component of type Short_Real.	Cannot occur
- When reading from a non-test file using Read(f,v) the value of the buffer variable f↑ of type Long_Real is not within the value range of the variable v of type Short_Real	Cannot occur
- When writing to a non-text file using Write(f,a) the value of the expression a of type Long_Real is not within the value range of buffer variable f↑ of type Short_Real.	Cannot occur

RANGE_ERROR

Error	Error detection
- In an assignment the value of the expression (right side) of an ordinal type is not within the value range of the type of the variable or of the function identifier (left side).	When Check=On
- For transfer of the value parameter the value of the actual parameter of an ordinal type is not in the value range of the type of formal parameter.	When Check=On
- In an aggregate the value of an aggregate element of an ordinal type is not within the value range of the type of the associated aggregate component.	When Check=On
- When reading from a non-text file using Read(f,v) the value of the buffer variable f↑ of an ordinal type is not within the value range of the type of variable v.	When Check=On
- When writing to a non-text file using Write(f,a) the value of expression a of an ordinal type is not within the value range of the type of the buffer variable f↑.	When Check=On
- Character value Chr(x) is not within the value range of type Char.	When Check=On
- The result of Succ(x) is not within the value range of type x.	When Check=On
- The result of Pred(x) is not within the value range of type x.	When Check=On
- When executing the statement within a FOR statement the begin or end value of the FOR statement is not within the value range of the type of the run variable.	When Check=On
- When reading an integer from a text file (using Read(f,x)) or from a character string expression using Readstring(s,x)) the value of the number is not within the value range of the variable x and no Read_Error (see below) has occurred.	When Check=On
- For Write(f,a:l1:l2) or Writestring(s,a:l1:l2) the total output length l1 is < 1 or the number of digits after decimal point l2 is <1. For Write(f,a:l1) or Writestring(s,a:l1) the total output length l1 is < 1 or l1 < 0 if a is of a string type.	When Check=On

SET_ERROR

Error	Error detection
- In an assignment the value of the expression (right side) of a set type is not within the value range of the type of the variable or of the function identifier (left side).	When Check=On
- For transfer of the value parameter the value of the actual parameter of a set type is not within the value range of the type of the formal parameter.	When Check=On
- In an aggregate the value of an aggregate element of a set type is not within the value range of the type of the associated aggregate component.	When Check=On
- When reading from a non-text file using Read(f,v) the value of the buffer variable f↑ of a set type is not within the value range of the type of variable v.	When Check=On
- When writing to a non-text file using Write(f,a) the value of the expression a of a set type is not in the value range of the type of buffer variable f↑.	When Check=On
- In a set constructor the value of an element definition is not within the value range of the basic type of the set constructor.	When Check=On
- For Setmin(s) or Setmax(s) the value of the expression s is equal to the empty set.	When Check=On

STRING_ERROR

Error	Error detection
- In an assignment the current length of the string (right side) is greater than the maximum length of the string type of the variable or of the function identifier (left side).	When Check=On
- In an assignment the current length of the string expression of a string type (right side) is not equal to the length of the fixed-length string type of the generalized variable or of the function identifier (left side) (10.1.2).	When Check=On
- For transfer of the value parameter the current length of the character string of the actual parameter is greater than the maximum length of the string type of the formal parameter.	When Check=On
- For transfer of the value parameter the current length of the character string of the actual parameter is not equal to the length of the fixed-length string type of the formal parameter.	When Check=On
- In an aggregate the current length of a character string of an aggregate element is greater than the maximum length of the string type of the associated aggregate component.	When Check=On
- In an aggregate the current length of a character string of an aggregate element (of a string type) is not equal to the length of the associated component of the aggregate (of a fixed-length string type).	When Check=On
- When reading from a non-text file with Read(f,v) the current length of the character string of a string type in the buffer variable f↑ is greater than the maximum length of the type in string variable v.	When Check=On
- When reading from a non-text file with Read(f,v) the current length of the character string of a string type in the buffer variable f↑ is not equal to the length of the fixed-length string type in variable v.	When Check=On
- When writing to a non-text file with Write(f,a) the current length of the character string is greater than the maximum length of the string type of the buffer variable f↑.	When Check=On
- When writing to a non-text file with Write(f,a) the current length of the character string a of a string type is not equal to the length of the buffer variable f↑ of a fixed-length string type.	When Check=On
- For Read(f,v) or Readstring(s,v) the maximum length of string variable v is less than the length of the character string read.	Always detected

Continued next page

Error	Error detection
- For Readstring(a,v1,...,vn) the character string expression a does not contain as many characters as requested by read parameter v1,...,vn.	Always detected
- For Writestring(s,p1,...,pn) the maximum length of string variable s is less than that of the character string formed from write parameters p1,...pn.	Always detected
- For Delete(s,i,l), i<1 or l<0 or (i+l-1) > Length(s).	When Check=On
- For Insert(s1,s2,i), i<1, or Length(s2) + Length(s1) > Maxlength(s2).	When Check=On
- For Substring(s,i,l), i<1, or l<0 or (i+l-1) > Length(s).	When Check=On
- For Pack(a,i,z) the maximum length of the string variable z is too short to accommodate all characters from unpacked array a as of index i.	When Check=On

INDEX_ERROR

Error	Error detection
- When indexing an array variable, an array constant, an array aggregate or a function result of an array type, the value of the index expression is not within the value range of the index type of the array type.	When Check=On
- When indexing a variable, a constant or a function result of a string type, the value of the index expression is less than 1 or greater than the current length of the character string.	When Check=On
- For a conformant array parameter, the index type of the actual parameter is not a subrange of the index type of the conformant array parameter.	When Check=On
- For Pack(a,i,z) the value of expression i is not within the value range of the index type of unpacked array parameter a.	When Check=On
- For Pack(a,i,z) the index range of a is exceeded when transferring components from the unpacked array a as of index i to packed array z.	When Check=On
- For Unpack(z,a,i) the ordinal value of expression i is not within the value range of the index type of unpacked array parameter a.	When Check=On
- For Unpack(z,a,i) the specified range in unpacked array a as of index i is too small to accommodate all components of packed array z.	When Check=On
- For Unpack(z,a,i) character string expression z contains more characters than can be transferred to unpacked array a as of index i.	When Check=On

POINTER_ERROR

Error	Error detection
- When dereferencing a pointer, the value of the variable, constant or function result of a pointer type is equal to nil.	When Check=On
- When calling Dispose(p), p has the value nil.	Always detected
- When calling Release(p), the reference value of p was not generated by a Mark call.	Always detected

VARIANT_ERROR

Error	Error detection
- A non-active variant of a variable, constant, aggregate or function result of a record type is accessed.	When Check=On

CASE_ERROR

Error	Error detection
- In a Case statement, no case constant corresponds to the value of the case index and no ELSE alternative has been specified.	When Check=On

FILE_ERROR

Error	Error detection
- Before calling Page(f), Put(f), Write(f,...) or Writeln(f,...), file f was not yet opened for writing.	Always detected
- Before calling Page(f), Put(f), Write(f,...) or Writeln(f,...), file f was still undefined.	Always detected
- Before calling Page(f), Put(f), Write(f,...) or Writeln(f,...) the current file position was not the end of file position, i.e. Eof(f) was false.	Always detected
- Before calling Get(f) or Read(f,...), file f was not yet opened for reading.	Always detected
- Before calling Get(f) or Read(f,...), file f was still undefined.	Always detected
- Before calling Read(f,...), buffer variable f↑ of file f was still undefined.	Always detected
- Before calling Eof(f), file f was still undefined.	When Check=On
- Before calling Eoln(f), file f was still undefined.	When Check=On
- For Assignfile(f,ext) the definition of the external file in operand "ext" is invalid.	Always detected

EOF_ERROR

Error	Error detection
- When calling Get(f) or Read(f,...) the end of file was already reached, i.e. Eof(f) is True.	Always detected
- When calling Eoln(f) the end of file was already reached, i.e. Eof(f) is True.	When Check=On

OPEN_ERROR

Error	Error detection
- When calling Reset or Rewrite the predefined text file Input or Output was specified.	Always detected
- When calling Reset(f) file f was still undefined.	Always detected
- For Reset(f), the file linked to f but outside the program was not opened for reading.	Always detected
- For Rewrite(f), the file linked to f but outside the program was not opened for writing.	Always detected

READ_ERROR

Error	Error detection
- When reading an integer or a real number from a text file (with Read(f,v)) or from a character string expression (with Readstring(s,v)), then either - the syntax of the character string read is not correct or - the character string read forms a number that cannot be represented internally. In the case of integers the value lies outside the range Long_Minint .. Long_Maxint and in the case of real numbers the value lies outside the range -Long_Maxreal .. Long_Maxreal.	Always detected

MEMORY_ERROR

Error	Error detection
- Execution of the program cannot be continued due to insufficient memory (e.g. when a subprogram is called or in the case of New).	Always detected

ELAB_ERROR

Error	Error detection
- Initialization of the packages of a program cannot be continued, as elaborate cycles are created on initialization as the result of the use of the predefined procedure.	Always detected

Miscellaneous errors (without error number assignment)

The errors given in the following table have no fixed error assignment. The occurrence of such errors is not detected and leads to secondary errors, the effects of which are undefined.

Error	Error detection
- In a statement the type of expression (right side) is of the generic pointer type and the pointer value of the expression refers to a dynamic variable whose type is different from the domain type of the type of generalized variable or function identifier (left side).	Not detected
- For transfer of the value parameter the type of the actual parameter is of the generic pointer type and the pointer value of the expression refers to a dynamic variable whose type is different from the domain type of the type of formal parameter.	Not detected
- In an aggregate the type of a pointer value is of the generic pointer type and the pointer value of the expression refers to a dynamic variable whose type is different from the domain type of the type of the associated aggregate component.	Not detected
- The length of a string variable is changed, although there is still a reference to a component of the string variable.	Not detected
- In Writestring(s,p1,...,pn), one of the write parameters p1,...,pn contains a reference to the string variable s.	Not detected
- In Convert(x,t) the storage representation of x does not represent a permissible value of type t.	Not detected

Continued next page

Error	Error detection
- The variant of a record variable is not active for the entire duration of each reference to any of its components.	Not detected
- The file pointer of a file variable f is changed (e.g. by reading or writing) although there is still a reference to buffer variable f.	Not detected
- When dereferencing a pointer the value of the variable, constant or function result of a pointer type is undefined.	Not detected
- With Dispose(q) a value pointing to a dynamic variable is deallocated although a reference to the dynamic variable still exists.	Not detected
- When calling Dispose(p) the value of p is still undefined.	Not detected
- Before calling Dispose(p), p↑ was already generated by New(p,c1,...,cn) or New(p,c1,...,cn,e) or New(p,e).	Not detected
- Before calling Dispose(p,k1,...,km), the dynamic variable p↑ was generated by New(p,c1,...,cn), where m is not equal to n.	Not detected
- In Dispose(p,k1,...,kn) or Dispose(p,k1,...kn,e) variants other than those specified by selector constants k1 through kn have been set in the dynamic variable p↑.	Not detected
- Before calling Dispose(p,e), p↑ was already generated by means of New(p,a), where a is not equal to e. By analogy this also applies to Dispose(p,c1,...,cn,e) and New(p,k1,...kn,a).	Not detected
- In Dispose(p,e) or Dispose(p,c1,..., cn, e) the value of e is not within the value range of the index type of the corresponding ARRAY type or is less than 1 or greater than the maximum length of the corresponding string type.	Not detected
- For an indexed ARRAY or string object the ARRAY or string object was generated in abbreviated form by calling New(p,e) or New(p,c1,...,cn,e) and the value of the index expression in the indexed object is greater than e.	Not detected
- In an indexed string object the value of the string object has not been defined (irrespective of the fact whether the indexed object occurs in an expression or, e.g., as a generalized variable on the left side of an assignment).	Not detected

Continued next page

Error	Error detection
- In a dynamic variable generated by New(p,c1,...,cn) or New(p,c1,...,cn,e) a variant other than the one specified by selector constants c1 through cn was set.	Not detected
- A dynamic string variable generated by New(p,e), or the last component of a dynamic string variable generated by New(p,c1,...,cn,e), is assigned a character string that is longer than e.	Not detected
- A dynamic variable generated by New(p,e), New(p,c1,...,cn) or New(p,c1,...,cn,e), occurs in its entirety in an expression or as the left side in a value assignment, or is transferred as the parameter.	Not detected
- In New(p,e) or New(p,c1,...,cn,e) the value of e is not within the value range of the index type of the corresponding ARRAY type, or is less than 1 or greater than the maximum length of the corresponding string type.	Not detected
- The reference value p, transferred when calling Release(p), was destroyed by another Release(q) call.	Not detected
- Before calling Put(f), Put(f,c1,...,cn), Put(f,e) or Put(f,c1,...,cn,e) the buffer variable f↑ is still undefined.	Not detected
- With abbreviated output of an array with the aid of Put(f,e) or Put(f,c1,...,cn,e) the value of the index expression e is not within the value range of the index type of the array.	Not detected
- With abbreviated output of a character string of variable length by means of Put(f,e) or Put(f,c1,...cn,e) the value of index expression e is less than 1 or greater than the current length of the character string.	Not detected
- A generalized variable used as the object in an expression, has an undefined value at the time the expression was evaluated.	Not detected
- The result of a function is undefined after the function block is executed if no value has been assigned to the function identifier.	Not detected
- In Unpack(z,a,i) one or the other component of the packed array z is undefined.	Not detected
- In Pack(a,i,z) a component of unpacked array a, which is undefined, is accessed [D.27].	Not detected
- The names of program parameters of the main program and all associated packages are not paired differently, with the exception of Input and Output.	Not detected
- The names of all packages belonging to a program and the name of the main program are not paired differently.	Not detected

10.4 System error codes

The following tables list the possible system error codes for the predefined exceptions, as returned by the system_code function of the predefined package ERRORS (see appendix A.7). System error codes greater than 9999 are system error messages and must first be converted to a hexadecimal number in order to query the corresponding texts of the errors using system command HELP. The meanings of the interrupt priorities are found in [6].

A Pascal program always supplies System_Code 0 if any of the exceptions Numeric_Error, Range_Error, Set_Error, String_Error, Index_Error, Pointer_Error, Variant_Error or Case_Error is detected as the result of the Check option being activated.

SYSTEM_ERROR (-1)

System_Code	Meaning
1201	Calls a 24-bit subprogram with parameters in the 31-bit range
1405	HEAPSUPPORT.select_heap was applied to an invalid heap pointer
1406	HEAPSUPPORT.release_heap was applied to an invalid heap pointer
1407	HEAPSUPPORT.release_heap was applied to the current_heap
1408	HEAPSUPPORT.release_heap was applied to the default_heap
1409	NEW and DISPOSE: These procedures cannot be invoked in UTM applications
1413	Error on return to the operating system from main memory
1414	No storage space for setting up a Glue Page
1503	Pascal file variable must be of the TEXT type
2250	Error in loading EDT
2251	Error in initializing EDT
2252	Error in SUBMIT_EDT
2254	Error in executing EDT
2255	Error in reading from virtual file
2256	SUBMIT_EDT: Line length is greater than 256
2257	Error in writing to the virtual file of EDT
3001	Error in calling the TABLE macro
4002	Error in the list of global files
4003	Error in the list of local files
5001	Reraise: No exception condition has yet occurred
5002	Error occurred in an external subprogram
other	Interrupt priority (see [6])

NUMERIC_ERROR (-2)

System_Code	Meaning
1001	EXP: The argument is too large
1002	LN: The argument is not positive
1003	SIN or COS: The absolute value of the argument is greater than or equal to pi * 2**50
1004	SQRT: The argument is negative
other	Interrupt priority (see [6])

STRING_ERROR (-5)

System_Code	Meaning
0	Error in a string assignment or in INSERT, DELETE or SUBSTRING (detected on account of {$CHECK=ON})
1012	WRITESTRING: The output length is invalid
1013	WRITESTRING: The string is too short to accept a character
1014	WRITESTRING: The string is too short to accept an integer value
1015	WRITESTRING: The string is too short to accept a real number
1016	WRITESTRING: The string is too short to accept a string
1017	WRITESTRING: The string is too short to accept a Boolean value
1025	READSTRING: The end of the string variable has been reached
1030	READSTRING, READ: The string variable is too short to accept the rest of the line

POINTER_ERROR (-7)

System_Code	Meaning
0	Dereferencing a NIL pointer (detected on account of {$CHECK=ON}) or a non-initialized pointer (detected on account of {$INITIALIZE=ON, CHECK=ON})
1404	RELEASE: The pointer value was not generated by means of MARK or MARK was invoked for another heap
1410	DISPOSE: Specification of an invalid pointer value
1411	DISPOSE: Specification of an invalid length
1412	DISPOSE: The memory to be released is reserved by a dynamic variable of another type
Other	Interrupt priority (see [6]) (addressing error detected by the hardware)

FILE_ERROR (-10)

System_Code	Meaning
1090	Buffer overflow for the text file Output 1)
1091	Buffer overflow for the text file Input 1)
1095	WRITE: The output length is invalid (also detected where Check=Off)
1096	PUT: The input length is invalid
1500	ASSIGNFILE: The length of the external description is invalid
1501	ASSIGNFILE: Syntax error in the external description
1502	ASSIGNFILE: Invalid specifications in the external description
2010	WRITELN: Device error
2011	READLN: Device error

Continued next page

1) A line that has been read in is longer than the internal buffer of the runtime system (see also Open_Error).

System_Code	Meaning
2045	Input/output to an unopened file
2046	The file is not permitted for this call
2049	UPDATE: Call with an invalid record length
2105	Library element cannot be closed
2106	Library cannot be closed
2109	PLAM access error
2201	Error when reading an EAM file
2202	Error in writing to an EAM file
2203	Error in closing an EAM file
4001	Error in the runtime system
Other	DMS error codes

OPEN_ERROR (-12)

System_Code	Meaning
1091	Buffer overflow on opening a text file [1)]
1603	The file name cannot be determined
1604	RESET: Open a non-existent file
1605	Invalid open mode for the file
1606	No more main memory available for block/record buffer
1607	RESET, REWRITE, REPLACE, EXTEND and CLOSE cannot be applied to the predefined Input and Output text files
1608	Incorrect or missing parameter in the FILE command (FCBTYPE, RECFORM, KEYLEN, KEYPOS, RECSIZE, BLKSIZE, SPACE)
1609	An attempt was made to open a file after an illegal Assignfile
2100	The system library for access to PLAM libraries does not exist
2101	Incorrect or unknown library type
2102	Library cannot be opened
2103	Library element not present
2104	Library element cannot be opened
2110	Invalid version specification for PLAM library elements
2200	Error on opening an EAM file
Other	DMS error codes

1) When opening a text file for reading the first line has already been read and can be longer than the internal buffer of the runtime system (see also File_Error).

READ_ERROR (-13)

System_Code	Meaning
1092	Error in reading an integer number
1094	Error in reading a real number

MEMORY_ERROR (-14)

System_Code	Meaning
1401	Main memory overflow in an internal request
1402	Main memory overflow in a stack request
1403	Main memory overflow in a heap request
1601	No more main memory available to create an FCB

A Appendix

A.1 Comparison between Pascal (BS2000) version 3.x and Pascal-XT

All language attributes of Pascal (BS2000) version 3.1B (abbreviated as "Version 3") and Pascal-XT are compared. The comparison is in keyword form only ; details can be found in the relevant language reference manuals.

In the tables below all attributes of the language are listed. The presence or absence of an attribute in a language is shown by:

X attribute present
– attribute absent.

The essential differences between Pascal version 3 and Pascal-XT are:

- Different approaches to separate compilation
- Different approaches to programmed exception handling
- Pascal-XT also includes level 1 of the ISO standard
- Aggregates (structured values) in Pascal-XT
- Static expressions as constants in Pascal-XT
- Influence on storage representation in Pascal-XT
- Inline subprograms in Pascal-XT
- Different standard procedures
- ISAM files are not integrated in the Pascal-XT language, but are instead provided in a package
- Include-mechanism omitted in Pascal-XT

Record lengths for non-text files

In certain cases a Pascal-XT program cannot process non-text files generated by Version 3. The reason for this is that in Version 3 the record length is always rounded to an integral multiple of the alignment of the component type of the file, as opposed to Pascal-XT. To read a file like this, the component type of the file in Pascal-XT must be expanded to include a dummy field, the size of which is the product of the difference between record lengths (see Example). The record length is calculated as follows:

```
T   is the component type of a file type (FILE OF T)
L   is the record length of the BS2000 file
LF  is the size of the record length field = 4  if RECFORM = V
                                           = 0  if RECFORM = F
sizeof  (T) :  size of the component type (in bytes)
alignof (T) :  alignment of the component type (1, 2, 4 or 8)

V3.x        :  L := ((sizeof (T) + LF) + alignof (T) - 1)
                           div alignof (T)  *  alignof (T)
               Therefore the size of the record is always the smallest
               integral multiple of the alignment of T, which is greater
               than (sizeof (T) + LF)

Pascal-XT   :  L := sizeof (T) + LF
               The size of the record is not rounded off.
```

Example

```
type
    rec = record                (* Definition of the record in Version 3 *)
              i: integer;
              c: char;
            end;
var
    my_file: file of rec;

The following applies to this file:
   sizeof  (rec) = 5
   alignof (rec) = 4
```

	Record length if RECFORM = V	Record length if RECFORM = F
V3.x	12	8
Pascal-XT	9	5

To read such a file generated by Version 3 the size of the record type "record" in Pascal-XT must be increased by 3 bytes:

```
type
    rec = record                (* Definition of the record in Pascal-XT *)
             i: integer;
             c: char;
             d: packed array [1..3] of char;  (* Dummy field *)
           end;
```

Keywords

Additional keywords in Pascal-XT are:

```
BODY      ENTRY      EXIT      EXCEPTION      FROM
INLINE    PACKAGE    USE
```

Compilation units

The approaches to separate compilation in the two languages are not compatible. Version 3 takes a modular approach, with no checking of interfaces. In Pascal-XT, interfaces and package body are separate and interface checking is done at compile time.

Compilation unit	Version 3	Pascal-XT
program	X	X
procedure module	X	are covered by the package approach
function module	X	
module	X	
entry procedure	X	
package	-	X
package body	-	X

Directives and language interfaces

In Version 3 all external subprograms are designated by the directive "external". In Pascal-XT, for COBOL subprograms the directive "cobol" is to be specified, for FORTRAN subprograms the directive "fortran" and for all other subprograms the directive "external".
For language interfacing using a mix of program sections, Pascal-XT allows any calling sequence without explicit initializations; Version 3 requires initializations of the runtime environment to be specified explicitly.
Entry procedures in Version 3 are a separate compilation unit, in Pascal-XT they are defined in package specifications (see compilation units).

Directive	Version 3	Pascal-XT
module	X	-
forward	X	X
internal	X	X
external	X	X
fortran	-	X
cobol	-	X

Pseudo-comments and compiler options

The compiler options of the two compilers are not compatible. Approximately equivalent options are compared in the table, but in general they do not cover the same functions. Version 3 provides the control statements SKIPON and SKIPOFF for conditional compilation of program parts. In Pascal-XT this can be simulated with the IF statement, if the IF condition is a static expression. If the condition has the value "True", only code for the THEN part is generated; otherwise only code for the ELSE part is generated. In a CASE statement with static case index only code for the corresponding case list element is generated.

Version 3	Pascal-XT
(*%MAP*)	} the two control statements are
(*%COPY*)	} covered by the package concept
(*%SKIPON*)	see above
(*%SKIPOFF*)	see above
(*$...*)	(*$...}
R, E, K, F, H, S	-
J, Q, V, T, Z	-
O	OPTIMIZE = on | off
D	CHECK = on | off
G	GENERATE = on | off
N	automatic, if testing with PATH
Y, U	DEBUG = on | off | restricted
L	LIST = on | off
L n	-
X	XREF = on | off
P	PAGE (without title specification)
-	TITLE = '...' (without page change)
C	ASSEMBLER = on | off
W	STANDARD = on | off
I	MAP = on | off

Program parameter list

In Version 3 the buffer size for external text files can be specified following the file identifier in the program parameter list. In Pascal-XT the buffer size is fixed.

Constant definitions

In Pascal-XT constant definitions may define constants of any type, even of structured types. Any static expression may appear on the right side of the definition.

Constant definition	Version 3	Pascal-XT
character literal (char)	X	X
enumerated type (boolean)	X	X
numbers (integer, real)	X	X
strings	X	X
constant identifier	X	X
static expression	-	X
- arithmetic or logical expression	-	X
- variable string expression	-	X
- (qualified) set constructor	-	X
- array aggregate	-	X
- record aggregate	-	X

Predefined constants

Identifier / keyword	Version 3	Pascal-XT
TRUE	X	X
FALSE	X	X
MAXINT	X	X
SHORT_MAXINT	-	X
LONG_MAXINT	-	X
MININT	-	X
SHORT_MININT	-	X
LONG_MININT	-	X
MINREAL	X	X
SHORT_MINREAL	-	X
LONG_MINREAL	-	X
MAXREAL	X	X
SHORT_MAXREAL	-	X
LONG_MAXREAL	-	X
NIL	X	X

Predefined types and new types

In Pascal-XT constants in type definitions may also be static expressions. String types are defined in accordance with the ISO standard as packed array [1..n] of char, variable string types as string [n], where n specifies the maximum string length. In Version 3 a string type need not be packed and the index lower limit may be a value other than 1. Pascal-XT adheres strictly to the ISO standard.

Type	Version 3	Pascal-XT
CHAR	X	X
BOOLEAN	X	X
INTEGER	X	X
SHORT_INTEGER	-	X
LONG_INTEGER	-	X
REAL	X	X
SHORT_REAL	-	X
LONG_REAL	-	X
enumerated type	X	X
subrange type	X	X
pointer type	X	X
generic pointer type	-	X
private pointer type	-	X
string type	X	X (as per ISO standard)
variable string type	X	X
- default length =	252	254
- length field =	4	2
- maximum length =	$2^{16}-1$	$2^{15}-1$
record type	X	X
- specifications regarding representation	-	X
- else part in the variant part	-	X
- range specification for selector constants	-	X
array type	X	X
file type	X	X
- ISAM files	X	-
generic file type	-	X
set type	X	X
corid type	X	-

Variables, expressions and operators

In Pascal-XT, in addition to components of variables there are also components of values of a structured type.

Expressions which can be evaluated at compile time are called static expressions. In Pascal-XT static expressions can be used as constants both in constant definitions and in places where the ISO standard requires a constant.

Operators

Operator	Version 3	Pascal-XT
monadic operators:		
+	X	X
-	X	X
arithmetic operators:		
+	X	X
-	X	X
*	X	X
/	X	X
** (exponentation)	-	X
div	X	X
mod	X	X
Boolean operators:		
or	X	X
and	X	X
not	X	X
or else	-	X
and then	-	X
set operators:		
+	X	X
-	X	X
*	X	X
/ (symmetric difference)	-	X
relational operators:		
=	X	X
<>	X	X
<	X	X
>	X	X
<=	X	X
>=	X	X
in	X	X

In comparisons, string types in Version 3 may be of different lengths; in Pascal-XT, in accordance with the ISO standard, the string types must contain the same number of characters. With variable string types in both languages only the current length is relevant.

Statements

Statement	Version 3	Pascal-XT
Assignment [1)]	X	X
goto	X	X
exit	-	X
return	X	X
Compound statement	X	X
if	X	X
case	X	X
- Ranges for case constants	-	X
repeat	X	X
while	X	X
for	X	X
with	X	X
Procedure calls	X	X
Inline procedure calls	-	X

1) In Version 3 a variable of a string type may also be assigned a shorter string; in Pascal-XT, in accordance with the ISO standard, the string expression must be of the same type as the string variable.

Procedure and function declarations and formal parameters

In Pascal-XT the formal parameter list can be repeated.

Entry procedures

In Version 3 entry procedures are independent compilation units. In Pascal-XT entry procedures may be declared only in package specifications.

Inline subprograms

In Pascal-XT procedures and functions may be declared as inline.

Value parameters

With formal parameters of a string type, in Version 3 the actual parameters may also be string expressions of a shorter length. In Pascal-XT, in accordance with the ISO standard, the actual parameters must have the same types. Pascal-XT also permits conformant array schemata.

Variable parameters

Pascal-XT also permits conformant array schemata.

Procedural parameters and functional parameters

No differences.

Programmed exception handling

The approaches to programmed exception handling in the two languages are completely different. In Version 3 an exception handler counts as being activated from the point of its activation until a new one is activated or until the block that directly contains details of its activation ends. In Pascal-XT an exception handler may be defined for an individual compound statement or for the entire body of a subprogram.

Standard procedures and standard functions

Procedure/function	Version 3	Pascal-XT
PACK	X	X
UNPACK	X	X
ABS	X	X
SQR	X	X
SIN	X	X
COS	X	X
EXP	X	X
LN	X	X
SQRT	X	X
ARCTAN	X	X
TRUNC	X	X
SHORT_TRUNC	-	X
LONG_TRUNC	-	X
ROUND	X	X
SHORT_ROUND	-	X
LONG_ROUND	-	X
LONG	-	X
ORD	X	X
CHR	X	X
SUCC	X	X
PRED	X	X
ODD	X	X

Procedures for input/output

Procedure/function	Version 3	Pascal-XT
RESET	X	X
REWRITE	X	X
GET	X	X
PUT	X	X
PUT (f,c1,...,cn,e)	X	X
EOF	X	X
READ	X	X
READ (f,str:len)	X	-
WRITE	X	X
WRITE (f,str:len1:len2)	X	-
WRITE (set:len HEX)	X	-
WRITE (f,expr HEX)	X	-
READLN	X	X 3)
WRITELN	X	X 3)
EOLN	X	X
PAGE	X	X
PAGE (f,exp1,exp2,...)	X	-
EXTEND	X	provided in the predefined package DMSIO
REPLACE	X	provided in the predefined package DMSIO
CLOSE	X	provided in the predefined package DMSIO
ELIM	X	provided in the predefined package DMSIO
ELIMKEY	X	provided in the predefined package DMSIO
GETKEY	X	provided in the predefined package DMSIO
GETBACK	X	provided in the predefined package DMSIO
UPDATE	X	provided in the predefined package DMSIO
INSERT	X	provided in the predefined package DMSIO
POS	X	provided in the predefined package DMSIO
POSKEY	X	provided in the predefined package DMSIO
POSBACK	X	provided in the predefined package DMSIO
MOVEKEY	X	provided in the predefined package DMSIO
NOKEY	X	provided in the predefined package DMSIO
LOCK	X	provided in the predefined package DMSIO
RECLEN	X	provided in the predefined package DMSIO
KEYPOS	-	provided in the predefined package DMSIO
KEYLEN	-	provided in the predefined package DMSIO
CLPAM	X	-
OPPAM	X	-
RDPAM	X	-
WRPAM	X	-
EAM	X	via local files

Procedures for heap management

Procedure/function	Version 3	Pascal-XT
NEW	X	X
DISPOSE	X	X
MARK	X	X
RELEASE	X	X
PUSHEAP	X	} can be emulated with the functions of the predefined package HEAPSUPPORT
POPHEAP	X	
KILLHEAP	X	
RELMEM	X	} provided in the predefined package MEMORYMANAGER
REQMEM	X	
GETMAP	X	
CHPTR (p,e)	X	- 1)
ASSPTR (p,q)	X	- 2)

1) p := CONVERT (CONVERT (p,integer)+e, pointer)

2) p := CONVERT (q, pointer)

3) For generation of an empty line in Pascal-XT at least one blank must be output.

Procedures for programmed exception handling

Procedure/function	Version 3	Pascal-XT
RAISE	X	X
ERROREXIT	X	-
SYSERROR	X	-
ERROR_NUMBER	-	X

Subprograms for string processing

Procedure/function	Version 3	Pascal-XT
DELETE	X	X
INSERTSTRING	X	X = INSERT
READSTRING	X	X
WRITESTRING	X	X
LENGTH	X	X
POSITION	X	X
CONCAT	X	X
SUBSTRING	X	X

Attribute functions

Function	Version 3	Pascal-XT
ALIGNOF	X	X 1)
SIZEOF	X	X 1)
BITSIZEOF	-	X 1)
CARD	X	X
SETMIN	-	X
SETMAX	-	X
FIRST	-	X
LAST	-	X
MAXLENGTH	-	X

1) Applicable to variables and types

Co-routines

Co-routines are at present not supported in Pascal-XT.

Procedure/function	Version 3	Pascal-XT
ALLOCATE	X	-
NEWCOR	X	-
TRANSFER	X	-
ENDCOR	X	-
USED	X	-

Other subprograms

Version 3	Pascal-XT
-	CONVERT (unchecked type conversion)
-	ELABORATE (expl. package initialization)
EDT	EDTADAPTER . call_edt
EDT (cmd)	EDTADAPTER . submit_edt (cmd)
EDP	-
EDOR	-
WRTRD	-
GETSWITCH BREAK CMD	} these and additional BS2000 macro calls are provided in the predefined package BS2000CALLS
DATE TIME	} are provided in modified form by the predefined package CLOCK
COMPDATE	-
VERSION	-
ASSEXT	-
LINK	-
LINKCALL	-
STXIT	is not required
FINALIZE	is not required

A.2 Compiler listings

The various compiler listings generated for a sample program are shown below. Repetition of the prolog and the source listing is omitted in the assembler listing, cross-reference listing and map listing. The error listing arose as the result of the insertion of a few errors into the sample program.

The sample program used the TOOLS package from the PLAM library PLAM.SUPPORT. For a given integer the hex function from this package returns the corresponding hexadecimal value as a string.

```
*** SOURCE LISTING ***    BS2000 SIEMENS PASCAL-XT COMPILER  V2.2A...


GLOBAL OPTIONS FOR THIS COMPILATION

DEBUG      =    OFF          BY DEFAULT
GENERATE   =    ON           BY DEFAULT
MAP        =    OFF          BY DEFAULT
STANDARD   =    OFF          BY DEFAULT
XREF       =    OFF          BY DEFAULT


LIST OF RECOMPILED PACKAGE SPECIFICATIONS (SOURCE FILES)

               ($USERID.PLAM.MANUAL,TOOLS.SPEC(*STD,S))



CURRENT COMPILATION UNIT (SOURCE FILE)

               ($USERID.PLAM.MANUAL,EXAMPLE.PROG(*STD,S))

     1      with TOOLS;
     2
     3      program EXAMPLE (input,output);
     4
     5      const
     6            txt = 'hexadecimal value = #';
     7
     8      var
     9            i : integer;
    10
    11      begin
    12         repeat
    13            read    (input, i);
    14            writeln (output, txt, TOOLS.hex(i));
    15         until i = 0;
    16      end (* EXAMPLE *).

***********************************************************
*                   COMPILATION SUMMARY                   *
***********************************************************
*  ERRORS DETECTED         :        0                     *
*  WARNINGS                :        0                     *
*  NOTES                   :        0                     *
*  SIZE OF CODE MODULE     :      540 BYTES               *
*  SIZE OF DATA MODULE     :      104 BYTES               *
*  COMPILATION TIME        :    0.142 SEC                 *
***********************************************************
```

```
*** SOURCE LISTING ***    BS2000 SIEMENS PASCAL-XT COMPILER  V2.2A...

GLOBAL OPTIONS FOR THIS COMPILATION

DEBUG       =    OFF           BY DEFAULT
GENERATE    =    ON            BY DEFAULT
MAP         =    OFF           BY DEFAULT
STANDARD    =    OFF           BY DEFAULT
XREF        =    OFF           BY DEFAULT

CURRENT COMPILATION UNIT (SOURCE FILE)

               ($USERID.PLAM.MANUAL,ERROR.PROG(*STD,S))

     1      {$Check=Of}
--------------1
>>> 1: NOTE       3: INVALID ARGUMENT NAME OF OPTION
>>>    NOTE       5: OPTION IGNORED

     2      prgram ERROR (input);
------------1
>>> 1: ERROR    723: REPLACED BY "PROGRAM"

     3
     4      const
     5          toobig = maxint + 1;
--------------------------------1
>>> 1: ERROR    404: NUMERIC ERROR IN COMPUTATION

     6
     7      var
     8         i,j : integer;
-----------------1
>>> 1: NOTE      50: IDENTIFIER IS DECLARED BUT NOT USED

     9
    10      begin
    11         for i := 1 to 10 do
    12            writeln (output, 2 ** i);
------------------1--------2
>>> 1: ERROR    361: STANDARD FILE MUST BE PROGRAM PARAMETER
>>> 2: ERROR    302: UNDECLARED OR INVISIBLE IDENTIFIER

    13         i := maxint + 1;
---------------------------1
>>> 1: WARNING 3404: NUMERIC ERROR IN COMPUTATION

    14      end (* ERROR *).

***********************************************************
*                   COMPILATION SUMMARY                   *
***********************************************************
*  ERRORS DETECTED         :        4                     *
*  WARNINGS                :        1                     *
*  NOTES                   :        3                     *
*  SIZE OF CODE MODULE     :        0 BYTES               *
*  SIZE OF DATA MODULE     :        0 BYTES               *
*  COMPILATION TIME        :    0.070 SEC                 *
***********************************************************
```

```
** ASSEMBLER LISTING *    BS2000 SIEMENS PASCAL-XT COMPILER  V2.0A...

     PROCEDURE: EXAMPLE

     LOCTN   OBJECT CODE     ADDR1  ADDR2    STMNT    ASSEMBLY CODE

     000098  C2C5C9E2D7C9C5D3                    1             DC    C'BEISPIEL'
     0000A0  00000088                            2             DC    F'136'
     0000A4  00000238                            3             DC    A(START+568)
     0000A8  0000018E                            4             DC    F'398'
     0000AC  00000000                            5             DC    F'0'
     0000B0  00000000                            6             DC    A(START+0)
     0000B4  50 C0 B030                          7             ST    12,48(0,11)
     0000B8  50 90 B0AC                          8             ST    9,172(0,11)
     0000BC  50 A0 B0B0                          9             ST    10,176(0,11)
     0000C0  50 B0 B0B4                         10             ST    11,180(0,11)
     0000C4  50 D0 B034                         11             ST    13,52(0,11)
     0000C8  58 30 A018      0000B0             12             L     3,24(0,10)
     0000CC  58 10 308C                         13             L     1,140(0,3)
     0000D0  58 F0 1028                         14             L     15,40(0,1)
     0000D4  58 10 D05C                         15             L     1,92(0,13)
     0000D8  50 10 B068                         16             ST    1,104(0,11)
     0000DC  58 D0 D05C                         17             L     13,92(0,13)
     0000E0  45 E0 9030                         18             BAL   14,48(0,9)
     0000E4  58 D0 B034                         19             L     13,52(0,11)
     0000E8  58 30 A018      0000B0             20             L     3,24(0,10)
     0000EC  58 10 3090                         21             L     1,144(0,3)
     0000F0  58 F0 1028                         22             L     15,40(0,1)
     0000F4  58 20 D060                         23             L     2,96(0,13)
     0000F8  50 20 B06C                         24             ST    2,108(0,11)
     0000FC  58 D0 D060                         25             L     13,96(0,13)
     000100  45 E0 9030                         26             BAL   14,48(0,9)
     000104  58 D0 B034                         27             L     13,52(0,11)
     000108  58 30 A018      0000B0             28             L     3,24(0,10)
     00010C  58 00 D030                         29             L     0,48(0,13)
     000110  5A 00 90C8                         30             A     0,200(0,9)
     000114  50 00 D030                         31             ST    0,48(0,13)
     000118  5B 00 90C0                         32             S     0,192(0,9)
     00011C  50 00 B048                         33             ST    0,72(0,11)
     000120  58 30 3094                         34             L     3,148(0,3)
     000124  50 30 B07C                         35             ST    3,124(0,11)
     000128  58 F0 3028                         36             L     15,40(0,3)
     00012C  58 D0 D064                         37             L     13,100(0,13)
     000130  05 E9                              38             BALR  14,9
     000132  58 D0 B034                         39             L     13,52(0,11)
     000136  58 30 A018      0000B0             40             L     3,24(0,10)
     00013A  58 10 B068                         41             L     1,104(0,11)
     00013E  58 20 B06C                         42             L     2,108(0,11)
     000142  41 00 1034                         43             LA    0,52(0,1)
     000146  50 00 B080                         44             ST    0,128(0,11)
     00014A  58 10 3080                         45             L     1,128(0,3)
     00014E  50 10 B070                         46             ST    1,112(0,11)
     000152  41 20 2034                         47             LA    2,52(0,2)
     000156  50 20 B074                         48             ST    2,116(0,11)
     00015A  41 30 302C                         49             LA    3,44(0,3)
```

```
00015E  50 30 B078                 50          ST    3,120(0,11)
                                   51    ***** LINE 13
000162  50 00 B048                 52          ST    0,72(0,11)
000166  58 F0 1038                 53          L     15,56(0,1)
00016A  58 D0 D050                 54          L     13,80(0,13)
00016E  45 E0 9030                 55          BAL   14,48(0,9)
000172  58 D0 B034                 56          L     13,52(0,11)
000176  58 10 B070                 57          L     1,112(0,11)
00017A  58 20 B074                 58          L     2,116(0,11)
00017E  58 30 B078                 59          L     3,120(0,11)
000182  41 00 0013      000013     60          LA    0,19(0,0)
000186  58 40 B04C                 61          L     4,76(0,11)
00018A  50 40 D034                 62          ST    4,52(0,13)
                                   63    ***** LINE 14
00018E  50 20 B048                 64          ST    2,72(0,11)
000192  50 30 B04C                 65          ST    3,76(0,11)
000196  50 00 B050                 66          ST    0,80(0,11)
00019A  50 00 B054                 67          ST    0,84(0,11)
00019E  58 F0 1058                 68          L     15,88(0,1)
0001A2  58 D0 D050                 69          L     13,80(0,13)
0001A6  45 E0 9030                 70          BAL   14,48(0,9)
0001AA  58 D0 B034                 71          L     13,52(0,11)
0001AE  58 30 B07C                 72          L     3,124(0,11)
0001B2  58 20 D034                 73          L     2,52(0,13)
0001B6  41 10 B060                 74          LA    1,96(0,11)
0001BA  58 F0 302C                 75          L     15,44(0,3)
0001BE  58 D0 D064                 76          L     13,100(0,13)
0001C2  05 E9                      77          BALR  14,9
0001C4  58 D0 B034                 78          L     13,52(0,11)
0001C8  58 10 B070                 79          L     1,112(0,11)
0001CC  58 20 B074                 80          L     2,116(0,11)
0001D0  41 00 B060                 81          LA    0,96(0,11)
0001D4  41 30 0008      000008     82          LA    3,8(0,0)
0001D8  50 20 B048                 83          ST    2,72(0,11)
0001DC  50 00 B04C                 84          ST    0,76(0,11)
0001E0  50 30 B050                 85          ST    3,80(0,11)
0001E4  50 30 B054                 86          ST    3,84(0,11)
0001E8  58 F0 1058                 87          L     15,88(0,1)
0001EC  58 D0 D050                 88          L     13,80(0,13)
0001F0  45 E0 9030                 89          BAL   14,48(0,9)
0001F4  58 D0 B034                 90          L     13,52(0,11)
0001F8  58 10 B070                 91          L     1,112(0,11)
0001FC  58 20 B074                 92          L     2,116(0,11)
000200  50 20 B048                 93          ST    2,72(0,11)
000204  58 F0 105C                 94          L     15,92(0,1)
000208  58 D0 D050                 95          L     13,80(0,13)
00020C  45 E0 9030                 96          BAL   14,48(0,9)
000210  58 D0 B034                 97          L     13,52(0,11)
000214  58 00 B080                 98          L     0,128(0,11)
000218  58 10 B070                 99          L     1,112(0,11)
00021C  58 20 D034                100          L     2,52(0,13)
000220  12 22                     101          LTR   2,2
000222  47 70 A0CA      000162    102          BC    7,202(0,10)
000226  58 E0 B038                103          L     14,56(0,11)
00022A  58 90 B024                104          L     9,36(0,11)
00022E  58 A0 B028                105          L     10,40(0,11)
000232  58 B0 B02C                106          L     11,44(0,11)
000236  07 FE                     107          BCR   15,14
```

```
*** XREF LISTING ***      BS2000 SIEMENS PASCAL-XT COMPILER  V2.0A...

hex
   FUNCTION ... ARRAY .......................... AT     7 IN TOOLS

       14

i
   VARIABLE ... INTEGER ....................... AT     9

       13    14    15

integer
   TYPE ....... INTEGER .... 4 ................. PREDEFINED

        9

read
   PROCEDURE .................................. PREDEFINED

       13

TOOLS
   PACKAGE .................................... AT     1

       14

txt
   CONSTANT ... ARRAY .......................... AT     6

       14

writeln
   PROCEDURE .................................. PREDEFINED

       14


*** MAP LISTING ***       BS2000 SIEMENS PASCAL-XT COMPILER  V2.0A...

                    PROCEDURE ENTRY VECTOR

       PEV-ADDRESS      MODULE-OFFSET        PROCEDURE / FUNCTION
      40 (00000028)       0 (00000000)   INITIAL PROCEDURE

                    GLOBAL CONSTANTS OF THE UNIT

     MODULE-OFFSET     TYPE       NAME          VALUE
      44 (0000002C)   STRING   txt             'hexadecimal value = #'

                    GLOBAL VARIABLES OF THE UNIT

      52 (00000034)   i
```

A.3 Predefined packages

Together with the Pascal-XT programming system, users are offered a series of predefined packages providing frequently used functions and procedures. Users can employ these packages just as if they had written them themselves.

The specifications of the predefined packages are provided in PLAM library **PASSUP-XT**; the object code of the packages is contained in the library of the runtime system, namely PASLIB-XT. Predefined packages for UTM applications are described in [11].

Notes

The predefined packages are not portable.

The predefined package CLOCK is described in the Language Reference Manual [1] because it is provided by all Pascal-XT implementations. The specification of the package is contained in the library PASSUP-XT, the body in the library PASLIB-XT of the runtime system.

The package names of the predefined packages cannot be used as program or package names in an application program if these predefined packages are to be used.

To be able to use the predefined packages they must first be made known in the defined project directory. This is accomplished simply by compiling the package specifications of the required packages. Each package specification is stored in an element of the library (of PASSUP-XT), whose element name is formed from the package name and the suffix ".SPEC".

Example

```
/EXEC $PASCAL-XT
%  BLS0500 PROGRAM 'PASCALXT', VERSION '22A00' OF ...  LOADED<
%  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG.1990. ALL RIGHT
S RESERVED
//DEFINE DIRECTORY
//MC ($PASSUP-XT,), *DUMMY
//C (,DMSIO.SPEC)
//C (,BS2000CALLS.SPEC)
```

A.4 BS2000CALLS

The BS2000CALLS package provides macros specific to BS2000.

```
(*********************************************************************
*                                                                   *
*       COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG 1990   *
*                    ALL RIGHTS RESERVED                            *
*                                                                   *
*********************************************************************)
{$DEBUG=ON}
package BS2000CALLS;

            (**********************************************)
            (*  The body of this package is part of the     *)
            (*  Pascal-XT runtime system.                   *)
            (*                                              *)
            (*  Required Pascal-XT Runtime System: >= 2.1A  *)
            (*  Available in V2.1A and higher:              *)
            (*     CREATE_CMD_BUFFER                        *)
            (*     RELEASE_CMD_BUFFER                       *)
            (*     RESET_CMD_BUFFER                         *)
            (*     CMDX                                     *)
            (*     GET_CMD_LINE                             *)
            (**********************************************)

const

   batch_task    =  0;


type

   switches      = set of 0 .. 31;
   ret_code      = packed array [ 1..4 ] of char;
   byte          = 0..255;
   cmdstring     = string [254];
   name8         = packed array [ 1..8 ] of char;

   unpacked_decimal = packed array [ 1..4 ] of char;

   taskinfo = packed record
      tasktype    (0):  byte;
      bufsize     (1):  short_integer;
      priority    (3):  byte;
      tsn         (4):  unpacked_decimal;
      user_id     (8):  name8;
      account    (16):  name8;
      time       (24):  long_integer;
      privilege  (28):  byte;
      line_length(29):  byte;
      station    (30):  byte;
      prog_name  (32):  name8;
      logon_name (40):  name8;
      end;


function tmode : taskinfo;
   (* returns attributes of the user process                        *)

procedure break;
   (* interrupts the program currently running                      *)

procedure cmd (     command: string;
                var errors: boolean );
   (* the (BS2000-) command is executed; errors returns true, if an *)
   (* error occurred, otherwise false.                              *)
```

```
function get_switches : switches;
   (* returns the currently set process switches                       *)

procedure set_switches     ( s: switches );
   (* switch i is set if element (31 - i) is set in s                  *)

procedure reset_switches   ( s: switches );
   (* switch i is reset if element (31 - i) is set in s                *)

procedure set_return_code  ( c: ret_code );
   (* return code c will be set at program termination                 *)

function  return_code: ret_code;
   (* returns the return code set by the last call of                  *)
   (* set_return_code                                                  *)

procedure create_cmd_buffer ( var buffer : pointer );
   (* creates a new buffer of 32 KB and returns the pointer            *)
   (* of this buffer; returns nil, if allocation failed                *)

procedure release_cmd_buffer ( var buffer : pointer );
   (* returns the buffer of 32 KB to the BS2000-System                 *)
   (* buffer is set to NIL                                             *)

procedure reset_cmd_buffer ( buffer : pointer );
   (* after this procedure was called the buffer@ can be read          *)
   (* again from the beginning                                         *)

procedure cmdx (     command: string;
                     sysout:  boolean;
                     buffer:  pointer;
                 var errors:  boolean );
   (* Precondition:
   (* command contains the bs2000-command; sysout contains true, if *)
   (* the output of the command shall be directed both to SYSOUT    *)
   (* and buffer@, sysout contains false, if the output shall be    *)
   (* directed only to buffer@. The information (strings) in        *)
   (* buffer@ can be read by means of procedure GET_CMD_LINE;       *)
   (* Postcondition:                                                *)
   (* error = false  the command was executed and the return        *)
   (*                information is available in buffer@            *)
   (*       = true   an error occurred, the contents of buffer@ is  *)
   (*                undefined.                                     *)

procedure get_cmd_line (     buffer: pointer;
                         var line:   cmdstring;
                         var more:   boolean );
   (* Precondition:                                                 *)
   (* buffer@ contains the value returned by a previous call of CMDX*)
   (* Postcondition:                                                *)
   (* the first call of GET_CMD_LINE after a call of CMDX returns   *)
   (* the first line of buffer@, the following calls of GET_CMD_LINE*)
   (* return the next lines of buffer@ until the end of the         *)
   (* information in buffer@ is reached. In this case the empty     *)
   (* string is returned. The next call of GET_CMD_LINE again       *)
   (* returns the first line of buffer@. More is true, if the       *)
   (* buffer's line is longer than cmdstring. With following calls  *)
   (* of GET_CMD_LINE the remainder of the line can be read.        *)

end (* package BS2000CALLS *).
```

TMODE

returns status information on the user task.

BREAK

causes a program interrupt and transfer to BS2000 command mode. Execution of the program can be continued with the BS2000 command RESUME.

CMD (command, errors)

permits execution of a BS2000 command from within a Pascal program. The string specified in this command is interpreted as a BS2000 command (macro CMD). The Boolean variable "errors" returns the value "True" if the command was executed with errors, otherwise "errors" has the value "False". Both uppercase and lowercase letters can be specified in the command.

Caution is recommended when using commands which overload the Pascal program and therefore terminate it, such as EXEC.

GET_SWITCHES

supplies the set of task switches currently set. The order of the switches is reversed compared to the descriptions in [6] and [7], i.e. switch i is set if (31-i) is set in the returned set.

SET_SWITCHES (s)

sets the task switches contained in set s. The switches are numbered from 0 through 31. The order is reversed as compared to the descriptions in [6] and [7], so that switch i is set if (31-i) is applicable in s.

RESET_SWITCHES (s)

resets the task switches contained in set s. The switches are numbered from 0 through 31. The order is reversed as compared to the descriptions in [6] and [7], so that switch i is reset when (31-i) is applicable in s.

SET_RETURN_CODE (code)

accepts "code" as the termination code for the TERM macro to be executed at program end (see [6]). If several SET_RETURN_CODE macros are present in a program, the code specified in the last macro is used as the termination code in the TERM macro. If a job variable was specified for program monitoring when the program was called, "code" is transferred to this job variable when the program is terminated.

RETURN_CODE

supplies the termination code last set with SET_RETURN_CODE.

CREATE_CMD_BUFFER (buffer)

This procedure requests 32 K of main memory. This is the maximum size of the result area for the SYSOUT log when the CMD macro is called.

buffer after the execution of CREATE_CMD_BUFFER it contains the address of the result area. If the request for memory cannot be satisfied, then "buffer" contains the value NIL.

RELEASE_CMD_BUFFER (buffer)

By calling this procedure the memory area requested by CREATE_CMD_BUFFER is returned to the operating system.

buffer contains the address of the result area to be returned and is then subsequently set to NIL.

RESET_CMD_BUFFER (buffer)

This procedure is used to reposition the pointer to the start of the buffer so that the lines can be read through again.

buffer contains the address of the result area.

CMDX (command, sysout, buffer, errors)

This procedure enables a BS2000 command to be issued together with the transfer of the SYSOUT log into the buffer requested by CREATE_CMD_BUFFER. The SYSOUT log is returned in an unedited form and therefore access to this information should only be effected via the GET_CMD_LINE procedure.

command	contains the BS2000 command to be issued.
sysout	if "sysout" is TRUE, then an additional output of the log occurs (through the system macro) to the system file SYSOUT.
buffer	buffer, in which the information supplied from the system macro cmd is stored. This buffer must have previously been created using CREATE_CMD_BUFFER.
errors	is TRUE, if an error occurs when issuing the command, otherwise it is FALSE.

GET_CMD_LINE (buffer, line, more)

Using this procedure lines can be read from the buffer one at a time without knowing the block/record structure specific to the operating system. After a CMDX call each GET_CMD_LINE call causes the next line to be read from the buffer, i.e
- the first GET_CMD_LINE call delivers the first line
- the second GET_CMD_LINE call delivers the second line
and so on.

If there are no further lines, GET_CMD_LINE returns an empty string and resets its pointer to the start of the buffer so that the first line can be read again with the next GET_CMD_LINE. If a line delivered from the system macro cmd is longer than Maxlength (line), then "line" contains the first part of the line and the variable "more" is set to TRUE.
The following GET_CMD_LINE call delivers the next part of the line and "more" is assigned the value FALSE, if "line" contains the remainder of the line.

buffer	is the buffer specified in the preceding CMDX call.	
line	after the GET_CMD_LINE call it contains the next line from the buffer (as a Pascal string).	
more	= TRUE	the line read cannot be fully stored in "line". The remainder can be read by issuing further GET_CMD_LINE calls.
	= FALSE	the complete line is contained in "line".

A.5 DMSIO

The DMSIO package provides additional functions for opening and closing files and a series of functions for accessing ISAM files (index sequential files).

When a file is opened for reading, the Pascal system takes the current file attributes from the catalog entry (when the file does not exist an OPEN_ERROR is generated). If a FILE command with specification of the link name and file attributes was issued before it was opened, an OPEN_ERROR is generated if these attributes do not match those in the catalog (see also section 5.3).

When a file is opened for writing, the Pascal system sets up a SAM file with variable record length as a standard procedure. If the current file is to be a SAM file with fixed record length or an ISAM file, a FILE command must be issued using the desired file attributes before opening the file. The FILE command must also include the name of the Pascal file as the link name so that the file attributes can be transferred to the catalog entry (see section 5.3). Attributes marked with "(*)" must definitely be specified in the FILE command for an ISAM file, while the default values are used for the other attributes (see [7]). For reasons of reliability and documentation, the attribute RECFORM should always be specified because the value of KEYPOS is dependent on the record format.

FCBTYPE (*) ISAM must be specified as the access method.

KEYPOS (*) Specifies the position of the first character of the record key within the record. In a Pascal file, the record key corresponds to a record field within the basic type of the file type. The relative offset n of the record field to the beginning of record in the case of fixed record length (RECFORM=F) corresponds to the position n+1; in the case of variable record length (RECFORM=V) it corresponds to the position n+5, because in this case the 4-byte long record length field must be taken into account.

The relative offset of the record field containing the key should be defined using the predefined function OFFSETOF (see also examples).

KEYLEN (*) Specifies the length of the record key in bytes. The length of the record field defining the record key can be determined using the standard function SIZEOF.

RECFORM Defines whether fixed or variable length records are to be processed. Variable length records are automatically given a 4-byte record length field.

RECSIZE Specifies the record length (including length field) in bytes.

BLKSIZE	Defines the buffer length for I/O operations.
DUPEKY	Defines whether duplicate keys are permissible. This attribute is not stored in the catalog and is valid only for as long as the associated FILE command is valid.
SHARUPD	Defines whether a file can be updated by more than one task at the same time if the file was opened with Replace. Further details on access synchronization are contained in [7,8]. This attribute is not stored in the catalog and is valid only for as long as the associated FILE command is valid.

```
(*****************************************************************
*                                                                 *
*       COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG 1990  *
*                      ALL RIGHTS RESERVED                        *
*                                                                 *
*****************************************************************)
 {$DEBUG=ON}
 package DMSIO;
                  (************************************)
                  (*  The body of this package is part   *)
                  (*  of the Pascal-XT runtime system    *)
                  (************************************)

 procedure extend     (var f: any_file);
    (* same as rewrite but the file's content is not lost *)

 procedure replace    (var f: any_file);
    (* opens the file for reading and writing *)

 procedure close      (var f: any_file);
    (* closes the file *)

 procedure elim       (var f: any_file);
    (* deletes the last read record in the file *)

 procedure elimkey    (var f: any_file);
    (* deletes the record with the given key *)

 procedure getkey     (var f: any_file);
    (* reads the record with the specified key *)

 procedure getback    (var f: any_file);
    (* reads backwards the next record *)

 procedure update     (var f: any_file);
    (* writes a previously read record back into the file *)

 procedure pos        (var f: any_file);
    (* positions to the beginning of the file *)

 procedure poskey     (var f: any_file);
    (* positions to the record with the specified key *)

 procedure posback    (var f: any_file);
    (* positions to the file's end *)

 procedure movekey    (var f: any_file; var stv: string);
    (* returns the key of the last read record *)

 function  bof        (var f: any_file): boolean;
    (* returns begin of file *)

 function  nokey      (var f: any_file): boolean;
    (* returns false if the specified record is in the file *)

 function  lock       (var f: any_file): boolean;
    (* is true if the specified record is locked *)

 function  reclen     (var f: any_file): integer;
    (* returns the length of the last read record *)

 function  keypos     (var f: any_file): integer;
    (* returns the file key's position *)

 function  keylen     (var f: any_file): integer;
    (* returns the file's keylength *)

 end (* package DMSIO *).
```

Opening modes for files

CLOSE (f)

At program end all files opened by Pascal are automatically closed. The user can also close a file independently using Close(f). In the case of a text file, an implicit Writeln(f) is executed for Close(f) if the last line has not yet been completed using Writeln(f). In the case of non-text files, the contents of a file buffer which has not been output are lost. If Close is used on the predefined Input or Output file, then an Open_Error (system error code 1607) will occur.

EXTEND (f)

opens the file for writing. The previously existing contents of the file are not lost. Buffer f↑ is undefined. Put permits further records to be stored at the end of the file. Extend can only be used for SAM and ISAM files and the standard files *SYSOUT, *SYSLST and *DUMMY. Calling Extend while specifying Input or Output leads to an Open_Error (system error code 1607).

REPLACE (f)

opens the file for reading and writing so that records can be updated. If the file does not exist, a new one is created. If the file does exist, the first record is then read into the file buffer and in the case of ISAM files is locked by specifying SHAREUPD=YES. The lock can be lifted by calling Pos(f). Replace can only be used for SAM and ISAM files. Calling Replace while specifying Input or Output leads to an Open_Error (system error code 1607).

General functions

BOF (f)

indicates whether the beginning of file f was reached. This function is symmetrical to Eof. If Bof becomes True after calling Getback or Eof True, then a further Getback is not permissible, whereas a Get is permissible.

POS (f)

positions to beginning of file so that the first record of the file is read using the next Get(f). After Pos(f), Bof(f) is True. Eof(f) is True for a dummy file, otherwise it is False.

POSBACK (f)

positions to end of file so that the last record of the file can be read using the next Getback(f). After Posback(f), Eof(f) is True. Bof(f) is True for a dummy file, otherwise it is False.

RECLEN (f)

provides the record length of the record last read or written as an integer result for file f. When the file status is undefined the value -1 is returned.

UPDATE (f)

rewrites a record processed with Get(f), Getkey(f) or Getback(f) to the file. In this case the record length and, for ISAM files the key field, must not be changed if file f has been opened with Replace. The record lock is lifted. Update can only be used if file f has been opened with Replace.

Processing of ISAM files

ELIM (f)

deletes from an ISAM file the record last read with Get(f) and leaves f↑ undefined. If this was the last record of the file, then Eof(f) is True; if it was the first one, Bof(f) remains True.

ELIMKEY (f)

deletes from an ISAM file the record whose key is in the key field. If this record is locked it is not deleted, instead Lock(f) is True. If such a record does not exist, Nokey(f) is True after the call. If there is more than one record with the same key, the first one is deleted. After Elimkey(f), Eof(f) and Bof(f) are False.

GETBACK (f)

reads the next record from the file in the direction beginning of file. If the previously read record was the first one in the file, f↑ is undefined and Bof(f) is True. If the record is locked, i.e Lock(f) is True, then f↑ is undefined and a new Getback attempts to read the locked record again.

GETKEY (f)

reads the record whose key is specified in the key field. If this record is locked, f↑ is undefined and Lock(f) is True. If such a record does not exist, Nokey(f) is True after the call. When more than one record has the same key, the first one is read and the others can be read with Get. If the file has been opened with Replace, the record is locked after a successful Getkey(f), but not after Reset. Getkey(f) sets (Eof)f and Bof(f) to False.

KEYLEN (f)

specifies the length of the record key in bytes for an ISAM file.

KEYPOS (f)

specifies the position of the first character of the record key for an ISAM file.

LOCK (f)

is True and f↑ is undefined if the required record is locked at the time. When the access function is repeated the locked record is accessed again. In the case of ISAM files with equal keys (DUPEKY=YES) it is not possible to determine whether the same record is being accessed again or not (this is determined by the processing in the Retry macro).

MOVEKEY (f,stv)

moves the last record key read of file f to the string variable stv in the case of read access to ISAM text files. In the case of files with sequential access or undefined file status, an empty string is returned.

NOKEY (f)

is True if the required record in file f does not exist.

Note

After Poskey(f), Nokey is also False if the record does not exist. If the file has been opened with Replace, then an interrogation must be made as to whether the record is locked (Lock call) before calling Nokey.

POSKEY (f)

positions to the record whose key is in the key field so that it can be read with the next Get(f). If such a record does not exist, positioning is to the place where such a record should exist. With the next Get(f), the first record with the next highest key is read. After Poskey(f), Eof(f), Bof(f), Lock(f) and Nokey(f) are False.

Extensions to the standard procedures

In addition to the above-mentioned procedures and functions, standard procedures and functions defined in Pascal-XT can be used on ISAM files.
The procedures Put, Get and the function Eof are extended for ISAM files in the following manner:

GET (f)

reads the next record from the file in the direction end of file. If this record is locked, Lock(f) is True and f↑ is undefined. If the file has been opened with Replace, a record is locked when Get is successful, but not in the case of Reset. If there are no more records, f↑ is undefined and Eof(f) is True.

PUT (f)

appends a record to the end of the file in the case of a SAM file. For an ISAM file opened with Rewrite or Extend, a record is likewise appended to the end of the file. However the record key must be larger than the keys of all previously written records. If the file has been opened with Replace, the record is stored in accordance with its key. If a record with the same key already exists, it is overwritten when DUPEKY=NO has been specified, otherwise it is appended to the list of records with the same key. If the record to which writing is to take place is locked, no action is taken and Lock(f) is True.

EOF (f)

shows whether the end of file f has been reached. If Eof is True after calling Posback or Get, any further Get is not permitted, whereas Getback is permitted.

The table below shows all action macros with respect to their permissibility as a function of the Open mode and type of file.

Open mode →	RESET		REWRITE		EXTEND		REPLACE	
File type →	SAM	ISAM	SAM	ISAM	SAM	ISAM	SAM	ISAM
Action macro ↓								
ELIM								x
ELIMKEY								x
GET	x	x					x	x
GETKEY		x						x
GETBACK		x						x
PUT			x	(1)	x	(1)		x
UPDATE							x	x
POS	x	x					x	x
POSKEY		x						x
POSBACK	x	x					x	x

x The action macro is permitted

(1) Permitted at the end of the file, i.e. only with increasing keys

The table below outlines the influence of the results of functions Bof, Eof, Nokey and Lock as well as the buffer contents of f↑ with respect to the procedures.

	BOF	EOF	NOKEY	LOCK	Buffer contents
RESET	(1)	(1)	FALSE	FALSE	(A)
REWRITE	-	TRUE	FALSE	FALSE	(U)
EXTEND	-	TRUE	FALSE	FALSE	(U)
REPLACE	(1)	(1)	FALSE	(2)	(A,B)
CLOSE	-	-	FALSE	FALSE	(U)
ELIM	-	-	FALSE	(2)	(U)
ELIMKEY	FALSE	FALSE	(3)	(2)	(U)
GET	FALSE	(4)	FALSE	(2)	(A)
GETBACK	(5)	FALSE	FALSE	(2)	(A,B)
GETKEY	FALSE	FALSE	(3)	(2)	(B)
POS	TRUE	(1)	FALSE	FALSE	(U)
POSBACK	(1)	TRUE	FALSE	FALSE	(U)
POSKEY	FALSE	FALSE	FALSE	FALSE	(U)
PUT	FALSE	-	FALSE	(2)	(U)
UPDATE	-	-	FALSE	FALSE	(U)

– Unchanged, irrelevant.

(1) True if the file is empty, otherwise False.

(2) True if the record is locked, otherwise False.

(3) True if the record does not exist, otherwise False.

(4) True if an attempt was made to read beyond the end of file.

(5) True if an attempt was made to read before the beginning of file.

(A) Undefined if Eof is True, otherwise defined.

(B) Undefined if Lock or Nokey is True, otherwise defined.

(U) Undefined.

Example 1

The sample program generates an ISAM file, writes records to this file, and then reads a specific record. Assignment of the physical file to the Pascal file is made before the program executes by issuing the FILE command. The first time the program is called, fixed length records are processed, the second time variable length records.

```
/EXEC $USERID.PASCAL-XT
%  BLS0500 PROGRAM 'PASCALXT', VERSION '22A00' OF ...  LOADED<
%  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG.1990. ALL RIGHT
S RESERVED
//D DIRECTORY
//C ($USERID.PASSUP-XT, DMSIO.SPEC), *D
   >>> COMPILATION SUCCESSFUL (WARNINGS: 0;  NOTES: 0)
//C (PLAM.EXAMPLE, ISAM1.PROG), *SYSOUT, *OMF(Y)
*** SOURCE LISTING ***    BS2000 PASCAL-XT COMPILER  V2.2A00...

GLOBAL OPTIONS FOR THIS COMPILATION

DEBUG       =    OFF          BY DEFAULT
GENERATE    =    ON           BY DEFAULT
MAP         =    OFF          BY DEFAULT
STANDARD    =    OFF          BY DEFAULT
XREF        =    OFF          BY DEFAULT

LIST OF RECOMPILED PACKAGE SPECIFICATIONS (SOURCE FILES)

               ($USERID.PASSUP-XT,DMSIO.SPEC(*STD,S))


CURRENT COMPILATION UNIT (SOURCE FILE)

               ($USERID.PLAM.EXAMPLE,ISAM1.PROG(*STD,S))

     1      (*$TITLE = 'Generate an ISAM file'*)
     2
     3      with DMSIO;
     4
     5      program ISAM1 (output, isamfile);
     6
     7      const
     8         keypos   = 4;
     9
    10      type
    11         record type = record
    12                      c (0)      : char;
    13                      i (keypos) : long_integer;
    14                      end;
    15         file type = file of record type;
    16
    17      var
    18        isamfile : file type;
    19
    20      procedure generate_file;
    21      begin
    22         rewrite (isamfile);
    23         isamfile↑.i := 1;
```

```
    24        isamfile↑.c := 'A';
    25        put (isamfile);
    26        isamfile↑.i := 2;
    27        isamfile↑.c := 'B';
    28        put (isamfile);
    29     end (* generate_file *) ;
    30
    31     procedure print_record (key : long_integer);
    32     begin
    33        reset (isamfile);
    34        isamfile↑.i := key;
    35        DMSIO.getkey (isamfile);
    36        if DMSIO.nokey (isamfile) then
    37           writeln ('Record not found')
    38        else
    39           writeln (isamfile↑.i:1, isamfile↑.c:3);
    40     end (* print_record *) ;
    41
    42     begin
    43        generate_file;
    44        print_record (2);
    45     end (* ISAM1 *).
```

```
***********************************************************
*                    COMPILATION SUMMARY                  *
***********************************************************
*  ERRORS DETECTED          :        0                    *
*  WARNINGS                 :        0                    *
*  NOTES                    :        0                    *
*  SIZE OF CODE MODULE      :      968 BYTES              *
*  SIZE OF DATA MODULE      :      472 BYTES              *
*  COMPILATION TIME         :    0.210 SEC                *
***********************************************************
   >>> COMPILATION SUCCESSFUL (WARNINGS: 0;  NOTES: 0)
//SY FILE FILE-1,LINK=ISAMFILE,FCBTYPE=ISAM,KEYLEN=4,KEYPOS=5,RECFORM=F   (01)
//RUN                                                                     (02)
2  B                                                                      (03)
//SY FSTAT FILE-1,ALL                                                     (04)
0000003 :V:$USERID.FILE-1
  FCBTYPE = ISAM              VSNTYPE = PUB
  LASTPG  = 0000002           2ND ALLO= 00003
  SHARE   = NO                ACCESS  = WRITE
  ACL     = NO                AUDIT   = NONE             DESTROY = NO
  CRDATE  = 1990-11-27        EXDATE  = 1990-11-27       LADATE  = 1990-11-27
  RDPASS  = NONE              WRPASS  = NONE             EXPASS  = NONE
  ACCESS# = 002               VERSION = 001
  LARGE   = NO                BACKUP  = A                MIGRATE = ALLOWED
  BLKTYPE = STD               BLKSIZE = 002048           BLKCTRL = PAMKEY
  RECFORM = (F,N)             RECSIZE = 000008
  KEYLEN  = 004               KEYPOS  = 00005
  VSN/DEV/EXT  =    PUBV02 / D3480    / 001
  EXTCNT  =   1
:V:    PUBLIC:     1  FILE  RES=        3  FREE=        1  REL=        0 PAGES
//SY FILE FILE-2,LINK=ISAMFILE,FCBTYPE=ISAM,KEYLEN=4,KEYPOS=9,RECFORM=V   (05)
//RUN
2  B
```

```
//SY FSTAT FILE-2,ALL  ————————————————————————————————————————————  (06)
0000003 :V:$USERID.FILE-2
  FCBTYPE = ISAM              VSNTYPE = PUB
  LASTPG  = 0000002           2ND ALLO= 00003
  SHARE   = NO                ACCESS  = WRITE
  ACL     = NO                AUDIT   = NONE            DESTROY = NO
  CRDATE  = 1990-11-27        EXDATE  = 1990-11-27      LADATE  = 1990-11-27
  RDPASS  = NONE              WRPASS  = NONE            EXPASS  = NONE
  ACCESS# = 002               VERSION = 001
  LARGE   = NO                BACKUP  = A               MIGRATE = ALLOWED
  BLKTYPE = STD               BLKSIZE = 002048          BLKCTRL = PAMKEY
  RECFORM = (V,N)             RECSIZE = 000012
  KEYLEN  = 004               KEYPOS  = 00009
  VSN/DEV/EXT  =    PUBV04 / D3480    / 001
  EXTCNT  =   1
:V:    PUBLIC:     1  FILE  RES=        3  FREE=        1  REL=        0 PAGES
//
```

(01) Before execution of the program, FILE command is used to create a file specifying the link name and the file attributes. In this instance fixed length records are processed. As a consequence, KEYPOS must have the value 5.

(02) Execution of the program. The name of the Pascal file is assumed as the link name.

(03) Message issued by the program.

(04) Output of information from the file catalog for FILE-1.

(05) The FILE command is used to define a new file. KEYPOS is given the value 9 because this time variable length records are processed.

(06) Output of information from the file catalog for FILE-2.

Example 2

The program writes records to a file. These records have the same key in pairs. Then all records of a certain key are output. The name of the physical file is read by the program. From within the program the FILE command is called to set the parameters of the ISAM file. Assignment of the physical file to the Pascal file is accomplished by means of the ASSIGNFILE macro.

```
/EXEC $USERID.PASCAL-XT
%  BLS0500 PROGRAM 'PASCALXT', VERSION '22A00' OF ...  LOADED<
%  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG.1990. ALL RIGHT
S RESERVED
//D DIRECTORY
//C ($USERID.PASSUP-XT, DMSIO.SPEC), *D
   >>> COMPILATION SUCCESSFUL (WARNINGS: 0;  NOTES: 0)
//C ($USERID.PASSUP-XT, BS2000CALLS.SPEC), *D
   >>> COMPILATION SUCCESSFUL (WARNINGS: 0;  NOTES: 0)
//C (PLAM.EXAMPLE, ISAM2.PROG), *SYSOUT, *OMF(Y)
*** SOURCE LISTING ***    BS2000 PASCAL-XT COMPILER  V2.2A00...

GLOBAL OPTIONS FOR THIS COMPILATION

DEBUG      =    OFF          BY DEFAULT
GENERATE   =    ON           BY DEFAULT
MAP        =    OFF          BY DEFAULT
STANDARD   =    OFF          BY DEFAULT
XREF       =    OFF          BY DEFAULT

LIST OF RECOMPILED PACKAGE SPECIFICATIONS (SOURCE FILES)

               ($USERID.PASSUP-XT,BS2000CALLS.SPEC(*STD,S))

               ($USERID.PASSUP-XT,DMSIO.SPEC(*STD,S))


CURRENT COMPILATION UNIT (SOURCE FILE)

               ($USERID.PLAM.EXAMPLE,ISAM2.PROG(*STD,S))

     1      with BS2000CALLS, DMSIO;
     2
     3      program ISAM2 (input, output, isamfile);
     4
     5      type
     6         key type     = long_integer;
     7         record type  = record
     8                      c   : char;
     9                      key : key type;
    10                      end;
    11         file type    = file of record type;
    12
    13      var
    14        isamfile  : file type;
    15        attr      : string;
    16        file name : string;
    17        error     : boolean;
    18
```

```
19      procedure init;
20      begin
21         (* Format of the FILE command without specification of a file *)
22         writestring (attr, '/FILE ',
23                            ',LINK=ISAMFILE',
24                            ',FCBTYPE=ISAM',
25                            ',KEYPOS=', (offsetof (record type, key) + 5):1,
26                            ',KEYLEN=', sizeof (key type):1,
27                            ',RECFORM=V',
28                            ',DUPEKY=YES');
29         writeln ('Please enter file name:');
30         readln; read (file name);
31         BS2000CALLS.cmd (attr, error);
32         if error then raise (1);
33         assignfile (isamfile, file name);
34      end (* init *) ;
35
36      procedure generate_file;
37      begin
38         rewrite (isamfile);
39         isamfile↑.key := 1; isamfile↑.c := 'A'; put (isamfile);
40         isamfile↑.key := 1; isamfile↑.c := 'a'; put (isamfile);
41         isamfile↑.key := 2; isamfile↑.c := 'B'; put (isamfile);
42         isamfile↑.key := 2; isamfile↑.c := 'b'; put (isamfile);
43         isamfile↑.key := 3; isamfile↑.c := 'C'; put (isamfile);
44         isamfile↑.key := 3; isamfile↑.c := 'c'; put (isamfile);
45      end (* generate_file *) ;
46
47      procedure print_record (value : key type);
48      begin
49         reset (isamfile);
50         isamfile↑.key := value;
51         DMSIO.getkey (isamfile);
52         if DMSIO.nokey (isamfile) then
53            writeln ('Record not found')
54         else
55            repeat
56               writeln (isamfile↑.key:1, isamfile↑.c:3);
57               get (isamfile);
58            until (isamfile↑.key <> value) or eof (isamfile);
59      end (* print_record *) ;
60
61      begin
62         init;
63         generate_file;
64         print_record (2);
65      end (* ISAM2 *).
```

```
***********************************************************
*                   COMPILATION SUMMARY                   *
***********************************************************
*  ERRORS DETECTED         :        0                     *
*  WARNINGS                :        0                     *
*  NOTES                   :        0                     *
*  SIZE OF CODE MODULE     :     2368 BYTES               *
*  SIZE OF DATA MODULE     :      988 BYTES               *
*  COMPILATION TIME        :    0.326 SEC                 *
***********************************************************
   >>> COMPILATION SUCCESSFUL (WARNINGS: 0;  NOTES: 0)
//RUN
Please enter file name:
TESTFILE
2  B
2  b
//SY FSTAT TESTFILE,ALL
0000033 :V:$USERID.TESTFILE
  FCBTYPE = ISAM              VSNTYPE = PUB
  LASTPG  = 0000002           2ND ALLO= 00032
  SHARE   = NO                ACCESS  = WRITE
  ACL     = NO                AUDIT   = NONE              DESTROY = NO
  CRDATE  = 1990-11-27        EXDATE  = 1990-11-27        LADATE  = 1990-11-27
  RDPASS  = NONE              WRPASS  = NONE              EXPASS  = NONE
  ACCESS# = 002               VERSION = 001
  LARGE   = NO                BACKUP  = A                 MIGRATE = ALLOWED
  BLKTYPE = STD               BLKSIZE = 002048            BLKCTRL = PAMKEY
  RECFORM = (V,N)             RECSIZE = 000012
  KEYLEN  = 004               KEYPOS  = 00009
  VSN/DEV/EXT  =    PUBV03 / D3480    / 001
  EXTCNT  =   1
:V:    PUBLIC:     1  FILE  RES=       33  FREE=       31  REL=       30 PAGES
//
```

A.6 EDTADAPTER

The EDTADAPTER package permits a program to call the EDT editor or to transfer data and EDT commands to the EDT editor.

```
(*********************************************************************
*                                                                    *
*       COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG 1990    *
*                    ALL RIGHTS RESERVED                             *
*                                                                    *
*********************************************************************)
{$DEBUG=ON}
package EDTADAPTER;

          (***********************************************)
          (*  The body of this package is part of the     *)
          (*  Pascal-XT runtime system.                   *)
          (*                                              *)
          (*  Required EDT version:  >= 16.1              *)
          (*  Required Pascal-XT Runtime System: >= 2.1A  *)
          (*  Available in V2.1A and higher:              *)
          (*     EDT_VERSION                              *)
          (*     EDT_INTERFACE_VERSION                    *)
          (*     SET_EDT_INTERFACE_VERSION                *)
          (***********************************************)


procedure call_edt;
   (* calls the editor EDT for interactive use *)

procedure submit_edt (command: string);
   (* sends a string or an edt-command to the editor edt *)


const
                                (* Constant values defined by EDT *)
   command_string_len = 296;    (* must be equal to EUPCMDM *)
   message_string_len =  80;    (* must be equal to EUPMSGM *)
   workfile_name_len  =   8;    (* must be equal to L'EGLFILE *)
   version_number_len =  12;    (* must be equal to EGLVERSL *)

type
   command_string     = string [ command_string_len ];
   message_string     = string [ message_string_len ];
   workfile_name      = string [ workfile_name_len ];
   version_number     = string [ version_number_len ];


procedure edt_command (      command  : command_string;
                             message  : message_string;
                         var result   : message_string;
                         var workfile : workfile_name );
   (* this procedure executes F-mode commands as specified in      *)
   (* operand 'command'; the string passed in operand 'message'    *)
   (* is displayed in line 23 of the EDT; in case of an error the  *)
   (* EDT's error message is returned in 'result'; the name of the *)
   (* current active workfile is returned in operand 'workfile'.   *)

function edt_version : version_number;
   (* returns the EDT version number *)
   (* format: 'EDT V16.xAxx'         *)

function edt_interface_version : integer;
   (* returns the EDT subprogram interface version *)
   (* which is currently supported, e.g. 1 or 2    *)
```

```
procedure set_edt_interface_version (    vers   : integer;
                                     var errors : boolean  );
   (* this procedure sets the EDT subprogram         *)
   (* interface version                              *)
   (* 'vers'  : specifies the interface version as   *)
   (*           described in [1] (subprogram control *)
   (*           block EDTUPCB, field EUPVERS)        *)
   (* 'error' : result of the operation              *)
   (*           = true  : an error occurred          *)
   (*           = false : no error                   *)

end (* package EDTADAPTER *).
```

CALL_EDT

deactivates execution of the Pascal program and calls EDT. As a result of the EDT command HALT or @RETURN or pressing the K1 key, EDT is exited from and execution of the Pascal program continues. The contents of the EDT working areas are retained.

SUBMIT_EDT (line)

passes the string designated by "line" to EDT. The version of the EDT subprogram interface being used defines how the string is to be interpreted (as data or as an EDT command). After EDT has processed the string, control is returned to the calling subprogram.

EDT_COMMAND (command, message, result, workfile)

This procedure services the new subprogram interface of EDT starting with version 16.1 (see [13]). This interface permits passing of multiple EDT commands and enables EDT to be called. In addition, a text can also be passed on, one which EDT outputs in the message line. If an error occurs, the error message text of EDT is returned. When returning from EDT, the name of the EDT working area exited from is given. Both uppercase and lowercase letters can be specified in "command" and "message".

command — Contains a string of F mode commands (separated by semicolons) which are processed in order of precedence. Permissible commands are described in the EDT manual. Switchover of the EDT working area is effected by: '$0' ,..., '$9', invocation of EDT by: 'DIALOG'

message — The specified string is output in line 23 of EDT. When the empty string ('') is specified, EDT produces standard output.

result — If errors occur when "command" is processed, then "result" contains either the error text from EDT - or, if this is not present, the main return code from EDT in the form
" PASCAL-XT : EDT MAIN-RETURNCODE xxxx", where xxxx represents the hexadecimal value of the main return code. The empty string is returned in the event of error-free execution.

workfile — Always supplies the current working area of EDT (in der Form '$0 ', '$1 ',..., '$9 ').

The subprograms call_edt and submit_edt likewise use the new EDT interface and have the following effect:

call_edt: edt_command('DIALOG', '', dummy, dummy)

submit_edt(cmd): edt_command(cmd, '', dummy, dummy)

where "dummy" stands for the string variable.

EDT_VERSION

This function returns the version number of the EDT used in the system in a 12-character string in the form 'EDT V16.xAxx'. 'x' is a more detailed identification of the version and is specific to EDT.

EDT_INTERFACE_VERSION

This function returns the interface version of the EDT subprogram interface in the form of an integer variable. Possible values for the interface version and its significance can be found in the reference manual for the EDT subprogram interface [13] (currently a value of 1 designates the statement set supported by EDT version 16.1, while a value of 2 designates the statement set supported by EDT version 16.2).

SET_EDT_INTERFACE_VERSION (vers, errors)

On initialization of the interface version the Pascal-XT runtime system takes responsibility for presetting the values in the IEDTUPCB macro (see [13]). The interface version can be changed by using the procedure SET_EDT_INTERFACE_VERSION. The setting of the values is only possible before EDT is initialized, i.e. apart from EDT_INTERFACE_VERSION, EDT_VERSION and this procedure itself, no other procedure in the EDTADAPTER package may be called before this is done. All other calls result in EDT being initialized. Similarly, once an EDT procedure area has been used as a Pascal file or EDT has been called in the programming environment or in the debugging aid PATH, it is no longer possible to change the interface version.

vers: specifies the required interface version; it must be defined as an integer (1,2,...). The validity of the parameter is not checked by the procedure.

errors: contains the result of the operation that has been executed.
- = FALSE: if execution is error-free
- = TRUE: if an inconsistency is detected in the EDT interface version (e.g. EDT V16.1 is called with interface version 2), or, if a command to EDT has already been issued and changing the interface is no longer possible.

Notes

- In the predefined procedure ASSIGNFILE the various working areas may be addressed. In addition to *EDT (working area currently set) the number of the working area can be specified in parentheses, e.g. (*EDT(1) ,..., *EDT(9))(see also section 5.2.1).
- If EDT is called from within a Pascal-XT program, then no Pascal-XT program can be executed by means of the EDT command @RUN. The reason for this is that both programs use the same runtime system, which will generate an error when the program started with @RUN terminates. Programs other than Pascal-XT will execute smoothly.
- The programming system and the debugging aid also issue commands to EDT via the subprogram interface. A program started with //RUN cannot change the interface version if EDT has already been called in the programming system or in the debugging aid.

A.7 ERRORS

The ERRORS package provides additional information on the more precise classification of the error when an exception occurs. The meanings of the system error codes is described in section 10.4.

```
(***********************************************************************
*                                                                      *
*       COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG 1990      *
*                     ALL RIGHTS RESERVED                              *
*                                                                      *
***********************************************************************)
{$DEBUG=ON}
package ERRORS;

            (**********************************************)
            (*  The body of this package is part of the     *)
            (*  PASCAL-XT runtime system.                   *)
            (*                                              *)
            (*  Required Pascal-XT Runtime System: >= 2.1A  *)
            (*  Available in V2.1A and higher:              *)
            (*     INTR_MESSAGE                             *)
            (*     PRINT_ERROR_INFO                         *)
            (*     RERAISE                                  *)
            (**********************************************)


function system_code: integer;
   (* returns the system error code as additional information about *)
   (* the last error. 0 is returned if there is no system error     *)
   (* code.                                                         *)

function interrupt_address: integer;
   (* returns the interrupt address of the last exception. 0 is     *)
   (* returned if there is no error.                                *)

function file_info: string;
   (* file_info returns file information if an exception was raised *)
   (* by a file error. For other errors the empty string is         *)
   (* returned.                                                     *)

function intr_message: string;
   (* returns the most recent /INTR message if an exception was     *)
   (* raised by /INTR; if /INTR was sent without message or no      *)
   (* /INTR at all, the empty string is returned.                   *)

procedure print_error_info;
   (* Lists the dynamic stack chain as it existed at the instant    *)
   (* of the most recent exception. The format is the same as       *)
   (* in the case of an unhandled exception. If no exception has    *)
   (* occurred yet, nothing is listed.                              *)

inline procedure reraise;
begin raise(0) end;
   (* Resumes the raise of the most recently raised exception.      *)
   (* Raises system_error with system_code 5001, if no exception    *)
   (* has occurred yet.                                             *)

end (* package ERRORS *).
```

SYSTEM_CODE

returns the system error code of the last error which occurred (see 10.4). The value of the function is undefined in the case of exceptions which are not predefined or in the event of error-free execution.

INTERRUPT_ADDRESS

returns the interrupt address of the last interrupt or of the last call of standard procedure RAISE. The value of the function is undefined when execution is without error.

FILE_INFO

In the event of a file access error the name of the Pascal file, the external description specified in the standard procedure ASSIGNFILE, and the name of the current (not the temporary) BS2000 file are returned. The empty string is returned for the other errors.

PRINT_ERROR_INFO

This procedure outputs to SYSOUT and SYSLST the dynamic call chain as it was when the last exception occurred without the exception having to be triggered again or propagated. The format of the output is the same as that in the case of an unhandled exception. PRINT_ERROR_INFO can be called at any time either from within or without an exception handler. If no exception has occurred then nothing is output.

INTR_MESSAGE

This function returns the text of the message issued at the last [K2]/INTR interrupt. The empty string is returned if no text was specified in the last /INTR command or if no /INTR command has yet been issued in the program run.

RERAISE

This (inline) procedure continues the propagation of the most recent exception already processed by an exception handler. If there is no other exception handler present, the dynamic call chain as it was when the last exception occurred is output to SYSOUT and SYSLST.

RERAISE can be called from within or without an exception handler. If no exception has yet occurred, a call leads to a runtime error (SYSTEM_ERROR, System_Code=5001).

A.8 HEAPSUPPORT

The HEAPSUPPORT package permits the user to employ several heaps. Dynamically generated objects are always allocated in the current heap. Objects in one heap can also point to objects in other heaps. If this package is not used or if no heap is defined, objects are then allocated in the default heap set up by the runtime system as a standard procedure.

RELEASE(p) is used to release the heap level previously set with the relevant MARK(p).

The concept of heaps permits logically associated objects to be combined, or objects with approximately the same lifespan, all in one heap. Release of these objects is quick--simply by triggering this heap release function.

```
(***********************************************************************
*                                                                      *
*       COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG 1990      *
*                     ALL RIGHTS RESERVED                              *
*                                                                      *
***********************************************************************)
 {$DEBUG=ON}
 package HEAPSUPPORT;

             (*************************************)
             (*  The body of this package is part   *)
             (*  of the Pascal-XT runtime system    *)
             (*************************************)

 type

    heap  = heap_descriptor;   (* private *)

 procedure create_heap  ( var h : heap );
    (* Creates a new heap and returns the pointer of the heap descriptor *)
    (* for subsequent use by select_heap and release_heap.               *)
    (* Raises memory_error with system_code 1403 if creation of a new    *)
    (* heap is not possible.                                             *)
    (* Note : create_heap does not select the new heap.                  *)

 procedure select_heap  ( h : heap );
    (* Selects the heap h as the current heap. All subsequent calls to   *)
    (* the standard procedure new will allocate memory on this heap.     *)
    (* select_heap(default_heap) is allowed, select_heap(current_heap)   *)
    (* has no effect.                                                    *)
    (* select_heap on an already released heap raises system _rror with  *)
    (* system_code 1404.                                                 *)

 function  current_heap  : heap;
    (* Returns the pointer of the heap descriptor of the current heap.   *)
    (* Note : this may be the default heap.                              *)

 function  default_heap  : heap;
    (* Returns the pointer of the heap descriptor of the default heap.   *)
    (* Note : The Initialization routine of the Runtime-System creates   *)
    (*        and selects a heap before transferring control to the      *)
    (*        user program. This heap is called the "default heap".      *)

 procedure release_heap  ( h : heap );
    (* All dynamic variables allocated in the heap h will be "disposed"  *)
    (* and their memory together with the heap descriptor returned to    *)
    (* the operating system.                                             *)
    (* release_heap(current_heap) raises system_error, system_code 1405  *)
    (* release_heap(default_heap) raises system_error, system_code 1406  *)

 end (* package HEAPSUPPORT *).
```

CREATE_HEAP (h)

creates a descriptor for a new heap and returns a pointer referring to this heap, the pointer being used for the procedures select_heap and release_heap. An explicit switch to a newly created heap must be made (by means of the select_heap procedure) before it can be used.

DEFAULT_HEAP

returns a pointer referencing the heap selected by the runtime system as a standard procedure. Release of this default_heap is not possible and raises a runtime error (SYSTEM_ERROR, system_code = 1408).

CURRENT_HEAP

returns a pointer referencing the heap which was last selected by the select_heap procedure. In the event select_heap has not yet been used, the current_heap = default_heap. Release of the curren_ heap is not possible and raises a runtime error (SYSTEM_ERROR, system_code = 1407).

SELECT_HEAP (h)

A switch is made to the heap specified by h. This heap must have been generated by means of create_heap if this is not the default_heap. Subsequent calling of the standard procedure NEW generates objects in this heap.
Calling select_heap (current_heap) has no effect, whereas calling select_heap (default_heap) switches over to the default_heap, provided that current_heap = default_heap does not already apply.

RELEASE_HEAP (h)

The heap specified by h is released and the reserved main memory is returned to the operating system. Release of the current heap or of the default_heap raises a runtime error.

Example

The HEAPSTACK package simulates the functions pushheap, popheap and killheap provided by Pascal (BS2000) version 3.

```
*** SOURCE LISTING ***    BS2000 SIEMENS PASCAL-XT COMPILER  V2.0A00...

GLOBAL OPTIONS FOR THIS COMPILATION

DEBUG       =    OFF          BY DEFAULT
GENERATE    =    ON           BY DEFAULT
MAP         =    OFF          BY DEFAULT
STANDARD    =    OFF          BY DEFAULT
XREF        =    OFF          BY DEFAULT

CURRENT COMPILATION UNIT (SOURCE FILE)

               ($USERID.PLAM.MANUAL,HEAPSTACK.SPEC(*STD,S))

     1       (*$TITLE = 'Simulation of heap functions for Pascal version 3'*)
     2       package HEAPSTACK;
     3
     4       procedure pushheap;
     5       procedure popheap;
     6       procedure killheap;
     7
     8       end (* package HEAPSTACK *).

***********************************************************
*                   COMPILATION SUMMARY                   *
***********************************************************
*  ERRORS DETECTED         :        0                     *
*  WARNINGS                :        0                     *
*  NOTES                   :        0                     *
*  SIZE OF CODE MODULE     :        0 BYTES               *
*  SIZE OF DATA MODULE     :        0 BYTES               *
*  COMPILATION TIME        :    0.071 SEC                 *
***********************************************************

*** SOURCE LISTING ***    BS2000 SIEMENS PASCAL-XT COMPILER  V2.0A00...

GLOBAL OPTIONS FOR THIS COMPILATION

DEBUG       =    OFF          BY DEFAULT
GENERATE    =    ON           BY DEFAULT
MAP         =    OFF          BY DEFAULT
STANDARD    =    OFF          BY DEFAULT
XREF        =    OFF          BY DEFAULT

LIST OF RECOMPILED PACKAGE SPECIFICATIONS (SOURCE FILES)

               ($PASSUP-XT,HEAPSUPPORT.SPEC(*STD,S))

               ($USERID.PLAM.MANUAL,HEAPSTACK.SPEC(*STD,S))

CURRENT COMPILATION UNIT (SOURCE FILE)

               ($USERID.PLAM.MANUAL,HEAPSTACK.BODY(*STD,S))
```

```
 1      with HEAPSUPPORT;
 2      from HEAPSUPPORT use heap       , default_heap,
 3                           create_heap, select_heap,  release_heap;
 4
 5      package body HEAPSTACK;
 6
 7      type
 8         heapstack     = ↑heapctrlblock;
 9         heapctrlblock = record
10                            curr : heap;
11                            rest : heapstack;
12                            end;
13
14      var
15         hs       : heapstack;
16         lastheap : heap;
17
18      procedure pushheap;
19      var
20         hsnew   : heapstack;
21         newheap : heap;
22
23      begin
24         create_heap (newheap);
25         new (hsnew);
26         hsnew↑.curr := newheap;
27         hsnew↑.rest := hs;
28         hs          := hsnew;
29         lastheap    := nil;
30         select_heap (newheap);
31      end (* pusheap *) ;
32
33      procedure popheap;
34      begin
35         if hs↑.curr <> default_heap then begin
36            lastheap := hs↑.curr;
37            hs       := hs↑.rest;
38            select_heap (hs↑.curr);
39            end;
40      end (* popheap *) ;
41
42      procedure killheap;
43      begin
44         if lastheap <> nil then begin
45            release_heap (lastheap);
46            lastheap := nil;
47            end;
48      end (* killheap *) ;
49
50      begin
51         new (hs);
52         hs↑.curr := default hea_;
53         hs↑.rest := nil;
54         lastheap := nil;
55      end (* package HEAPSTACK *).
```

```
***********************************************************
*                   COMPILATION SUMMARY                   *
***********************************************************
*  ERRORS DETECTED         :        0                     *
*  WARNINGS                :        0                     *
*  NOTES                   :        0                     *
*  SIZE OF CODE MODULE     :      896 BYTES               *
*  SIZE OF DATA MODULE     :      112 BYTES               *
*  COMPILATION TIME        :    0.245 SEC                 *
***********************************************************
```

A.9 MEMORYMANAGER

The procedures provided in the MEMORYMANAGER package execute the functions provided by the BS2000 macros GTMAP, REQM and RELM (see [8]). Normally a Pascal program for memory management only uses the predefined procedures NEW, DISPOSE, MARK and RELEASE. To obtain information on the current storage space reservation, calling MEMORYMANAGER.GETMAP is recommended. Invocation of MEMORYMANAGER.REQMEM and MEMORY_MANAGER.RELMEM is reserved for special applications only. In particular, calling of RELMEM for memory pages is **not permissible** if the pages have not been previously obtained by means of an explicit MEMORYMANAGER.REQMEM request. Incorrect use generally results in a program crash.

Note that the runtime system automatically performs memory management and does not always return released pages to the operating system. This is particularly true if it is expected that released pages will soon be needed again.

```
(************************************************************************
*                                                                       *
*       COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG 1990       *
*                    ALL RIGHTS RESERVED                                *
*                                                                       *
************************************************************************)
 {$DEBUG=ON}
 package MEMORYMANAGER;

             (*************************************)
             (*  The body of this package is part   *)
             (*  of the Pascal-XT runtime system    *)
             (*************************************)

 const

      max_page             = 2047;
      max_pages_per_reqmem = 1792; (* BS2000 V7.5 and higher *)
      any_page             = 0;    (* code for non-specific page request *)

 type

      page_range           = 0 .. max_page;
      reqmem_page_range    = 1 .. max_pages_per_reqmem;

      page_map             = set of 0..max_page;

 procedure getmap  ( var map  : page_map  );
    (* returns a map of the used pages of the user's memory *)

 procedure reqmem  (     number_of_pages : reqmem_page_range;
                         page_number     : page_range; (* or "any_page" *)
                     var first_page      : pointer );
    (* request a contiguous memory area of pages *)

 procedure relmem  ( number_of_pages : page_range;
                     first_page      : pointer);
    (* releases the specified number of pages starting at "first_page" *)

 end (* package MEMORYMANAGER *).
```

GETMAP (map)

provides information on the use of pages in the first 16 Mb of user memory. A page i is free whenever "i in map" holds true (cf. [7]). This procedure provides no information on the use of pages in the upper address space (> 16 Mb).

REQMEM (number_of_pages, page_number, first_page)

requests a contiguous memory area containing "number_of_pages", beginning with the page specified by page_number. If page_number = any_page, any arbitrary free page is used. If the request can be met by the BS2000 system, a pointer referencing the allocated memory area is returned in the "first_page" result parameter, otherwise the value NIL is returned.

RELMEM (number_of_pages, first_page)

releases the number of memory pages specified in number_of_pages, beginning with the page specified by first_page.

A.10 Compatibility problems between Pascal-XT V2.2 and V3.0

In the next Pascal-XT version, a new language will replace the current language set. This will cover the language set defined in ISO standard 10206 for Extended Pascal and some extensions. To keep migration problems to a minimum, a tool will be provided for converting Pascal-XT V2.2 source programs to V3.0 source programs. Certain language constructs, however, will pose problems in this respect. The Pascal-XT V2.2A compiler therefore already issues notes at points where these language constructs occur, to alert the user to such problems. The compiler does not issue notes for language constructs which the migration tool will convert to equivalent Pascal-XT V3.0 constructs without difficulty.

The notes issued by the compiler relate to the following language constructs:

Standard procedure Elaborate(p)
(see the Pascal-XT Language Reference Manual [1], section 15.12)

This procedure, which enables the user to influence the sequence in which packages are initialized, will no longer be supported in Pascal-XT V3.0. Therefore with cyclical package references, package blocks may no longer contain statements.

Domain type of pointers
(see the Pascal-XT Language Reference Manual [1], section 6.4)

In Pascal-XT V3.0, the domain type of a pointer type and the pointer type must be defined in the same declaration part.

The compiler cannot point precisely to this problem in every case.

Procedures and functions with the directive Forward
(see the Pascal-XT Language Reference Manual [1], section 8.6)

In Pascal-XT V3.0, if a procedure or function is declared with the directive "FORWARD", the header and block of the procedure or function must be declared in the same declaration part. This means that there must be no variable, type or constant declarations between them.

Qualified set constructors
(see the Pascal-XT Language Reference Manual [1], section 9.4)

In Pascal-XT, the ordinal values of the base type of an unqualified set constructor must be in the range 0..2047 (see 4.2., No.11). With qualified set constructors it is possible to construct sets outside this range. In Pascal-XT V3.0, the base type of a set constructor is set to ordinal values in the range 0..2047; sets can no longer be constructed outside this range.

Indexing and selection of aggregate members

(see the Pascal-XT Language Reference Manual [1], section 9.5)

In Pascal-XT V3.0, a single expression may no longer be used when specifying an aggregate from which a member is selected by indexing or selection. The aggregate can be defined as a constant or, in the case of non-static members, a variable, as required.

Smallest representable real number

(see 4.2, Nos.1 and 2)

The constant MINREAL stands for the smallest real number that can be represented on a given machine. In Pascal-XT V3.0, the constant corresponds to the smallest normalized real number, so it has a somewhat greater value.

References

[1] **Pascal-XT (SINIX, BS2000)**
Language Reference Manual

Target group
Pascal-XT users working under the BS2000 operating system.
Contents
Structure and elements of a Pascal program which complies with the standard, and extensions of Pascal-XT as compared with the standard.

[2] BS2000
Introductory Guide to the SDF Dialog Interface
User Guide

Target group
BS2000 users.
Contents
The various input options offered with SDF in system operation; operating instructions and examples relating to optional user guidance via menus.
Applications
General.

[3] **LMS** (BS2000)
ISP Format
Reference Manual

Target group
BS2000 users
Contents
Description of the LMS statements in ISP format for creating and managing PLAM libraries and the members these contain.
Frequent applications are illustrated by means of examples.

[4] BS2000
Utility Routines
User Guide

Target group
BS2000 users (non-privileged).
Contents
Utility routines for non-privileged BS2000 users.
Applications
BS2000 timesharing mode.

[5a] BS2000
System User, Part 1
System Reference Guide

Target group
Experienced BS2000 users
Contents
Overview of commands and macros in BS2000, of essential tables and registers, of code tables and systems standards.
Applications
BS2000 interactive/batch mode

[5b] BS2000
System User, Part 2
System Reference Guide

Target group
Experienced BS2000 users
Contents
Overview of
- Assembler instructions and statements.
- Statements for the software products and EDT, SORT, ARCHIVE, PERCON, LEASY, TSOSLNK, DCAT, PASSWORD, FDEXIM, FDRIVE, DPAGE, SODUMP, $PCCNTRL, TPCOMP2, PRSERVE

Applications
BS2000 interactive/batch mode

[6] BS2000
Executive Macros
User Guide

Target group
- BS2000 assembly language programmers (non-privileged)
- System administrators

Contents
- All Executive macros in alphabetical order with detailed explanations and examples; selected macros for DMS and TIAM
- Macro overview according to application areas
- Comprehensive training section dealing with eventing, serialization, inter-task communication, contingencies

Applications
BS2000 application programs

[7a] BS2000
User Commands (ISP Format)
User Guide

Target group
BS2000 users (non-privileged)

Contents
- All BS2000 system commands in alphabetical order with detailed explanations and examples
- The following products are dealt with:
  BS2000-GA, MSCF, JV, FT, TIAM

Applications
BS2000 interactive/batch mode, procedures

[7b] BS2000
User Commands (SDF Format)
User Guide

Target group
BS2000 users
Contents
This manual describes the commands of the SDF command interface which can be used by the non-privileged user in the basic configuration of BS2000 V10.0A. The following are also dealt with:

- SDF V2.0A
- SDF-P V1.0A
- SYSFILE V10.1
- RSO V2.1B
- SPOOL V2.5B
- FT V4.0A
- HSMS V1.1A
- JV V10.0A
- RFA V10.0A
- SECOS V1.0A

The descriptions are arranged alphabetically according to command names. They include the command format and a description of the operands.

[8] BS2000
Job Variables
User Guide

Target group
BS2000 users
Contents
- Applications for job variables in controlling and monitoring jobs and program runs
- Conditional job control
- All the necessary commands and macros
- Application examples

Applications
BS2000 timesharing mode

[9] **UTM** (TRANSDATA, BS2000)
Planning and Design
User Guide

Target group
- Dp managers
- Application planners
- Programmers

Contents
- Introduction to UTM; description of the program memory and interface concept, access to data and files, and the interoperation with databases
- Notes on the design, optimization and performance of UTM applications, as well as details of data protection and linked applications

Applications
BS2000 transaction processing

[10] **UTM** (TRANSDATA, BS2000)
Generating and Administering Applications
User Guide

Target group
- System administrators
- UTM administrators

Contents
- Creation, generation and operation of UTM applications
- Working with UTM messages and error codes

Applications
BS2000 transaction processing

[11] **UTM** (TRANSDATA)
Supplement for Pascal-XT
User Guide

Target group
Programmers of UTM Pascal-XT applications

Contents
- Translation of the KDCS program interface into the language Pascal-XT
- All the information required by programmers of UTM Pascal-XT applications

Applications
BS2000 transaction processing

[12] **UTM** (TRANSDATA)
Programming Applications
User's Guide

Target group
Programmers of UTM applications
Contents
- Language-independent description of the KDCS program interface
- Structure of UTM programs
- KDCS calls
- Testing UTM applications
- All the information required by programmers of UTM applications

Applications
BS2000 transaction processing

[13] BS2000
Assembler Instructions
Reference Manual

Target group
BS2000 assembly-language programmers
Contents
This manual describes in alphabetical order all (nonprivileged) assembler instructions of the CPUs supported by BS2000. For each instruction the following is described:
- its function
- its assembler format, i.e. how to write it in assembly language
- its machine format, i.e. how it is represented in the CPU
- its execution sequence in detail
- any condition codes values which it sets
- possible program interrupts when it is executed
- programming notes
- one or more examples

Applications
BS2000 assembly-language application programmers

[14a] **EDT** (BS2000)
Statements
User Guide

Target group
- EDT newcomers
- End users

Contents
- Processing of SAM and ISAM files and elements from program libraries
- Introduction to the basic principles of EDT and description of the operating modes
- Creation of EDT procedures
- Descriptions of all the EDT statements. Frequent applications are illustrated with the aid of numerous examples.

Applications
File editing

[14b] **EDT** (BS2000)
Subroutine Interfaces
User Guide

Target group
- Experienced EDT users
- Programmers

Contents
- Processing of SAM and ISAM files and elements from program liberaries
- Incorporation of EDT functions into self-written programs
- Description of the EDT subroutine interface
- Calling EDT as a subroutine
- Functions of the subroutine interface
- Structure and generation of the control blocks
- Creating external statement routines
- Calling user programs from EDT

Applications
Program engineering

[15] BS2000
DMS Introductory Guide and Command Interface
User Guide

Target group
Non-privileged BS2000 users
Contents
- Functions of DMS in BS2000
- Processing of disk and tape file
- Access methods UPAM, SAM, BTAM, EAM, ISAM
- DMS commands

[16] **ASSEMBH** (BS2000)
User Guide

Target group
Assembly language users under BS2000
Contents
- Calling and controlling ASSEMBH
- Assembling, linking, loading, and starting programs
- Input sources and output of ASSEMBH
- Runtime system, structured programming
- Language interfacing
- Assembler Diagnostic Program ASSDIAG
- Advanced Interactive Debugger AID
- ASSEMBH messages
- Machine instruction formats

The publication(s) marked with an * is/are not published by Siemens Nixdorf Informationssysteme AG or by Siemens AG.

Ordering manuals

The manuals listed above and the corresponding order numbers are to be found in the *List of Publications* issued by Siemens Nixdorf Informationssysteme AG, which also tells you how to order manuals. New publications are listed in the *Druckschriften-Neuerscheinungen (New Publications)*.

You can arrange to have both of these sent to you regularly by having your name placed on the appropriate mailing list. Your local office will help you.

Index

% character 199
%LOCAL 226
%PARAM 227
*LIBRARY 19
*OMF 99, 135
  in the ADD-TOOL statement 21
  in the COMPILE statement 28
  in the RUN statement 45
*UNCHANGED 20, 41, 42
:HEX 227
? 237

A
abbreviation rules 18
abnormal program termination 13
abort 240
action commands 197, 226
ADD-TOOL 11, 21
alias names for operands 18
alias names for statements 18
alias names for tools 11
alignment 96
ARRAY 214
assembler listing 108
ASSIGN command 197, 214, 230
Assignfile 124
  attribute description 127
  change line length 129
  file description 125
  LINK specification 128
  SPACE specification 127
  syntax 124
assignment 214, 230, 237
AT command 197, 216
AT DO 216

availability of test tables 237
AWAKE command 197, 223

B
batch mode 16
BEGIN command 197
BEGIN END 232
bit range specification 94, 98
block 198, 199, 226, 227
block mode 10
block qualification 199, 212, 226
BOOLEAN 231
BREAK ERROR 261
Break_Error 10
BS2000 files 114
BS2000CALLS 307

C
call chain, dynamic 262
CALL-STATEMENT-FILE 22
calling the compiler 24
calling the EDT 33
calling the programming system 6
CASE ERROR 261
character set 94
code module 99
combination of commands 232
combined testpoint and debugging aid end message 243
command mode 57
comment, continuation character 105
comments 202
common memory pool 137
compatibility, compiler and runtime system 137
compatibility problems 341
compatible data types, ILCS 162
compilation sequence 92
compilation summary 105
compilation unit 198, 205, 237
  current 198, 203, 205
compilation units 79
  compilation sequence 92
  relationships 80
COMPILE-UNIT 24
compiler
  call 24

generated listings 103
generated object modules 99
implementation-defined attributes 93
compiler limitations 95
compiler listing 106
compiler options 101
default values 101
priorities 101
specification 101
validity 101
compiler version, compatibility with runtime system 137
complete test table 214, 237
components 227
compound command 197, 232
condition 231
constant identifiers, predefined 212
constants
predefined 93
presettings 93
values 93
control statements 101
cross reference listing 109
current compilation unit 198, 203, 205

D
data module 99
data structures, ILCS 160
data types
alignment 96
ILCS 162
memory requirements 96
packed 96
unpacked 96
DEBUG 214, 230
debugging aid, see PATH 195
debugging aid options 202
default output lengths 94
for Boolean 94
for Integer 94
default output lengths- for Real 94
default value 17
deferred action 215, 216, 218, 236
DEFINE-PROJECT-FILE 32
delete testpoints 220

dereferenced 201
different programming languages 157
directives 164
DISPLAY command 197, 226
DMSIO 312
DUMP command 197, 239
dynamic call chain 212, 238, 239, 262
dynamic linking 153

E
EAM object module file, see *OMF 99
echoing of input 242
EDIT command 197, 234
EDIT-UNIT 33
editor command 234
EDT
  call 33
  error messages 36
EDT work area 116
  in Assignfile 116
  in the CALL statement 23
  in the COMPILE statement 25
  in the EDIT statement 33
  in the SHOW statement 53
EDTADAPTER 327
ELAB ERROR 261
END 39
entry procedures, naming 100
entry subprograms 171
entry testpoint 209
EOF ERROR 261
error handling 255
  for internal interface 179
  output 260
  SEH events 257
  system error code 285
error message 242, 247
error propagation across language boundaries 255
ERRORS 331
errors 106
event classes 256
exception handler 255
exception propagation across language boundaries 255
exceptions 255

execution
  of a program 44
  of a tool 44
expert mode 9
Extend 114
extension of the programming system 11
External, directive 164
external functions, result types 164
external references 136
external subprograms 157
  naming 100

F
factor 201, 226, 230, 231
feed control character 127
FILE ERROR 261
files
  assignment with Assignfile 124
  default assignments 120
  external 113
  FILE command assignment 120
  in RUN statement 131
  ISAM files 116, 312
  local 113
  opening 116
  PLAM libraries 118
  SAM files 116
  standard files 116
  supported BS2000 files 114
  temporary 119
four-digit year number 103, 237
function 198
function keys 10
function return values, passing in ILCS 162

G
GETCMD command 197, 216, 218
global scope 198, 203, 213, 242
guided dialog 9

H
HEAPSUPPORT 333
help levels 9
hexadecimal 227
hexadecimal output 227
hidden identifiers 212
high-precision routines 93

I
IF command 197, 231
IF THEN ELSE 231
ILCS
  data types 162
  functions 158
  initialization 161
  linking program systems 163
  parameter passing 162
  passing function return values 162
  program communication interface 157
  program mask handling 161
  register conventions 159
  Standard Event Handler (SEH) 256
  Standard STXIT Handler (SSH) 256
  subprograms in other languages 164
ILCS capability 137
ILCS conventions 157
ILCS data structures 160
implementation-defined attributes 93
implicit testpoint 203
incarnation number 199
incarnation qualification 199, 212, 226, 238
indexed 201
initialization
  ILCS 161
  Pascal-XT runtime system 161
input in block mode 10
input mode 218
input prompt 9, 216
interactive mode 16
internal interface 176
  error handling 179
  parameter passing 177
  register conventions 176
ISAM file

open modes 117
text file 117
ISAM files 312

J
job step 13
job variable 13

K
keyword 17
keyword operands 18
KILL command 197, 240

L
language interface 157
error handling 166, 173
file processing 157
for entry procedures 171
for XS-compatible subprograms 165
internal interface 176
language mix 157
license module 156
license protection 156
line length 95
line number 199, 203
line number range 199, 204
link names, conventions 120
linking 135
code and data separate 143
dynamic 153
on XS processors 138, 153
prelinked modules 140
prelinking the runtime system 146
segmented 146
static 138
to a phase 138
with PATH 248
within the programming system 153
listing output
control 103
output file 103

M
main program 198
map listing 111
mathematical routines, high-precision 93
MAXLINELENGTH 122, 124, 129
MEMORY ERROR 261
MEMORYMANAGER 339
metasymbols 17
metavariable 18
minimum test table 214, 237
MODIFY-COMPILE 40
MODIFY-EDT 42

N
nesting of procedures 212
NK format 117
non-Pascal subprograms, ILCS 164
normal program termination 13
notational conventions 17
notes 106
NUMERIC ERROR 261

O
object module file, see *OMF 99
object modules 99
  entry procedures 100
  external references 136
  naming 99
  XS capability 99
offset specification 94, 97
OPEN ERROR 261
open modes 114, 116, 117, 118, 119
operand value
  *LIBRARY 19
  *UNCHANGED 20
operand value change
  COMPILE statement 40
  EDIT statement 42
option 202
output all variables 239
output lengths, see default output lengths 94
output medium 226

P

package 198, 237, 239
package concept 79
packed data types 96
PAGE 95
effect on input 95
effect on output 95
PAM key format 117
parameter 227
parameter passing, ILCS 162
partial test table 214, 237
Pascal program, cf. program 137
Pascal-XT, Pascal-XT V3.0, compatibility problems 341
Pascal-XT program section 157
PASLIB-XT 137
passing on errors across language boundaries 255
PASSUP-XT 306
PATH 195
testpoint before program start 249
PATH commands, canceling 250
PATH input medium 215, 218, 242
PATH output medium 215, 226, 239
PCD (Prosys Common Data Area) 160
PLAM element, see PLAM library 118
PLAM library 118
element designation 118
in statements 19
open modes 118
specification in the programming system 19
PLAM library element
see PLAM library 118
specification in Assignfile 118
POINTER ERROR 261
positional operands 18
postmortem testpoint 206, 242
potential testpoint 198, 203, 216
predefined constant identifiers 212
predefined constants 93
predefined packages 306
library PASSUP-XT 306
object modules 306
specifications 306
using 306
predefined types 93

prelinked module
  handling of entries 140, 143
  handling of external references 140, 143
prelinked modules 140
procedure 198
  recursive 196
program
  executable 135
  shareable 137
program aborted 243
program continued 243
program execution 44
program information 14
program interface 157
program library, see PLAM library 118
program mask handling, ILCS 161
program monitoring 13
program parameters 121
program system 157
program systems, linking in ILCS 163
program termination 13
  abnormal 13
  application program 155
  job step 13
  message 39
  normal 13
  program information 13
  programming system 14
  spin-off mechanism 13
  status indicator 13
  termination code 13
program under test 198, 237
programming languages other than Pascal 157
programming system 5
  call 6
  extension 11
  modifications 7
  terminating 39
project directory 79
  compiler access 91
  defining 81
  definition 32
  output contents 48
  processing 81

status information 80, 82
tasks 79
using 83
propagating errors across language boundaries 255
Prosys Common Data Area (PCD) 160

Q

qualified 201

R

R option 202
R-option 242
RAISE 255
RANGE ERROR 261
RDEBUG 214, 230
READ ERROR 261
real numbers
exponent overflow 93
exponent underflow 93
precision 93
significant positions 93
value range 93
recompilation 48
RECORD 214
RECORD type
bit range specification 98
offset specification 97
recursive procedure 196
recursive subprogram calls 238
register contents, ILCS 159
register conventions, ILCS 159
relational operator 231
REMOVE command 197, 220
REMOVE-DIRECTORY-ENTRY 43
Replace 114
Reset 114
restart point, in the programming system 56
result types, external functions 164
RESUME 216
RESUME command 197, 219
Rewrite 114
routines, high-precision mathematical 93
RUN-PROGRAM 44
runtime error, see error handling 255
runtime system 137

compatibility with compiler 137
module names 146
prelinking 146

S

SAM file, open modes 117
save area, ILCS 160
scope 212
global 198, 203, 213, 242
scope of identifiers 196, 212
SDF 7
abbreviation rules 18
input file 16
secondary error 263
segmented linking 146
SEH, Standard Event Handler, ILCS 256
SEH-NON-STXIT events 257
SEH-STXIT events 257
semantic error 242
semantic errors 196
semicolon 215
omission of 215
sequence of commands 232
SET 214
set constructor 94
SET ERROR 261
set testpoints 216
set type, number of elements 94, 96
setting testpoints 216
shared code 137
SHOW CALLS command 238
SHOW command 197, 235
show testpoints 235
SHOW UNITS command 237
SHOW WHERE command 235, 242, 243
SHOW-ATTRIBUTES 48
showing the testpoints 235
SLEEP command 197, 222
slice 201
source listing 104
spin-off mechanism 13, 16, 21, 22, 25, 32, 34, 41, 43, 44, 50
SSH, Standard STXIT Handler, ILCS 256
Standard Event Handler (SEH), ILCS 256
standard files, open modes 116

Standard STXIT Handler (SSH), ILCS 256
starter module 99
statement file 22
status indicator 13
STEP 56
storage unit 94
STRING ERROR 261
string type 94
  default length 94
structured values 227
STXIT handling 256
subprogram calls, recursive 238
subprogram incarnation 238
subprograms, non-Pascal 157
subprograms in other languages 157
  ILCS 164
SWITCH command 197, 241
syntactic error 242
syntax errors 196
SYSTEM command 197, 233
SYSTEM ERROR 261
system error code 285
system mode 233
SYSTEM-COMMAND 57

T

tag field 214
temporary file, open modes 119
temporary files 119
terminating the programming systems 39
termination code 14
test table 243
  complete 214, 237
  minimum 214, 237
  partial 214, 237
test table not complete or not available 243
test tables 248
  availability 237
  dynamic loading 250
  static loading 248
  validity 251
test tables module 99
testing with PATH 249
  response to errors 252

testpoint
implicit 203
potential 198, 203, 216
user-set 203
testpoint before program start 217, 242
testpoint commands 197
testpoint message 236, 242
testpoint specification 199, 203, 216, 235
testpoints 203
activate 223
deactivate 222
testpoints before program start 203
text editor 234
tool execution 44
tools 11
execution 12
information 12
loading 11
requirements 11
tracing program execution 236
type compatibility 214
types 214
predefined 93

U
ulp-precision 93
unguided dialog 9
USE clauses 226
user guidance 9
expert mode 9
user prompting
input prompt 9
unguided dialog 9
user-set testpoint 203
UTM
data types and constants 190
error handling 193
external files 192
linkage 189
linking 194
predefined packages 190
program structure 191

V
variable 226, 230
variable group 226
variables 239
VARIANT ERROR 261
variants 214

W
warnings 106
WITH statements 212

X
XS capability 99, 138, 140, 153
XS-compatibility 165

Y
year, four-digit format 237
year number, four-digit 103