PLI1 V4.1A

PL/I Compiler

Edition May 1991

Comments… Suggestions… Corrections…

The User Documentation Department would like to know your opinion on this manual. Your feedback helps us to optimize our documentation to suit your individual needs.

Feel free to send us your comments by e-mail to:
manuals@ts.fujitsu.com

Certified documentation according to DIN EN ISO 9001:2000

To ensure a consistently high quality standard and user-friendliness, this documentation was created to meet the regulations of a quality management system which complies with the requirements of the standard DIN EN ISO 9001:2000.

cognitas. Gesellschaft für Technik-Dokumentation mbH
www.cognitas.de

Copyright and Trademarks

Contents

1	**Preface**	**1**
1.1	Target group and summary of contents	1
1.2	Changes since the last version of the manual	3
2	**Overview of the use of PLI1**	**5**
2.1	General program execution	5
2.1.1	General program execution with ISP commands	6
2.1.2	General program execution with SDF commands	10
2.2	Examples	13
2.2.1	Batch processing: reading, printing	13
2.2.2	Interactive task: input/output to files via SYSFILE command	18
2.2.3	Batch processing: input/output via LINK	20
2.2.4	Use of libraries	28
2.2.5	Use of debugging aids	32
2.2.6	File organization CONSECUTIVE, INDEXED and REGIONAL (1)	37
3	**Compiling a PL/I source program**	**41**
3.1	Functions of the PLI1 compiler	41
3.2	Invoking the PLI1 compiler	43
3.2.1	Invoking the compiler by ISP command	43
3.2.2	Invoking the compiler by SDF command	43
3.2.3	Monitoring by monitoring job variable	44
3.2.4	Message text file	45
3.2.5	Examples	46
3.3	Controlling the PLI1 compiler	50
3.3.1	General rules for control statements	51
3.3.2	Error handling during control statement evaluation	53
3.3.3	Summary of the control statements for the PLI1 compiler	54
3.3.4	Controlling the compiler via the source program	63
3.3.5	Storage requirements of the compiler (STORAGE)	65
3.4	Controlling source input	66
3.4.1	SYSFILE command (SYSDTA reassigned)	68
3.4.2	Defining the source file (SOURCE)	70
3.4.3	Defining the INCLUDE library (COMLIB)	75
3.4.4	Source line format (MARGINS)	76

3.5	Controlling the listing output of the PLI1 compiler	78
3.5.1	List selection (LIST)	79
3.5.2	Selection of diagnostic information (DIAGNOST)	82
3.5.3	Additional listing output of messages (MESSAGE)	82
3.5.4	Output format (FORMAT)	83
3.6	Controlling object module generation	85
3.6.1	Switch for compiler phases (OBJECT)	85
3.6.2	Debugging aids in the object module (DEBUG)	86
3.6.3	Procedure status (OPTIONS)	87
3.6.4	Optimization (OPTIMIZE)	88
3.6.5	Debugging aid AID (SYMTEST)	90
3.6.6	Object module storing (MODULE)	91
3.7	Object module maintenance	93
3.7.1	Object module to *EAM file	95
3.7.2	Object module to LMS library	97
3.7.3	Table of contents of a library	98
3.8	Listings of the compiler (LIST =)	99
3.8.1	Options (OPTIONS)	100
3.8.2	Preprocessor (INSOURCE)	101
3.8.3	Source listing (SOURCE, EXPAND)	102
3.8.4	External names (ESD)	106
3.8.5	Include texts (IREF)	108
3.8.6	Cross-reference listing	109
3.8.7	Structure lengths (AGGREGATE)	112
3.8.8	Storage occupancy (MAP)	115
3.8.9	Offset listing (OFFSET)	119
3.8.10	Assembly code (ASSM)	119
3.8.11	Statistics (SUMMARY)	121
3.9	Diagnostic messages (DIAGNOST)	122
3.10	Format of the SAVLST file	125
3.11	Use of the preprocessor	126
3.12	Description of the operands of the SDF command START-PLI1-COMPILER	131
3.12.1	Overview of the operands	131
3.12.2	Description of the individual operands	132
	SOURCE operand	132
	INCLUDE-LIBRARY operand	133
	SOURCE-PROPERTIES operand	134
	PREPROCESSING operand	135
	COMPILER-ACTION operand	136
	MODULE-LIBRARY operand	137
	LISTING operand	138
	TEST-SUPPORT operand	141

	OPTIMIZATION operand	143
	COMPILER-TERMINATION operand	144
	MONJV operand	145
	LANGUAGE operand	146
3.12.3	Mapping of SDF operands to COMOPT operands	147
4	**Linking and loading a PL/I program**	**149**
4.1	General	149
4.2	Controlling the linkage editor (TSOSLNK)	150
4.2.1	Calling the linkage editor	151
4.2.2	Statements for the linkage editor	152
	PROGRAM statement	152
	INCLUDE statement	153
	RESOLVE statement	154
	END statement	154
4.3	Example of the linkage editor	155
4.4	Loading	156
4.5	Runtime system	157
4.5.1	Elementary runtime system	159
4.5.2	Prelinked runtime system	159
4.5.3	Storage of the runtime system	160
4.6	Name conventions for PL/I object modules	161
4.7	Extended address space (XS)	163
5	**Execution of the PL/I program**	**165**
5.1	General	165
5.2	Program execution	167
5.2.1	Execution with ISP command	167
5.2.2	Execution with SDF command	167
5.2.3	Monitoring by monitoring job variable	168
5.2.4	Examples	169
5.3	Controlling the PL/I program	172
5.3.1	General rules for control statements	172
5.3.2	Error handling for control statement evaluation	172
5.3.3	Controlling the listing output	173
5.3.4	Overview of the control statements for PL/I programs	174
5.4	Individual descriptions of the control statements for PL/1 programs	180
5.4.1	Parameter transfer (ARGUMENT)	180
5.4.2	Dump control (DUMP)	181
5.4.3	Output formats (FORMAT)	181
5.4.4	List selection (LIST)	182
5.4.5	Additional message output (MESSAGE)	182
5.4.6	Storage requirements (STORAGE)	183
5.4.7	System file assignment (SYSFILE)	184

5.4.8	Trace control (TRACE)	186
5.4.9	Setting tabs (TABULATOR)	186
5.4.10	Alignment control (CONTROL)	187
5.5	Check activation (ACTIVE)	187
5.6	Program interrupt	188
5.7	Description of the operands of the SDF command START-PLI1-PROGRAM	192
5.7.1	Overview of the operands	192
5.7.2	Description of the individual operands	193
	FROM-FILE operand	193
	CPU-LIMIT operand	194
	MONJV operand	194
	START-PARAMETERS operand	194
	LANGUAGE operand	194
	ASSIGN-SYSLST operand	195
	ASSIGN-SYSOUT operand	195
	ASSIGN-SYSDTA operand	195
	TEST-SUPPORT operand	196
	LISTING operand	198
	HEAP-ADMINISTRATION operand	199
	STACK-ADMINISTRATION operand	200
	TABULATOR-POSITION operand	201
5.7.3	Mapping of SDF operands to RUNOPT operands	202
6	**File access by PL/I programs**	**203**
6.1	General	203
6.2	BS2000 files	204
6.2.1	System files	204
6.2.2	User files	206
6.2.2.1	General	206
6.2.2.2	File name	207
6.2.2.3	File link name	208
6.2.2.4	File organization (access method)	209
6.2.2.5	Record structure	211
6.2.2.6	Volumes	217
6.2.2.7	Access authorization	218
6.2.2.8	File size	219
6.2.2.9	Other file specifications	221
6.2.3	Interfaces between the PL/I input/output system and BS2000	221
6.3	Program files and ENVIRONMENT attribute	222
6.3.1	Types of organization	224
6.3.2	Record structure	225
6.3.3	Key options	227

6.3.4	Carriage control	228
6.3.5	Controlling the VARYING variable	229
6.3.5.1	Reading a record using READ INTO	230
6.3.5.2	Writing a record using WRITE FROM	232
6.3.5.3	Reading a record using READ SET	232
6.3.5.4	Writing a record using LOCATE SET	234
6.3.6	Device control using TRANSIENT files	235
6.3.7	Other options	235
6.4	Assigning program files to BS2000 files	236
6.4.1	Rules governing the assignment of file names	236
6.4.2	Rules governing the assignment of organization methods	239
6.4.3	Relationship between FILE command and ENVIRONMENT specification	240
6.4.4	Determining the file characteristics	242
6.4.4.1	Characteristics for a new file	244
6.4.4.2	Extend old output file (EXTEND)	246
6.4.4.3	Characteristics for existing files	247
6.5	Stream-oriented input and output	248
6.5.1	Rules governing stream-oriented input and output	249
6.5.2	PRINT files	251
6.5.3	Stream-oriented input/output to interactive device	253
6.5.3.1	Input from interactive device (SYSDTA)	253
6.5.3.2	Output to interactive device (PRINT file to SYSOUT)	254
6.6	Record-oriented input and output	256
6.6.1	Rules governing CONSECUTIVE organization	258
6.6.1.1	Opening CONSECUTIVE files	259
6.6.1.2	Closing CONSECUTIVE files	259
6.6.1.3	Writing to a CONSECUTIVE file	259
6.6.1.4	Reading from a CONSECUTIVE file	260
6.6.1.5	Overwriting records in a CONSECUTIVE file	260
6.6.1.6	Deleting records in a CONSECUTIVE file	260
6.6.1.7	FILE command for CONSECUTIVE files	261
6.6.2	Rules governing INDEXED organization	263
6.6.2.1	Key specification	264
6.6.2.2	Opening an INDEXED file	265
6.6.2.3	Closing an INDEXED file	265
6.6.2.4	Writing to an INDEXED file	266
6.6.2.5	Reading from an INDEXED file	266
6.6.2.6	Overwriting records in an INDEXED file	267
6.6.2.7	Deleting records in an INDEXED file	267
6.6.2.8	FILE command for INDEXED files	267
6.6.3	Rules governing REGIONAL(1) organization	269
6.6.3.1	Key specification	271
6.6.3.2	Dummy records	271

6.6.3.3	Opening a REGIONAL(1) file	272
6.6.3.4	Closing a REGIONAL(1) file	273
6.6.3.5	Writing to a REGIONAL(1) file	273
6.6.3.6	Reading from a REGIONAL(1) file	273
6.6.3.7	Overwriting records in a REGIONAL(1) file	273
6.6.3.8	Deleted records in a REGIONAL(1) file	274
6.6.3.9	FILE command for REGIONAL(1) files	274
6.6.4	Rules governing REGIONAL(3) organization	275
6.6.4.1	Key specification	276
6.6.4.2	Dummy records	279
6.6.4.3	Opening a REGIONAL(3) file	279
6.6.4.4	Closing a REGIONAL(3) file	279
6.6.4.5	Writing to a REGIONAL(3) file	280
6.6.4.6	Reading from a REGIONAL(3) file	280
6.6.4.7	Overwriting a REGIONAL(3) file	280
6.6.4.8	Deleting a record in a REGIONAL(3) file	281
6.6.4.9	FILE command for REGIONAL(3) files	281
6.7	Magnetic tape	283
6.7.1	Access methods for tape files	283
6.7.2	File attributes	283
6.7.3	Accessing	284
6.7.4	Closing the file	285
7	**Procedure interface**	**287**
7.1	PL/I interfaces	288
7.1.1	Invocation interface	288
7.1.1.1	Invoking procedure	289
7.1.1.2	Prolog	289
7.1.2	Passing of parameters	292
7.1.2.1	Normal case (PL/I)	293
7.1.2.2	General assembler convention (VARIABLE)	296
7.1.2.3	Standard assembler convention (ASSEMBLER)	298
7.1.3	Problem processing	299
7.1.4	Return of the result	299
7.1.4.1	Return in register 1	300
7.1.4.2	Return in floating point registers	300
7.1.4.3	Return via parameters	300
7.1.4.4	Return when * is specified	301
7.1.5	Terminating a procedure	302
7.1.5.1	Return	303
7.1.5.2	Branch	304
7.1.6	Library procedure (LIBRARY)	305
7.1.7	(WXTRN) Linkage	306
7.2	Assembler procedures	307

7.2.1	Assembler procedure conforming to PLI1 conventions	308
7.2.2	Assembler procedures conforming to standard assembler conventions	309
7.2.3	Assembler procedures conforming to general assembler conventions (VARIABLE)	309
7.2.4	Invocation of PL/I procedures from assembly-language programs	310
7.3	FORTRAN and COBOL procedures	311
7.3.1	General	311
7.3.2	Matching the data	312
7.3.3	Declaration, call	316
7.3.4	Interrupt handling	317
7.3.5	Program termination	318
7.3.6	Invocation of PL/I procedures from FORTRAN and COBOL programs	319
7.4	ILCS procedures	320
7.4.1	General	320
7.4.2	Start handling	321
7.4.3	Declaration, call	321
7.4.4	Mapping of files	322
7.4.5	Interrupt handling	322
7.4.6	Termination handling	322
8	**Optimization facilities**	**323**
8.1	Overview	324
8.1.1	Compiler	324
8.1.2	Runtime system	325
8.2	Manual optimization	326
8.2.1	Running a program - stage 1	326
8.2.2	Tuning a program - stage 2	327
8.2.3	Tuning a program for virtual storage	331
8.2.4	Modular programming	333
8.3	In-line operations	334
8.3.1	Data conversion	334
8.3.2	String handling	339
8.4	Global optimization features	340
8.4.1	Common expressions	340
8.4.1.1	Interrupt handling	341
8.4.2	Transfer of invariant expressions or statements out of DO loops	342
8.4.3	Reduction of linear expressions in DO loops	343
8.4.4	ORDER and REORDER options	343
8.4.4.1	ORDER options	344
8.4.4.2	REORDER option	344
8.4.5	Elimination of side effects / reducible functions	346
8.4.6	Optimization of Boolean expressions	346
8.4.7	Expression simplification	347

8.4.8	Initialization of aggregates	348
8.4.9	Special code for aggregate assignment	348
8.4.10	Utilization of registers in DO statements	349
8.4.11	Internal procedure calls	349
8.4.12	Utilization of global optimization	349
8.4.12.1	Common expression elimination	350
8.4.12.2	Transfer of invariant expressions	352
8.4.12.3	Reduction of linear expressions in loops	352
8.4.12.4	Register and address optimization	352
8.4.12.5	Use of registers in DO statements	352
8.5	Optimization control (OPTIMIZE)	353
8.5.1	Time optimization (TIME)	353
8.5.2	Change enabling of conditions (ENABLING)	353
8.5.3	Sequence of statements modifiable (REORDER)	353
8.5.4	Overlapping (OVERLAP)	354
8.6	Programming notes	355
8.6.1	Source program and general syntax	355
8.6.2	Program control	356
8.6.3	Declarations and attributes	357
8.6.4	Assignment and initialization	360
8.6.5	Arithmetic expressions, Boolean expressions and conversions	362
8.6.6	DO groups	366
8.6.7	Aggregates	369
8.6.8	Strings	370
8.6.9	Functions and pseudo variables	370
8.6.10	Conditions and ON units	371
8.6.11	Input/output	372
8.6.12	Procedure functions with several entries	375
8.6.13	Variable length entry	377
8.6.14	Passing of parameters	377
8.6.15	Absolute bit pointer for XS	377
9	**Debugging aids**	**379**
9.1	Compiler control	380
9.2	Program control	381
9.3	Trace output	383
9.4	Activation of check points	383
9.5	Program interrupt	384
9.6	Dump	384
9.7	SNAP	385
9.8	Interface to the AID debugger	386

10	**Internal Representation**		**387**
10.1	Arithmetic variables		389
10.1.1	Fixed binary variables (FIXED BINARY)		390
10.1.2	Fixed decimal variables (FIXED DECIMAL)		393
10.1.3	Floating point variables (FLOAT)		396
10.1.4	Picture character string variable (PICTURE)		399
10.1.5	Complex variables		399
10.2	String variables		400
10.2.1	Bit string variables (BIT)		400
10.2.2	Character string variables (CHARACTER)		403
10.2.3	Picture variable (PICTURE)		405
10.3	Program control variables		406
10.3.1	Pointer (POINTER, OFFSET)		406
10.3.1.	Pointer if "*COMOPT OPTIONS = NOXS"		406
10.3.1.	Pointer with "*COMOPT OPTIONS = XS"		407
10.3.2	Area (AREA)		408
10.3.3	Label (LABEL)		410
10.3.4	Format (FORMAT)		411
10.3.5	Entry (ENTRY)		412
10.3.6	File (FILE)		413
10.4	Array (DIMENSION)		414
10.5	Structure (STRUCTURE)		416
10.5.1	Storage requirements		416
10.5.2	Aliased variables		419
10.5.3	Matching at the beginning		421
10.5.4	Self-defining structures		424
10.5.5	Record		426
10.6	Description of the data type		428
10.6.1	Data description		428
10.6.2	Picture description		433
10.7	Storage management		436
10.7.1	Static variables (STATIC)		436
10.7.2	Activation records (stack, AUTOMATIC)		437
10.7.3	Standard area (CONTROLLED, BASED)		442
10.7.4	Named area (AREA)		446
10.7.5	Reference chain for CONTROLLED variable		449
11	**Utilities**		**453**
	ADUMP	Dump from the standard area	455
	BS2SRT	Sort/merge	457
	CMD	Execute BS2000 command	462
	ERROUT	Error text output	464
	HEXDEC (a)	Hexadecimal characters	465
	NOTRACE	Trace off	467

	PLIRETC	Set return code	468
	RDUMP (a,b)	Dump	469
	RUNTIME	Computing time used	470
	SDUMP	Stack dump	471
	SNAP	Call nesting	473
	TRACE	Trace on	475
12	**Shareable programs**		**477**
12.1	Prerequisites		477
12.2	PL/I programs		478
12.2.1	STATIC variables		478
12.2.2	Input/output statements		479
12.2.3	CONTROLLED variables		479
12.3	Entry into class 4 memory		480
13	**PLI1 ASSEMBLER macro interface**		**481**
13.1	General		481
13.1.1	Table of macros		481
13.1.2	User considerations		482
13.2	Macros		483
	P$CALL		483
	P$ENTRY		486
	P$ENVIRM		488
	P$ERROR		490
	P$LINK		491
	P$PRV		493
	P$REGEQU		494
	P$RETURN		495
	P$STACK		497
	P$STOP		498
14	**Appendix**		**499**
14.1	List of compiler warnings and error messages		499
14.2	List of error messages from object programs (ONCODE values)		501
14.3	Constraints on implementation		521
14.4	Distinctions from PL/I-D		525
14.5	Examples of sorting		528
14.6	Additional information on information messages		535
14.7	Runtime modules		543
14.8	Messages of the PLI1 runtime system		552
14.9	Examples of PLI1 ASSEMBLER macros		556
15	**References**		**561**
	Index		**567**

1 Preface

1.1 Target group and summary of contents

This user guide is intended for programmers who use the software product PLI1. While the language reference manual for this PL/I compiler is largely free of peripheral conditions relating to the operating system, this manual describes the integration and operation of the compiler in BS2000. A general knowledge of this operating system is assumed. Explanations of system characteristics are given only if such information will be of use to the PL/I user.

The reader should be familiar with the PL/I programming language as described in the manual

PLI1 (BS2000)
PL/I Compiler
Language Reference Manual

The present manual is geared to applications in both interactive and batch mode. It does not cover operation of the PL/I (D level) compiler.

An initial overview is provided by the examples in chapter 2. The execution of some simple applications is explained in detail using the computer listings.

Chapter 3 explains in detail all the control facilities for the PLI1 compiler.

These include:

- selection of different listing formats
- preparation of the source program
- storage and characteristics of the generated object modules
- notes on the management of object modules in libraries.

Chapter 4 describes the linkage procedure for standard applications. Users who wish to segment (OVERLAY) their programs or perform other special operations during linkage will find the necessary information in the "Utility Routines" manual [3], and in the "Linkage Editor and Loaders" manual [12] if using BS2000 version 8.0 or higher.

Some effects of executable user programs can be controlled externally. This applies especially to

- activation of debugging and testing aids
- passing of parameters
- linking of system files
- selection of standard listings

These facilities are presented in chapter 5.

The use of files is described in detail in chapter 6. An overview of the file interface provided by BS2000 is followed by a discussion of various aspects of the peripheral conditions concerning PL/I programs.

In chapter 7 the interface for PL/I procedures is represented in machine-oriented format. This interface can also be used to set up interfaces for assembly-language programs that behave like PL/I procedures. Other interfaces are discussed that can be set up by means of special control options. A description is also given of interfaces to procedures written in other programming languages (see also chapter 13).

Chapter 8 shows how PL/I programs can be optimized.

Chapter 9 describes debugging aids applied within the framework of the PLI1 system.

Chapter 10 deals with the internal representation of data and listings from the viewpoint of their interest to users for debugging purposes.

Chapter 11 describes utilities that can be invoked in PL/I programs by a subroutine reference (CALL) or a function call.

Chapter 12 explains what to do when a program is to be shared by several users.

chapter 14 describes a number of macros that facilitate the writing of assembly-language procedures that are to behave exactly like PL/I procedures.

Where possible, important information is summarized in table form. The tables can also be found in the PLI1 Ready Reference. In some instances formal syntax notation is used; this conforms to the syntax rules defined in the language reference manual [1].

1.2 Changes since the last version of the manual

The most important changes compared with the previous edition of the manual are summarized below:

- The PLI1 compiler can also be invoked using the SDF command START-PLI1-COMPILER (sections 2.1.2 and 3.2).
- PLI1 programs can also be invoked using the SDF command START-PLI1-PROGRAM (sections 2.1.2 and 5.2).
- Descriptions of the operands of the SDF command START-PLI1-COMPILER and table showing the mapping of the SDF operands to the COMOPT operands (section 3.12).
- Descriptions of the operands of the SDF command START-PLI1-PROGRAM and table showing the mapping of the SDF operands to the RUNOPT operands (section 5.7).
- PLI1 language interfacing facilities also permit program-to-program communication at runtime in accordance with the conventions of the Inter Language Communication Services (ILCS). These conventions support the linkage of any combination of programs written in different programming languages (see section 7.4).

2 Overview of the use of PLI1

This section shows, in simple examples, how PL/I compilations and PL/I program runs can be performed in BS2000. The examples are a guide for processing common applications.
Normally, the examples apply to any kind of job (task); i.e. to processing in both batch and interactive mode. Any differences between batch and interactive mode are discussed.

2.1 General program execution

To understand the command sequences in the examples, the general flow for compiling and executing the program is briefly described below.

Using the PLI1 compiler, PL/I source programs are converted into object modules. In this process, the programs are checked for correct syntax and semantics and various listings are generated. The object modules are stored in the EAM file or in an LMS library and can be linked by the linkage editor (TSOSLNK) into load modules and stored in a file. Then the linked object program can be loaded and executed. The most important task within the linkage process is to combine the runtime library with the object modules generated by the user as a result of compilation. The runtime library contains all the ready-made modules for input/output and for condition handling as well as built-in functions and a program monitor which controls the correct execution of the object program.

The user is able to control the PL/I compiler and object program. By means of control statements to the PL/I compiler, the user can define the listings, checks, optimizations etc. to be performed and the attributes that will be assigned to the generated object module and the ultimate program. All control statements have default (preset) values which become effective if the user does not specify any options. Examples of simple program execution are described in the following chapters. The user can choose between BS2000 commands in ISP format [2] or SDF commands (System Dialog Facility) [19].

2.1.1 General program execution with ISP commands

The following example of a simple command procedure (using BS2000 commands in ISP format) illustrates the commands required to compile, link and execute a source program:

```
/ PROCEDURE C,(&NAME, &LIST=SOURCE, &COMOPT='COMLIB=NO'), SUBDTA=&       1)
/         ERASE *
/         SYSFILE SYSDTA=(SYSCMD)
/ REMARK ..................................................... TRANSLATE
/         EXEC $PLI1                                                     2)
               *COMOPT FORMAT=PRINTER(64,72),
               *COMOPT SOURCE=&NAME,
               *COMOPT LIST=&LIST,
               *COMOPT &COMOPT,
               *END
/ REMARK ..................................................... LINK       3)
/         EXEC $TSOSLNK
               PROGRAM &NAME, FILENAM=PROG.&NAME, MAP=NO
               INCLUDE *
               END
/ REMARK ..................................................... EXECUTE    4)
/         EXEC PROG.&NAME
/ REMARK ..................................................... END
/         STEP
/         SYSFILE SYSDTA=(PRIMARY)                                       5)
/         ENDP
```

The various commands have the following effect:

1) PROCEDURE

   The procedure has three parameters:

   – &NAME for the name of the source to be compiled.

   – &LIST for controlling the listing output from the compiler.
     Default: SOURCE listing.

   – &COMOPT for specifying any other control statements to the compiler.
     COMLIB = NO serves to reserve space and corresponds to a common default.

   ERASE

   Erase the EAM file if it still contains object modules from previous compiler runs.

   SYSFILE

   Switch from SYSDTA to the command file so that the control statements for the compiler and linkage editor can be read from there.

2) Invoke PLI1 with variable control statements.

3) Call the linkage editor. Listing feature deactivated.

4) Call the user program.

5) Reset SYSDTA.

For example, assume that a source program is contained in the EXAMP21 file.

Then the procedure might be called as follows:

```
/DO Proc.(EXAMP21,COMOPT='LIST=NOF')
```

Runtime listing on SYSOUT

```
/DO PLI1.PROZ,EXAMP21,COMOPT='LIST=NOF'
/ PROCEDURE C,(&NAME, &LIST=SOURCE, &COMOPT='COMLIB=NO'), SUBDTA=&
/         ERASE *
/         SYSFILE SYSDTA=(SYSCMD)
/ REMARK ..................................................... TRANSLATE
/         EXEC $PLI1
%  BLS0500 PROGRAM 'PLI1', VERSION '4.0A' OF '88-10-03' LOADED.

..... THERE WAS NO DIAGNOSTIC MESSAGE
..... OBJECTMODUL 'EXAMP21'  GENERATED AND WRITTEN TO: *EAM
..... OBJECTMODUL 'EXAMP217' GENERATED AND WRITTEN TO: *EAM
END OF SIEMENS PLI1-COMPILER VERSION 4.0A , TIME USED:     1.43 SEC
/ REMARK ..................................................... LINK
/         EXEC $TSOSLNK
%  BLS0500 PROGRAM 'TSOSLNK', VERSION '21.0C40' OF '87-09-28' LOADED.
%  LNK0500 PROG BOUND
%  LNK0503 PROG FILE WRITTEN: PROG.EXAMP21
%  LNK0504 NUMBER PAM PAGES USED:      7
/ REMARK ..................................................... EXECUTE
/         EXEC PROG.EXAMP21


%  BLS0500 PROGRAM 'EXAMP21', VERSION ' 'OF 88-10-06' LOADED.
END OF PROGRAM EXAMP21 , RTS 4.0A-AAA, TIME USED:   0.16 SEC
/ REMARK ..................................................... END
/         STEP
/         SYSFILE SYSDTA=(PRIMARY)
/         ENDP
```

Compiler control statement listing on SYSLST (*COMOPT LIST = OPTIONS)

```
COMPILER-OPTIONS USED


  STORAGE = (STACK(16,4),AREA(16,16,975))
  LIST    = (NOESD,NOTERMINAL,NOSUMMARY,OPTIONS,SAVLST,NOMAP,
             NEST,IREF,NOXREF,SOURCE,NOINSOURCE,NOAGGREGATE,
             OFFSET,NOASSM,NOOUTTEXT,NOLINECNT)
  FORMAT  = (TERMINAL(0,80),PRINTER(64,72),ENGLISH)
  MESSAGE = NOSYSLST
  SOURCE  = EXAMP21
  MARGINS = (TEXT(2,72),PAD,NOLINID,NOASACNTRL,GAMKEY(0,0),CHAR60,
             NOSAVMAC)
  DIAGNOST= (NOTERMINAL,NOSAVLST,WARNING)
  COMLIB  = NO
  OBJECT  = (ERROR(32767),ABORT(500),OUT)
  OPTIONS = (NOISO,NOMAIN,NOINTERRUPT,NOMACRO,NOXS,NOBITPTR)
  OPTIMIZE= (NOTIME,NOOVERLAP,NOENABLING,NOREORDER)
  DEBUG   = (NOSTMT,NOPROCTRACE,NOLABTRACE,NOCALLTRACE,
             NOGOTOTRACE,NORETURNTRACE,NOBREAKPOINT)
  SYMTEST = MAP
  MODULE  = *
```

Source program listing on SYSLST (*COMOPT LIST = SOURCE)

```
1     EXAMP21: PROC OPTIONS(MAIN);
2              DCL SYSPRINT FILE;
3              PUT PAGE LIST('VERY EASY TO HANDLE');
4              END;
```

Linkage editor (TSOSLNK) listing on SYSLST

```
PROGRAM EXAMP21, FILENAM=PROG.EXAMP21, MAP=NO
INCLUDE *
END
PROG BOUND
PROGRAM FILE WRITTEN : PROG.EXAMP21
NUMBER PAM PAGES USED:      7
```

Object program output on SYSLST

```
VERY EASY TO HANDLE
```

A simple procedure like the above is not sufficient if:
- programs become more complex,
- control of the object program is desired,
- several files are to be processed,
- precompiled program sections are stored in a library,
- compiler and compiler system files were not cataloged under $TSOS in the recommended manner.

The control facilities for the compiler are explained in detail in chapter 3. The object controls are discussed in chapter 5. The use of libraries by the linkage editor is described in chapter 4.

The user must be aware that an executable program (load module) can only be created if all the necessary object modules are available.

These are:

- Object modules resulting from the compilation of user-own procedures. If these compilations do not run ahead within the current task, then the private object modules must be supplied from one of the user libraries.
- Object modules which are part of the system and as such are linked on a standard basis but depend on the specific language elements being used (runtime system). For the following examples, the runtime system was available under $TSOS.TASKLIB.

  For example, if the runtime system is stored in the $TSOS.PLI1.MODLIB file, the following statement must be inserted every time the user wishes to link a module:

  ```
  RESOLVE,$TSOS.PLI1.MODLIB
  ```

Then the object program generated by the user with the linkage editor (TSOSLNK) and stored in a file can be executed as often as required. It can be modified by control statements issued to the object program.

Files to be accessed by an object program must be made available by the user before the start of the program, making sure that the file characteristics and manipulations described in the PL/I program are compatible with the file characteristics supported by BS2000. For details of the relationship between PL/I programs and BS2000, refer to chapter 6.

2.1.2 General program execution with SDF commands

Besides the functions of the BS2000 command language in ISP format, there are two SDF commands available, one for compiling a PL/I source program and another for running a PL/I object program.

The SDF command START-PLI1-COMPILER causes the PLI1 compiler to initiate a compilation; almost all the control options are available as compiler options. The SDF command START-PLI1-PROGRAM starts a PL/I object program generated by the PLI1 compiler; again, almost all the control options are available as object program options.

The following SDF command sequence corresponds almost exactly to the ISP commands given in the previous section:

```
/START-PLI1-COMPILER SOURCE = example.source                    1)

/EXEC $TSOSLNK                                                  2)
 PROGRAM example,FILENAM=example.object,MAP=NO
 INCLUDE *
 END

/ START-PLI1-PROGRAM FROM-FILE = example.object                 3)
```

The individual commands have the following effect:

1) Call to the PLI1 compiler with control statements for compiling a PL/I procedure contained in the file with the name example.source. Prerequisites: the PLI1 procedure and the PLI1 syntax file for SDF must be installed and the PLI1 compiler must be stored in the $TSOS.PLI1 file.
2) Call to the linkage editor with the control statement for linking the object module generated by the PLI1 compiler with the name example from the EAM library, then storing the generated load module in the file with the name example.object. Prerequisites: the object module generated by the PLI1 compiler must be available in the EAM library. The PLI1 runtime system must be present in the $TSOS.TASKLIB library, otherwise the tasklib would have to be redirected to the PLI1 runtime library using the SYSFILE command.
3) Call to the linked PL/I object program from the file with the name example.object. Prerequisites: the PLI1 procedure and PLI1 syntax file for SDF must be installed.

The following listings are generated:

Runtime listing to the SYSOUT system file:

```
/START-PLI1-COMPILER SOURCE=EXAMPLE.SOURCE
% BLS0500 PROGRAM 'PLI1', VERSION '4.1A' OF '91-04-23' LOADED. COPYRIGHT...
———— THERE WAS NO DIAGNOSTIC MESSAGE
———— OBJECTMODUL 'EXAMPLE' GENERATED AND WRITTEN TO: *EAM
———— OBJECTMODUL 'EXAMPLE7'GENERATED AND WRITTEN TO: *EAM
END OF SIEMENS PLI1-COMPILER VERSION 4.1A , TIME USED:     2.11 SEC
/EXEC $TSOSLNK
%  BLS0500 PROGRAM 'TSOSLNK', VERSION 'V21.0D12' OF '90-05-10' LOADED.
%  LNK0500 PROGRAM BOUND
%  LNK0503 PROGRAM FILE WRITTEN: EXAMPLE.OBJECT
%  LNK0504 NUMBER PAM PAGES USED:     11
/START PLI1-PROGRAM EXAMPLE.OBJECT
%  BLS0500 PROGRAM 'EXAMPLE', VERSION ' ' OF '91-05-02' LOADED.
END OF PROGRAM EXAMPLE , RTS 4.1A-DDD, TIME USED:     0.80 SEC
```

Compiler listing to the SYSLIST system file:

```
COMPILER-OPTIONS USED

STORAGE = ( STACK(16,4), AREA(16,16,1775))
LIST    = ( NOESD, NOTERMINAL, NOSUMMARY, OPTIONS, NOSAVLST, NOMAP, NEST,
            IREF, NOXREF, SOURCE, NOINSOURCE, NOAGGREGATE, OFFSET, NOASSM,
            NOOUTTEXT, NOLINECNT )
FORMAT  = ( TERMINAL(0,80), PRINTER(64,132), ENGLISH )
MESSAGE = NOSYSLIST
SOURCE  = EXAMPLE.SOURCE
MARGINS = ( TEXT(2,72), PAD, NOLIND, NOASACNTRL, GAMEKEY(0,0), CHAR60,
            NOSAVMAC )
DIAGNOST= ( NOTERMINAL, NOSAVLST, WARNING )
COMLIB  = NO
OBJECT  = (ERROR(32767), ABORT(500), OUT )
OPTIONS = ( NOISO, NOMAIN, NOINTERRUPT, NOMACRO, XS, NOBITPTR )
OPTIMIZE= ( NOTIME, NOOVERLAP, NOENABLING, NOREORDER )
DEBUG   = ( NOSTMT, NOPROCTRACE, NOLABTRACE, NOCALLTRACE, NOGOTOTRACE,
            NORETURNTRACE, NOBREAKPOINT )
SYMTEST = MAP
MODULE  = *

        1      EXAMPLE: PROC OPTIONS(MAIN);
        2               DCL SYSPRINT FILE;
        3               PUT PAGE LIST('VERY EASY TO HANDLE');
        4               END;
```

Linkage editor listing

```
PROGRAM EXAMPLE,FILENAM=EXAMPLE.OBJECT,MAP=NO
INCLUDE *
END
PROG BOUND
PROGRAM FILE WRITTEN : EXAMPLE.OBJECT
NUMBER PAM PAGES USED:     11
```

Output of the PL/I program:

```
VERY EASY TO HANDLE
```

For more complicated program execution, more control options will be required. A more detailed description of all the control options is given in chapter 3 for the PLI1 compiler, in chapter 4 for the linkage editor and in chapter 5 for the PLI1 object program.

2.2 Examples

The following examples demonstrate common command sequences as required for frequent operations involving PL/I programs. General prerequisites:

- PLI1 compiler stored under $TSOS,
- PLI1 runtime system available under $TSOS.TASKLIB, and
- PLI1 message text files cataloged under $TSOS.PLI1.TEXT.D or E.

2.2.1 Batch processing: reading, printing

Task:

Every time a call is issued, the following program uses GET LIST to read values for 3 variables from the SYSIN file, each requiring 11 data elements. The internal procedure PROCESS symbolizes the processing of the read-in data, which in this case consists merely of formalized output to SYSPRINT.

Prerequisite:

The PL/I program is executed as a batch task. The data for the program is read in from SYSIN and follows the source program cards. There are two ways of running this task:

- The card deck shown below is entered via card reader and processed by BS2000.
- The commands, control statements, sources, and data corresponding to the card deck shown below are contained in a file (e.g. entered by the DATA command). In this case, the task is activated using the ENTER command, e.g.

```
E/ENTER filename, TIME=...
```

Structure of the card deck:

Data cards read in by object programs

```
  15   ,  23  , 'AB'  ,  'CD'  'EF'  ,  15 30 , 16  32 , 25 50
 16 24,'AC' 'CE' 'EG',16 31,17 33,26 51
17 25 'AD' 'CF' 'EH' 17 32 18 34 27 52
```

Runtime listing on SYSOUT

```
/ REMARK ..................................................... TRANSLATE
/ EXEC $PLI1                                                          1)
%  BLS0500 PROGRAM 'PLI1', VERSION '4.0A' OF '88-10-03 LOADED.
          *COMOPT LIST=(NO,SOURCE)
          *END
```

Source program (for program text see source listing on SYSLST, next page)

```
..... THERE WAS NO DIAGNOSTIC MESSAGE
..... OBJECTMODUL 'EXAM221'  GENERATED AND WRITTEN TO: *EAM              2)
..... OBJECTMODUL 'EXAM2217' GENERATED AND WRITTEN TO: *EAM
END OF SIEMENS PLI1-COMPILER VERSION 4.0A , TIME USED:     1.31 SEC
/ REMARK ..................................................... LINK
/ EXEC $TSOSLNK                                                       3)
%  BLS0500 PROGRAM 'TSOSLNK', VERSION '21.0C40' OF '87-09-28' LOADED.
          PROGRAM EXAM221, MAP=NO, FILENAM=EXAM221
          INCLUDE *
          END



%  LNK0500 PROG BOUND
%  LNK0503 PROG FILE WRITTEN: EXAM221
%  LNK0504 NUMBER PAM PAGES USED:      9
/ REMARK ................................................ EXECUTE
/ EXEC EXAM221                                                        4)
%  BLS0500 PROGRAM EXAM221, VERSION   OF 88-10-06 LOADED.
```

Input data (see previous page)

```
END OF PROGRAM EXAM221 , RTS 3.2A-666, TIME USED:     0.17 SEC
/ REMARK ..................................................... END
```

1) Start the PLI1 compiler using the EXEC command. The LIST = (NO, SOURCE) control statement included in the deck ensures that only the listing of the source program will be output to SYSLST.

2) The following object modules are generated and stored in the EAM area:

   – code module EXAM221,
   – data module EXAM2217.

   The rule of formation for these names can be found in section 4.6.

3) Initiate the linkage editor.

4) Start the load module. The name of the program can be obtained from the PROGRAM statement for the linkage editor.

Source listing on SYSLST

```
 1      EXAM221: PROCEDURE OPTIONS(MAIN);
 2
 3               DCL SYSPRINT      FILE INTERNAL;
 4               DCL A             FIXED BIN     DIM(2)    INIT(0,0);
 5               DCL B             CHAR(5)       DIM(3)    INIT((3)(1)'');
 6               DCL 1 C                         DIM(3),
 7                   2 NO          FIXED DEC                 INIT ((3)0),
 8                   2 VALUE       FIXED DEC                 INIT(0,0,0);
 9               DCL SYSIN         FILE STREAM INPUT;
10
11               OPEN FILE(SYSIN);
12               ON ENDFILE(SYSIN) GOTO END;
13               ON CONVERSION  BEGIN;
14   1                          ON CONVERSION SYSTEM;
15   1                          PUT DATA;
16   1                          END;
17
18            E: GET FILE(SYSIN) LIST(A,B,C);
19               CALL PROCESS;
20               GOTO E;
21      PROCESS: PROCEDURE;
22   1           PUT SKIP(1) EDIT (A) (      (2)F(3))
23   1                            (B) (X(2), (3)A(5))
24   1                            (C) (         F(3));
25   1           PUT SKIP(1);
26   1           END;
27
28          END: PUT SKIP(1) LIST('—— END EXAMPLE 1');
29               END;
```

```
PROGRAM EXAM221,MAP=NO,FILENAM=EXAM221
INCLUDE *                                    5)
END
PROG BOUND
PROGRAM FILE WRITTEN : EXAM221
NUMBER PAM PAGES USED:      9
```

5) Listing of the control statements for the linkage editor. The PROGRAM statement specifies that no linkage editor listing (MAP) is to be generated. The object program is to be stored in file EXAM221. The INCLUDE * statement indicates that the contents of the EAM area are to be linked. The EAM area contains the modules generated by the preceding compiler run.

Caution

INCLUDE * causes all modules from the EAM area to be processed, including those which may have originated from other compilations of the same task. Therefore, the EAM area should be cleared by the /ERASE * command before another compiler run begins.

Load module output to SYSLST

```
15 23  AB   CD   EF    15 30 16 32 25 50

16 24  AC   CE   EG    16 31 17 33 26 51

17 25  AD   CF   EH    17 32 18 34 27 52

... END EXAMPLE 1
```

Note

This operating mode is not suited to interactive tasks. Although source lines can be supplied interactively to the PLI1 compiler, they cannot be retrieved again. For another compilation of the same or possibly corrected program, all entries would have to be repeated. The same applies to the data. An example of a typical interactive task is shown in section 2.2.2.

2.2.2 Interactive task: input/output to files via SYSFILE command

Task:

Same as example 2.2.1.

Prerequisite:

The source program is contained in the SOURCE222 file. The data for the PL/I file SYSIN are in INPUT222. Result output via SYSPRINT is to be sent to the OUTPUT222 file. No compiler listing is requested.

Runtime listing on SYSOUT

```
/REMARK ..................................................... TRANSLATE
/EXEC $PLI1
%  BLS0500 PROGRAM PLI1, VERSION 32A OF 84-07-26 LOADED.
          *COMOPT SOURCE=SOURCE222, LIST=NO
          *END

..... THERE WAS NO DIAGNOSTIC MESSAGE
..... OBJECTMODUL 'EXAM222'  GENERATED AND WRITTEN TO: *EAM
..... OBJECTMODUL 'EXAM2227' GENERATED AND WRITTEN TO: *EAM
END OF SIEMENS PLI1-COMPILER VERSION 4.0A , TIME USED:     1.76 SEC
/REMARK ....................................................... LINK
/EXEC $TSOSLNK
%  BLS0500 PROGRAM 'TSOSLNK', VERSION '21.0C40' OF '87-09-28' LOADED.
          PROGRAM EXAM222, MAP=NO, FILENAM=PROG.EXAM222
          INCLUDE *
          END
%  LNK0500 PROG BOUND
%  LNK0503 PROG FILE WRITTEN: PROG.EXAM222
%  LNK0504 NUMBER PAM PAGES USED:      9
/REMARK ..................................................... FILES
/SYSFILE SYSLST=OUTPUT222                                              1)
/SYSFILE SYSDTA=INPUT222
/REMARK ..................................................... EXECUTE
/EXEC PROG.EXAM222
%  BLS0500 PROGRAM 'EXAM222', VERSION ' ' OF '88-10-06' LOADED.
END OF PROGRAM EXAM222 , RTS 4.0A-AAA, TIME USED:     0.17 SEC
/REMARK ..................................................... END
```

The result of the PRINT command is the same output as for example 2.2.1.

Explanation of the run:

1) The PL/I program outputs to SYSPRINT.
   SYSPRINT is assigned to the system file SYSLST. However, as the result is to appear on a file rather than on a runtime listing, the SYSLST system file is assigned to the OUTPUT222 file via the SYSFILE command.

The program is read via SYSIN. By default, SYSIN is identical with SYSDTA; therefore, SYSDTA is reassigned to INPUT222 via the SYSFILE command.

Note on the storage of source programs and data:

The storage of source programs and input data in files permits the use of the file editing systems EDT and EDOR for any corrections to the program or data that may be necessary.

2.2.3 Batch processing: input/output via LINK

Task:

Structured fixed-length records are to be read from a file called STOCK223. The data are to be checked for the value of a quantity (MG). All records for which MG ≤ 1100 are to be restructured and written to the LIST223 file, which is then to be printed.

Prerequisite:

Batch task; the input file exists under the name of STOCK223 but must be chained using the LINK name CARD. The output file must be created. The source program and the control statements are contained in the input deck. All listings generated during the compilation are to be printed.

Input data in the STOCK223 file

Index Data record

```
10000000 11111SOAP            10001033000
20000000 12134PERFUME         01120200637
30000000 14667DISH-CLOTH      00510100123
50000000 15678PAPERBAGS       12450500021
60000000 74567NATRON          03060500173
70000000 90000POWDER          01340501073
80000000 91000PAPER           12450100714
```

Result of the run in output file LIST223

```
11111 SOAP           1000 10 33000
12134 PERFUME        0112 02 00637
14667 DISH-CLOTH     0051 01 00123
74567 NATRON         0306 05 00173
90000 POWDER         0134 05 01073
```

SYSOUT listing

```
/REMARK ..................................................... TRANSLATE
/EXEC $PLI1
%  BLS0500 PROGRAM PLI1, VERSION 32A OF 84-07-26 LOADED.
         *COMOPT LIST=ALL, MARGINS=T(2,60),
         *END
```

Source lines (omitted here; refer to compiler listing)

```
..... THERE WAS NO DIAGNOSTIC MESSAGE
..... OBJECTMODUL 'EXAM223'  GENERATED AND WRITTEN TO: *EAM
..... OBJECTMODUL 'EXAM2237' GENERATED AND WRITTEN TO: *EAM
END OF SIEMENS PLI1-COMPILER VERSION 4.0A , TIME USED:     1.81 SEC
/REMARK ..................................................... LINK
/EXEC $TSOSLNK
%  BLS0500 PROGRAM 'TSOSLNK', VERSION '21.0C40' OF '87-09-28' LOADED.
          PROGRAM EXAM223, MAP=NO, FILENAM=EXAM223
          INCLUDE *
          END
%  LNK0500 PROG BOUND
%  LNK0503 PROG FILE WRITTEN: EXAM223
%  LNK0504 NUMBER PAM PAGES USED:      7
/REMARK ..................................................... FILES
/FSTAT STOCK223,ALL                                                  1)
0000003 :B:$STSPL1.STOCK223
  FCBTYPE = SAM     VSNTYPE = PUB       LASTPG  = 0000001   2ND ALLO= 00006
  SHARE   = NO      ACCESS  = WRITE
  ACCESS# = 001     CRDATE  = 85-08-22  EXDATE  = 85-08-22  LADATE  = 85-08-22
  RDPASS  = NONE    WRPASS  = NONE      EXPASS  = NONE
  VERSION = 001     BACKUP# = 000       LARGE   = NO        BACKUP  = A
  DESTROY = NO      AUDIT   = NONE
  BLKTYPE = STD     BLKSIZE = 002048    RECFORM = (V,N)     RECSIZE = 00000
  VSN/DEV/EXT =     PUBB02/D3475/001
  EXTCNT  =   1
:B:  PUBLIC:      1 FILE.  RES=        3, FREE=        2, REL=        0 PAGES
/FILE STOCK223,LINK=CARD                                             2)
/FILE LIST223,LINK=LIST, FCBTYPE=SAM, RECFORM=F, RECSIZE=39, BLKSIZE=STD
/REMARK ..................................................... EXECUTE
/SETSW ON=(1)                                                        3)
/EXEC EXAM223
%  BLS0500 PROGRAM 'EXAM223', VERSION ' ' OF '88-10-06' LOADED.
          *RUNOPT LIST=(OPTIONS,SUMMARY), FORMAT=PRINTER(,120)
          *END
END OF PROGRAM EXAM223 , RTS 4.0A-AAA, TIME USED:     0.29 SEC
/REMARK ..................................................... END
```

Explanation of the run in example 2.2.3.

1) The FSTAT command was issued to show that the file has the required parameters.

2) LINK name declaration for the STOCK223 file. Creating the LIST223 file with the required characteristics.

3) The object program was supplied with control statements at initiation time. Therefore, task switch 1 must be set before the call.

Control statements effective at compile time, on SYSLST (*COMOPT LIST = OPTIONS)

```
COMPILER-OPTIONS USED


  STORAGE = ( STACK(16,4), AREA(16,16,975))
  LIST    = ( ESD, NOTERMINAL, SUMMARY, OPTIONS, NOSAVLST, MAP, NEST, IREF,
            FULLXREF, EXPAND, NOINSOURCE, AGGREGATE, OFFSET, ASSM,
            OUTTEXT, NOLINECNT )
  FORMAT  = ( TERMINAL(0,80), PRINTER(64,132), ENGLISH )
  MESSAGE = NOSYSLST
  SOURCE  = EXAM223
  MARGINS = ( TEXT(2,60), PAD, NOLINID, NOASACNTRL, GAMKEY(0,0), CHAR60,
            NOSAVMAC)
  DIAGNOST= ( NOTERMINAL, NOSAVLST, WARNING )
  COMLIB  = NO
  OBJECT  = ( ERROR(32767), ABORT(500), OUT )
  OPTIONS = ( NOISO, NOMAIN, NOINTERRUPT, NOMACRO, NOXS, NOBITPTR )
  OPTIMIZE= ( NOTIME, NOOVERLAP, NOENABLING, NOREORDER )
  DEBUG   = ( NOSTMT, NOPROCTRACE, NOLABTRACE, NOCALLTRACE, NOGOTOTRACE,
            NORETURNTRACE, NOBREAKPOINT )
  SYMTEST = MAP
  MODULE  = *
```

Source listing on SYSLST (*COMOPT LIST = EXPAND)

```
 1      EXAM223: PROCEDURE OPTIONS(MAIN);
 2
 3               DCL CARD           FILE RECORD INPUT,
 4                   LIST           FILE RECORD OUTPUT;
 5               DCL 1 IN,
 6                     2 NO         CHAR(5),
 7                     2 BZ         CHAR(15),
 8                     2 MG         CHAR(4),
 9                     2 ME         CHAR(2),
10                     2 PR         CHAR(5);
11               DCL 1 OUT,
12                     2 NO         CHAR(5),
13                     2 FILL1      CHAR(2)   INIT (’  ’),
14                     2 BZ         CHAR(15),
15                     2 FILL2      CHAR(2)   INIT (’  ’),
16                     2 AMG        CHAR(4),
17                     2 FILL3      CHAR(2)   INIT (’  ’),
18                     2 AME        CHAR(2),
19                     2 FILL4      CHAR(2)   INIT (’  ’),
20                     2 APR        CHAR(5);
21
22               OPEN FILE(CARD), FILE(LIST);
23               ON ENDFILE(CARD) GOTO END;
24
25           A1: READ FILE(CARD) INTO(IN);
26               IF IN.MG <= ’1100’ THEN DO;
27   1              OUT.NO  = IN.NO;
28   1              OUT.BZ  = IN.BZ;
29   1              AMG     = IN.MG;
```

```
30   1              OUT.AME = ME;
31   1              APR     = IN.PR;
32   1              WRITE FILE(LIST) FROM(OUT);
33   1              END;
34                GOTO A1;
35
36                END: END;
```

Listing of external names in the code module on SYSLST (*COMOPT LIST = ESD)

```
LIST OF EXTERNAL NAMES IN THE CODE MODULE

  NAME      TYPE    MA    ADDR    L/ESID

EXAM223   CSECT    04   000000   00041C
EXAM2237  EXTERN   00   000000   404040
P$3#20##  EXTERN   00   000000   404040
P$RECIO#  EXTERN   00   000000   404040
P$OPNEX#  EXTERN   00   000000   404040
CARD      COMMON   00   000000   000200
LIST      COMMON   00   000000   000200
P$START#  ENTRY    00   000078   000001
```

Object code listing on SYSLST (detail) (*COMOPT LIST = ASSM)

```
          000260 05 EF                BALR  14,15
26        000262 D2 03 D04C D050      MVC   76(4,13),80(13)
          000268 D5 03 D25C B13C      CLC   604(4,13),316(11)     MG,
          00026E 47 20 A2C2           BC    2,706(0,10)           C.1
27        000272 D2 04 D267 D248      MVC   615(5,13),584(13)     NO, NO
28        000278 D2 0E D26E D24D      MVC   622(15,13),589(13)    BZ, BZ
29        00027E D2 03 D27F D25C      MVC   639(4,13),604(13)     AMG, MG
30        000284 D2 01 D285 D260      MVC   645(2,13),608(13)     AME, ME
31        00028A D2 04 D289 D262      MVC   649(5,13),610(13)     APR, PR
32        000290 58 60 B038           L     6,56(0,11)            LIST
          000294 50 60 D0A0           ST    6,160(0,13)
```

Listing of external names in the STATIC module on SYSLST (*COMOPT LIST = ESD)

```
LIST OF EXTERNAL NAMES IN THE STATIC MODULE

  NAME      TYPE    MA    ADDR    L/ESID

EXAM2237  CSECT    00   000000   000008
EXAM223   EXTERN   00   000000   404040
CARD      COMMON   00   000000   000200
LIST      COMMON   00   000000   000200
```

Storage map listing on SYSLST (*COMOPT LIST = MAP)

```
                M A P - L I S T

SOURCE-REF.     TYPE            ADDR   OFFSET    NAME


          0     ROOTBLOCK

          1     ENTRY CONST        0             EXAM223



          1     EXT PROCEDURE                    EXAM223

          3     ESD #              3             CARD
          4     ESD #              4             LIST
          5     AUTO             248             IN
                  MEMBER                  0       NO      IN IN
                  MEMBER                  5       BZ      IN IN
                  MEMBER                 14       MG      IN IN
                  MEMBER                 18       ME      IN IN
                  MEMBER                 1A       PR      IN IN
         11     AUTO             267             OUT
                  MEMBER                  0       NO      IN OUT
                  MEMBER                  5       FILL1     IN OUT
                  MEMBER                  7       BZ      IN OUT
                  MEMBER                 16       FILL2     IN OUT
                  MEMBER                 18       AMG     IN OUT
                  MEMBER                 1C       FILL3     IN OUT
                  MEMBER                 1E       AME     IN OUT
                  MEMBER                 20       FILL4     IN OUT
                  MEMBER                 22       APR     IN OUT
         25     LABEL CONST      236             A1
         36     LABEL CONST      2C6             END



         23     ON UNIT                          ENDFILE



 * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Listing of the structure length table on SYSLST (*COMOPT LIST = AGGREGATE)

```
                                S T R U C T U R E   L E N G T H   T A B L E
                                                        OFFSET          ELEMENT LENGTH          TOTAL LENGTH
SOURCE-REF   LEVEL IDENTIFIER                DIMENSION   DEC    HEXDEC     DEC    HEXDEC        DEC    HEXDEC

     5     1 IN                                          0       0                                 31       1F
             2 NO                                        0       0          5       5
             2 BZ                                        5       5         15       F
             2 MG                                       20      14          4       4
             2 ME                                       24      18          2       2
             2 PR                                       26      1A          5       5

    11     1 OUT                                         0       0                                 39       27
             2 NO                                        0       0          5       5
             2 FILL1                                     5       5          2       2
             2 BZ                                        7       7         15       F
             2 FILL2                                    22      16          2       2
             2 AMG                                      24      18          4       4
             2 FILL3                                    28      1C          2       2
             2 AME                                      30      1E          2       2
             2 FILL4                                    32      20          2       2
             2 APR                                      34      22          5       5


********** END OF STRUCTURE LENGTH TABLE **********************************************************
```

Listing of the cross-reference table on SYSLST (*COMOPT LIST = FULLXREF)

```
                C R O S S - R E F E R E N C E - T A B L E     -   REFERENCED IDENTIFIERS   -
IDENTIFIER      DIMENSION DATATYPE                              STORAGE            REFERENCES

AME                       CHAR (2)                  UNAL     MEM-2 (OUT)         DCL 18 30
AMG                       CHAR (4)                  UNAL     MEM-2 (OUT)         DCL 16 29
APR                       CHAR (5)                  UNAL     MEM-2 (OUT)         DCL 20 31
A1                        LABEL                              CONSTANT            DCL 25 34
BZ                        CHAR (15)                 UNAL     MEM-2 (OUT)         DCL 14 28
BZ                        CHAR (15)                 UNAL     MEM-2 (IN)          DCL 7 28
CARD                      FILE INPUT RECORD              EXT CONSTANT            DCL 3 22 23 25
END                       LABEL                              CONSTANT            DCL 36 23
ENDFILE                   CONDITION                                              +++ 23
EXAM223                   ENTRY                          EXT CONSTANT            DCL 1
FILL1                     CHAR (2)                  UNAL     MEM-2 (OUT)         DCL INIT 13 11
FILL2                     CHAR (2)                  UNAL     MEM-2 (OUT)         DCL INIT 15 11
FILL3                     CHAR (2)                  UNAL     MEM-2 (OUT)         DCL INIT 17 11
FILL4                     CHAR (2)                  UNAL     MEM-2 (OUT)         DCL INIT 19 11
IN                        STRUCTURE                 UNAL     AUTOMATIC           DCL 5 25
LIST                      FILE OUTPUT RECORD             EXT CONSTANT            DCL 4 22 32
ME                        CHAR (2)                  UNAL     MEM-2 (IN)          DCL 9 30
MG                        CHAR (4)                  UNAL     MEM-2 (IN)          DCL 8 26 29
NO                        CHAR (5)                  UNAL     MEM-2 (IN)          DCL 6 27
NO                        CHAR (5)                  UNAL     MEM-2 (OUT)         DCL 12 27
OUT                       STRUCTURE                 UNAL     AUTOMATIC           DCL 11 32
PR                        CHAR (5)                  UNAL     MEM-2 (IN)          DCL 10 31
..... THERE ARE  N O  IDENTIFIERS  N O T  REFERENCED .....................................

********** END OF CROSS-REFERENCE-TABLE ***************************************************
```

Compiler program statistics on SYSLST (*COMOPT LIST = SUMMARY)

```
SUMMARY OF VIRTUAL MAIN STORAGE REQUIREMENTS
PROCEDURE STACK:    20 PAGES; SYSTEM CALLS:     2 REQM,     0 RELM
STANDARD AREA:      21 PAGES; SYSTEM CALLS:     2 REQM
ASSIGNED MEMORY:   237 PAGES CLASS 6   AND:    66 PAGES CLASS 5
```

Linkage editor map on SYSLST

```
PROGRAM EXAM223, MAP=NO, FILENAM=EXAM223
INCLUDE *
END
PROG BOUND
PROGRAM FILE WRITTEN : EXAM223
NUMBER PAM PAGES USED:      7
```

Control statement effective at object run time, on SYSLST
(*RUNOPT LIST = SUMMARY)

```
RUNTIME-OPTIONS USED


 STORAGE = ( STACK(16,4), AREA(16,16,1071))
 LIST    = ( NOTERMINAL, SUMMARY, OPTIONS )
 FORMAT  = ( TERMINAL(0,80), PRINTER(64,120), ENGLISH )
 ARGUMENT= ''
 MESSAGE = NOSYSLST
 SYSFILE = ( SYSDTA(SYSIN), SYSLST(SYSPRINT), SYSOUT(SYSOUT),
           LINESIZE(0,0,0), PAGESIZE(0,0), DISPLAY(SYSDTA))
 DUMP    = ( COND, NOSNAP, NOAREA, NOSTACK, NORANGE )
 TRACE   = ( TERMINAL, NOLABTRACE, NOPROCTRACE, NORETURNTRACE,
           NOGOTOTRACE, NOCALLTRACE, NOIFTRACE )
 TABULATOR = (1,11,21,31,41,51,61,71,81,91,101,111,121)
 CONTROL = NOALIGN
 ACTIVE  = YES
```

Object run statistics on SYSLST (*RUNOPT LIST = SUMMARY)

```
SUMMARY OF VIRTUAL MAIN STORAGE REQUIREMENTS
PROCEDURE STACK:    20 PAGES; SYSTEM CALLS:     2 REQM,     0 RELM
STANDARD AREA:       1 PAGES; SYSTEM CALLS:     1 REQM
ASSIGNED MEMORY:   101 PAGES CLASS 6   AND:    57 PAGES CLASS 5
```

2.2.4 Use of libraries

Task:

To demonstrate operation with a user library.
First, the procedures AVG (determination of average) and AVGDEV (mean deviation from average) are compiled and entered in the PLIBIB library.
This is followed by the compilation of a main procedure which requires the above functions. The functions are obtained from the library at link-edit time. The main procedure reads in data and calculates their average and mean deviation from average.

Prerequisite:

The three procedures AVGDEV224, AVG224 and EXMPL4 are contained in files called AVGDEV224, AVG224, and SOURCE224.

Each procedure is compiled with the following options:

```
*COMOPT SOURCE = file, LIST = (NO, SOURCE)
```

The input data are entered via SYSIN, immediately following the /EXEC EXAM224 command in the deck. Result output is to SYSLST.

Explanation of the run in example 2.2.4 (next page)

1) Erase the EAM area if it still contains object modules from previous compiler runs.
2) Copy the object modules of previous compiler runs from the EAM area into a library, using the LMR utility (see section 3.6.4). For the control of the utility, see [11].
3) Erase the contents of the EAM area, thus simulating a condition as if the procedures AVG and AVGDEV had been compiled and entered in the library in a previous task.

Listing on SYSOUT

```
/ERASE *                                                                    1)
/REMARK ..................................... TRANSLATE FOR LIBRARY
/EXEC $PLI1
%  BLS0500 PROGRAM 'PLI1', VERSION '4.0A' OF '88-10-03' LOADED.
          *COMOPT SOURCE=AVGDEV224,
          *COMOPT LIST=(NO,SOURCE)
          *END

..... THERE WAS NO DIAGNOSTIC MESSAGE
..... OBJECTMODUL 'AVGDEV' GENERATED AND WRITTEN TO: *EAM
END OF SIEMENS PLI1-COMPILER VERSION 4.0A , TIME USED:     1.39 SEC
/EXEC $PLI1
%  BLS0500 PROGRAM 'PLI1', VERSION '4.0A' OF '88-10-03' LOADED.
          *COMOPT SOURCE=AVG224,
          *COMOPT LIST=(NO,SOURCE)
          *END

..... THERE WAS NO DIAGNOSTIC MESSAGE
..... OBJECTMODUL 'AVG' GENERATED AND WRITTEN TO: *EAM
END OF SIEMENS PLI1-COMPILER VERSION '4.0A' , TIME USED:     2.74 SEC
/REMARK ..................................... INCLUDE IN LIBRARY
/EXEC LMR                                                                   2)
%  BLS0500 PROGRAM LMR.266, VERSION 266 OF 83-03-11 LOADED.
          MODLIB MODLIB224
          COPYALL SOURCE=*
          END
LMR (BS2000) VERSION V26.6506
LMR (BS2000) VERSION V26.6506 NORMAL END
/STEP
/ERASE *                                                                    3)
/REMARK ..................................................... TRANSLATE
/EXEC $PLI1
%  BLS0500 PROGRAM 'PLI1', VERSION '4.0A' OF '88-10-03' LOADED.
          *COMOPT SOURCE=SOURCE224,
          *COMOPT LIST=(NO,SOURCE)
          *END

----- THERE WAS NO DIAGNOSTIC MESSAGE
----- OBJECTMODUL 'EXAM224'  GENERATED AND WRITTEN TO: *EAM
----- OBJECTMODUL 'EXAM2247' GENERATED AND WRITTEN TO: *EAM
END OF SIEMENS PLI1-COMPILER VERSION 4.0A , TIME USED:     2.91 SEC
/REMARK ..................................................... LINK
/EXEC $TSOSLNK
%  BLS0500 PROGRAM 'TSOSLNK', VERSION '21.0C40' OF '87-09-28' LOADED.
          PROGRAM EXEM224, MAP=NO, FILENAM=EXAM224
          INCLUDE *
          RESOLVE, MODLIB224
          END
%  LNK0500 PROG BOUND
%  LNK0503 PROG FILE WRITTEN: EXAM224
%  LNK0504 NUMBER PAM PAGES USED:      8
/REMARK ..................................................... EXECUTE
/EXEC EXAM224
%  BLS0500 PROGRAM 'EXAM224', VERSION ' ' OF '88-10-06' LOADED.
8 3 5 6 4 2 7 1 9
END OF PROGRAM EXAM224 , RTS 4.0A-AAA, TIME USED:     0.16 SEC
/REMARK ..................................................... END
```

Source procedure AVGDEV (function) on SYSLST (*COMOPT LIST = SOURCE)

```
 1       AVGDEV: PROCEDURE (N, AVERAGE, X) RETURNS (FLOAT BIN);
 2
 3               DCL X                          DIM(*);
 4               DCL (AVERAGE,SUM)              FLOAT BIN;
 5               DCL (N,K)                      FIXED BIN;
 6
 7               SUM = 0;
 8               DO K = 1 TO N;
 9   1              SUM = SUM + ABS(AVERAGE - X(K));
10   1              END;
11               RETURN (SUM/N);
12               END;
```

Source procedure AVG (function) on SYSLST (*COMOPT LIST = SOURCE)

```
 1      AVG: PROCEDURE (N, X) RETURNS (FLOAT BIN);
 2
 3           DCL X                     DIM(*);
 4           DCL SUM                   FLOAT BIN;
 5           DCL (N, K)                FIXED BIN;
 6
 7           SUM = 0;
 8           DO K = 1 TO N;
 9   1          SUM = SUM + X(K);
10   1          END;
11           RETURN (SUM/N);
12           END;
```

Modules entered in library (LMR listing) on SYSLST

```
PROGRAM  LMR/26:65 STARTED

MODLIB MODLIB224

COPYALL SOURCE = *

END
```

Source procedure EXAM224 (main procedure) on SYSLST (*COMOPT LIST = SOURCE)

```
 1      EXAM224: PROCEDURE OPTIONS(MAIN);
 2
 3               DCL X                   DIM(1000);
 4               DCL (N,K)               FIXED BIN;
 5               DCL (AVERAGE,AVDEV)     FLOAT BIN;
 6               DCL (AVG, AVGDEV)       EXTERNAL ENTRY RETURNS(FLOAT BIN);
 7               DCL (SYSIN,SYSPRINT)    FILE INTERNAL;
 8
 9               ON ENDFILE(SYSIN) GOTO END;
10          S10: GET LIST(N);
11               PUT SKIP LIST(N);
12               DO K = 1 TO N;
13   1              GET LIST(X(K));
14   1              PUT SKIP LIST(K, X(K));
15   1              END;
16               AVERAGE = AVG(N,X);
17               AVDEV = AVGDEV(N,AVERAGE,X);
18               PUT PAGE LIST('     AVERAGE', 'AVERAGE DEVIATION');
19               PUT SKIP LIST(AVERAGE, AVDEV);
20               GO TO S10;
21
22          END: PUT SKIP LIST('——END EXAMPLE 224');
23               END;
```

Linkage editor map on SYSLST

```
PROGRAM EXAM224,MAP=NO,FILENAM=EXAM224
INCLUDE *
RESOLVE, MODLIB224 ————————————————————————— Resolve the main
END                                           procedure references
PROG BOUND                                    to AVG and AVGDEV by
PROGRAM FILE WRITTEN : EXAM224                referring to the
NUMBER PAM PAGES USED:      8                 library.
```

Object run result on SYSLST

```
       8
       1  3.00000E+00
       2  5.00000E+00
       3  6.00000E+00
       4  4.00000E+00
       5  2.00000E+00
       6  7.00000E+00
       7  1.00000E+00
       8  9.00000E+00
———.———.———.———.———.———.———.———.———.———.
    AVERAGE        AVERAGE DEVIATION
4.625000E+00        2.125000E+00                    new page
---END EXAMPLE 224
```

2.2.5 Use of debugging aids

Task:

The use of the TRACE debugging aid is to be shown. This example uses two internal procedures. UP1 lists all the elements of a two-dimensional array supplied as a parameter. UP2 does the same for one-dimensional arrays. At the beginning of the program, the elements of array A are assigned the values 1-9. The elements of array B are assigned the values 10-19.
The effect of the procedures is to print the arrays or parts of them.

Prerequisite:

The program requires no input; its output is to SYSPRINT. The program source is read in via SYSDTA. To follow the flow of the program, activate the procedure trace facility.

Runtime listing on SYSOUT

```
/ERASE *
/REMARK ..................................................... TRANSLATE
/EXEC $PLI1
%  BLS0500 PROGRAM 'PLI1', VERSION '4.0A' OF '88-10-03' LOADED.
          *COMOPT DEBUG=(PROCTRACE,LABTRACE),
          *COMOPT MARGINS=T(1,50)
          *END
```

Source lines (omitted here; refer to compiler listing)

```
..... THERE WAS NO DIAGNOSTIC MESSAGE
..... OBJECTMODUL 'EXAM225'  GENERATED AND WRITTEN TO: *EAM
..... OBJECTMODUL 'EXAM2257' GENERATED AND WRITTEN TO: *EAM
END OF SIEMENS PLI1-COMPILER VERSION 4.0A , TIME USED:     1.70 SEC
/REMARK ..................................................... LINK
/EXEC $TSOSLNK
%  BLS0500 PROGRAM 'TSOSLNK', VERSION '21.0C40' OF '87-09-28' LOADED.
          PROGRAM EXAM225, FILENAM=EXAM225
          INCLUDE *
          END
%  LNK0500 PROG BOUND
%  LNK0503 PROG FILE WRITTEN: EXAM225
%  LNK0504 NUMBER PAM PAGES USED:      7
/REMARK ..................................................... EXECUTE
/SETSW ON=(1)
/EXEC EXAM225
%  BLS0500 PROGRAM 'EXAM225', VERSION ' ' OF '88-10-06' LOADED.
          *RUNOPT TRACE=PROCTRACE,
          *END
END OF PROGRAM EXAM225 , RTS 4.0A-AAA, TIME USED:     0.16 SEC
/REMARK ..................................................... END
```

Source listing (*COMOPT LIST = SOURCE)

```
 1      EXAM225: PROC OPTIONS(MAIN);
 2
 3               DCL A              DIM(3,3) BIN FIXED,
 4                   B              DIM(3,3) BIN FIXED,
 5                   C              DIM(3)   BIN FIXED;
 6               DCL SYSPRINT       FILE INTERNAL;
 7               DCL (K,I,J)        FIXED BIN;
 8
 9               K = 0;
10               DO I = 1 TO 3;
11   1              DO J = 1 TO 3;
12   2                 K      = K+1;
13   2                 A(I,J) = K;
14   2                 B(I,J) = K+9;
15   2                 END;
16   1              END;
17                CALL UP1(A);
18                C = B(1,*);
19                PUT SKIP(3) LIST(C);
20                CALL UP2(C);
21                CALL UP2(B(1,*));
22
23          UP1: PROC(A1);
24   1            DCL A1(*,*) BIN FIXED;
25   1            PUT SKIP(1) LIST(A1);
26   1            END;
27
28          UP2: PROC(B1);
29   1            DCL B1 DIM(*) BIN FIXED;
30   1            PUT SKIP(1) LIST(B1);
31   1            END;
32
33          END;
```

Control statements effective at compile time (*COMOPT LIST = OPTIONS)

```
STORAGE = ( STACK(16,4), AREA(16,16,975))
LIST    = ( NOESD, NOTERMINAL, NOSUMMARY, OPTIONS, NOSAVLST, NOMAP, NEST,
          IREF, NOXREF, SOURCE, NOINSOURCE, NOAGGREGATE, OFFSET,
          NOASSM, NOOUTTEXT, NOLINECNT )
FORMAT  = ( TERMINAL(0,80), PRINTER(64,132), ENGLISH )
MESSAGE = NOSYSLST
SOURCE  = EXAM225
MARGINS = ( TEXT(1,50), PAD, NOLINID, NOASACNTRL, GAMKEY(0,0), CHAR60,
          NOSAVMAC )
DIAGNOST= ( NOTERMINAL, NOSAVLST, WARNING )
COMLIB  = NO
OBJECT  = ( ERROR(32767), ABORT(500), OUT )
OPTIONS = ( NOISO, NOMAIN, NOINTERRUPT, NOMACRO )
OPTIMIZE= ( NOTIME, NOOVERLAP, NOENABLING, NOREORDER )
DEBUG   = ( NOSTMT, PROCTRACE, LABTRACE, NOCALLTRACE, NOGOTOTRACE,
          NOIFTRACE, NORETURNTRACE, NOASSIGNTRACE, NOBREAKPOINT )
SYMTEST = NOAID
MODULE  = *
```

Offset listing (*COMOPT LIST = OFFSET)

LISTING OF THE OFFSETS OF THE CODE MODULE

SOURCE-REF.	ADDR	SOURCE-REF.	ADDR	SOURCE REF.	ADDR	SOURCE-REF.
1	000000	6	000098	9	0000EE	10
11	0000FC	12	000104	13	000110	14
15	00014E	16	000166	17	00017E	18
19	000194	20	000216	21	00022C	33
23	000240	25	0002A2	26	0003FE	28
30	000476	31	000554			

END ADDRESS: 000650

Linkage editor map

```
PROGRAM EXAM225, FILENAM=EXAM225
INCLUDE *
END
PROG BOUND
PROGRAM FILE WRITTEN : EXAM225
NUMBER PAM PAGES USED:      7
```

Linkage editor map

```
                                       BS2000  LINK EDITOR ... PROGRAM MAP
***************************************************************************************************************
**                                                                                                           **
**                                                                                                           **
**                   PROGRAM: EXAM225                                            SEGMENT: %ROOT              **
**                   FILE: EXAM225                                                                           **
**                                                                                                           **
**                                                                                                           **
***************************************************************************************************************
                     DEC                        HEX      DEC

 SEGMENT NUMBER:       1       LOAD ADDR.:     000000         0
 NO. OF MODULES:       3       SEGMENT LENGTH: 0035B4    13.748
 NO. OF ENTRY PTS.:  124

     MODNAME / DATE        LOAD           MODULE                NO.OF     BIND   / MODULE : CONTAINING OML
     COMNAME / COMMON     ADDRESS         LENGTH       TRAITS   ENTRYS    METHOD / COMMON : CONTAINING SEGMENT
                        HEX       DEC   HEX       DEC

     EXAM225 /------    000000       0  000684  1.668  RO            2     EXPLICIT/EAM OBJMOD FILE
     EXAM2257/------    001000   4.096  000208    520                1     EXPLICIT/EAM OBJMOD FILE
     ITP#AOD#/880816    002000   8.192  0015B4  5.556  RO          121     AUTOLINK/TASKLIB
```

Listing with trace and output data (*COMOPT DEBUG = (PROCTRACE, LABTRACE) *RUNOPT TRACE = PROCTRACE)

```
 *P: EXAM225  LEVEL:  0    R1=0003C170   R2=FFFEFFF8   R3=00000001   R4=00000004   R14=00000082   R15=6E000000 1)

 *P: UP1      LEVEL:  1    R1=0003F32C   R2=FFFEF624   R3=00000001   R4=00000004   R14=6E000184   R15=0000026C 2)

 *P: EXAM225  LEVEL:  0    R1=0003D170   R2=FFFEFFF8   R3=00000001   R4=00000004   R14=00000082   R15=6E000000

 *P: UP2      LEVEL:  1    R1=0004032C   R2=000005F0   R3=00000001   R4=00000004   R14=6E00018E   R15=00000240

          1          2          3          4          5          6          7          8          9                3)
         10         11         12

 *P: UP2      LEVEL:  0    R1=000401A4   R2=00000628   R3=000401A8   R4=0400800F   R14=4E00022C   R15=00000414 4)

         10         11         12                                                                               5)

 *P: UP2      LEVEL:  1    R1=0004033E   R2=00000628   R3=000401A8   R4=00000004   R14=6E00023C   R15=00000240 6)

         10         11         12
```

1) TRACE output on activation of the EXPL5 procedure by the system. Output for procedure trace:
   - Qualifier *P
   - Name of the called procedure
   - Depth of procedure nesting
   - Registers R1-R4 and R14-R15. The registers contain:
     R1-R4: Information about the first 4 arguments. May be references to the arguments or the actual values. For details, see chapter 7.
     R14: Address of the branch point (return address).
     R15: Address of the entry point to the procedure.
2) Invoke procedure UP1. Same output as for 1).
3) Result of the output in line 21 of the source program.
4) Same procedure trace as for 1), caused by calling UP2 from line 26 of the source program.
5) Output of result from line 30 of procedure UP2.
6) Same procedure trace as for 1), caused by calling UP2 from line 33.
7) Output of result from line 30 of procedure UP2.

2.2.6 File organization CONSECUTIVE, INDEXED and REGIONAL (1)

Task:

To show the use of various file organizations and access methods. The program uses CONSECUTIVE, INDEXED, and REGIONAL (1) files whose LINK names are SEQ, KEY, and DIR. First, 6 records in the format shown below are written into each of these files in sequential order:

```
LINE: n where 1 ≤ n ≤ 6
```

Then the program modifies those records.

Prerequisite:

All files have a fixed record length of 27 characters. The SEQ file is mapped to the SAM access method. The ISAM access method is used for KEY, and PAM is used for DIR. The example shows which attributes are used for each of the OPEN statements. The only meaningful way of examining the contents of the DIR file is by means of the DPAGE utility.

Listing on SYSOUT

```
/                  SYSFILE SYSDTA = (SYSCMD)
/ERASE *
/REMARK ..................................................... TRANSLATE
/EXEC $PLI1
%  BLS0500 PROGRAM 'PLI1', VERSION '4.0A' OF '88-10-03' LOADED.
         *COMOPT LIST = (NO,SOURCE)
         *END
```

Source procedure (omitted here; see source listing further below)

```
..... THERE WAS NO DIAGNOSTIC MESSAGE
..... OBJECTMODUL 'EXAM226'  GENERATED AND WRITTEN TO: *EAM
..... OBJECTMODUL 'EXAM2267' GENERATED AND WRITTEN TO: *EAM
END OF SIEMENS PLI1-COMPILER VERSION 4.0A , TIME USED:     2.01 SEC
/REMARK ..................................................... LINK
/EXEC $TSOSLNK
%  BLS0500 PROGRAM 'TSOSLNK', VERSION '21.0C40' OF '87-09-28' LOADED.
         PROGRAM EXAM226, MAP=NO, FILENAM=EXAM226
         INCLUDE *
         END
%  LNK0500 PROG BOUND
%  LNK0503 PROG FILE WRITTEN: EXAM226
%  LNK0504 NUMBER PAM PAGES USED:      8
/REMARK ..................................................... FILES
/FILE CONSE226,LINK=SEQ,FCBTYPE=SAM,RECFORM=F,RECSIZE=27,BLKSIZE=STD
/FILE INDEX226,LINK=KEY,FCBTYPE=ISAM,BLKSIZE=STD,SPACE=(3,3)
/FILE REG1226, LINK=DIR,FCBTYPE=PAM, BLKSIZE=STD
/REMARK ..................................................... EXECUTE
/EXEC EXAM226
%  BLS0500 PROGRAM 'EXAM226', VERSION ' ' OF '88-10-06' LOADED.
END OF PROGRAM EXAM226 , RTS 4.0A-AAA, TIME USED:     0.99 SEC
/REMARK ..................................................... END
/PRINT CONSE226, ERASE
%  SCP0810 SPOOLOUT OF :B:$STSPL1.CONSE226 ACCEPTED: TSN: 5708, PNAME: PL1
/PRINT INDEX226, ERASE
%  SCP0810 SPOOLOUT OF :B:$STSPL1.INDEX226 ACCEPTED: TSN: 5709, PNAME: PL1
 DPAGE       VER=V21.0A10    CR DATE=840531
ENTER DPAGE-COMMAND
      OPEN REG1226
FILE 'REG1226' OPENED.
      PRINT 2,1-216
      HALT
```

Source listing on SYSLST (*COMOPT LIST = SOURCE)

```
 1  EXAM226: PROCEDURE OPTIONS(MAIN);
 2
 3           DCL SEQ        FILE;
 4           DCL KEY        FILE  ENV(INDEXED,     F(27), KEYLOC(1),
 5                                                        KEYLENGTH(6));
 6           DCL DIR        FILE  ENV(REGIONAL(1), F(27));
 7           DCL SYSPRINT   FILE INTERNAL;
 8
 9           DCL KEY3       FIXED BIN(4)  INIT(3);
10           DCL KEY5       FIXED BIN(4)  INIT(5);
11           DCL I          FIXED BIN(4);
12           DCL Z          POINTER;
13           DCL SOURCE     CHAR(27) INIT(' ');
14           DCL RECORD     CHAR(27) BASED(Z);
15           DCL CHANGED    CHAR(27) INIT(' ');
16
17           OPEN FILE(SEQ) RECORD OUTPUT SEQL;
18           OPEN FILE(KEY) RECORD OUTPUT SEQL KEYED;
19           OPEN FILE(DIR) RECORD OUTPUT DIRECT KEYED;
20           DO I=1 TO 6;
21              PUT STRING(SOURCE) EDIT ('LINE: ',I) (X(7),A(6),F(2));
22              WRITE FILE(SEQ) FROM(SOURCE);
23              WRITE FILE(KEY) FROM(SOURCE) KEYFROM(I);        fill
24              WRITE FILE(DIR) FROM(SOURCE) KEYFROM(I);        files
25              END;
26           CLOSE FILE(KEY);
27           CLOSE FILE(DIR);
28           CLOSE FILE(SEQ);
29
30           OPEN FILE(SEQ) RECORD UPDATE SEQL;
31           READ FILE(SEQ) IGNORE(2);
32           READ FILE(SEQ) INTO(SOURCE);                    CONSECUTIVE
33                CHANGED = SUBSTR(SOURCE,1,16) || ' - CHANGED';file
34           REWRITE FILE(SEQ) FROM(CHANGED);
35           READ FILE(SEQ) SET(Z);
36                SUBSTR(Z->RECORD,17) = ' - CHANGED';
37           REWRITE FILE(SEQ);
38           CLOSE FILE(SEQ);
39
40           OPEN FILE(KEY) RECORD UPDATE SEQL;
41           READ FILE(KEY) IGNORE(2);
42           READ FILE(KEY) INTO(SOURCE);
43                CHANGED =  SUBSTR(SOURCE,1,16) || ' - CHANGED';
44           REWRITE FILE(KEY) FROM(CHANGED);                INDEXED
45           CLOSE FILE(KEY);                                file
46
47           OPEN FILE(KEY) RECORD UPDATE DIRECT KEYED;
48                CHANGED = (6)' ' || ' - CHANGED';
49           REWRITE FILE(KEY) FROM(CHANGED) KEY(KEY5);
50           CLOSE FILE(KEY);
51
52           OPEN FILE(DIR) RECORD UPDATE DIRECT KEYED;
53                CHANGED = ' - CHANGED';                    REGIONAL(1)
54           REWRITE FILE(DIR) FROM(CHANGED) KEY(KEY3);      file
55           CLOSE FILE(DIR);
56
```

```
57            OPEN FILE(DIR) RECORD SEQL UPDATE;
58            READ FILE(DIR) IGNORE(4);
59            READ FILE(DIR) SET(Z);
60                 SUBSTR(Z->RECORD,17) = ' - CHANGED';
61            REWRITE FILE(DIR);
62            CLOSE FILE(DIR);
63            END;
```

Result of the run in the CONSE226 file

```
      LINE:  1
      LINE:  2
      LINE:  3  - CHANGED
      LINE:  4  - CHANGED
      LINE:  5
      LINE:  6
```

Result of the run in the INDEX226 file

```
1   LINE:  1
2   LINE:  2
3   LINE:  3  - CHANGED
4   LINE:  4
5   - CHANGED
6   LINE:  6
```

Result of the run in the REG1226 file, listed by $DPAGE (8 x 27 bytes)

```
LOGICAL PAGE = 1

KEY — (0001)  (000) BB5AEAB0 01000001 00000000 00000000                                                .!..............

(0001)  (000) 001B0000 00000003 4040D9C5 C7F1F2F2  (010) F6404040 40404040 40404040 40404040   ........  REG1226
(0033)  (020) 00000000 00000000 00000000 00000000  (030) 00000000 00000000 00000000 00000000   ................................
          NEXT 62 LINES ARE IDENTICAL TO ABOVE LINE.

LOGICAL PAGE = 2

KEY — (0001)  (000) BB5AEAB0 01000002 00000000 00000000                                                .!..............

(0001)  (000) FF000000 00000000 00000000 00000000  (010) 00000000 00000000 00000040 40404040   -.......................
(0033)  (020) 4040D3C9 D5C57A40 40F14040 40404040  (030) 40404040 40404040 40404040 40D3C9D5     LINE:  1                  LIN
(0065)  (040) C57A4040 F2404040 40404040 40404040  (050) 40406040 C3C8C1D5 C7C5C440 40404040   E:  2             - CHANGED
(0097)  (060) 40404040 40404040 40404040 40404040  (070) 404040D3 C9D5C57A 4040F440 406040C3                      LINE:  4  - C
(0129)  (080) C8C1D5C7 C5C44040 40404040 4040D3C9  (090) D5C57A40 40F54040 40404040 40404040   HANGED        LINE:  5
(0161)  (0A0) 40404040 40404040 40D3C9D5 C57A4040  (0B0) F6404040 40404040 40404040 40FF0000            LINE:  6            -..
(0193)  (0C0) 00000000 00000000 00000000 00000000  (0D0) 00000000 00000000 FF000000 00000000   .......................-.......
```

Dummy records 0 and 7 can be identified by ’FF’B4.

3 Compiling a PL/I source program

3.1 Functions of the PLI1 compiler

By means of the PLI1 compiler, PL/I source programs are checked for compliance with syntactic and semantic rules and compiled into object modules. Control statements, entered by the compiler before the source program is processed, are responsible for controlling the compile operation, i.e. locating the source program, selecting the listings to be output, etc.

The user must be aware that the PLI1 compiler does not evaluate the /PARAM command.

The PLI1 compiler is started with the EXECUTE command and begins by reading in the control statements from SYSDTA. Then it enters the source program, either from SYSDTA or from a file specified in the control statements. INCLUDE texts from files, GAM files, or macro libraries can be inserted in the source program by the compiler. A number of libraries or GAM files may be specified, with the ability to define a search sequence for the texts to be inserted.

The source program can be listed on SYSLST - complete with inserted INCLUDE texts if required. If the PL/I compiler finds errors in the source program, they are reported on SYSLST. Depending on the error type, the messages distinguish between warnings, minor errors, and severe errors. If a severe error occurs, compilation is terminated before code generation. Compilation may be terminated on the occurrence of a minor error, a warning, or in any of the three cases as desired.

In addition to the source program listing, the following listings can be generated:

- Enabled control statements
- Storage map
- Reference listing
- OFFSET listing
- INCLUDE references
- Object code listing in assembler format
- Storage occupancy statistics

The listings and/or messages can also be sent to a special file (SAVLST option in the LIST or DIAGNOST control statements).
The object code generated is entered into the EAM area of the object module file if compilation was not terminated abnormally. The compiler ends its run by outputting an end message to SYSOUT.
If the compiler run is terminated as the result of an error, all subsequent commands up to STEP or LOGOFF are ignored.

The PLI1 compiler is not yet provided with the XS capability:
it cannot run in the address space above 16 MB.

3.2 Invoking the PLI1 compiler

3.2.1 Invoking the compiler by ISP command

he PLI1 compiler is invoked, i.e. started, by the EXECUTE command, as follows:

```
/{EXECUTE }
 {        }$PLI1[,MONJV=jvname]
 {EXEC    }
```

Further parameters may be specified after the word $PLI1, as explained in the manual "Control System Command Language" [2]. Their use is not mandatory.

3.2.2 Invoking the compiler by SDF command

The PLI1 compiler can alternatively be started by the SDF command

```
/START-PLI1-COMPILER
```

Additional operands are available for this command. The operands and their meaning are described in section 3.12 at the end of this chapter.

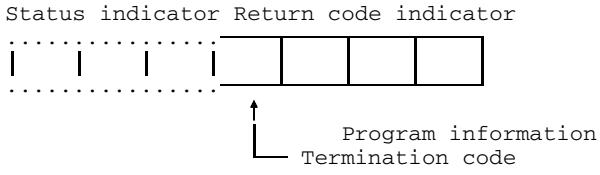
Input of SDF commands and their operands in guided and unguided dialog is described in detail in the manual "Introductory Guide to the SDF Dialog Interface".

3.2.3 Monitoring by monitoring job variable

For monitoring the compiler run, a monitoring job variable can be specified by entering 'MONJV=jname' in the EXECUTE command. The status indicator (digits 1 through 3) of the job variable is set by the system; it may have the following values:

```
'$R␣' = running
'$T␣' = normally terminated
'$A␣' = abnormally terminated.
```

The return code indicator (digits 4 through 7) is set at end of compiler run; it has the following format:



Program information is given the value of the highest error weight that occurred, in format PIC'999', where the following values are allocated to the error weights:

0 No diagnostic message
1 Information messages were issued
2 Warnings were issued
3 Errors occurred
4 Severe errors occurred
5 A fatal error occurred
6 A fatal compiler error occurred

The termination code is set, depending on the type of termination and the highest error weight that occurred. The type of termination can be influenced by the control specification *COMOPT OBJECT.

Allocation is as follows:

Type of termination \ Highest error weight	0	1	2	3	4	5	6	
Normal ($T)	0	1	1	1	-	-	-	Termination code
Abnormal ($A)	-	-	2[1)]	2[1)]	2	2	3	

1) Depending on the control statement *COMOPT OBJECT

This feature presupposes software product JV.

3.2.4 Message text file

For outputting all the messages, the compiler requires either of the following files:

```
$TSOS.PLI1.TEXT.D    for German
$TSOS.PLI1.TEXT.E    for English
```

If these files exist under different names or user IDs, then one of them must be made available by

```
/FILE file,LINK=TEXTLINK
```

in which case the language control feature (German or English) is ineffective.

3.2.5 Examples

The following examples of compiler calls illustrate various forms of compiler control. The control statements used depend on whether processing is in interactive or batch mode and where the source program and the control statements are located.

The examples only concern the compiler call environment. BS2000 often uses procedures to perform the compile/link-edit/execute sequence, an example of which can be found in chapter 2.
The examples 1-3 illustrate the standard methods for compilation in batch mode. It is of no importance whether the commands are entered via a card deck or from a file which is activated by the /ENTER command.

Example 1

The control statement and source program are supplied in the form of a card deck. Compilation is only to be performed up to and incl. the semantics run (no object module will be generated) and only errors are to be reported (warnings will be suppressed).

```
   .
   .
   .
/EXEC $PLI1
* COMOPT DIAGNOST=E,OBJECT=SE
* END
  source program lines
   .
   .
   .
```

Example 2

The source program exists in the SOURCE file. No control statements are required.

```
   .
   .
   .
/SYSFILE SYSDTA=SOURCE
/EXEC $PLI1
/SYSFILE SYSDTA=(SYSCMD)
   .
   .
   .
```

Note

Since the first line in SYSDTA (SOURCE file) does not begin with either *COMOPT or *END, the compiler knows that no control statements are specified.

Example 3

The source program contains INCLUDE statements in the %INCLUDE MACRO1 format. The INCLUDE texts are contained in libraries LIBA, LIBB, and LIBC and are to appear in the source listing.

```
   .
   .
   .
/EXEC $PLI1
* COMOPT COMLIB=(LIBA,LIBB,LIBC)
* COMOPT LIST=EX
* END
  source program
   .
   .
   .
```

Note

The name specified for %INCLUDE is interpreted as an element (member) name. LIBA, CBB, and LIBC refer to GAM (see section 3.4 and [4]) or MLU (see section 3.4 and [3]) files which are searched in the specified order until the element is found. Examples 4 through 7 apply mainly to interactive mode.

Example 4

The source program exists in a file A. Control statements are required.

```
   .
   .
   .
/EXEC $PLI1
* COMOPT SOURCE=A,MARGINS=(T(1,72)) ┐ entry
                                    } as
* END                               ┘ prompted
   .
   .
   .
```

Example 5

The source program is in the SOURCE file, the control statements are in the CONTROL file.

```
a) .
   .
   .
/SYSFILE SYSDTA=CONTROL
/EXEC $PLI1
/SYSFILE SYSDTA=(SYSCMD)
   .
   .
   .

b) .
   .
   .
/SYSFILE SYSDTA=CONTROL
/EXEC $PLI1
-Breakpoint (prompt)
/SYSFILE SYSDTA=SOURCE
/RESUME
   .
   .
   .
```

CONTROL file:

```
*COMOPT SOURCE = SOURCE

*END
```

CONTROL file:

```
*COMOPT OBJECT = E (5)

*END/
```

Example 6

The source program and the control statements are entered via terminal.

```
   .
   .
   .
/EXEC $PLI1
*COMOPT LIST=MAP,OPTIONS=MAIN                          ⎫
*END                                                   ⎪
EXAMPLE:   PROC;                                       ⎪
           DCL SYSPRINT FILE;                          ⎬ enter as
           PUT SKIP (2) LIST ('——EXAMPLE 6——');        ⎪ prompted
           END;                                        ⎪
   .                                                   ⎭
   .
   .
```

Example 7

The source program exists in the SOURCE file, the control statements are entered via terminal.

```
   .
   .
   .
/EXEC $PLI1
*COMOPT LIST=FULLXREF                    ┐
                                         } according to input request
*END/                                    ┘

-Breakpoint (prompt)

/SYSFILE SYSDTA=SOURCE                   ┐
/RESUME                                  │ alternative to
   .                                     } COMOPT SOURCE =
   .                                     │ SOURCE
   .                                     ┘
```

3.3 Controlling the PLI1 compiler

To select its functions, the PLI1 compiler requires options which must be supplied in the form of control statements. To avoid unnecessary programming effort, defaults (preset options) are provided, which become effective in the order shown below:

1. PLI1 Default
   The defaults specified for each control statement are evaluated first.
2. Computing Center Default
   If there is a file by the name of $TSOS.PLI1.OPTIONS, the control statements contained in that file are evaluated, overwriting the values determined so far.
3. User Default
   If there is a file by the name of $user.PLI1.OPTIONS under the current user ID, the control statements contained in that file are evaluated, overwriting the values determined so far.
4. Start Default
   Further control statements may be specified at the time the compiler PL/I program is started. Following the start, they are read in from system file SYSDTA and evaluated, overwriting the values determined so far.

This control statement concept is valid for both the compiler (*COMOPT) and the PL/I program (*RUNOPT). The following applies to the $TSOS.PLI1. OPTIONS and PLI1.OPTIONS files:

- SAM or ISAM file with variable-length records
- Key length for ISAM: 8 characters
- Maximum number of characters evaluated per record: 72
- COMOPT and RUNOPT may appear in combination
- Initial * character may be omitted
- END option may be omitted

Otherwise, control statements are governed by the same rules as described below.

3.3.1 General rules for control statements

All control statements are read from SYSDTA after the compiler is started. They are in the following format:

```
 /EXEC $PLI1

[[_...][*]COMOPT[keyword=[specification        ]]
                [        [(specifications,...)]],...],...

 [_...][*]END[/]
```

The following rules apply here:

1. A number of lines with control statements may be supplied. Control statement *END indicates the end of the control statements.
2. Control statement format:
   - Control statements may be indented, i.e. lines may begin with blanks.
   - The * may be omitted without affecting the meaning.
   - Two or more control statements are separated by commas.
   - Control statements may be continued on the next line but *COMOPT must also precede the continuation line. The options on the lines following *COMOPT are interpreted as one contiguous entry, which means:

     The end of a line cannot replace a comma.
   - Control statements consist of a keyword followed by the equal symbol and one or more specifications.

     Example

     ```
     *COMOPT SOURCE = FILE1, LIST = (SOURCE, XREF),
     *COMOPT DBG = ALL
     *END
     ```
3. If a given control statement allows a number of specifications, they must be enclosed in parentheses as a list.

In addition, the following applies:

– A control statement may be in any of the following formats:

```
keyword = {empty | ? | STD | specification | list}
```

Meaning

empty The current value is not modified.

? In interactive mode, inquiry to the user terminal. In batch mode, the same effect as 'empty'.

STD The specification defined under 'default' (preset options) are used.

– Keywords and specifications of a keyword nature can be abbreviated. The characters used to form the abbreviation are underlined in the discussion of each control statement.

– Specifications may always appear as a list, in the form: (specification 1, specification 2, ...)

If inconsistencies arise, the last option prevails.

– If errors are found during the processing of a *COMOPT line, the options previously evaluated still apply.

– Control statements are only valid for the particular compiler run for which they were specified.

– If a control statement is specified more than once, the last supplied value applies.

4. If the system defaults (preset values) are to be used, you do not need the *COMOPT lines and the *END option should be supplied.

Note

Normally, the compiler can operate without error even if the *END option is omitted, but there may be situations where characters read in via SYSDTA are misinterpreted.

5. *END/ means that a breakpoint is set after the control statements have been processed, i.e. the program (the compiler) is interrupted but remains loaded. At this point, the user may e.g. issue a /SYSFILE command to reassign SYSDTA. A subsequent RESUME command ends the interrupt. The use of *END/ in batch tasks is similar. The program is interrupted and the next command is executed. A /RESUME command permits continuation from the point of interrupt unless another program has been loaded in the meantime.

3.3.2 Error handling during control statement evaluation

Any errors detected during the syntactic or semantic checking of control statements are registered. Only at the end of processing (*END) will the error be reported and displayed on SYSOUT with the invalid string.

A missing initial asterisk is not reported; the control line is evaluated. A missing *END option is only reported if a valid COMOPT has been processed before.

After an error message is issued, the terminal displays, in interactive mode, the invalid control statement and the prompt "ENTER CORRECT OPTION OR '_' OR '@':". The following entries are allowed:

- Control statements
  The invalid control statement may be corrected and returned. Additional control statements may be entered as well. Entry is made without *COMOPT.

  Caution
  Any error that may be made while entering a correction will be reported also; however, any additional control statements following the invalid control statement will be lost.

- @ (commercial at)
  Processing of control statements is terminated. This ensures that the table of valid control statements in the compiler is in a defined status.

- Blanks (_)
  The invalid control statement is ignored. The next error is displayed.

After all the error messages have been displayed, the user is prompted in interactive mode as follows:

```
***CONTINUE (Y/N).
```

If the response is 'Y', compilation continues. In batch mode, compilation can be continued by setting task switch 0. If 'N' is entered, or if switch 0 is not set, then compilation is aborted.

The above rules apply accordingly to error handling in object programs.

3.3.3 Summary of the control statements for the PLI1 compiler

Fig. 3-1 provides a summary of all the control statements that may be used in compilation.

The individual control statements, combined into groups, are described in section 3.4 and later.

Most control statements and specifications can be abbreviated. Some specifications may be supplied in a negative form (e.g. TERMINAL and NOTERMINAL). The outline description in the "Effect" column refers to the positive option.

Compiler control statement	Abbr.	Meaning	Specification	Abbr.	Effect	Default
COMLIB	CML	Library option	NO	N	No library	NO
			(libname,...)		Library or library hierarchy	
DEBUG	DBG	Debugging aids option	ALL	ALL	≙(ST,P,L,C,G, R)	
			NO	NO	≙(NST,NP,NL, NC,NG,NR, NBK)	NO
			STMT	ST	Insert source line number	
			NOSTMT	NST		NST
			PROCTRACE	P	Instructions are inserted for PROCEDURE entries Labels CALLs GOTOs RETURNs	
			NOPROCTRACE	NP		NP
			LABTRACE	L		
			NOLABTRACE	NL		NL
			CALLTRACE	C		
			NOCALLTRACE	NC		NC
			GOTOTRACE	G		
			NOGOTOTRACE	NG		NG
			RETURNTRACE	R		
			NORETURNTRACE	NR		NR
			BREAKPOINT ({[i-]z[.a]},.)	BK	Insert a breakpoint for INCLUDE text 'i', line 'z', statement 'a'	
			NOBREAKPOINT	NBK		NBK

Fig. 3-1 Control statements for the PLI1 compiler (part 1)

Compiler control statement	Abbr.	Meaning	Specification	Abbr.	Effect	Default
DIAGNOST	DIAG	Diagnostic message output	INFORMATION WARNING ERROR SEVERE	I W E S	Messages in ascending order from specified severity to SYSLST	W
			TERMINAL NOTERMINAL	T NT	In interactive mode, above messages additionally displayed on terminal	NT
			SAVLST NOSAVLST	SV NSV	Above messages additionally output to file	NSV
FORMAT	FM	Controls the number of lines per page, line length, and language for messages	PRINTER ([m_1][,m_2])	P	Controls output to SYSLST and, in batch mode, also to SYSOUT; m_1= number of lines/page m_2= length of line.	P(64,132)
			TERMINAL (|n_1||,n_2|)	T	Controls output to terminal; n_1 = number of lines n_2 = number of characters	T (O,k)[1]
			ENGLISH DEUTSCH	E D	All compiler output is in English or German	E

1) 'k' is the physical line length of the current output device.

Fig. 3-1 Control statements for the PLI1 compiler (part 2)

Compiler control statement	Abbr.	Meaning	Specification	Abbr.	Effect	Default
LIST	LST	List control	ALL	ALL	Corresponds to (EX,NOT,NLC, N,M,FX,I,E,A, NOF,OP,SM)	
			NO	NO	Corresponds to (NSC,NOT,NLC, NN,NM,NX,NI, NE,NOF,NA,NOP, NSM)	
			SOURCE	SC	Source listing with %INCLUDE statements	SC
			NOSOURCE	NSC		
			EXPAND	EX	Source listing with inserted INCLUDE texts	
			OUTTEXT [(c)]	OT	Display source text, header, trailer, and frame character 'c' if any	
			NOOUTTEXT[(c)]	NOT		NOT
			NOLINECNT	NLC	Index without leading zeros or line count	NLC
			LINECNT (EDOR)	LC(ER)	Index according to EDOR	
			LINECNT (EDT)	LC(ET)	Index according to EDT	
			LINECNT (PRINT)	LC(PR)	Index according to PRINT	
			INSOURCE	ISC	Log preprocessor input	
			NOINSOURCE	NISC		NISC
			NEST	N	Identification of nestings	N
			NONEST	NN		
			MAP	M	Output of storage maps	
			NOMAP	NM		NM

Fig. 3-1 Control statements for the PLI1 compiler (part 3)

Compiler control statement	Abbr.	Meaning	Specification	Abbr.	Effect	Default
LIST (continued)			SHRTXREF	SX	References and attributes of identifiers (explicit only)	NX
			FULLXREF	FX	References and attributes of identifiers (all)	
			NOXREF	NX		
			AGGREGATE	AGG	Output of the aggregate listing	NAGG
			NOAGGREGATE	NAGG		
			IREF	I	INCLUDE reference listing	I
			NOIREF	NI		
			ESD	E	External names listing	NE
			NOESD	NE		
			OFFSET [({a,e},...)]	OF	Table to assign source statement to hexadecimal program addresses in area a, e	OF
			NOOFFSET	NOF		
			ASSM [({a,e},...)]	A	Output of an object code listing in area a, e	NA
			NOASSM	NA		
			OPTIONS	OP	Output of the enabled control statements	OP
			NOOPTIONS	NOP		
			SUMMARY	SM	Output of program statistics	NSM
			NOSUMMARY	NSM		
			SAVLST	SV	Copy all listings into a file	NSV
			NOSAVLST	NSV		
			TERMINAL	T	Copy all listings to the terminal	NT
			NOTERMINAL	NT		

Fig. 3-1 Control statements for the PLI1 compiler (part 4)

Compiler control statement	Abbr.	Meaning	Specification	Abbr.	Effect	Default
MARGINS	MAR	Formal structure of source program and INCLUDE texts	CHAR 48 CHAR 60	C48 C60	48- or 60-character set is used	C60
			TEXT (t_1,t_2)	T	Source program text begins in column t_1 and ends in col.t2 2	T (2,72)
			LINID(l_1,l_2) NOLINID	L NL	Line identification begins in column l_1 and ends in col.l_2	NL
			PAD NOPAD	P NP	Pad source line with blanks	P
			ASACNTRL (a) NOASACNTRL	A NA	Carriage control character for source program listing is in column 'a'	NA
			GAMKEY(n,c)	G	Conversion of an element name to form a group key n = name length c = pad char.	G (0,0)
			SAVMAC NOSAVMAC	SM NSM	Source program is a SAVMAC file	NSM
MESSAGE	MSG	Message control	SYSLST NOSYSLST	S NS	Additionally, messages to SYSLST in interact. mode	NS
MODULE	MOD	Output destination for object modules	* Library (*[(version)]) Library (element[(version)])	*	Temporary *EAM file in LMS library *: Module name becomes element name	*

Fig. 3-1 Control statements for the PLI1 compiler (part 5)

Compiler control statement	Abbr.	Meaning	Specification	Abbr.	Effect	Default
OBJECT	OBJ	Generating the object module			Compilation up to and including:	
			SYNTAX	SY	Syntax run (SY)	O
			SEMANTIC	SE	Semantics run (SE)	
			CODE	C	Code generation (C) or	
			OUT	O	Output to object module file	
			WARNING (n)	W	Compilation	
			ERROR (n)	E	aborted if n errors of the corresponding severity have occurred	E(32767)
			ABORT(n)	A	Compilation aborted after n errors and warnings	A(500)
			MACRO	MAC	Terminate after preprocessor run	
OPTIMIZE	OPT	Optimization	NO	NO	Corresponds to (NE,NOL,NT,NR)	NO
			ALL	ALL	Corresponds to (E,OL,T,R)	
			ENABLING	E	Conditions OFL,UFL,ZDIV preset	
			NOENABLING	NE	Conditions CONV,FOFL,OFL,UFL,ZDIV preset	NE
			OVERLAP	OL	Overlapping possible by assignment	NOL
			NOOVERLAP	NOL		
			TIME	T	Optimize runtime	NT
			NOTIME	NT		
			REORDER	R	Optimize sequence of statements	NR
			NOREORDER	NR		

Fig. 3-1 Control statements for the PLI1 compiler (part 6)

Compiler control statement	Abbr.	Meaning	Specification	Abbr.	Effect	Default
OPTIONS	OPN	Program classification	MAIN NOMAIN	M NM	Program is main program	NM
			ISO	I	Source program conforms to PL/I standard	N
			NOISO	N	Source program conforms to Industry Standard	
			MACRO NOMACRO	MAC NMAC	Invoke pre-processor	NMAC
			INTERRUPT NOINTERRUPT	INTR NINTR	Include interrupt points for ATTENTION condition	NINTR
			XS NOXS	X NX	Generate XS object modules	NX
			BITPTR NOBITPTR	B NB	Generate absolute bit pointer	NB
SOURCE	SRC	Specification of input volume for source programs	*	*	SYSDTA, following the control statements	*
			File name		LINK or FILE name of the file containing the source program	
			Library (element)		Element from GAM, LMS, or MLU file	
			Library (element [(version)])		Element from LMS library	

Fig. 3-1 Control statements for the PLI1 compiler (part 7)

Compiler control statement	Abbr.	Meaning	Specification	Abbr.	Effect	Default
STORAGE	STR	Capability for controlling storage management at compiler run time and achieving optimizations	AREA ([q_1][,[q_2][,[q_3]]])	A	A storage area of 'q_1' pages is initially provided for the standard area. Additional capacity, if required, is added in increments of 'q_2' pages up to a maximum of 'q_3' pages, when error 30 may be reported. (1 page = 4KB)	A(16,16,
			STACK ([s_1][,s_2])	S	The stack is used in segments of 's_1' pages. A reserve of 's_2' is provided for error and end handling in case of insufficient storage space. (1 page = 4KB)	S(16,4)
SYMTEST	SMT	Debugging aid AID	ALL	A	Generate debug aid info.	M
			NO	N	Do not generate debug aid info.	
			MAP	M	Generate only ESD info "compilation unit"	

Fig. 3-1 Control statements for the PLI1 compiler (part 8)

3.3.4 Controlling the compiler via the source program

The OPTIONS entries in the PROCEDURE statement allow the specification of a number of compiler controls in the source program. They are supplied in the following format:

Identifier: PROCEDURE OPTIONS (option,...)

The following options are supported:

Option	Effect	Description
MAIN NOMAIN	This procedure is compiled as a main procedure.	Section 3.6.3
ISO NOISO	Compilation according to standard.	Section 3.6.3
OVERLAP NOOVERLAP REORDER NOREORDER ENABLING NOENABLING	Optimization	Section 3.6.4
REENTRANT	None; all PL/I procedures are reentrant	
XS NOXS	XS-capable objects are to be generated.	Section 3.6.3
BITPTR NOBITPTR	Bit pointers are to be generated (only relevant for OPTIONS (XS))	Section 3.6.3

When used, these options take precedence over the same entries for *COMOPT OPTIONS = or *COMOPT OPTIMIZE =.

The OPTIONS attribute, when used in the declaration of external entries, defines the way the compiler will generate calls for user program procedures.

Format:

DCL identifier ENTRY [(parameter-list)] OPTIONS (option[,]...);

The following entries are supported for the OPTIONS attribute:

Option	Effect	Description
{ASSEMBLER ASM}	The compiler generates the invocation of the procedure thus declared in accordance with Industry Standard compatible conventions.	Section 7.1.2.3 1)
[INTER]	INTER has no additional significance.	
PLI1	This ensures error handling for ASSEMBLER programs according to PLI1 conventions.	Section 7.2.1
COBOL [INTER]	This entry refers to a COBOL program. INTER has no additional significance.	Section 7.3
FORTRAN	This entry refers to a FORTRAN program. Interrupts are handled by PL/I.	Section 7.3
FORTRAN INTER	This entry refers to a FORTRAN program. Interrupts not dealt with by FORTRAN are handled by PL/I.	Section 7.3
ILCS	The compiler generates a call for the agreed procedure in accordance with ILCS conventions.	Section 7.4
LIBRARY	Special option for entry name.	Section 7.1.6
VARIABLE	For invoking the procedure thus declared, the compiler generates a detailed parameter list which contains all available information on the arguments and their number.	Section 7.1.2.2 1)
WXTRN	The entry (weak external) is not handled automatically by the linkage editor but only if linkage is requested explicitly or if the entry is declared somewhere else without WXTRN. It is the user's responsibility to make sure control will not be transferred to an unlinked entry.	[3] TSOSLNK [12] as of BS2000 V8.0

1) This option is required if the invoked procedure can be supplied with a variable number of parameters (e.g. assembler procedures). For details of the interface, see chapter 7.

3.3.5 Storage requirements of the compiler (STORAGE)

STORAGE is a statement for controlling the use of the virtual user address space by the compiler. It can be used to specify the initial size and extensions of the virtual storage for the standard area and stack.

Operating system capacity:

A virtual user address space of 1 MB, although theoretically sufficient, would limit the program size to approx. 200 - 1000 statements; therefore, an address space of 2 - 4 MB should be provided (a storage capacity of 2 MB gives you approx. 1000 - 3000 statements; 3 MB, approx. 2500 - 7000; and 4 MB, approx. 4000 - 10000 statements).

The storage statistics (LIST = SUMMARY) indicate the optimum sizes for standard area and stack.

```
STORAGE={specification       }
        {                    }
        {(specification,...) }
```

Default:

```
STORAGE=(AREA (16,16,),STACK (16,4))
```

The following specifications are allowed:

AREA ([q_1][,[q_2][,[q_3]]])

An initial storage area of 'q_1' virtual pages of 4 KB each is supplied for building the workfile during compilation. Necessary extensions are added in increments of 'q_2' pages up to a maximum of 'q_3' pages. If 'q_3' is not specified, maximum storage size is allocated. If that size is exceeded, the system reports error no. 30.

Depending on the size of the allowable user address space, a storage requirement of up to 1500 (6 MB) pages is reasonable for the standard area.

STACK ([s_1][,s_2])

The space for the stack requirements of the compiler is used in segments of 's_1' virtual pages of 4 KB each. A reserve of 's_2' pages is provided for error and end handling in cases of insufficient storage space.

The maximum depends on the size of the allowable user address space. Generally, the default 16 should be sufficient.

3.4 Controlling source input

The PLI1 compiler provides various facilities for entering the source program. The source is read in either via system file SYSDTA or from a cataloged file. The source program may be present in the form of a SAM or ISAM file or it may be an element in a macro-organized library or stored in a group file. The read-in process is controlled by the SYSFILE command and the SOURCE control statement. SOURCE defines whether the source program will be taken from a file or library or whether it should be read in from system file SYSDTA. If it is read in from SYSDTA, entry is usually via a card deck (batch mode) or from a terminal (interactive mode). Moreover, the role of SYSDTA may be assigned to a user file by means of the SYSFILE command. Input via SYSDTA is not possible, however, if the source program is supplied as an element of a library or as a group file.

Libraries can be created and maintained by the MLU or LMS utilities (see Utility Routines [3] and LMS [13]).

Note

The MLU utility, when processing source texts, scans columns 1 and 2 in order to detect control statements. If it finds a blank there, followed by an alphabetic character, that line will be regarded as control information. It is therefore recommended that PL/I programs are either written from column 1 or from column 3 if they are to be stored in MLU libraries.

The term ’group file’ refers to the set of records in an ISAM file whose keys begin with a given string, i.e. the name (group index) of the file (see the "EDOR" reference manual [4]).

Another means of controlling source input is the compiler statement %INCLUDE. It should be specified in the source program and indicates a text which will be obtained from a library or file and inserted in the source program in place of the %INCLUDE statement. The library is selected by means of the control statements COMLIB.

The facilities described above merely serve to define the source for the read-in process. More precise control such as the selection of code, line numbering, and columns as well as the definition of a group file element to be compiled is accomplished through the MARGINS control statement.

When the source program is entered via terminal, the request for further lines can be terminated by

```
*END
```

This statement is only effective if the end of the source program has been encountered before. It must be supplied as part of the MARGINS option.

Another way is to enter the following:

```
BREAK function
/EOF
/R
```

Again, source input terminates.

3.4.1 SYSFILE command (SYSDTA reassigned)

The SYSFILE command allows the user to modify the assignment of system file SYSDTA.

General Format:

```
                    ⎡ (PRIMARY)            ⎤
                    ⎢ (SYSCMD)             ⎥
                    ⎢ filename             ⎥
/SYSFILE SYSDTA = { ⎢                      }
                    ⎢ library (element)    ⎥
                    ⎢ (CARD)               ⎥
                    ⎣ (device)             ⎦
```

For example, the command

```
/SYSFILE SYSDTA = filename
```

assigns SYSDTA to a cataloged file, and

```
/SYSFILE SYSDTA = (CARD)
```

assigns it to a card reader. Further details can be found in the "Control System Command Language" reference manual [2] and in section 6.2.1.

The compiler initially expects its control statements in system file SYSDTA. Depending on the SOURCE control statement, the source program may then be entered via SYSDTA also.

If SYSDTA was reassigned before compilation, it is usually necessary to reset the SYSDTA assignment after compilation. This can be done by the command

```
/SYSFILE SYSDTA = (SYSCMD) or
/SYSFILE SYSDTA = (PRIMARY)
```

The different effects of these two commands should be noted. SYSCMD always resets SYSDTA to the data stream of the current spool-in file (batch processing), ENTER file or DO procedure. PRIMARY has the same effect in batch mode but in interactive mode it always assigns the current terminal. This applies even when the SYSFILE command is part of a DO procedure.

Example 1

Enter the control statements (and source program) from card reader:

```
/SYSFILE SYSDTA = (CARD)
/EXEC $PLI1
/SYSFILE SYSDTA = (SYSCMD)
```

Example 2

Enter the control statements (and source program) from the CARD file:

```
/SYSFILE SYSDTA = CARD
/EXEC $PLI1
/SYSFILE SYSDTA = (SYSCMD)
```

Note that the parenthesized CARD of example 1 refers to card reader whereas the unparenthesized CARD of example 2 refers to a file whose name is CARD.

3.4.2 Defining the source file (SOURCE)

The SOURCE control statement defines the file or library from which the source program will be read.

```
SOURCE={*                              }
       {filename                       }
       {library (element[(version)])   }
```

Default: SOURCE = *

*
After the control statements are processed, further input is expected from system file SYSDTA.

filename
If the source program is stored in a SAM or ISAM file, either the file name or the file LINK name is specified here.

library (element[(version)])
If the source program is stored as a library element, the library name, the source program name in parentheses and, optionally, a version specification, also in parentheses, must be entered. The library must have been created in accordance with LMS or MLU or GAM (see EDOR) conventions.

Names of LMS library elements may be up to 64 characters in length. If they are longer, they will be truncated to 64 characters. Names of MLU library elements may be up to 8 characters in length. If they are longer, they will be shortened to the first 5 and the last 3 characters.

Names of GAM file elements may be no longer than indicated in the GAMKEY specification of the MARGINS statement. If they are longer, they are shortened in accordance with the rule described under the GAMKEY specification (see section 3.4.4).

The version specification is only evaluated for LMS libraries. It may be up to 24 characters in length. If no version specification is entered, the element with the highest version number will be addressed when the library is an LMS library.

'filename' and 'library' are first interpreted as link names. If that fails, they are regarded as file names. The same applies to the identification of libraries referenced in %INCLUDE statements.

Note on application

Assume e.g. a control statement SOURCE = ORIGIN within a DO procedure, which permanently assigns a source file name for several compiler runs. Any source programs may be compiled then if a /FILE command with the name of the current source file is issued before each call of the DO procedure. In this case:

```
/FILE PROGR1,LINK=ORIGIN
```

Example 1

Entering a source program from a cataloged file A.

```
   .
   .
/SYSFILE SYSDTA=(SYSCMD)
/EXEC $PLI1
*COMOPT SOURCE=A
*END
/next command
   .
   .
   .
```

Example 2

Entering a source program from a library LIB1. Assume the name of the source program is PROGR5

```
   .
   .
/SYSFILE SYSDTA=(SYSCMD)
/EXEC $PLI1
*COMOPT SOURCE=LIB1(PROGR5)
*END
/next command
   .
   .
   .
```

Example 3

Entering a source program from a GAMLIB file. The record groups whose records begin with the key 00003 are to be compiled. The KEY defined for the file by the FILE command is assumed to be 8 characters in length, the first 5 of which will be interpreted as the group key (in this case, 00003). This means e.g. that all records with keys ranging from 00003000 to 00003999 will be compiled.

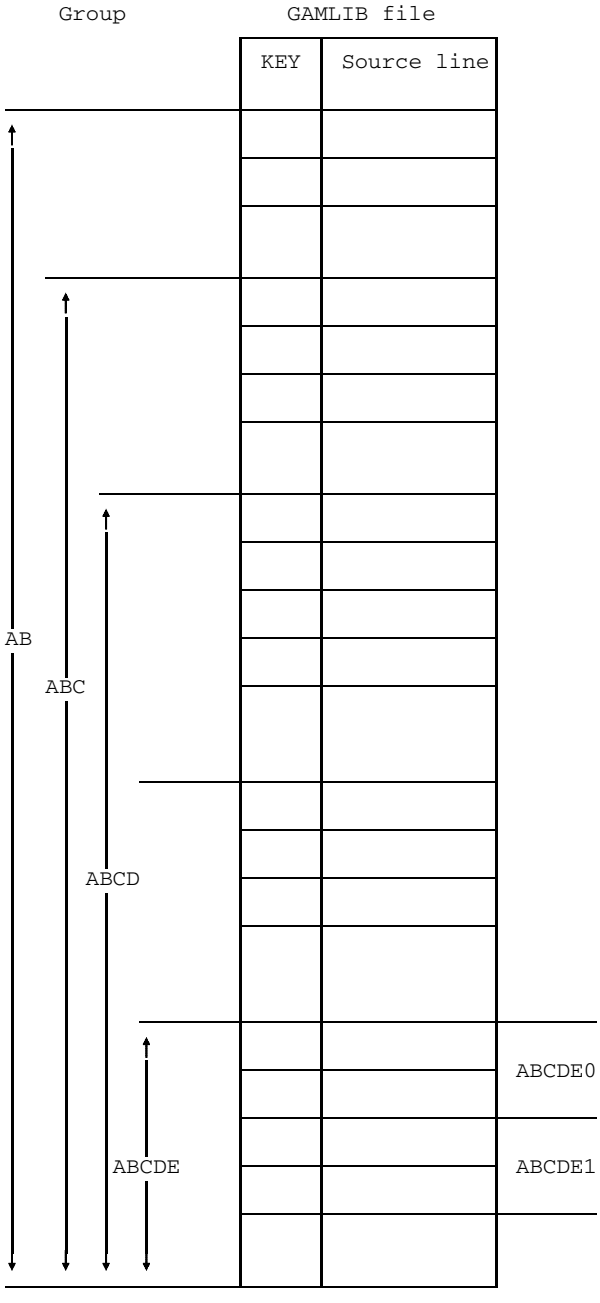
```
   .
   .
/SYSFILE SYSDTA=(SYSCMD)
/EXEC $PLI1
*COMOPT SOURCE=GAMBIB(00003),MARGINS=GAMKEY(5,0)
*END
/next command
   .
   .
   .
```

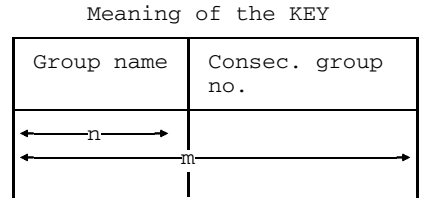
Example 4

Entering a source program from a file GAMLIB. Assume that the LINK name is GROUP. All records whose group name is ABCDEO are to be compiled.

```
   .
   .
/FILE GAMBIB,LINK=GROUP
/SYSFILE SYSDTA=(SYSCMD)
/EXEC $PLI1
*COMOPT SOURCE=GROUP(ABCDE),MARGINS=GAMKEY (6,0)
*END
/next command
   .
   .
```

Examples of the GAMLIB file





n: first option from GAMKEY specification

m: KEYLEN

KEYLEN: option in the FILE command

If the user, as in example 4, supplies the options SOURCE = GROUP(ABCDE) and MARGINS = GAMKEY(6,0) then all records with the group name ABCDE0 are compiled because the character "0" is added to the 5-digit element name to generate the 6-digit group name.

For MARGINS = GAMKEY (2,0), the records whose group name is AB would be compiled, i.e. all the records of our sample file. For MARGINS = GAMKEY (4,0), the group name ABCE would be formed from the element name. This group is not included in the above sample file. The rules of formation for group names are explained in section 3.4.4.

Example 5

The control statements for compiling a source program are contained in a file OPT. The source file is not to be defined until the control statements have been processed. The source is in the PROG file.

Contents of the OPT file:

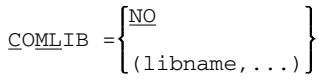
```
*COMOPT SOURCE = *
 further control statements
 *END/
```

Command sequence at the terminal:

```
   .
   .
   .
/SYSFILE SYSDTA=OPT
/EXEC $PLI1

breakpoint is taken

/SYSFILE SYSDTA=PROG
/RESUME
   .
   .
   .
```

3.4.3 Defining the INCLUDE library (COMLIB)

The COMLIB control statement is required if %INCLUDE statements without library options occur in a PL/I source program, i.e. if the format of the statement is %INCLUDE name;.
The names of the libraries containing the names (elements) specified in the %INCLUDE statements must be supplied in COMLIB. The libraries are searched in the order they were specified in COMLIB.



Default: COMLIB=NO

NO — No library; the name supplied in %INCLUDE is the name of a SAM or ISAM file.

libname — The name specified in %INCLUDE is interpreted as an element name. 'libname' identifies GAM, LMS, or MLU files which are searched in the specified order until the element is found. In each case, 'libname' is first interpreted as a LINK name. If this LINK name cannot be found, the search continues for a file of the same name. A maximum of 8 libnames may be supplied.

Note

From the list of INCLUDE references (LIST = IREF control statement), the user can learn in which library the appropriate INCLUDE text was found.

3.4.4 Source line format (MARGINS)

The MARGINS control statement describes the line format of source programs and INCLUDE texts.

```
MARGINS={specification       }
        {                    }
        {(specification,...) }
```

Default: MARGINS = (C60,T(2,72),NL,NA,G(0,0),P)

Possible specifications:

CHAR60
CHAR48
: Source program and INCLUDE texts use only the 60-character set or additionally the replacement symbols of the 48-character set. Differences between the character sets are explained in section 3.2 of the PL/I language reference manual [1].

TEXT(t_1, t_2)
: The source program text begins in column t_1 and ends in column t_2, where $1 \le t_1 \le t_2 \le 255$. The same applies to INCLUDE texts. If the t_1 - t_2 range is outside the text ranges, the line is ignored.

LINID(l_1,l_2)
: LINID refers to the line identifiers contained in the source file. The line identification field begins in column l_1 and ends in columns l_2, again with $1 \le l_1 \le l_2 \le 255$.
Out of the characters in the field, a maximum of 8 positions (from right to left) are evaluated. These characters appear unmodified in the source listing. For references etc., only the numeric part is internally evaluated (from right to left) and stored (in extreme cases, 0). LINID only refers to the source indicated by SOURCE. It does not apply to INCLUDE texts.

NOLINID
: There is no line identification in the source lines. The ISAM key or consecutive numbering is used for the listing.

PAD
NOPAD
: Source lines are padded with blanks to give them the length (e - a + 1) specified in TEXT (a, e).

ASACNTRL (a)
: Column 'a' of the record contains an ASA carriage control character to be taken into account when the source program is listed ($1 \le a \le 255$).

NOASACNTRL

There are no ASA carriage control characters in the records of the source program.

GAMKEY(n,c)

This specification describes the editing of an element name from the SOURCE control statement and the %INCLUDE statement in order to form the group key for GAM files. In GAM files, the first 'n' characters (n = 1 thru 15) of the record key (ISAM KEY) are interpreted as the group key. If z is the length of the element name from the SOURCE control statement or a %INCLUDE statement, then for

n = z: the element name is the group key;

n > z: the element name is padded with the character 'c' to length 'n';

n < z: the first 'k' and the last 'n-k' characters of the element name form the group key, where

```
k = 1 + FLOOR (n/2).
```

For examples of GAMGEY, refer to section 3.4.2.

Default: GAMKEY (0,0)

This means that 'n' is calculated from the option supplied for COMLIB, SOURCE or %INCLUDE.

SAVMAC This specification is used to read and print a SAVMAC file generated by a preprocessor run. This ensures that the compiler listings as well as the runtime messages will refer to the original source line numbers or the original include numbers of the preprocessor input. This option need only be supplied for separate preprocessor and compiler runs; it will be set internally if preprocessing and compilation follow each other immediately. This specification is equivalent to

```
MARGINS = (LINID(4,11), TEXT(17,124)).
LIST = NOOUTTEXT
```

with an overwriting effect.

NOSAVMAC

The source program is not in accordance with SAVMAC file conventions. See section 3.11.

The areas provided within a record for the text (t_1 - t_2), line identifiers (l_1 - l_2), and ASA control characters must not be overlapping.

3.5 Controlling the listing output of the PLI1 compiler

During compilation, the PLI1 compiler generates a number of listings containing information on the structure of the program being compiled and the flow of the compilation process.

All output can be classified as follows:

- Compiler messages (M)
- Source listings (L)
- Diagnostic information on the source program (D)

Depending on the type, output generated during compilation is directed to either SYSLST or SYSOUT. In addition, various control statements enable the user to direct output to a second system file. The following table provides a summary.

CLASS	Object of compiler output	Output file		
		Default	Additional output to file	controlled by
L	Source and INCLUDE texts as well as other output according to LIST control statement	SYSLST	SYSOUT only allowed in interactive mode	LIST = TERMINAL (no effect in batch mode)
D	Source-related error messages and warnings	SYSLST	SYSOUT only allowed in interactive mode	DIAGNOST = TERMINAL (no effect in batch mode)
M	Compiler messages	SYSOUT	SYSLST	MESSAGE = SYSLST

The ability to direct SYSLST to a user file by means of the SYSFILE command provides another control facility. Format of the SYSFILE command:

```
                  ⎧(PRIMARY)              ⎫
                  ⎪file-name              ⎪
/SYSFILE SYSLST=  ⎨                       ⎬
                  ⎪(file-name, EXTEND)    ⎪
                  ⎩*DUMMY                 ⎭
```

For details, refer to the "Control System Command Language" reference manual [2] and section 6.2.1.

3.5.1 List selection (LIST)

The LIST control statement is provided to select listings which arise as a result of the compilation of a PL/I program and are output by the compiler to SYSLST. The same control statement can also be used at runtime of the PL/I program, but then only the options for OPTIONS, SUMMARY and TERMINAL are relevant.

```
LIST={specification       }
     {                    }
     {(specification,...) }
```

Default: LIST = (SC, NOT, NLC, N, NM, NX, I, NE, OF, NA, OP, NSM, NSV, NT, NISC)

ALL equivalent to:
(EX, NOT, NLC, N, M, FX, I, E, NOF, A, OP, SM)

NO equivalent to:
(NSC, NOT, NLC, NN, NM, NX, NI, NE, NOF, NA, NOP, NSM)

NO and ALL can be combined with other values, and the contradiction rule (last appropriate option valid) becomes applicable.

The following specifications can be used:

EXPAND Source listing with inserted INCLUDE texts.

If OUTTEXT is specified as well, then both the %INCLUDE statement and the INCUDE text are listed.

SOURCE
NOSOURCE

Output of the source listing with %INCLUDE statements (without inserted INCLUDE texts).

OUTTEXT [(c)]

The whole line is listed, not only the source text selected for compilation by MARGINS = TEXT (a, e). Moreover, all % statements will be listed.

A single character may be supplied for 'c', which is then inserted on both sides of the source text in the listing, giving the source text a lateral frame. For variablelength lines, it may happen that the right-hand frame characters are not vertically aligned if MARGINS = NOPAD is specified.

NOOUTTEXT [(c)]

Only the source text specified by MARGINS = TEXT (a, e) is listed, including the frame as supplied by 'c' if applicable.

NOLINECNT

Specifies in which form the reference to a source line is to be output in a listing.

LINECNT

The source reference has the following format:

(EDOR | EDT | PRINT)

"[Include] line [:statement]" See also section 3.8. The line specification can be controlled as follows:

NOLINECNT	without leading zeros
EDOR	without trailing zeros
EDT	without trailing zeros and without leading zeros and with a period between the 4th and 5th position.
PRINT	unchanged, same as for PRINT command.

INSOURCE

Listing of input for the preprocessor.

NOINSOURCE

NEST
NONEST

Identifies nestings in the source listing (PROCEDURE, BEGIN, DO, SELECT, END), comment continuation lines, string constants occupying more than one line.

MAP
NOMAP

Output of a memory map

SHRTXREF

Output of a reference listing which includes the identifiers (referenced identifiers only)

FULLXREF

Reference listing and attributes (all identifiers)

NOXREF

No reference and attribute listing

AGGREGATE
NOAGGREGATE

Output of a structure length table (aggregate listing)

IREF
NOIREF

INCLUDE reference listing indicating the relationship between the INCLUDE numbers used in the program listing and the associated INCLUDE names (file, library, element)

ESD
NOESD

Listing of all identifiers with the EXTERNAL attribute

OFFSET[({a,e},...)]
NOOFFSET

Listing which shows the relationship between the source statement and hexadecimal offset (procedure-relative address) for the whole compilation unit or the line areas specified by a, e.
If a = e, e can be omitted, but not the associated comma. The offset listing is used for locating errors in the source program, since normally error messages are issued with reference to the hexadecimal address. The offset listing can be dispensed with if DEBUG = STMT is supplied (see section 3.6.2).

ASSM[({a,e},...)]
NOASSM

Listing of the compiler-generated machine code in assembler-like format for the entire compilation unit or the line areas indicated by 'a, e'. In the latter case, this listing is generated as part of the OFFSET listing (see above).

OPTIONS
NOOPTIONS

Listing of the enabled control statements for this compilation.

SUMMARY
NOSUMMARY

Statistics including the following:
- Maximum stack request
- Maximum allocation of the virtual address space (total)
- Number of requests for additional storage and storage releases by the PLI1 system.

These statistics are relevant to optimization (see chapter 8).

SAVLST
NOSAVLST

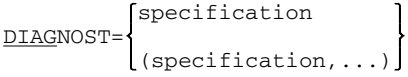
Copies all the above listings into a file whose name is assigned implicitily.
First, the system tries to access a file whose LINK name is SAVLNK; if that is unsuccessful, the file name SAVLNK is retrieved in the user catalog.
If there is no such file, a file by the name of SAVLST.PLI1.tsn is created in the user catalog; it is renamed as SAVLST.PLI1.module-name if the syntax run proceeds correctly. This file will be overwritten by a subsequent compiler run. The same applies to the SAVLST entry in the DIAGNOST control statement.

TERMINAL
NOTERMINAL

Additional output of the above-mentioned listings to SYSOUT (terminal) in interactive mode. No effect in batch mode.

3.5.2 Selection of diagnostic information (DIAGNOST)

The control statement DIAGNOST effects the output of diagnostic information on the source program. This includes error messages or warnings referring to the translated program and the INCLUDE texts.



Default: DIAGNOST = (W, NT, NSV)

The following specifications may be entered:

INFORMATION
WARNING
: Diagnostic messages starting from the severity level specified are output to SYSLST.

ERROR
SEVERE
: Information messages 500 - 504 are only displayed if some optimization is required by *COMOPT OPTIMIZE=.

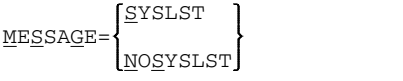
TERMINAL
NOTERMINAL
: The diagnostic messages are also output to the terminal

in interactive mode. No effect in batch mode.

SAVLST
NOSAVLST
: The above messages are also output to a file. See also the notes on SAVLST in the LIST control statement.

3.5.3 Additional listing output of messages (MESSAGE)

The MESSAGE control statement defines whether messages from the PLI1 compiler should also be output to SYSLST.



Default: MSG = NS

SYSLST
: All messages from the compiler are also output to SYSLST.

NOSYSLST
: Messages from the compiler are only output to SYSOUT.

3.5.4 Output format (FORMAT)

The FORMAT control statement controls the number of lines per page and the line length of all compiler outputs to the printer, the terminal or the SAVLST file. It also permits specification of the output language (English or German). This control statement can be used in the same way for both PL/I compilations and PL/I user programs (*RUNOPT FORMAT = ...).

```
FORMAT={specification       }
       {(specification,...)}
```

Default: FORMAT = (P(64,132), T(O,k),E)

k = physical line length of the device concerned according to the TMODE macro.

Possible specifications:

PRINTER([m_1][,m_2])

This control affects all output which is sent to SYSLST and which, in batch mode, may be additionally output to SYSOUT. After m_1 lines have been output, a new page is started. When a line length of m_2 characters is reached, the line is split up, i.e. another line is started.

$0 \leq m_1 \leq 255$
$1 \leq m_2 \leq 255$

If $m_1 = 0$ no page feed is performed, i.e. printing continues across the paper fold.

TERMINAL([n_1][,n_2])

This control affects all output for the terminal. When n_1 lines have been reached, the following message is displayed at the terminal:

```
WEITER (J/N/NDT/NL) or
CONTINUE (Y/N/NDT/NL)
```

The user is then able to set the cursor to a suitable position on the screen and make one of the following entries:

J or Y:	Continue output
NL	The TERMINAL option in the LIST control statement is disabled immediately. This only affects listings which are also output to the terminal.

NDT	The TERMINAL option in the TRACE or DIAGNOST control statements is disabled immediately. The output of debugging messages and diagnostic messages to the terminal is inhibited.
N	As NL + NDT

When n_2 characters per line are reached, a new line is started.
If $n_1 = 0$ (default), the inquiry is not displayed on the terminal.

ENGLISH DEUTSCH	All compiler output is in either English or German. The files containing the error texts must be available to the compiler. See also section 3.2.

3.6 Controlling object module generation

This group of control statements determines whether an object module is to be generated, where it will be stored, which test aids will be incorporated, the form in which messages will be output at object time etc.

3.6.1 Switch for compiler phases (OBJECT)

The OBJECT control statement is used to determine whether an object module is to be generated, and, if applicable, how many errors may be accepted by the compiler.

```
OBJECT={specification       }
       {(specification,...) }
```

Default: OBJECT = (O, E (), A(500))

Possible specifications:

MACRO Only preprocessing is performed.

SYNTAX
SEMANTIC
If the compilation was not previously aborted (see below), it should be performed up to and including:
CODE syntax run (SY)
OUT semantics run (SE)
code generation (C)
code output to object module file (O)

WARNING[(n)]
ERROR[(n)]
The compilation is terminated prior to code output at the latest if the specified number (n) of errors of or greater than the weight indicated have occurred. If the option n is omitted, then n = 1 is assumed.

ABORT (n)
The compilation is terminated immediately if n errors or warnings have occurred. $1 \leq n \leq 32767$

3.6.2 Debugging aids in the object module (DEBUG)

The DEBUG control statement is required for instructing the compiler to insert code for the debugging aids in the object module.

```
DEBUG={specification       }
      {(specification,...) }
```

Default: DEBUG = NO

The following specifications are permitted:

ALL	equivalent to DEBUG = (ST, P, L, C, G, R)
NO	equivalent to DEBUG = (NST, NP, NL, NC, NG, NR, NBK)
STMT NOSTMT	Generation of a statement table in the object module. It permits messages of the PL/I program to refer to the SOURCE listing.

Tracing instruments are incorporated for:

PROCTRACE NOPROCTRACE	PROCEDURE entries
LABTRACE NOLABTRACE	Labels
CALLTRACE NOCALLTRACE	CALL statements
GOTOTRACE NOGOTOTRACE	GOTO statements
RETURNTRACE NORETURNTRACE	RETURN statements
BREAKPOINT({[i-]z[.a]},...) NOBREAKPOINT	Breakpoints are inserted at the specified positions. i: consecutive number for INCLUDE texts z: number of the source line a: number of the statement (if omitted, defaulted to 1)

3.6.3 Procedure status (OPTIONS)

The OPTIONS control statement defines whether the compiled procedure is to be declared a main procedure. This control statement also determines whether the source program is to be compiled according to the rules of the industrial standard or according to the rules of the PL/I standard.

```
        ⎧specification       ⎫
OPTIONS=⎨                    ⎬
        ⎩(specification,...) ⎭
```

Default: OPTIONS = (NOMAIN,NOISO,NOINTERRUPT,NOMACRO, NOXS,NOBITPTR)

Possible specifications are:

MAIN
NOMAIN
: The procedure being compiled is the main procedure. Only one procedure of a program consisting of several procedures must be the main procedure. If OPTIONS (MAIN) is supplied in the PROCEDURE statement of the program, the procedure is compiled as the main procedure regardless of the option used in this control statement.

ISO
NOISO
: The source program is compiled according to the rules of the PL/I standard (ISO) or those of the industry standard.

INTERRUPT
NOINTERRUPT
: Interrupt points for the ATTENTION condition are to be inserted in the program. See also chapter 5.

MACRO
NOMACRO
: A preprocessor run is carried out. See also section 3.11 or the PL/I Reference Manual [1], chapter 13.

XS
: XS objects are to be generated (see section 4.7).

NOXS
: Non-XS objects are to be generated (see section 4.7).

BITPTR
NOBITPTR
: Bit pointers are to be generated (only relevant for OPTIONS = XS).

REENTRANT:
: This option, although accepted by the compiler, is irrelevant since all procedures for the PLI1 compiler have this characteristic.

Some of the above may also be supplied in the PROCEDURE statement under OPTIONS (see section 3.3.4). If so, they take precedence over values specified by these control statements.

3.6.4 Optimization (OPTIMIZE)

The OPTIMIZE control statement defines which optimizations are to be used on the object module.

```
OPTIMIZE={specification      }
         {                   }
         {(specification,...)}
```

Default: OPTIMIZE = (NE,NOL,NT,NR)

The following specifications are permitted:

NO corresponds to OPTIMIZE = (NE,NOL,NT,NR)

ALL corresponds to OPTIMIZE = (E,OL,T,R)

ENABLING As a default for computational conditions, only OFL, UFL, and ZDIV are enabled.

NOENABLING
The system default is assumed for computational conditions.

OVERLAP Optimization is carried out for overlapping: If there is a scalar string variable on the right of an assignment, and the compiler is unable to find out whether the source and target variables overlap, it will assume that they do not and will generate optimized code. A warning will be issued in all cases of this kind.

NOOVERLAP
No optimization is carried out for the overlap condition. In cases where it cannot be ascertained whether the source and target variables overlap, it is assumed that they do and safe code is generated.

TIME Time optimization is carried out.

NOTIME No time optimization is carried out.

REORDER This option has the same effect as a REORDER option in the first PROCEDURE statement of an external procedure. An explicit option in the PROCEDURE statement overwrites the REORDER option.

NOREORDER

This option is equivalent to the ORDER option in the PROCEDURE statement. Otherwise the above options apply accordingly. In PL/I, ORDER is the system default.

Optimization in accordance with ENABLING can also be achieved by explicitly disabling (NO...) the appropriate conditions in the source program.

Some of the above may also be supplied in the PROCEDURE statement under OPTIONS (see section 3.3.4), in which case they take precedence over any values specified in these control statements.

3.6.5 Debugging aid AID (SYMTEST)

These control statements are used to indicate that the compiler is to add information which allows the use of the debugging aid AID.

```
SYMTEST={specification       }
        {                    }
        {(specification,...) }
```

Default: SYMTEST = MAP

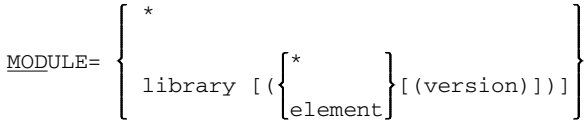
The following specifications are supported:

ALL	Information is generated for the AID debugging system
NO	No information is generated for the AID debugging system
MAP	Same as NO, but also ESD information of type "compilation unit" is generated.

For an extensive description on the use of AID see [18] "AID Debugging PL/1 Programs".

3.6.6 Object module storing (MODULE)

The control statement "*COMOPT MODULE=destination" is used to specify where the generated object modules are to be stored. If this option is omitted, the object modules will be stored in the temporary *EAM file. If two object modules are generated (code module and static module), all specifications will apply to both object modules. For further options see section 3.7. A precondition for the storing of object modules in a library is that the PLAM access method is integrated in the operating system.



Default: MODULE = *

library
The name of a file that contains a library created in accordance with LMS [13] conventions. If this file does not exist, an appropriate library will be created. Specification of a file generation group is not possible. The library name may also be a link name. If the specified name exists both as a link name and as a file name, the link name will be used.

element
*
The name that is given to the object module in the library. If * is specified, the first entry name of the compiled external procedure is taken.

The element names may be up to 7 characters in length: longer names are abbreviated to the first 4 and the last 3 characters. An underscore (_) is replaced with the dollar sign ($). The following characters may be used as element names:

- Characters A through Z
- Digits 0 through 9, but not as the first character
- Special characters
  - $ dollar
  - # number sign
  - @ commercial at
- Punctuation marks
  - \- hyphen
  - _ underscore
  - . period

Punctuation marks must not appear in the first or last position. The hyphen must not be on the right of a special character or punctuation mark. Two identical punctuation marks must not appear in juxtaposition.

If the entry name of the PL/I procedure is adopted, the rules for PL/I names apply.

If two object modules are created during compilation, the version name dealt with above is given to the code module. For the static module the element name is derived from the code module name by filling out the element name with @ characters to a length of 7 characters and appending a digit which indicates the original length of the name (1 to 7).

version

Version name which contains the element. If this option is not specified, the element is given the highest version that is possible in an LMS library (X'FF'):
This will not be logged.

The version name may consists of up to 24 characters, which may be

- Letters A through Z
- Digits 0 through 9
- Punctuation marks
  - `-` Hyphen
  - `.` Period

  Punctuation marks must not appear in the first or last position. Two identical punctuation marks must not appear in juxtaposition. A period must not be followed by a hyphen.

If the specified library already contains an element with the same element name and version name, this element will be overwritten. On each entry for an element its variant (see LMS [13]) is incremented by 1. The element type (see LMS [13]) is always 'R'.

In the *EAM file the element is always entered under the first name of the compiled external procedure. If two elements of the same name are entered, the result is undefined.

A log is sent to SYSOUT stating which module is output and where it is output.

3.7 Object module maintenance

When an external procedure is compiled, normally two object modules will be created: the code module and the static module. The code module is generated anyway; the static module is generated under special conditions only. Chapter 12 explains under which conditions a static module is created, when it is undesirable and how it can be avoided.

Where the object modules are to be stored can be specified using the control statement

*COMOPT MODULE = destination

See section 3.6.6. There are two options which are detailed in the following subsections.

- Temporary *EAM file
  This file exists only for the duration of a task (interactive or batch operation). When a task terminates the object modules stored in it are lost. They must be processed further before that point in time, if desired (e.g. linked with the linkage editor $TSOSLNK), or be stored in a library (e.g. by the LMS or LMR programs).
- Library (LMS)
  This is always a library created according to LMS [13] conventions. A precondition is that the access method PLAM is integrated in the operating system.

Fig. 3-2 Data flow for object modules

Caution

If two object modules are generated, these always belong together and must be linked jointly.

3.7.1 Object module to *EAM file

If no entry is specified for storing the object modules or if the entry

```
*COMOPT MODULE = *
```

is specified, the object modules are added to the temporary *EAM file from which they may be further processed.

The names of the object modules are formed from the first entry name of the compiled external procedure (first name of the PROCEDURE statement). Further details regarding the formation of names are given in section 4.6.

The contents of the *EAM file are lost at end of task. Further processing should therefore be done beforehand. Generally this will be the following:

– Linking the modules into a program

  In the simplest case the object modules are linked into a program by the $TSOSLNK linkage editor (see [3] or, as of BS2000 V8.0, [12]). The control statement

INCLUDE *	causes all object modules
INCLUDE a,*	causes the object module a
INCLUDE (a,...),*	causes the object modules a

  of the *EAM file to be included in the linking process. See also chapter 4. The following example will illustrate this point.

```
/ REMARK " ..................................... COMPILATION"
/         EXEC $PLI1
               *COMOPT SOURCE=EXAMPLE1,
               *COMOPT MODULE=*,
               END
/ REMARK " ....................................... LINKAGE"
/         EXEC $TSOSLNK
               PROGRAM EXAMPLE1,FILENAM=OBJECT,MAP=N
               INCLUDE *
               END
/ REMARK " ..................................... START RUN"
          EXEC OBJECT
```

  The external procedure in file EXAMPLE1 is compiled and the generated object modules are entered in the *EAM file. The linkage editor links all modules of the *EAM file into load module EXAMPLE1 which is stored in the file OBJECT.

– Adding the module to a library

  If the module is to be retained for later processing, it must be included in a library using program $LMR [3] or program $LMS [13]. This is illustrated in the following example.

```
/ REMARK " ..................................... COMPILATION"
/        EXEC $PLI1
              *COMOPT SOURCE=EXAMPLE1,
              *COMOPT MODULE=*
              END
/ REMARK " ..................................... INCLUSION"
/        EXEC $LMR
              MODLIB  LIBRARYONE
              COPYALL SOURCE=*
              END
/ REMARK " ..................................... LINKAGE"
/        EXEC $TSOSLNK
              PROGRAM EXAMPLE1,FILENAM=OBJECT,MAP=N
              INCLUDE  *,LIBRARYONE
              END
/ REMARK " ..................................... START RUN"
/        EXEC OBJECT
```

The external procedure in file EXAMPLE1 is compiled and the generated object modules are added to the *EAM file. Subsequently all object modules of the *EAM file are added to the LMR library LIBRARYONE. The linkage editor links all object modules contained in the LIBRARYONE library into one load module.

3.7.2 Object module to LMS library

If object modules must be retained, it is advisable to have them added to an LMS library by the compiler straightaway. This can be effected using the control statement

```
*COMOPT MODULE = library (*)
*COMOPT MODULE = library (element)
```

A precondition is that the access method PLAM is integrated in the operating system. A complete description is given in section 3.6.6.

The names under which the object modules must be stored - the element names - are derived either from the first entry name of the compiled external procedure, i.e. the first name of the first PROCEDURE statements (if * is specified), or from the specified element name. More details about the construction of element names are to be found in section 3.6.6.

The following example will illustrate this point.

```
/ REMARK " ..................................... COMPILATION"
/         EXEC $PLI1
               *COMOPT SOURCE=LIBRARY(EXAMPLE1),
               *COMOPT MODULE=LIBRARY(*),
               *END
/ REMARK " ....................................... LINKAGE"
/         EXEC $TSOSLNK
               PROGRAM EXAMPLE1,FILENAM=OBJECT,MAP=N
               INCLUDE   (EXMPLE1,EXMPLE7),LIBRARY
               RESOLVE                    ,LIBRARY
               END
/ REMARK " ..................................... START RUN"
/         EXEC OBJECT
```

The external procedure in element EXAMPLE1 (type S) of the LMS library LIBRARY is compiled and the object module is stored in the same library (type R). The object modules EXMPLE1 and EXMPLE7 from the library and, if required, other object modules (RESOLVE) from the library are linked into one load module EXAMPLE1, which is stored in the file OBJECT.

3.7.3 Table of contents of a library

Which elements (object modules) are contained in an LMS library can be ascertained by way of the program $LMS [13]. This is illustrated in the following example.

```
/ EXEC $LMS
       LIB     BIBLIOTHEK
       PRT     (LST)                        see Note
       TOCR     *
       END
```

Note

```
PRT (LST)   to SYSLST     preset in batch mode
PRT (CON)   to SYSOUT                } only in inter-
PRT (BOTH)  to SYSLST and SYSOUT     } active mode
```

3.8 Listings of the compiler (LIST =)

All listings of the compiler are output to SYSLST. Every page has a header line which contains the following:

- Name of the compiler and its version number
- Name of the file containing the source procedure (for SOURCE = file) or "SYSDTA" (for SOURCE = *)
- Date and time-of-day compiled.
- Consecutive number of the page.

Which of the listings is to be output is determined by the control statements

*COMOPT LIST = output

For details, refer to chapter 3.

Some of the listings explained below include a reference to a source line, which always pertains to a line reference given in the source listing (see section 3.8.3). The line references are represented in all listings; they have a common format:

```
[Include] Line [:Statement]
```

Line: The line number is always present; it is formed as follows:

- Source in ISAM file: index from the file
- Source in SAM file: consecutive number
- COMOPT MARGINS=LINID(p,n): index with n characters as position p (see section 3.4.4)

If indices are used that are not numeric, then only the numeric part will be used. This may lead to ambiguous line numbers; a warning is issued.

Include: Lines that are inserted as a result of a %INCLUDE statement, are prefixed with a consecutive include number. The consecutive include number is listed in the list of include numbers (see section 3.8.5).

Statement: "statement" specifies the consecutive number of the statement within a line. The value ":1" is suppressed. The statement number is not listed in some listings.

In some listings the line references are output one under the other, in tabular form; in other listings they are packed, i.e. output without blanks.

3.8.1 Options (OPTIONS)

This listing shows all option values which have been set for the compiler run. The output of this listing can be controlled as follows:

```
*COMOPT LIST=OPTIONS               Listing of control statements
           =NOOPTIONS              No listing
```

```
COMPILER-OPTIONS USED

  STORAGE = (STACK(16,4),AREA(16,16,975))
  LIST    = (NOESD,NOTERMINAL,NOSUMMARY,OPTIONS,SAVLST,NOMAP,
             NEST,IREF,NOXREF,SOURCE,NOINSOURCE,NOAGGREGATE,OFFSET,NOASSM,
             NOOUTTEXT,NOLINECNT)
  FORMAT  = (TERMINAL(0,80),PRINTER(64,72),ENGLISH)
  MESSAGE = NOSYSLST
  SOURCE  = EXAMP21
  MARGINS = (TEXT(2,72),PAD,NOLINID,NOASACNTRL,GAMKEY(0,0),CHAR60,
             NOSAVMAC)
  DIAGNOST= (NOTERMINAL,NOSAVLST,WARNING)
  COMLIB  = NO
  OBJECT  = (ERROR(32767),ABORT(500),OUT)
  OPTIONS = (NOISO,NOMAIN,NOINTERRUPT,NOMACRO,NOXS,NOBITPTR)
  OPTIMIZE= (NOTIME,NOOVERLAP,NOENABLING,NOREORDER)
  DEBUG   = (NOSTMT,NOPROCTRACE,NOLABTRACE,NOCALLTRACE,NOGOTOTRACE,
             NORETURNTRACE,NOBREAKPOINT)
  SYMTEST = MAP
  MODULE  = *
```

Fig. 3-3 Sample options listing of the compiler (detail)

3.8.2 Preprocessor (INSOURCE)

Those texts are listed which were used as input data by the preprocessor. These are:

- one listing for the source text according to SOURCE =
- and one listing for each Include text inserted.

Each of these listings begins on a new page. The output of the listings can be controlled as follows:

```
*COMOPT LIST = INSOURCE              listing output
             = NOINSOURCE            no listing
```

These listings have the same format as the source listing (see 3.8.3), without the comment continuation and nesting level codes.

3.8.3 Source listing (SOURCE, EXPAND)

The source listings contains the source text to be compiled, line numbering, and any further information. The output of the source listing can be controlled as follows:

```
*COMOPT LIST = EXPAND                source listing with Include texts
             = SOURCE                source listing without Include texts
             = NOSOURCE              no source listing

*COMOPT LIST = NEST                  nesting level is shown
             = NONEST                block nesting level is not shown

*COMOPT LIST = NOLINECNT             line index without leading zeros
             = LINECNT (EDOR)        line index according to EDOR
             = LINECNT (EDT)         line index according to EDT
             = LINECNT (PRINT)       line index as for PRINT

*COMOPT MARGINS = TEXT  (a, e)       source text
        MARGINS = LINID (a, e)       index
        MARGINS = PAD                pad source texts with blanks

*COMOPT LIST = OUTTEXT (c)           source text leader and trailer
                                     + framing if applicable
             = NOOUTTEXT (c)         framing if applicable
```

The source listing may consist of the following columns as a maximum:

line reference	Include number index (or consecutive no.)
qualifier	* (comment continuation) nesting level (or empty)
source line	source text leader (optional) frame character left (optional) source text frame character right (optional) source text trailer (optional)

Descriptions of each column follow:

- Line reference
  The way line numbers are shown depends on the LIST = LINECNT control statement and the input medium or on the MARGINS = LINID (a, e) control statement. If LINID is used, the following discussion is subject to the same rules as apply to ISAM files.

NOLINECNT	(default):
ISAM file	Include number index without leading zeros

otherwise	Include number consecutive number

LINECNT:

ISAM file	Include number index (according to EDOR/EDT/PRINT)
otherwise	Include number consecutive number (according to EDOR/EDT/PRINT)

The line reference has the following format:

[Include] Line

– Include
The Include number is shown only if the source line comes from an Include text; otherwise it is empty. The Include number is a consecutive number which is assigned to the Include texts being used (see also 3.8.5).

– Line
This is the index from the ISAM file or a consecutive number for SAM files or according to the MARGINS = LINID (a, b) control statement. However, a maximum of 8 digits are used. Characters other than digits are not accepted. The way the index is shown depends on the LIST = LINECNT (a) control statement as follows:

NOLINECNT	without leading zeros
LINECNT (EDOR)	without trailing zeros
LINECNT (EDT)	period between the 4th and 5th digit, without leading and trailing zeros
LINECNT (PRINT)	with all zeros (same as for PRINT command)

- Qualifier
Between the line numbering and source lines, there are two qualifiers which refer to source line structure:

*	This line continues a comment from the previous line.
Nesting level	A number indicates the static nesting of blocks and DO groups. No number is shown for the top level (0). The number of the first nesting level is 1, etc. The line which follows the opening PROCEDURE, BEGIN, DO or SELECT statement has a level number incremented by 1. The corresponding END statement resets the level by 1 so that the line following the END statement contains the decremented level number.

Both qualifiers can be suppressed by *COMOPT LIST=NONEST.

```
            PROCEDURE  OPTIONS(MAIN);

            DCL X         DIM(10,10);
            DCL (I,J);

            B: PROCEDURE;
1              PUT SKIP(2);
1              DO I = 10 TO 10;
2                 DO J = 1 TO 10 BY 1;
3                    X(I,J) = I*10 + J;
3                    END;
2                 BEGIN;
3                    DO I = 1 TO 10;
4                       PUT SKIP;
4                       DO J= 1 TO 10;
5                          PUT EDIT (X(I,J)) (F(5));
5                          END;
4                       END;
3                    SELECT (I);
4                       WHEN (4) PUT SKIP LIST ('4');
4                       OTHER;
4                       END;
3                    /* A COMMENT EXTENDING
3                       OVER 2 LINES */
3                     END;
2                 END;
1              END;
            END;
```

Fig. 3-4 Example of a source procedure listing with nesting level indicators

- Source line
  From the source line supplied by the input medium, source text from column 'a' through column 'e' can be extracted by the control statement MARGINS = TEXT (a, e); only this text will be compiled. The OUTTEXT control statement can be used if you wish to include the texts before column 'a' (leader) and after column 'e' (trailer) in the printout. As well, the character (c) supplied in OUTTEXT (c) can be inserted preceding and following the source text on every line. The source line can consist of the following columns:

  – Leader, trailer
    A leader is created based on the control statement MARGINS = TEXT (a,e) if 'a' is greater than 1.

    A trailer is created if there are any characters following column 'e'. Both appear only if the control statement LIST = OUTTEXT has been specified.

Leader and trailer are no longer available after a preprocessor run; in this case, they do not appear in the source listing.

– Frame character
Frame character 'c' is set before and after the source text when specified in LIST = OUTTEXT (c) or LIST = NOOUTTEXT (c). For example, OUTTEXT (") shows how many blanks exist at the beginning and end of the source text.

If the source text is of variable length, the right-hand frame character may vary its position. If you wish to have it always in the same position, you need an additional control statement: MARGINS = PAD. Thus, the effect of MARGINS = PAD, LIST = OUTTEXT (|) would be to frame both sides of the source text by a vertical bar.

– Source text
This is the text which constitutes the PL/I procedure and will be compiled. Blanks at the beginning and end of the text can be made visible by frame characters (see above).

3.8.4 External names (ESD)

This listing contains all external names of the procedure which were either declared in the source or generated by the compiler. It supplies information on the external (linkable) names of the procedure, which are used to associate the procedure with other procedures, runtime procedures, and foreign procedures. The ESD information listed corresponds to the ESD records which are generated in the process of compilation.

CSECT/COMMON sections are built as follows:

```
Type/Level-1 item           Number of ESID nos. assigned
EXTERNAL VARIABLE                      1
EXTERNAL VARIABLE INIT                 2
FILE EXTERNAL CONSTANT                 2
ENTRY EXTERNAL CONSTANT                1
```

The output of this listing can be controlled as follows:

```
*COMOPT LIST = ESD                   listing output
             = NOESD                 no listings
```

The system prints a listing for the code module and as far as generated, a listing for the static module.

```
LIST OF EXTERNAL NAMES IN THE CODE MODULE

  NAME      TYP     MA   ADR    LNG/ESID
BH$384     CSECT   04  000000  000238
BH$384@6   EXTERN  00  000000  404040
P$3#00##   EXTERN  00  000000  404040
EXTINIT    COMMON  00  000000  000004
EXTSTAT    COMMON  00  000000  000004
EINGANG    VKONST  00  000000  404040
P$START#   ENTRY   00  000074  000001
EINGNG1    ENTRY   00  000004  000001
EINGNG2    ENTRY   00  000124  000001
```

```
LIST OF EXTERNAL NAMES IN THE STATIC MODULE

  NAME      TYP     MA   ADR    LNG/ESID
BH$384@6   CSECT   00  000000  000008
BH$384     EXTERN  00  000000  404040
EXTINIT    COMMON  00  000000  000008
EXTNIT     CSECT   00  000008  000008
EXTSTAT    COMMON  00  000000  000008
```

Fig. 3-5 Listing of external names in the code module and static module (example)

The meaning of each column is as follows:

- NAME
  External name declared in the procedure or generated by the compiler. Names which are too long are reduced to the first 4 and last 3 characters or to the first 8 characters (see section 4.6).
- TYP
  The type of external name:

CSECT	control section
ENTRY	link address
EXTERN	external link address
COMMON	common dummy section
VKONST	external link address
CU	compilation unit; only if *COMOPT SYMTEST ≙ NO

- MA

Characteristics:	00	no entry
	04	write-protected CSECT

- ADR
  Address relative to the module. Only for entries based on a statement ENTRY unequal to 0.

LNG/ESID	
for CSECT, COMMON	length in bytes
for ENTRY	ESID number of the CSECT
for CU	1st byte: total number of modules 2nd byte: empty 3rd byte: consecutive no. of the module
rest	blanks (X'404040')

3.8.5 Include texts (IREF)

This listing shows the Include texts called in the procedure and the file from which they were obtained. The output of this listing can be controlled as follows:

```
*COMOPT LIST = IREF                  listing output
             = NOIREF                no listing
```

```
                I N C L U D E - R E F E R E N C E S

SOURCE-REF.   # LEVEL FILENAME                  MEMBERNAME          VERSION

      5       1   1   $PLI1.SRC.INCL           ALLE$EXT            A01
      6       2   1   $PLI1.SRC.INCL           BLOCK000            002
      7       3   1   $PLI1.SRC.INCL           LABEL000            015
      8       4   1   $PLI1.SRC.INCL           SYMBOL00            B01
      9       5   1   $PLI1.SRC.INCL           REFERNCE            C00
     10       6   1   $PLI1.SRC.INCL           TOKEN000            C00
     11       7   1   $PLI1.SRC.INCL           ARRAY000            003
     12       8   1   $PLI1.SRC.INCL           CROSSNCE            003
     13       9   1   $PLI1.SRC.INCL           NODES000            001
     14      10   1   $PLI1.SRC.INCL           LIST0000            002
     15      11   1   $PLI1.SRC.INCL           DCL$TYPET           A02
```

Fig. 3-6 Sample listing of Include texts (detail)

The meaning of each column is as follows:

- SOURCE REF.
  This is the number of the source line and, where applicable, the number of the Include text line that contains the INCLUDE statement.
- #
  Consecutive numbering of the Include texts in the order they were inserted in the procedure.
- LEVEL
  The nesting level which was supplied in the INCLUDE statement. A ’1’ means that the statement is in the source procedure; any value greater than ’1’ means that the statement is in an Include text.
- FILENAME
  Name of the file from which the Include text was obtained.
- MEMBERNAME
  Name of the Include text by which it was called in the INCLUDE statement. Names that are too long are reduced to the first 4 and last 3 characters. Names which are too short are padded with zeros to a length of 7 characters.
- VERSION
  Version designation of the library element (member) that contains the Include text. The version designation may be up to 3 characters in length.

3.8.6 Cross-reference listing

The cross-reference listing contains for each identifier (name) of the external procedure the complete attribute set and a listing of all line numbers where the particular identifier is explicitly or implicitly used. The listing consists of two parts. The first shows all identifiers which are referenced in the procedure. The second contains all identifiers which although declared are not referenced. Both parts are sorted in alphabetic order by identifiers. The output of this listing can be controlled as follows:

```
*COMOPT LIST = FULLXREF              Referenced and non-referenced
                                     identifiers
             = SHRTXREF              Referenced identifiers only
             = NOXREF                No cross-reference listing
```

As far as the items in this listing have corresponding items in PL/I, the notation of the language has been used wherever possible. The following special notations are used in addition:

@ This symbol represents an item (reference or expression) which the system was unable to identify.

< > An item enclosed with < > (a reference) has been truncated to the right because it was too long.

```
          C R O S S - R E F E R E N C E - T A B L E   -   REFERENCED IDENTIFIERS  -

IDENTIFIER      DIMENSION DATATYPE                          STORAGE             REFERENCES


$PRINT_CPDCL              BIT (1)                 ALIG      MEM-2 <ALLE_EXT_ST> DCL 1-1400 33150000

$ROOT                     POINTER                 ALIG      MEM-2 <ALLE_EXT_ST> DCL 1-8800 13150000

$SHRTXREF                 BIT (1)                 ALIG      MEM-2 <ALLE_EXT_ST> DCL 1-1100 13220000

ADDR                      BUILTIN                                               DCL 11250000 13040000

ALGOL                     BIT (1)                 UNAL      MEM-8 (SYMBOL)      DCL INIT 4-165000 75490000

ALIGNED                   BIT (1)                 UNAL      MEM-4 (SYMBOL)      DCL INIT 4-77000 641500000

ALLOCATED                 BIT (1)                 UNAL      MEM-3 (SYMBOL)      DCL INIT 4-4000 33240000 33330

ALT_ZEILENNUMMER          CHAR (13)          VAR  UNAL      AUTOMATIC           DCL INIT 51270000 67080000 811

ANFANG                    FIXED BIN (15)          ALIG      AUTOMATIC           DCL INIT 51340000 53210000 624
                                                                                76130000

ANFANG                    FIXED BIN (15)          ALIG      AUTOMATIC           DCL 83070000 83100000 83160000

ANFANGSPOSITION           FIXED BIN (15)          ALIG      PARAMETER           DCL 83060000 83030000 83100000

ANFANGSZEIGER             POINTER                 ALIG      AUTOMATIC           DCL INIT 51100000 53010000 530
                                                                                53160000 53190000 53240000 533
                                                                                53410000 53450000 68060000

AREA                      BIT (1)                 UNAL      MEM-4 (SYMBOL)      DCL INIT 4-61000 63360000

ARITHMETIK_ERMITTELN      ENTRY (                       INT CONSTANT            DCL 71030000 63110000 63150000
                           POINTER ALIG)

ARRAY                     BIT (1)                 UNAL      MEM-3 (LABEL)       DCL INIT 3-500 62330000

ARRAY                     POINTER                 ALIG      MEM-2 (SYMBOL)      DCL INIT 4-21000 62090000 6212

ASSEMBLER                 BIT (1)                 UNAL      MEM-6 (SYMBOL)      DCL INIT 4-161000 75470000
```

Fig. 3-7 Sample cross-reference listing (detail)

Related attributes are shown in one column. The meaning of each column is as follows:

- IDENTIFIER
  The identifiers start in this column. They must always be supplied at full length.
- DIMENSION
  This column contains all dimensions which apply to the identifier, including those (if any) inherited from a higher-order (containing) structure.
- DATATYPE
  This column shows the attributes which determine the data type. The attribute VAR and CPLX are right-justified.

Together with ENTRY are shown the data attributes of the parameters as well as RETURNS with the associated data attributes. An indent by 1 position indicates the nesting. Dimensional information on the parameters and the RETURNS value can be found in the DIMENSION column.

The PICTURE attribute, after PIC, shows the parenthesized length of the variables in storage (without V, F, K). Furthermore, the attribute set (CHAR, FIXED, FLOAT) associated with the mask and the precision, if applicable, are shown.

"(USER)", when added to the CONDITION attribute, shows that this is a user rather than system condition.

- Alignment (no header)
  This column provides - as far as relevant - ALIG (for ALIGNED) or UNAL (for UNALIGNED).
- Scope (no header)
  This column contains EXT or INT where applicable.
- STORAGE
  This column provides - as far as relevant - the storage class. The reference is included for BASED and DEFINED.

  For the members of the structure, this column shows "MEM-n (haupt)", where ’n’ is the normalized level no. and ’haupt’ is the identifier of the main structure.
- REFERENCES
  This column contains the numbers of the lines where the identifiers were used explicitly or implicitly. If an identifier is used more than once on a given line, the line number is usually listed just once.

  Depending on how an identifier was declared, this column begins with:

  DCL if by DECLARE statement or if by statement prefix (mask, format, entry) or if by parameter on an entry;

  `+++` if contextually (area, file, pointer, condition, builtin function);

  `---` if implicitly.

  "INIT" follows in cases of initialization. The first line number is generally the number of the line containing the declaration.

  The line reference is listed in the same format as in the source listing. The Include number precedes the line number and is separated from it by a hyphen in this listing.

3.8.7 Structure lengths (AGGREGATE)

The structure length table is a listing of all structures of the external procedure as well as showing the decimal and hexadecimal length of the storage space required by the structures and their contained substructures, arrays, and elementary members. The output of this listing can be controlled as follows:

```
*COMOPT LIST = AGGREGATE             Listing output
             = NOAGGREGATE           No listing
```

The OFFSET and length entries include the following information also:

- for elements with the VARYING attribute, the two bytes for the current length,
- for arrays and structures, the fillers required for alignment and internal representation.

```
 8     P: PROCEDURE;
 9  1
10  1     DCL 1 STRUKTUR1          ALIGNED,
11  1           2 BIT1         BIT(3),
12  1           2 BIT2         BIT(6),
13  1           2 UNTER             DIM(3),
14  1             3 CHAR       CHAR(3),
15  1             3 VARCHAR    CHAR(5) VAR,
16  1             3 VARBIT     BIT(5)  VAR,
17  1           2 FIXED        FIXED BIN;
18  1
19  1     DCL 1 STRUKTUR2          BASED,
20  1           2 DIM            FIXED BIN,
21  1           2 LAENGE         FIXED BIN,
22  1           2 UNTER1 ,
23  1             3 ZEICHEN      CHAR(3) DIM(5 REFER (DIM)),
24  1             3 BIT          BIT(5 REFER (LAENGE)),
25  1           2 UNTER2,
26  1             3 ZEICHEN      CHAR(P1),
27  1             3 BIT          BIT(P2),
28  1           2 ZEIGER         POINTER;
29  1
30  1     DCL 1 STRUKTUR3,
31  1           2 FEST           FIXED DEC,
32  1           2 UNTER1               DIM(3),
33  1             3 CHARVAR      CHAR(5) VAR,
34  1             3 BITS         BIT(3),
35  1           2 UNTER2               DIM(P1),
36  1             3 CHAR         CHAR(5),
37  1             3 BIT          BIT(3) DIM(3);
38  1
39  1      END;
40
```

Fig. 3-8 Sample program for Figure 3-9

```
                                S T R U C T U R E   L E N G T H   T A B L E
                                                    OFFSET          ELEMENT LENGTH      TOTAL LENGTH
SOURCE-REF   LEVEL IDENTIFIER   DIMENSION        DEC    HEXDEC      DEC    HEXDEC      DEC    HEXDEC

        10   1 STRUKTUR1                          0       0                              52       34
               2 BIT1                             0       0          0.3      0.3
               2 BIT2                             1       1          0.6      0.6
               2 UNTER                    3       2       2           15        F       48       30
                 3 CHAR                   3       2       2            3        3    INTERLEAVED
                 3 VARCHAR                3       6       6            7        7    INTERLEAVED
                 3 VARBIT                 3      14       E          2.5      2.5    INTERLEAVED
               2 FIXED                            50      32            2        2

        19   1 STRUKTUR2                          0       0                           VARIABLE
               2 DIM                              0       0            2        2
               2 LAENGE                           2       2            2        2
               2 UNTER1                           4       4                           VARIABLE
                 3 ZEICHEN              @         4       4            3        3       VARIABLE
                 3 BIT                          VARIABLE             VARIABLE
               2 UNTER2                         VARIABLE                             VARIABLE
                 3 ZEICHEN                      VARIABLE             VARIABLE
                 3 BIT                          VARIABLE             VARIABLE
               2 ZEIGER                         VARIABLE           4        4

        30   1 STRUKTUR3                          0       0                           VARIABLE
               2 FEST                             0       0            3        3
               2 UNTER1                   3       3       3          7.3      7.3      24       18
                 3 CHARVAR                3       3       3            7        7    INTERLEAVED
                 3 BITS                   3      10       A          0.3      0.3    INTERLEAVED
               2 UNTER2                   @      27      1B          6.1      6.1       VARIABLE
                 3 CHAR                   @      27      1B            5        5    INTERLEAVED
                 3 BIT                  @,3      32      20          0.3      0.3    INTERLEAVED
```

Fig. 3-9 Structure length table (for sample program in Fig. 3-8)

Some general information on the way items are shown on the listing follows:

- Bytes.bits
  Values are shown in bytes of 8 bits each. If a value contains any bits beyond the bytes, these bits follow the byte entry, separated by a period.
- VARIABLE
  This word indicates that a value could not be determined at compile time.
- INTERLEAVED
  This word means that the particular elements are interleaved with other elements so that they are not connected in storage and consequently do not add up to a connected total length.
- @
  This symbol indicates that a value could not be determined at compile time.

The meaning of each column in this listing is as follows:

- SOURCE-REF.
  This is the number of the source line where the structure was declared. If it is an Include text, it is preceded with the number of the Include text to the left (see section 3.8).
- LEVEL IDENTIFIER
  These are the normalized level no. and the identifier. They are indented 2 positions for the second and each subsequent level. Identifiers which are too long are truncated to the right.
- DIMENSION
  This is for each dimension, the number of elements, each separated by a comma. Those inherited from a containing structure are listed also. If the number of elements of a given dimension cannot be determined at compile time, the @ symbol appears.
- OFFSET
  This column shows the byte and bit, relative from the beginning of the main structure, where the substructure or structure element begins, counting from 0. If the value of the bit is 0, no information is displayed.

  This entry may also be interpreted as indicating how many bytes and bits precede the particular element in the main structure.
- ELEMENT LENGTH
  This column provides the following information:
  - If no dimension exists, this column remains empty for structures whereas it contains the length for the elementary members of a structure.
  - If one dimension exists, this column contains the length of the array element both for structures and elementary members of a structure. Dimensions in lower-order (contained) structure members are included but the own dimension is not. This length multiplied by the own dimension is the total storage requirement, which (provided it is connected and not INTERLEAVED) is shown in the TOTAL LENGTH column.
- TOTAL LENGTH
  This column provides the total length for structures and arrays. For non-connected storage space, the word "INTERLEAVED" appears here. Such variables are subject to certain restrictions in PL/I. The storage requirement should be obtained from the containing structure or it should be determined by means of the DIMENSION and ELEMENT LENGTH columns or using the OFFSET column.

3.8.8 Storage occupancy (MAP)

This listing contains a sublisting for each block of the external procedure. Each of these sublistings consists of the following:

- a header with block type information and
- a listing of constants and variables of the block and storage arrangement information.

The number of the source line on which the item is declared and the declared name complete the listing.

The output of this listing can be controlled as follows:

```
*COMOPT LIST = MAP                    Listing output
             = NOMAP                  No listing
```

```
                    M A P - L I S T

  SOURCE-REF.       TYPE          ADDR     OFFSET  NAME

             0      ROOTBLOCK

             2      ENTRY CONST      0            BH_388


             2      EXT PROCEDURE                 BH_388
             6      ESD #            A            O$P
             9      AUTO            98            STRUKTUR
                      MEMBER                0     BIT1   IN STRUKTUR
                      MEMBER                0(3)  BIT2   IN STRUKTUR
                      MEMBER                2     UNTER  IN STRUKTUR
                      MEMBER                2     CHAR   IN UNTER IN STRUKTUR
                      MEMBER                9     BIT    IN UNTER IN STRUKTUR
                      MEMBER               20     FIXED  IN STRUKTUR
            18      ENTRY CONST     1AC           P
            27      AUTO             80           Z
            26      ESD #             3           SYSPRINT


            18      INT PROCEDURE                 P

            20      STATIC(FILE)     18           DATEI
            21      STATIC            8           STATISCH
            24      AUTO             80           ZEIGER
  1-        60      CONSTANT        10C           TOKEN_NODE
            35      AUTO             90           KOPFZEILE
            38      LABEL CONST     4B4           SCHLEIFE
            41      ENTRY CONST     4D4           SCHREIBEN


            28      ON UNIT                       O.ENDFILE*1


            41      INT PROCEDURE   360           SCHREIBEN( QUICK ) OWNER: P


            44      BEGIN BLOCK     3A8            ( QUICK ) OWNER: P
```

Fig. 3-10 Storage map listing (sample)

The meaning of each column of this listing is as follows:

- SOURCE-REF.
  This column contains the number of the source line and if applicable the number of the Include text where the block or the variable or constant was declared (see section 3.8).
- TYPE
  This column contains as its header the type of block or else the data type. For block type, the following information is shown:

  – ROOTBLOCK (this is always the first block)
  – EXT PROCEDURE
  – INT PROCEDURE
  – BEGIN BLOCK
  – ON UNIT

  For data type, the following information is listed:

  – LABEL CONST scalar label constants
  – ENTRY CONST entry constants
  – CONSTANT label arrays, internal STATIC constants, unnamed format constants
  – STATIC internal STATIC variables
  – STATIC (CTL) anchor for CONTROLLED variables
  – STATIC (FILE) internal file constants
  – AUTO AUTOMATIC variables
  – ESD # ESD number for external variables and external entries
  – MEMBER member of a structure
- ADDR
  The information shown in this column depends on the TYPE column. It may contain the following values:

  – For INT PROCEDURE or BEGIN BLOCK, which according to the NAME column are of the QUICK type:
    starting address of the activation record relative to the beginning of the activation record of the father.

  – For LABEL CONST, ENTRY CONST, CONSTANT:
    starting address in the constant section of the code module.

  – For STATIC, STATIC (CTL), STATIC (FILE):
    starting address in the static module.

– For AUTO:
starting address relative to the beginning of the activation record.

– For ESD #:
ESD number of the external name.

This column is empty in all other cases.

- OFFSET
This column is only used for the MEMBER type and contains the byte address related to the beginning of the main structure (in hexadecimal) as well as the bit address unless the value of the latter is zero.

- NAME
This column contains the names declared in the particular case. It remains empty for the BEGIN BLOCK and ON UNIT types since there are no entry names for those types.

  For blocks which as "QUICK" blocks share the activation record of the father block, the NAME column contains an additional entry pointing to the father block. If the father block is of the PROCEDURE type, the additional entry is as follows:

  (QUICK) OWNER:procedure name

  If the father block is of the BEGIN BLOCK type, the additional entry is this:

  (QUICK) OWNER:BEGIN BLOCK IN ZEILE n

  where ’n’ is the number of the source line where the block was declared (see LINE column).

3.8.9 Offset listing (OFFSET)

This listing shows the relationship between line numbers and addresses relative to the beginning of the procedure. For an address supplied in a runtime message, the user can thus determine the line number of the statement which caused the message.

3.8.10 Assembly code (ASSM)

This listing presents the machine code generated and its retranslation in assembler format. Therefore, knowledge of the assembler language is needed to read this listing properly. The output of this listing can be controlled as follows:

```
*COMOPT LIST = ASSM                          Whole procedures
             = ASSM ({beginning, end};...)   Parts of procedure
             = NOASSM                        No listing
```

For 'beginning' and 'end', a line must be specified in accordance with the source listing.

The meaning of each column is as follows:

- QUELL-BEZUG (SOURCE REF.)
  This column shows in decimal notation the line number as it appears in the source listing, the number of the Include text if any, and the consecutive number of the statement within the line.
- ADR INTERNCODE
  This is in hexadecimal notation the address of the machine instruction and the instruction itself.
- EXTERNCODE
  This column shows a retranslation of machine instructions in assembler code format. The comments refer to the source program names. Additional comments are printed for optimized loops.
- COMMENTS
  These consist of references to source names. For statements processed by the loop optimizer routine, the following information is displayed:

  'advance invariant code'
  'initialize induction variable'
  'initialize increment'
  'increment induction variable'

If only part of the assembler code listing is requested (*COMOPT LIST = ASSM (a,e)), the listing is combined with the offset listing (see 3.8.9); that is, those parts of the program whose assembler code is not supplied are output in the form of the offset listing, in which case there will be no offset listing in its own right.

```
                                OBJECTCODEPROTOKOLL DES CODE-MODULES
QUELL-BEZUG            ADR  INTERNCODE                  EXTERNCODE

                    000000                   BH$387    START O,READ
                                                       ENTRY P$START#
                                                       EXTRN BH$387@6
                                                       EXTRN P$3#00##
                                                       EXTRN P$PPREP#
                                                       EXTRN P$PTERM#
                                                       EXTRN P$PVL###
                                                          .
                                                          .
                                                          .
         5          000260 05 EF                       BALR  14,15
                    000262 D2 03 D04C D050             MVC   76(4,13),80(13)
                    000268 D5 03 D25C B13C             CLC   604(4,13),316(11)       MG,
                    00026E 47 20 A2C2                  BC    2,706(0,10)             C.1
                    000272 D2 04 D267 D248             MVC   615(5,13),584(13)       NO, NO
                    000278 D2 0E D26E D24D             MVC   622(15,13),589(13)      BZ, BZ
                    00027E D2 03 D27F D25C             MVC   639(4,13),604(13)       AMG, MG
                    000284 D2 01 D285 D260             MVC   645(2,13),608(13)       AME, ME
                    00028A D2 04 D289 D262             MVC   649(5,13),610(13)       APR, PR
                    000290 58 60 B038                  L     6,56(0,11)              LIST
                    000294 50 60 D0A0                  ST    6,160(0,13)
                    000272 D2 04 D267 D248             MVC   615(5,13),584(13)       NO, NO
                    000278 D2 0E D26E D24D             MVC   622(15,13),589(13)      BZ, BZ
                    00027E D2 03 D27F D25C             MVC   639(4,13),604(13)       AMG, MG
                    000284 D2 01 D285 D260             MVC   645(2,13),608(13)       AME, ME
                    00028A D2 04 D289 D262             MVC   649(5,13),610(13)       APR, PR
                    000290 58 60 B038                  L     6,56(0,11)              LIST
                    000294 50 60 D0A0                  ST    6,160(0,13)
                    000278 D2 0E D26E D24D             MVC   622(15,13),589(13)      BZ, BZ
                    00027E D2 03 D27F D25C             MVC   639(4,13),604(13)       AMG, MG
                    000284 D2 01 D285 D260             MVC   645(2,13),608(13)       AME, ME
                    00028A D2 04 D289 D262             MVC   649(5,13),610(13)       APR, PR
                    000290 58 60 B038                  L     6,56(0,11)              LIST
```

Fig. 3-11 Sample assembly code listing (detail)

3.8.11 Statistics (SUMMARY)

This listing shows how much virtual storage was used by the compiler and how often additional storage space was requested. See also the STORAGE control statement.

```
STATISTIK DER BELEGUNG DES VIRTUELLEN KERNSPEICHERS
PROZEDUR-STACK:    20 SEITEN; SYSTEM-AUFRUFE:     2 REQM,     0 RELM
STANDARD-AREA:     22 SEITEN; SYSTEM-AUFRUFE:     2 REQM
SPEICHERBEDARF:   219 SEITEN KLASSE 6   UND:    18 SEITEN KLASSE 5
```

Fig. 3-12 Compiler statistics (sample)

The meaning of each item in the summary is as follows:

Procedure stack: maximum number of memory pages (4K bytes) used in the procedure stack.

Standard area: maximum number of memory pages (4K bytes) used in the standard area.

Storage requirement:
maximum number of memory pages (4K) requested by the whole program (procedure stack, standard area, load module) in memory classes 5 and 6.

REQM number of memory requests to the system.

RELM number of memory releases by the system.

3.9 Diagnostic messages (DIAGNOST)

Diagnostic messages are information about the flow of compilation. According to importance, there are the following groups:

1. Fatal errors (compiler aborts immediately)
2. Severe errors (no code is generated)
3. Errors (compiler tries to recover)
4. Warnings (references to suspected errors)
5. Information (notes on optimization)

Some information messages include a note for which chapter 14.6 provides further information. There are:

– Information message no. 500
  The number of the out-line sequence is explained in chapter 14.6.

– Information message no. 503 The name of the out-line sequence is explained in chapter 14.6.

– Information message no. 504 The type of the out-line sequence for conversion is explained in chapter 14.6.

Information messages 500 through 504 appear only if some optimization is requested by *COMOPT OPTIMIZE=.

Which type of diagnostic messages occurred during the compiler and preprocessor runs is listed to SYSOUT. The diagnostic messages as such are listed to SYSLST.

The output of diagnostic messages can be controlled as follows:

```
*COMOPT DIAGNOST = SEVERE             messages of group 2 and above
                 = ERROR              messages of group 3 and above
                 = WARNING            messages of group 4 and above
                 = INFORMATION        messages of group 5

*COMOPT DIAGNOST = TERMINAL           also to terminal
                 = NOTERMINAL         only to SYSLST

*COMOPT DIAGNOST = SAVLST             additionally to file
                 = NOSAVLST           not to file

*COMOPT FORMAT   = DEUTSCH            messages in German
                 = ENGLISH            messages in English

*COMOPT OPTIMIZE =                    Capability for controlling
                                      information messages 500 through
                                      504 (see above)
```

```
 1    BH_39:                /*EXAMPLE OF DIAGNOSTIC LISTING*/
 2      PROCEDURE;
 3
 4      CALL LESEN;
 5      CALL SCHREIBEN;
 6
 7      DCL ANTON       CHAR(3) VARING;
 8      DCL BERTA       FIXED   BINARI;
 9
10      DCL DATEINAME  FILE STREAM  EXTERNAL;
11      PUT FILE(DATEINAME) SKIP(2);
12
13      BERTA = VERIFY(CHAR(BERTA),'+-0123456789');
14      ANTON = SEARCH(CHAR(ANTON),'0');
15
16      BERTA = VERIFY(CHAR(BERTA,'+-0123456789');
17      ANTON = SEARCH(CHAR('0',ANTON);
18
19      GOTO WEITER;
20      BERTA = 3;
21   WEITER:
22
23      DCL 1 STRUKTUR           ALIGNED,
24            2 BIT1         BIT(3),
25            2 UNTER            DIM(3),
26              3 VARCHAR    CHAR(5) VAR,
27              3 VARBIT     BIT(5) VAR,
28            2 FIXED        FIXED BIN INIT(1);
29
30       PUT LIST(STRUKTUR);
31
32
33       END;
```

Fig. 3-13 Procedure for the diagnostic messages for Fig. 3-14

```
      C O M P I L E R   D I A G N O S T I C   M E S S A G E S

SEVERE ERROR DIAGNOSTIC MESSAGES

+++++SEVERE ERROR NO 49
THE RIGHT-HAND SIDE OF THIS APPARENT ASSIGNMENT STATEMENT IS NOT AN EXPRESSION

     SOURCE REF.      SOURCE REF.      SOURCE REF.      SOURCE REF.      SOURCE REF.      SOURCE REF.
             16               17

ERROR DIAGNOSTIC MESSAGES

+++++ERROR NO 7
AN UNRECOGNIZABLE ATTRIBUTE HAS BEEN FOUND IN THE DECLARATION OF '.....'

     SOURCE REF.      '.....'                                  SOURCE REF.      '.....'
             7        'ANTON'                                           8       'BERTA'

+++++ERROR NO 64
THE UNDECLARED IDENTIFIER '.....' HAS BEEN USED AS AN ENTRY; IT HAS BEEN DECLARED AS AN EXTERNAL ENTRY
CONSTANT

     SOURCE REF.      '.....'                                  SOURCE REF.      '.....'
             4        'LESEN'                                           5       'SCHREIBEN'

WARNING DIAGNOSTIC MESSAGES

+++++WARNING NO 56 IN LINE 20
THIS STATEMENT CAN NEVER BE REACHED DURING EXECUTION DUE TO AN UNCONDITIONAL GOTO STATEMENT OR RETURN
STATEMENT IMMEDIATELY PRECEDING IT

+++++WARNING NO 234 IN LINE 14
AN ARITHMETIC VALUE HAS BEEN CONVERTED TO A STRING VALUE

+++++WARNING NO 304
THE EXTERNAL NAME '.....' IS TRUNCATED TO A CONCATENATION OF THE FIRST 4 AND THE LAST 3 CHARACTERS

     SOURCE REF.      '.....'                                  SOURCE REF.      '.....'
             10       'DATEINAME'                              ——/——            'SCHREIBEN'
```

Fig. 3-14 Diagnostic messages for the procedure in Fig. 3-13

The messages are shown in their order of importance (group 1 first), each group under its own heading. Within a given group, identical messages are united and the message text appears just once. The text is followed by the numbers of the source lines and the consecutive number of the statement within the line which caused the message, and finally the additional information for the message text.

The message texts are self-explanatory. Note that for executable statements, the message refers to the line in which the statement begins.

Note also that messages may be the consequences of other messages.

3.10 Format of the SAVLST file

The compiler control statements

```
*COMOPT LIST     = SAVLST
*COMOPT DIAGNOST = SAVLST
```

can be used if listings and messages of the compiler are to be additionally stored in a file. See on this sections 3.5.1 and 3.5.2.

The file must be an ISAM file and must have a key length of 8 bytes, with record format V. If no file exists, a new file is created, according to the following options:

```
/FILE SAVLST.PLI1.tsn,LINK=SAVLINK,SPACE=(3,3),FCBTYPE=ISAM,KEYPOS=5,      -
 KEYLEN=8,RECFORM=V,BLKSIZE=STD
```

At the end of the run, the name of the file is changed to:

```
SAVLST.PLI1.module-name
```

The records of the file have the following structure:

length entry - 4 bytes
key - 8 bytes
information - up to 255 bytes

The header lines at top of page are included in the file. Carriage control characters do not exist.

3.11 Use of the preprocessor

The preprocessor is a separate program, integrated with the PLI1 compiler. At compiler start time, you can issue a control statement if you wish to precede the compiler with a preprocessor run.

The preprocessor expects source text to be written in the PL/I programming language. Among the PL/I statements, there are statements for the preprocessor. These statements, whose syntax and semantics are very similar to those of PL/I statements, are processed by the preprocessor. The result of that processing is generally a replacement of text, often causing substantial changes to the source procedure. When the preprocessor has finished processing, the source procedure is transferred to an intermediate file, where most of the preprocessor statements are being removed from the source procedure. Only statements which control the printout of the compiler listing are still contained in the source procedure.

The following control statements for the PLI1 compiler are relevant to the preprocessor:

```
OPTIONS = MACRO                     With preprocessor run
        = NOMACRO                   Without preprocessor run

LIST    = INSOURCE                  Preprocessor input listing
        = NOINSOURCE                No preprocessor input listing

OBJECT  = MACRO                     Preprocessor run only; no compilation

COMLIB  = (library,...)             Libraries supplying the texts to be
                                    inserted

LIST    = IREF                      INCLUDE reference listing
        = NOIREF
```

Fig. 3-15 Data flow for the preprocessor run and the control statements

The preprocessor reads source text according to the SOURCE = source control statement. If the LIST = INSOURCE control statement is specified, an input listing is printed, followed by a listing of the Include files.

Some statements are first processed by the preprocessor and then passed on to the compiler in the modified source procedure. These are the statements which have a controlling effect on the compiler listing.

The result of the preprocessor run is a source text file which is free of preprocessor statements except those mentioned above. The only purpose of this file is to serve as input to the actual compiler run without any user intervention. The file may be created as follows:

a) The user, before calling the compiler, issues the following FILE command:

```
/FILE file-name,LINK=SAVMAC[,SPACE=...]
```

In this case, the preprocessor writes its output to the specified file.

b) The preprocessor creates an output file by the name of:

```
PLI1.SAVMAC.task-sequence-number,
```

using the tsn of the current task.

The preprocessor writes its output to a file which has the fixed file link name of SAVMAC. If the user has not assigned a file to that name, the preprocessor creates its own file according to the following command:

```
/FILE PLI1.SAVMAC.task-sequence-number
      LINK=SAVMAC,
      FCBTYPE=ISAM,BLKSIZE=STD,RECSIZE=136,
      RECFORM=V,KEYPOS=5,KEYLEN=8
```

This file may be printed e.g. by one of the following commands:

```
/PRINT PLI1.SAVMAC.task-sequence-number

/PRINT PLI1.SAVMAC.task-sequence-number
       STARTNO=17,ENDNO=124
```

In both cases, the preprocessor-generated file can be used again for subsequent compilations, after making any corrections that may be necessary, as immediate input to the compiler. Generally, OPTIONS = NOMACRO is required in these cases since all preprocessor statements have been resolved. MARGINS = SAVMAC should be supplied.

Should errors be encountered as the preprocessor processes the preprocessor statements, appropriate messages are issued which in every respect are similar to the messages of the compiler. Further compilation may be suppressed, e.g. if severe errors occur and OBJECT = was specified accordingly by the user.

```
/FILE DATEI,LINK=SAVMAC                    Intermediate file
/EXEC $PLI1                                Compiler start
      *COMOPT SOURCE=QUELLE,               Source entry
      *COMOPT COMLIB=(...),                With preprocessor
      *COMOPT OPTIONS=MACRO,
      *COMOPT LIST=INSOURCE,               Input listing
      *COMOPT LIST=IREF,                   Preprocessor only
      *COMOPT OBJECT=MACRO,                Compiler listing
      *COMOPT LIST=SOURCE,
```

Fig. 3-16 Commands and control statements concerning the preprocessor

All forms of input for the sources are identical with preprocessor or without; that is, the preprocessor accepts the same forms of input as the PLI1 compiler.

For the %INCLUDE statement, the text to be inserted may have to be supplied as a file or library. Also, the control statement

```
COMLIB = (library,...)
```

may be required to establish a relationship with particular libraries.

The records of the output file of the preprocessor have the following format:

Position	Contents
1 - 3	Number of the Include text
4 - 11	Original source line number according to input file for the preprocessor: index from an ISAM file or consecutive no. for SAM file or in accordance with MARGINS = LINID (a,b).
12	Carriage control character from the input file; otherwise blank.
13	Blank
14 - 15	Level depth of replacement by the preprocessor or blanks if no replacement took place.
16	Blank
17 - 24	Source line as extracted by MARGINS = TEXT (a,b) and/or modified by the preprocessor. If the text area contains more than 108 characters, additional records which have the same original line number are created in the output file. If the text area is less than 108 characters, the output line is padded to 108 characters and blanks.

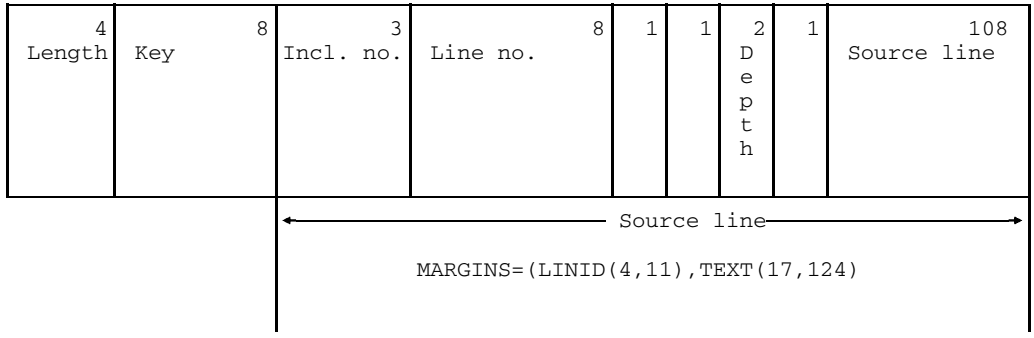


Fig. 3-17 Data record and source line as stored by the preprocessor in an intermediate file for the compiler

Note that the record is preceded with an 8-byte key and a 4-byte length field. The key is incremented in steps of 1, starting with 00000001.

3.12 Description of the operands of the SDF command START-PLI1-COMPILER

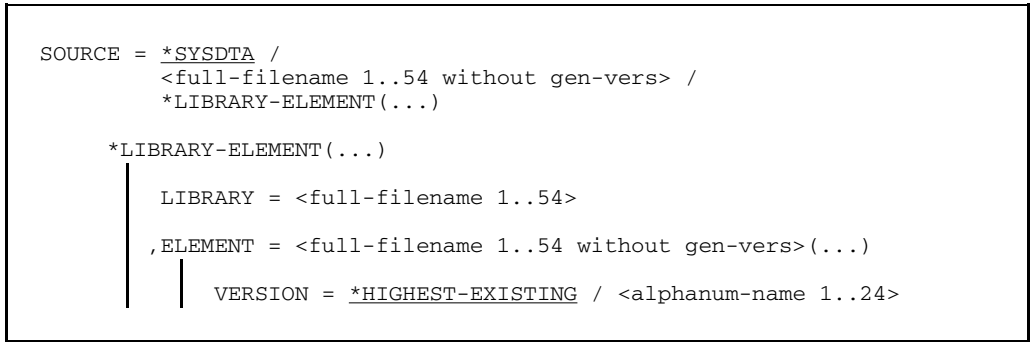
3.12.1 Overview of the operands

Name of the operand	Purpose
SOURCE	Defines the input source for the source program
INCLUDE-LIBRARY	Assigns the PLAM library in which the %INCLUDE elements are stored
SOURCE-PROPERTIES	Describes the input format of the source program
PREPROCESSING	Calls the preprocessor
COMPILER-ACTION	Performs a partial execution of the compilation run and controls certain properties of the generated objects
MODULE-LIBRARY	Specifies the name and output destination of the generated objects
LISTING	Controls listing output
TEST-SUPPORT	Controls the program testing tools
OPTIMIZATION	Controls optimization
COMPILER-TERMINATION	Defines a point at which the compilation run is to be terminated
MONJV	Monitors the compilation run using job variables
LANGUAGE	Selects the language (English or German) for output of compiler messages

3.12.2 Description of the individual operands

SOURCE operand

This operand determines whether the source program is to be read from SYSDTA, from a cataloged file, or from a PLAM library.



SOURCE = *SYSDTA
Input is from the SYSDTA system file by default; in interactive mode SYSDTA is assigned to the display terminal. Using the ASSIGN-SYSDTA command it is possible to assign SYSDTA to a cataloged file or a PLAM library element before the compiler is invoked.

SOURCE = <full-filename 1..54 without gen-vers>
The entry <full-filename> assigns a cataloged file as the input source.

SOURCE = *LIBRARY-ELEMENT(...)
This parameter is used to specify a PLAM library and an element stored in it.

LIBRARY = <full-filename 1..54>
The entry <full-filename> assigns a PLAM library as the input source.

ELEMENT = <full-filename 1..54 without gen-vers>(...)
<full-filename> is the fully-qualified name of an element from the specified PLAM library. The element must be type S (=source).

VERSION = *HIGHEST-EXISTING
If the element entry does not include a version identifier, the compiler will use the highest version.

VERSION = <alphanum-name 1..24>
The compiler uses the version specified by <alphanum-name>.

INCLUDE-LIBRARY operand

This operand determines the library or libraries from which the %INCLUDE elements are to be input.

```
INCLUDE-LIBRARY = *NONE / list-poss: <full-filename 1..54> /
```

INCLUDE-LIBRARY = *NONE
No %INCLUDE elements are to be read.

INCLUDE-LIBRARY = <full-filename 1..54>
<full-filename> is the file or link name of one or more PLAM libraries. The PLAM libraries are searched for %INCLUDE elements in the order in which they are specified. Finally, the system catalog ($TSOS) is also searched. <full-filename> is interpreted first as a link name, then as a file name.

SOURCE-PROPERTIES operand

This operand describes the input format of the source program.

```
SOURCE-PROPERTIES = STD / PARAMETERS(...)

   PARAMETERS(...)
      |  FROM-COLUMN = 2 / <integer 1..256>
      |  TO-COLUMN = 72 / <integer 1..256>
      | ,LANGUAGE-STANDARD = STD / ISO
      | ,SPECIAL-KEYWORDS = NO / YES
```

SOURCE-PROPERTIES = STD
The default operand values of the subsequent PARAMETERS structure are used.

SOURCE-PROPERTIES = PARAMETERS(...)

FROM-COLUMN = 2 / <integer 1..256>
Specifies the first column of the source text. The default value is column 2.

TO-COLUMN = 72 / <integer 1..256>
Specifies the last column of the source text. The default value is column 72.

LANGUAGE-STANDARD = STD / ISO
The source text complies with the industry standard (STD) or the PL/I standard (ISO).

SPECIAL-KEYWORDS = NO / YES
YES: Special keywords, such as B, CAT, LE for example, are treated as reserved names.

PREPROCESSING operand

This operand determines whether the preprocessor is to be called.

```
PREPROCESSING = NONE / PARAMETERS(...)

      PARAMETERS(...)
        |
        |  OUTPUT = *NONE / *STD
```

PREPROCESSING = NONE
The preprocessor is not called.

PREPROCESSING = PARAMETERS(...)
The preprocessor is called.

OUTPUT = *NONE
The processing result of the preprocessor is not output.

OUTPUT = *STD
The preprocessor writes the result to the SAVMAC file.

COMPILER-ACTION operand

This operand specifies whether an object is to be generated, and if so, how.

```
COMPILER-ACTION = SYNTAX-CHECK / MODULE-GENERATION(...)

      MODULE-GENERATION(...)
          |
          |  MAIN-PROGRAM = NO / YES
          |
          | ,EXTENDED-SYSTEM = NONE / PARAMETERS(...)
          |
          |       PARAMETERS(...)
          |         |  BIT-POINTER = NO / YES
```

COMPILER-ACTION = SYNTAX-CHECK
The compiler run is to terminate after the syntax check.

COMPILER-ACTION = MODULE-GENERATION(...)
A complete compiler run is performed. Certain properties of the objects to be generated can be determined via the parameters of the MODULE-GENERATION structure.

MAIN-PROGRAM = NO / YES
YES: The object is a main program.

EXTENDED-SYSTEM = NONE
The object has no XS capability.

EXTENDED-SYSTEM = PARAMETERS(BIT-POINTER = NO / YES)
The object has XS capability.
In the case of BIT-POINTERS=YES, all pointers are bit pointers.

MODULE-LIBRARY operand

This operand determines the library in which the generated object is to be stored, and the name under which it is stored.

```
MODULE-LIBRARY = *OMF / <full-filename 1..54 without gen-vers>(...)

  <full-filename 1..54 without gen-vers>(...)
     |
     |  ELEMENT = *STD(...) / <full-filename 1..54 without gen-vers>(...)
     |
     |     *STD(...)
     |       |
     |       |  VERSION = *UPPER-LIMIT / <alphanum-name 1..24>
     |
     |     <full-filename 1..54 without gen-vers>(...)
     |       |
     |       |  VERSION = *UPPER-LIMIT / <alphanum-name 1..24>
```

MODULE-LIBRARY = *OMF
The object is to be written to the temporary EAM file.

MODULE-LIBRARY = <full-filename 1..54 without gen-vers>(...)
Name of the PLAM library into which the object is to be written.

ELEMENT = *STD(...)
The element name is formed from the first entry name of the external procedure.

VERSION = *UPPER-LIMIT / <alphanum-name 1..24>
Modules with the same name can be distinguished by version identifiers. If there is no version option, the element with the highest version (*UPPER-LIMIT) can be used for the linkage run.

ELEMENT = <full-filename 1..54 without gen-vers>(...)
An element name can be assigned using <full-filename>.

VERSION = *UPPER-LIMIT / <alphanum-name 1..24>
See above

If no element name is specified the object is given the first entry name of the external procedure.

LISTING operand

```
LISTING = STD / NONE / PARAMETERS(...)

    PARAMETERS(...)
        OPTIONS = NO / YES
       ,PREPROCESSING-OUTPUT = NO / YES
       ,SOURCE = NONE / PARAMETERS(...)
           PARAMETERS(...)
              INCLUDE-EXPANSION = NO / YES
             ,PAGE-FRAME = NONE / PARAMETERS(...)
                   PARAMETERS(...)
                      LINE-DELIMITER = ' ' / <c-string 1..1>
             ,LINE-NUMBER-LAYOUT = STD / EDT / EDOR
       ,DATA-ALLOCATION-MAP = NO / YES
       ,CROSS-REFERENCE = NONE / REFERENCED / FULL
       ,INCLUDE-REFERENCE = YES / NO
       ,STRUCTURE-LAYOUT = NO / YES
       ,EXTERNAL-DICTIONARY = NO / YES
       ,STATEMENT-ADDRESS = YES / NO
       ,ASSEMBLER-CODE = NO / YES
       ,SUMMARY = NO / YES
       ,ADDITIONAL-OUTPUT = *NONE / *TERMINAL
```

LISTING = STD
The default operand values of the PARAMETERS structure are to be used.

LISTING = NONE
No lists are to be generated.

LISTING = PARAMETERS(...)
The following parameters determine which lists are to be generated. The value NO prevents the respective list from being generated.

OPTIONS=NO / YES
YES: Output of the effective control statements.

PREPROCESSING-OUTPUT = NO / YES
YES: Output of the preprocessor listing.

SOURCE = NONE
No source listing will be output.

SOURCE = PARAMETERS(...)
A source listing will be output.

INCLUDE-EXPANSION = NO / YES
YES: The source listing will contain the inserted INCLUDE texts.

PAGE-FRAME = NONE / PARAMETERS(LINE-DELIMITER = '_' / <c-string 1..1>
PARAMETERS: Output of the source program with leader and trailer, and with frame characters <c-string> if specified.

LINE-NUMBER-LAYOUT = STD / EDT / EDOR
Line format.
STD: unchanged, as in the PRINT command
EDT: without leading and trailing zeros, with a period between 4th and 5th position
EDOR: without trailing zeros

DATA-ALLOCATION-MAP = NO / YES
YES: Output of the memory map.

CROSS-REFERENCE = NONE / REFERENCED / FULL
Output of cross-references and attributes of identifiers.

INCLUDE-REFERENCE = YES / NO
YES: Output of INCLUDE references.

STRUCTURE-LAYOUT = NO / YES
YES: Output of the aggregate listing.

EXTERNAL-DICTIONARY = NO / YES
YES: Output of the list of external names.

STATEMENT-ADDRESS = YES / NO
YES: Output of the allocation table: statement for the hexadecimal address of the generated code.

ASSEMBLER-CODE = NO / YES
YES: Output of the object code listing.

SUMMARY = NO / YES
YES: Output of program statistics.

ADDITIONAL-OUTPUT = *NONE / *TERMINAL
*TERMINAL: The listings are also to be output on the display terminal.

TEST-SUPPORT operand

This operand selects options for support of program testing.

```
TEST-SUPPORT = NONE / PARAMETERS(...)

   PARAMETERS(...)
      |
      |  STATEMENT-TABLE = NO / YES
      |
      | ,TOOL-SUPPORT = NONE / AID
      |
      | ,TRACE-SUPPORT(TRC) = NONE / ALL / PARAMETERS(...)
      |
      |       PARAMETERS(...)
      |          |
      |          |  PROCEDURE-ENTRY = YES / NO
      |          |
      |          | ,PROCEDURE-EXIT = YES / NO
      |          |
      |          | ,PROCEDURE-CALL = YES / NO
      |          |
      |          | ,LABELLED-STATEMENT = YES / NO
      |          |
      |          | ,GOTO-STATEMENT = YES / NO
```

TEST-SUPPORT = NONE
No test support.

TEST-SUPPORT = PARAMETERS(...)
The following parameters determine which test support options are to be generated. The value NO prevents the respective test support from being generated.

STATEMENT-TABLE = NO / YES
YES: Preparation of source line numbers.

TOOL-SUPPORT = NONE / AID
AID: Generation of LSD information for the AID debugging aid.

TRACE-SUPPORT = NONE / ALL / PARAMETERS(...)
Generation of trace information.

PROCEDURE-ENTRY = YES / NO
YES: Procedure trace

PROCEDURE-ENTRY = YES / NO
YES: Trace when quitting a procedure

PROCEDURE-CALL = YES / NO
YES: Trace when invoking a procedure

LABELLED-STATEMENT = YES / NO
YES: Trace for statement labels

GOTO-STATEMENT = YES / NO
YES: Trace for GOTOs

OPTIMIZATION operand

This operand selects the type of optimization.

```
OPTIMIZATION = NONE / PARAMETERS(...)

    PARAMETERS(...)
       |  SUPPRESS-COND-CHECK = NO / YES
       | ,REORDER-EXPRESSION = NO / YES
       | ,IGNORE-STRINOVERLAP = NO / YES
```

OPTIMIZATION = NONE
Optimization is not activated.

OPTIMIZATION = PARAMETERS(...)

SUPPRESS-COND-CHECK = NO / YES
YES: Condition code generation is suppressed.

REORDER-EXPRESSION = NO / YES
YES: The sequence of the statements may be changed.

IGNORE-STRINGOVERLAP = NO / YES
YES: Assignments are not checked for overlapping.

COMPILER-TERMINATION operand

This operand controls termination of the compilation run.

```
COMPILER-TERMINATION = STD / PARAMETERS(...)

    PARAMETERS(...)
      |  CPU-LIMIT = JOB-REST / <integer 1..32767>
      | ,MAX-ERROR-NUMBER = 500 / <integer 1..32767>
```

COMPILER-TERMINATION = STD
The default operand values of the PARAMETERS structure are to be used.

COMPILER-TERMINATION = PARAMETERS(...)
Selection of the criteria for compiler termination

CPU-LIMIT = JOB-REST / <integer 1..32767>
Maximum compilation time in seconds

MAX-ERROR-NUMBER = 500 / <integer 1..32767>
Number of errors after which the compilation is to be terminated

MONJV operand

This operand determines whether the compiler run is to be monitored by a job variable.

```
MONJV = *NONE / <full-filename 1..54 without gen>
```

MONJV = *NONE
The compiler run is not to be monitored by a job variable.

MONJV = <fill-filename 1..54 without gen>
Name of the job variable which is to monitor the compiler run.

LANGUAGE operand

This operand determines the language in which the compiler messages are to be output.

```
LANGUAGE = ENGLISH / DEUTSCH
```

LANGUAGE = ENGLISH / DEUTSCH
The compiler messages can be output in English (default) or in German.

3.12.3 Mapping of SDF operands to COMOPT operands

SDF operand	**COMOPT operand**
SOURCE	SOURCE
INCLUDE-LIBRARY	COMPLIB
SOURCE-PROPERTIES	
FROM-COLUMN, TO-COLUMN	MARGINS=TEXT
LANGUAGE-STANDARD=ISO	OPTIONS=ISO
SPECIAL-KEYWORDS=YES	MARGINS=CHAR48
PREPROCESSING=YES	OPTIONS=MACRO
OUTPUT=*STD	MARGINS=SAVMAC
COMPILER-ACTION=	OBJECT=
SYNTAX-CHECK	CODE
MODULE-GENERATION	OUT
MAIN-PROGRAM=YES	OPTIONS=MAIN
EXTENDED-SYSTEM=YES	OPTIONS=XS
BIT-POINTER=YES	OPTIONS=BITPTR
MODULE-LIBRARY	MODULE
LISTING	LIST=
OPTIONS=YES	OPTIONS
PREPROCESSING-INPUT=YES	INSOURCE
SOURCE=YES	SOURCE
INCLUDE-EXPANSION=YES	EXPAND
PAGE-FRAME=YES	OUTTEXT
LINE-NUMBER-LAYOUT	LINECNT
DATA-ALLOCATION=YES	MAP
CROSS-REFERENCE= REFERENCED FULL	 SHRTXREF FULLXREF
STRUCTURE-LAYOUT=YES	AGGREGATE

SDF operand	COMOPT operand
LISTING (continued)	
EXTERNAL-DICTIONARY=YES	ESD
STATEMENT-ADDRESS=YES	OFFSET
ASSEMBLER-CODE=YES	ASSM
SUMMARY=YES	SUMMARY
ADDITIONAL-OUTPUT=*TERMINAL	TERMINAL
TEST-SUPPORT	DEBUG, SYMTEST
STATEMENT-TABLE=YES	DEBUG=STMT
TOOL-SUPPORT=AID	SYMTEST=ALL
TRACE-SUPPORT	
PROCEDURE-ENTRY=YES	DEBUG=PROCTRACE
PROCEDURE-EXIT=YES	DEBUG=RETURNTRACE
PROCEDURE-CALL=YES	DEBUG=CALLTRACE
LABELLED-STATEMENT=YES	DEBUG=LABTRACE
GOTO-STATEMENT=YES	DEBUG=GOTOTRACE
OPTIMIZATION	OPTIMIZE=
SUPPRESS-COND-CHECK=NO	ENABLING
REORDER-EXPRESSION=YES	REORDER
IGNORE-STRINGOVERLAP=YES	OVERLAP
COMPILER-TERMINATION	
CPU-LIMIT=	1)
MAX-ERROR-NUMBER=	OBJECT=ABORT()
MONJV=	1)
LANGUAGE=ENGLISH	FORMAT=ENGLISH

1) Entered in the EXECUTE command

4 Linking and loading a PL/I program

4.1 General

A PL/I object module arising as a result of compilation must be linked into an executable load module by linking it to other modules. These other modules are taken from the PL/I runtime library, or they result from separately compiled (PL/I) procedures. The runtime library contains all prefabricated modules of an object program, such as program monitor, input/ output system, built-in functions, condition and error handling, etc. These modules are also referred to as the runtime system in this manual. Two runtime systems are available (see section 4.5).
The linkage process links all the object modules named when the linkage editor was called, and also those explicitly or implicitly referenced by the named modules. Explicitly referenced are those object modules to which declarations (DCL...ENTRY EXTERNAL;) in the module in question apply, as long as the declaration does not also contain the option OPTIONS (WXTRN). Modules from the runtime library which are required for conversion, input/output, etc. are, however, implicitly referenced.
Generation of the address references between the EXTERNAL items of the modules and the fixing of the relative addresses in the object modules are other important functions also performed by the linkage editor. In BS2000 the load modules are entered in a cataloged file by the linkage editor (TSOSLNK). These load modules can be called from the cataloged file as often as required.

During the development and debugging phase, the dynamic linking loader (DLL) can be used instead of the TSOSLNK linkage editor for testing programs (short compilation time, subroutines from a single library, programmed as "shareable"). Linking and loading is initiated with the /EXEC* or /LOAD* statement.

4.2 Controlling the linkage editor (TSOSLNK)

The linkage editor links object modules into load modules. All the object modules named in the INCLUDE statement are firstly linked. If they contain references to other procedures or external names (external references), these procedures must also be linked. The search for procedures with definitions of external names is carried out in three stages. Firstly an attempt is made to find the definitions in the object modules specified in INCLUDE. An attempt is then made to satisfy the as yet unassigned external references from object modules libraries which may be expressly specified for this purpose (RESOLVE statement). For references still not resolved, appropriate definitions are sought in the [$TSOS.]TASKLIB library. If no definition is found here, the reference remains unresolved and an error message is issued. Procedures for which only so-called weak external references (OPTIONS(WXTRN)) exist are not linked, see section 3.3.4.

Further functions of the linkage editor include:

- modifying external names
- combining and reserving storage space for EXTERNAL items (COMMON)
- generating overlay structures
- defining the program start point

The linkage editor also generates printer listings, if required.

Control of the linkage editor (TSOSLNK) is peformed via its own control statements. The most important control statements are briefly described here with their parameters. More detailed information on the function of the linkage editor and its control statements may be found in the Utility Routines Reference Manual [3].

4.2.1 Calling the linkage editor

The linkage editor is called via the command

```
/{EXECUTE}
 {      }$TSOSLNK
 {EXEC  }
```

The linkage editor expects control statements and object modules as input which it reads in via SYSDTA. A special statement (INCLUDE statement) causes the linkage editor to read object modules from the EAM file of the current task (indicated by the symbol *) or from an object modules library, and to add them to the load module.

This means the user can select:

– out of the EAM object modules file, the whole file or specific modules;

– out of object modules libraries, only the specific modules.

The linkage editor writes the generated load module into a cataloged file. In a linkage run this file is rewritten from the beginning if it already exists. If a /FILE command for this file is not entered by the user before calling the linkage editor, cataloging and storage allocations are performed by the linkage editor.

If the PL/I program is linked statically, or if a prelinked module is created by dynamic linkage, it is advantageous to define the storage space with a /FILE command before linking takes place. This avoids unnecessary chopping up of the load module and reduces the loading time. Any unrequired storage space reserved can subsequently be released via a /FILE command with a negative SPACE option.

Error messages from the linkage editor are output to SYSOUT and SYSLST.

Example

```
.
.
/SYSFILE SYSDTA=PROG
/EXEC $PLI1
/SYSFILE SYSDTA=(SYSCMD)
/FILE PL1.PROGR.003,SPACE=(90,6)
/EXEC $TSOSLNK
PROGRAM P003,FILENAM=PL1.PROG.003
INCLUDE *
END
/FILE PL1.PROG.003,SPACE=-100
.
.
```

Note

SYSDTA was assigned to the source program file PROG for the compilation run and must be reset before the linkage editor is called.

4.2.2 Statements for the linkage editor

PROGRAM statement

The name of the load module is defined via the PROGRAM statement. This statement must be specified, but need not be the first statement. If several PROGRAM statements are specified, the last one is valid.

```
PROG[RAM] program-name[,FILENAM=file-name]
```

program-name — Name of the linked program (max. 8 characters)

file-name — Name of the cataloged file in which the linked program is to be stored. If this parameter is not specified, the program name also applies as the file name (name of the load module).

Of the remaining parameters in the PROGRAM statement, the following can also be used in special cases:

LET = Y[ES] — The program is also linked even if open external references are still present after the libraries have been searched. However in this case it should be ensured that these references will not be required when the program is executed, since otherwise the program will behave in an undefined manner.

PL1 = Y[ES] — This option prevents the linkage editor from issuing error messages if variables with the EXTERNAL attribute in several object modules are assigned an initial value.

Warning:

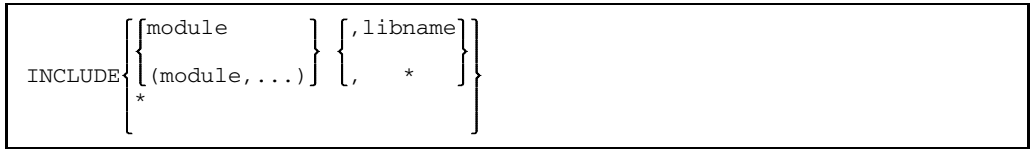
If this parameter is specified, no external names beginning with IQ can be used (see Linkage Editor Reference Manual [3]).

START = P$START — This option is only necessary if a non-PL/I module is linked as the first module.

The PROGRAM parameters not named have the usual effect or do not apply to PL/I programs.

INCLUDE statement

The INCLUDE statement retrieves one or more object modules from the specified library and joins them to the load module. This statement is mandatory.



module	Name of the object module to be added to the load module.
libname	Name of an object module library from which the modules are to be taken.
*	OMF of the current task.

Note

The effect of the INCLUDE * statement is to link together all object modules which exist in the OMF file even if they originate from several successive compilations, possibly by different compilers.

RESOLVE statement

The RESOLVE statement specifies an object module library in which the linkage editor should attempt to resolve external references. The object modules from the specified library which satisfy the external references are then also joined to the load module. This statement should only be specified if required.

```
        ⎧external reference       ⎫
RESOLVE[⎨                         ⎬],library name
        ⎩(external reference,...) ⎭
```

external reference

Name of an external reference to be resolved. If no references are specified the linkage editor tries to find definitions of all the currently unsatisfied references in the library.

library name — The name of the library in which the linkage editor should search for the suitable definition.

Remark

If a number of RESOLVE statements are specified, these are processed from back to front, which means that the open external references are sought first in the last library specified, then in the next to the last, etc. If external references still remain open after all the RESOLVE statements have been processed, the [$TSOS.]TASKLIB file is finally searched.

Note

If all PLI1 runtime library modules are kept in $TSOS.TASKLIB, the user need not supply any RESOLVE statements referring to them. On the other hand, if the system administrator creates a separate library for PLI1, e.g. to keep the system library catalog small, the name of that library must be communicated to the linkage editor by a RESOLVE statement without explicit module names. In these cases, the library name will be supplied by the system administrator.

END statement

The END statement terminates the statements to the linkage editor and must be specified.

```
END
```

4.3 Example of the linkage editor

Three PL/I procedures are compiled and then linked. They contain external references to the procedures FUNCT1 and FUNCT2, which are located in the library FUNCLIB. Examples illustrating how procedures are entered in libraries are given in sections 2.2.4 and 3.6.4.

```
/SYSFILE SYSDTA=(SYSCMD)                    ┐
/EXEC $PLI1                                 │
*COMOPT SOURCE=PL1A,OPTIONS=MAIN            │
*END                                        │
/EXEC $PLI1                                 │
*COMOPT SOURCE=PL1B                         ┘ 1)
*END                                        │
/EXEC $PLI1                                 │
*COMOPT SOURCE=PL1C                         │
*END                                        ┘

/EXEC $TSOSLNK
PROGRAM PRO05,FILENAM=PL1.PROGR.05      2)
INCLUDE *                               3)
RESOLVE,FUNCLIB                         4)
END
/EXEC PL1.PROGR.05                      5)
   .
   .
   .
```

1) The generated object modules - assume they are also called PL1A, PL1B, and PL1C - are written to the EAM file. Additionally, for each procedure compiled, a 'static module' may be created whose name is derived from the procedures name. See section 4.6. The main procedure must be in the first position.

2) Specification of the load module name and the cataloged file into which the load module is to be entered.

3) Statement for linking all the modules from the EAM file.

4) The external references FUNCT1 and FUNCT2 are to be resolved with definitions in object modules from the FUNCLIB file.

5) Execution of the compiled and linked program. PL1.PROGR.05 is the cataloged file in which the load module PRO05 was entered.

4.4 Loading

The loader loads the load module generated by the linkage editor.

More detailed information on the loader may be found in the Utility Routines Reference Manual [3]. Details on the control of the loaded and started program via RUNOPT are given in chapter 5.

The loader is called via the command

```
/LOAD filename[,TIME=number]
```

filename — Name of the cataloged file containing the load module. The name to be specified is that used for the linkage editor in the PROGRAM control statement.

number — The maximum CPU time in seconds for which this program should run. If this parameter is not specified, the time defined as the system default is assumed.

If a PL/I program is to be controlled via IDA statements (e.g. AT command), then the command sequence

```
/LOAD     filename
/AT...
/RESUME
```

can be used for loading and executing the program. The IDA statements, however, can only refer to virtual addresses which can be derived from the offset listing, or the object code listing (LIST = ASSM during compilation), and the linkage editor listing, and which were marked TRAITS READONLY = N as they were linked or entered in the library. For details refer to section 9.8.

4.5 Runtime system

Two runtime systems are available for PL/I programs. They both produce identical results when the program is executed, but behave differently for linking and loading and have different storage and machine time requirements.

The main difference is that in one case the main part of the object modules is prelinked into two large modules, which are linked dynamically when the program is executed. This means that the linkage process requires substantially less time and the load module is smaller. These relationships are outlined in the overview in Figure 4-1.

Each runtime system is complete in itself. Both use the same elementary object modules.

Which of the two runtime systems is used depends on whether the linkage strategy described above will first retrieve the ITP#AOS# or ITP#AOD# connection module.

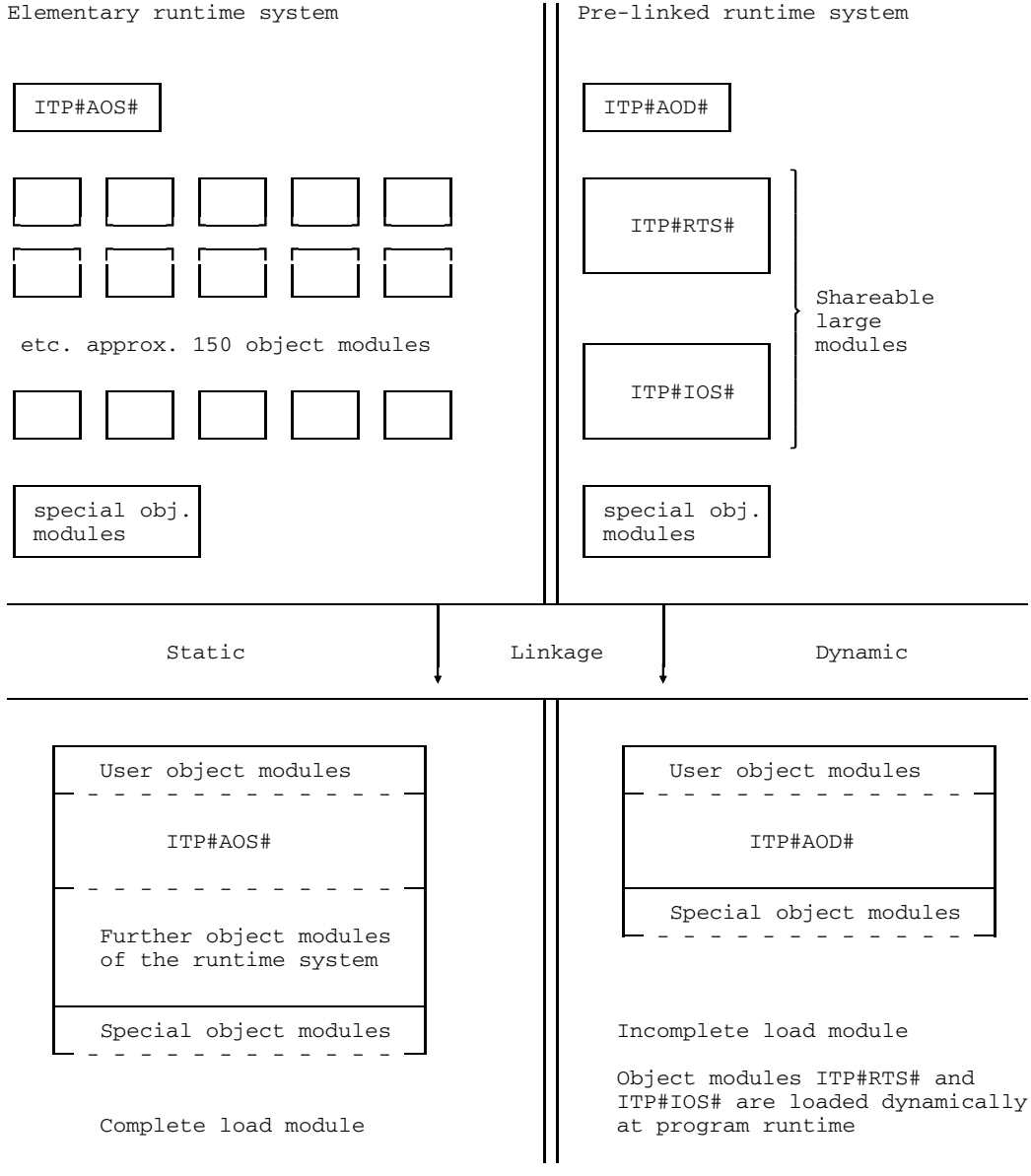


Fig. 4-1 Overview of the runtime systems and the resulting load modules

4.5.1 Elementary runtime system

In an elementary runtime system all the elementary object modules are present individually. The following object modules are present:

- Linkage module ITP#AOS#
- Approx. 150 further elementary object modules
- Special object modules (these are only listed separately here because they receive special treatment in the prelinked runtime system).

In the linkage process the object modules required by the object program are linked as a result of the linkage module ITP#AOS#. This produces a complete load module containing all the facilities required by the source program. For loading and execution the load module is always regarded as one unit.

A complete linkage and loading process is necessary for each source program. When linking takes place, a complete linkage editor listing is generated.

4.5.2 Prelinked runtime system

In the prelinked runtime system most of the elementary object modules are prelinked into two large modules, so that the runtime system consists of the following object module:

- Linkage module ITP#AOD#
- Prelinked modules ITP#RTS# and ITP#IOS#
  These modules can be entered in the "Share Table".
- Special object modules
  These include object modules for calling the sort program, etc. These modules are not prelinked, but remain elementary object modules.

All modules except ITP2SRT# and those used for language transfer are programmed to be 'reentrant'.

The linkage editor links together the object modules resulting from the compilation of the PL/I procedures and the ITP#AOD# module. As well, special object modules are linked as required. All these object modules together form an incomplete load module. As the two prelinked modules are not yet included, this load module is of limited size; it requires less storage space than a complete load module. Only a small number of object modules is involved in the linkage process, which is therefore executed very fast.

The prelinked modules required for executing the program are loaded dynamicall when the program has started. Since the modules have already been prelinked, the time required for this operation is minimal.

The prelinked modules are programmed to be "reentrant" and can therefore be entered in the "Share Table" of the operating system by the system administrator. In this case the time required for dynamic load is further reduced. If a number of PL/I programs use the same prelinked modules, the storage requirements can also be improved.

Since only a few modules are to be linked in the linkage process, only a limited linkage editor listing is produced. If a complete linkage editor listing is required for error detection, then the elementary runtime system should be used.

4.5.3 Storage of the runtime system

Which of the runtime systems is available on a particular installation is defined by the system administrator and cannot be directly controlled by the user. In many cases it is advantageous if both runtime systems are available at the same time.

For best results, the system administrator will store the preferred (or only) runtime system in system file TASKLIB, which allows the user to control the linkage process most easily.

The second runtime system can be stored in a library. The appropriate options (RESOLVE) should be supplied to control the linkage process.

4.6 Name conventions for PL/I object modules

Module names can be specified in the control statements of the linkage editor. The names of the modules and the names of other external program items are listed in the linkage editor listing. These names result directly from the appropriate identifiers of the linked PL/I procedures, where, however, due to certain restrictions in the linkage editor and other programming languages, the following rules should be observed:

- If entry constants are declared with OPTIONS (ASSEMBLER OR COBOL), the linkage editor always uses the first 8 characters.
- For all other identifiers with OPTIONS (PLI1 or FORTRAN or VARIABLE) and the EXTERNAL attribute, a maximum of 7 characters is placed in the object code generated; if the identifier is too long, its first 4 and last 3 characters are used.
- If a PL/I program identifier in addition to EXTERNAL, has the ENTRY attribute as well as (procedure name, entry), then all occurrences of "_" are also replaced with "$".

Compilation of a PL/I procedure usually creates two object modules, a code module and a data module. The data module is always generated if the PL/I procedure contains variables with the STATIC attribute. These modules are named in accordance with the following conventions:

- The code module contains the name of the first (primary) identifier having the ENTRY attribute, if applicable reduced to 7 characters and modified with $ according to the above rule. This is the first name of the outermost procedure block of a separately compiled procedure.
- The name of the data module is derived from the name of the code module, which is always developed into 8 characters. If the name was shorter it is firstly padded to 7 characters with the character "@". The eighth character is always one of the digits between "1" and "7" which indicates the length of the original name.

It should be noted that "_" is only replaced by "$" in the module names and entries, and not in other identifiers with the EXTERNAL attribute. Only the rule of reduction is used for the latter.

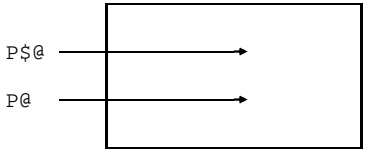
Example

The program reads:

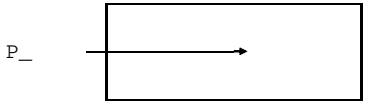
```
P_@ : PROCEDURE;
P@  : ENTRY;
DCL P_STATIC EXTERNAL INIT (0);
END;
```

It supplies the following modules:

Code module name: P$@



Data module name: P$@@@@@3



The external name P$START# is generated for the main procedure (MAIN), and identifies the position at which the PL/I program is to be started.

4.7 Extended address space (XS)

Processors having an address space larger than 16 Mbytes (up to 2 gigabytes) are referred to as XS systems. If this larger address space is to be utilized by a PL/I program, the compiler will have to generate modules that do not only use 24-bit addresses but also 31-bit addresses. Modules of this kind can be generated by the PLI1 compiler Version 4.0A. As a result, the following module types can be distinguished:

- old modules — generated by PLI1 compilers prior to Version 4.0A
- non-XS modules — generated by PLI1 compilers as of Version 4.0A if *COMOPT OPTIONS=NXS is specified (default)
- XS modules — generated by PLI1 compilers as of Version 4.0A if *COMOPT OPTIONS=XS is specified.

Linking conventions:

- Old modules and non-XS modules are compatible and are linked indiscriminately. The result is either a non-XS load module or a non-XS prelinked module, which can again be linked like a non-XS module.
- XS modules are not compatible with non-XS modules and old modules: thus XS modules can only be linked with XS modules. The result is either an XS load module or an XS prelinked module that, in turn, can be linked further like an XS module.

These conventions apply to both static and dynamic linking.

The PLI1 runtime system modules are compatible with XS and non-XS modules.

At load time, the LOADPT parameter determines the program's load address and thus its position in the storage area. The following applies in this context:

- Non-XS load modules can be loaded in the address space up to 16 Mbytes, anyway: either in a system with an extended address space or in a system without an extended address space.
- NX load modules can run in either system without any restrictions, i.e. in a system with or without extended address space.

It is the user's responsibility to ensure that these conventions are adhered to. Certain tests may be required at load time. Refer to the

`ARMODE-CHECK`

operand and the section on "XS support" in publication [12] Linkage Editors and Loaders as of Version 21.0B (BS2000 Version 9.0).

5 Execution of the PL/I program

5.1 General

Once the load module generated by the linkage editor for the object program has been placed in a file it can be loaded and executed via the /EXECUTE command. As with the PLI1 compiler, control statements can be transferred to the object program once it has been started.

Before a PL/I program is executed the following assignments must be made:

- The files required in the program must be set up or in the case of existing files, assigned to the program via FILE or CHANGE commands. All the necessary information is explained in chapter 6 of this manual.
- The control statements (*RUNOPT) must be provided if these are to be used to control the object program. In this case, task switch 1 should also be set.
- As for compilation, the message files must be available (see section 3.2).

After the object program has been started the runtime system reads the control statements from SYSDTA. If the control statement LIST = OPTIONS is specified then the enabled control statements are output to SYSLST. In addition all the messages generated by the runtime system during the object run appear in the system file SYSOUT.

On termination of the program run the message

```
END OF PROGRAM name, RTS v.www-ari, TIME USED: xxxxx.xx SEC
```

is output to SYSOUT.SEC

The specifications have the following meaning:

Name	Name of the load module from the PROGRAM statement of the linkage editor. The name does not appear if the program was started by EXEC * or EXEC (module).
x	Time used, in seconds.
v.w	Version number of the runtime system.

a Update code of module ITP#AOS# of the elementary runtime system, or ITP#AOD# of the prelinked runtime system.

r Update code of module ITP#RTS# of the prelinked runtime system, or 0 for the elementary runtime system.

i Update code of module ITP#IOS# of the prelinked runtime system, or 0 for the elementary runtime system.

Refer also to section 4.5. With the prelinked runtime systems, a check is made to ascertain whether the version numbers of the modules are compatible. If they are not, an error message is issued and the run terminated.

If an error terminates the run, subsequent commands are ignored until the STEP or LOGOFF command is entered.

5.2 Program execution

5.2.1 Execution with ISP command

The PL/I program present in a file as a load module is started with the EXECUTE command. The program is referenced via the file name defined in the PROGRAM statement of the linkage editor.

Call format:

```
/{EXECUTE}
 {       }filename [,TIME=t][,MONJV=jvname]
 {EXEC   }
```

The operating system permits other parameters in the EXEC command, which at present do not apply to PL/I programs or which could lead to errors. Specification of the TIME parameter is expedient to minimize the time requirements of the program should program errors occur.

5.2.2 Execution with SDF command

A PL/I program present as a load module in a file can alternatively be started with the SDF command

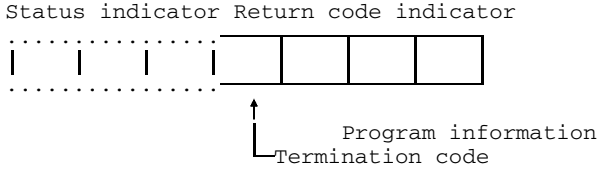
```
/START-PLI1-PROGRAM
```

with additional operands. The operands and their meaning are described in section 5.7.

Entering SDF commands and their operands in guided and unguided dialog is described in detail in the manual "Introductory Guide to the SDF Dialog Interface".

5.2.3 Monitoring by monitoring job variable

For monitoring the object run a monitoring job variable can be specified by entering 'MONJV=jvname' in the EXECUTE command. The status indicator (digits 1 through 3) of the job variable is set by the system (values '$R_', '$T_', '$A_'). The return code indicator (digits 4 through 7) is set at the end of the object run; it has the following format:



The termination code is allocated as follows:

'0' Normal program termination
'2' Abnormal termination (e.g. error condition occurred)
'3' Fatal error in the PLI1 runtime system

Program information is represented by '000'. By invoking the library procedure PLIRETC (see chapter 11 Utilities) program information can be set to other values (between '000' and '999').

This feature presupposes the software product JV.

5.2.4 Examples

Depending on whether control statements or files are required for the PL/I program, various alternatives are available, some of which are shown in the following examples:

Example 1

The PL/I program contains no input or output files; output is to SYSLST. A date is to be supplied to the main procedure (MAIN) as a parameter.

The following commands and control statements should be entered:

```
  .
  .
  .
/SETSW ON=1
/EXEC PROG1
*RUNOPT ARGUMENT='24.12.78'
*END
  .
  .
  .
```

Example 2

The program requires an input file with the PL/I name (TITLE) PLIIN and an output file with the PL/I name PLIOUT. No other control statements are provided. The commands appear as follows:

```
  .
  .
  .
/FILE INPUT,LINK=PLIIN
/FILE OUTPUT,LINK=PLIOUT,FCBTYPE=SAM,
/RECSIZE=160,BLKSIZE=STD,RECFORM=V
/EXEC PROG3
  .
  .
  .
```

Example 3

The control statements for the object run are located in file OPT.

The commands are structured as follows:

```
  .
  .
  .
/SYSFILE SYSDTA=OPT
/SETSW ON=1
/EXEC PROG2
/SYSFILE SYSDTA=(PRIMARY)
/RESUME
    data
  .
  .
  .
```

File OPT:

```
*RUNOPT LIST = OP,
*RUNOPT MESSAGE = S,
*RUNOPT DUMP = ST
*END/
```

A breakpoint is set with the control statement *END/. The command

```
/SYSFILE SYSDTA=(PRIMARY)
```

is specified before the RESUME command so that data can then be entered from SYSDTA.

Example 4

In this example output is effected via the PL/I file PLIOUT, while input data for the file PLIIN are to be taken from the system file SYSDTA.

The commands are structured as follows:

```
  .
  .
  .
/SETSW ON=(1)
/FILE OUTPUT,LINK=PLIOUT...
/EXEC PROG4
*RUNOPT SYSFILE=SYSDTA(PLIIN)
*END
   data
  .
  .
  .
```

Example 5

The program contains PLIIN and PLIOUT as input and output files. In addition, data for the PL/I file with the title A is to be entered via the system file SYSDTA, and output data from the PL/I file with the title B are to be output to the system file SYSLST. As a result of this requirement the system files must be reassigned with the SYSFILE control statement. All the control options used are to be listed at the start of the program.

The commands are structured as follows:

```
   .
   .
   .
/SETSW ON=(1)
/FILE INPUT,LINK=PLIIN...
/FILE OUTPUT,LINK=PLIOUT...
/EXEC PROG5
*RUNOPT SYSFILE=(SYSDTA(A),SYSLST(B)),
*RUNOPT LIST=OPTIONS
*END
   data
   .
   .
   .
```

Note

The TITLE options SYSPRINT and SYSIN must not be used in the program.

5.3 Controlling the PL/I program

As for the compiler, control facilities are also provided for the object program. These are the control statements ARGUMENT, DUMP, FORMAT, LIST, MESSAGE, STORAGE, SYSFILE, and TRACE.

5.3.1 General rules for control statements

The rules governing the provision and structure of user program control statements correspond to those for the PLI1 compiler, which may be found in section 3.3.1. The main differences are in the leading keyword - in this case *RUNOPT - and in a number of control statements, only some of which are identical.

The format of the control statements for the user program appears as follows:

```
*RUNOPT control statement...
```

As with the compiler, input is terminated with *END or *END/. All further rules may be found in the section mentioned above. Runtime control statements are likewise entered via SYSDTA, in which case it is required that task switch 1 is first set with the SETSW command.

5.3.2 Error handling for control statement evaluation

Error handling during the processing of control statements for the user program is identical with that in the compiler. The points discussed in section 3.3.2 therefore apply here.

5.3.3 Controlling the listing output

Depending on their type, messages generated during the object run are output to either SYSLST or SYSOUT. The user can also use various control statements to forward the output to a second system file. It is only possible to control output from the runtime system in this way, and not output programmed by PUT, DISPLAY or WRITE statements.

Type of runtime output	Output file		
	Default	Additional output to file	Additional output controlled by
- List of active control statements - Program statistics	SYSLST	SYSOUT	LIST=TERMINAL (ineffective in batch mode)
- Procedure and audit TRACE	SYSLST	SYSOUT	TRACE = TERMINAL (ineffective in batch mode)
- Condition message - Backtrace (SNAP) - Program interrupt - Runtime errors - End message - ERROUT procedure	SYSOUT	SYSLST	MESSAGE=SYSLST

Since SYSLST can be directed to a user file by the SYSFILE command, a further control facility exists. See also sections 3.3.4 and 6.2.1.

5.3.4 Overview of the control statements for PL/I programs

The following sections give an overview of the object program control statements, most of which can be abbreviated. In the individual descriptions the relevant letters are underlined. A negative form can be specified for some parameters (e.g. TERMINAL and NOTERMINAL) but the brief description in the "Effect" column refers to the positive parameter option.

Control statement for object program	Abb.	Meaning	Specification	Abb.	Effect	Default
ACTIVE	ACT	Activate check points	YES NO	YES NO	Activation No activation	YES
ARGUMENT	ARG	Transfer a character string to the parameter position of the MAIN procedure of the program	'The character string can be up to 253 characters long and is enclosed in apostrophes (single quotes)'		Interpretation of the string is left to the user	'' (empty or null character string)
CONTROL	CTL	Control	STD NO ALL ALIGN NOALIGN	STD NO ALL AL NAL	} corresponds to NOALIGN corresponds to ALIGN command simulation for alignment errors for alignment errors; ERROR with ONCODE=8098	 NAL

Control statement for object program	Abb.	Meaning	Specification	Abb.	Effect	Default
DUMP	DMP	Output of dumps and SNAP	ALL	ALL	corresponds to (ST,A,SN,R (X'0',X'7FFFFF'))	NO
			NO	NO	corresponds to (NST,NA,NR,NSN)	
			STACK NOSTACK	ST NST	Dump of the entire stack	
			AREA NOAREA	A NA	Dump of the entire standard area	
			RANGE(a,e) NORANGE	R NR	Dump of an area from a to e. (Address in hex.)	
			SNAP NOSNAP	SN NSN	Output of procedure nesting at end of program	
			COND UNCOND	C UC	All above outputs at program termination, only in the event of an error or unconditionally	C
FORMAT	FM	Control of the no. lines per page, the line length, language for messages	PRINTER ([m_1][,m_2])	P	Control of output to SYSLST and in batch mode to SYSOUT. m_1 = no. of lines/page, m_2 = line length.	P(64, 132)
			TERMINAL ([n1][,n2])	T	Control of output to terminal: n_1 = no. of lines n_2 = no. of characters	T(O,k) 1)
			ENGLISH DEUTSCH	E D	All output from the runtime system is in English or German	E

1) k represents the physical line length of the current output device.

Control statement for object program	Abb.	Meaning	Specification	Abb.	Effect	Default
LIST	LST	Listing control (only those parameters are described here which are relevant at object time)	ALL	ALL	corresponds to (OP,SM)	
			NO	NO	corresponds to (NOP,NSM)	
			OPTIONS NOOPTIONS	OP NOP	Listing of the enabled control statements	NOP
			SUMMARY NOSUMMARY	SM NSM	Output of program statistics	NSM
			TERMINAL NOTERMINAL	T NT	Copying of all runtime system listings to the terminal	NT
MESSAGE	MSG	Control of runtime system messages	SYSLST	S	Messages also output to SYSLST	NS
			NOSYSLST	NS	Messages output to SYSOUT only	
STORAGE	STR	Facility for controlling the storage administration of the runtime system and achieving optimizations	AREA ([q_1][,[q_2][,[q_3]]])	A	A storage area of q_1 pages is firstly provided for the standard area. Any necessary extensions are added in increments of q_2 pages up to a max. of q_3 pages. The STORAGE condition is then set, if applicable. (page = 4 KB)	A(16,16 16,)
			STACK ([s_1][,s_2])	S	The stack is occupied in segments up to s_1 pages. For error and end handling in the case of insufficient storage a reserve of s_2 pages is provided. (page = 4 KB)	S(16,4)

Control statement for object program	Abb.	Meaning	Specification	Abb.	Effect	Default
SYSFILE	SFL	Assignment of the PL/I TITLE to the BS2000 system files. Also modification to the	SYSLST (title)	SL	The file with the PL/I TITLE 'title' is mapped to the system file SYSDTA, SYSOUT or SYSLST.	SL (SYSPRINT)
			SYSOUT (title)	SO		SO (SYSOUT)
			SYSDTA (title)	SD		SD (SYSIN)
		PAGESIZE and LINESIZE defaults	LINESIZE (l_1,l_2,l_3)	LS	l_1,l_2,l_3 if ≠ 0, overwrite the LINESIZE defaults for SYSLST, SYSOUT and SYSDTA.	LS(0,0)
			PAGESIZE(p_1,p_2)	PS	p_1 and p_2 if ≠ 0, overwrite the PAGESIZE defaults for SYSLST and SYSOUT.	PS(0,0)
		as well as DISPLAY statement control	DISPLAY(SYSOUT) dia = SYSOUT	DP(SO)	In interactive mode, output for the DISPLAY statement is sent to the interactive terminal; input for DISPLAY REPLY is expected from the interactive terminal.	DP(SO)
			DISPLAY(SYSCON) dia = SYSCON	DP(SC)	In interactive mode output for the DISPLAY statement is sent to the operator console; input for DISPLAY REPLY is expected from the operator console.	

Control statement for object program	Abb.	Meaning	Specification	Abb.	Effect	Default
SYSFILE (contin-uation)			DISPLAY(SYSDTA) dia = SYSDATA	DP(SD)	In interactive mode, output for the DISPLAY state-ment is sent to the interactive terminal; input for DISPLAY REPLY is entered from the system file SYSDTA.	
			DISPLAY(SYSOUT) bat = SYSOUT	DP(SO)	In batch mode, output for DISPLAY is sent to the system file SYSOUT; input for DISPLAY REPLY is entered from the system file SYSDTA.	
			DISPLAY(SYSCON) bat = SYSCON	DP(SC)	In batch mode, output for the DISPLAY statement is controlled on the operator console, input for DISPLAY REPLY is expected from the operator console.	
			DISPLAY(SYSDTA) bat = SYSDTA	DP(SD)	In batch mode output for DISPLAY is entered into the system file SYSOUT. Input for DISPLAY REPLY is entered from the system file SYSDTA.	

Control statement for object program	Abb.	Meaning	Specification	Abb.	Effect	Default
TABULATOR	TAB	Set tabs	(n,...)		n become tab positions	(1,11, 21,31, 41,etc.)
TRACE	TRC	Enable and disable the TRACE options	ALL	ALL	≙(P,L,C,G,R)	NO
			NO	NO	≙(NP,NL,NC,NG,NR)	
			PROCTRACE	P	Enable or disable the trace for PROCEDURE entries, labels, CALL calls, GOTO branches, RETURN	NP
			NOPROCTRACE	NP		
			LABTRACE	L		NL
			NOLABTRACE	NL		
			CALLTRACE	C		NC
			NOCALLTRACE	NC		
			GOTOTRACE	G		NG
			NOGOTOTRACE	NG		
			RETURNTRACE	R		NR
			NORETURNTRACE	NR		
			TERMINAL	T	Additional output of traces to terminal	T
			NOTERMINAL	NT		

5.4 Individual descriptions of the control statements for PL/1 programs

The control statements for user programs can be divided into two groups:

- The pure object time control statements ARGUMENT, CONTROL, DUMP, SYSFILE, TABULATOR, TRACE and STORAGE. These control statements may only be used at the runtime of the user program.
- The control statements FORMAT, LIST and MESSAGE which apply in the same way for the compiler and the user program. They can be used for the same purpose in each case, although some of the permissible specifications will differ. If a source program is compiled and then executed in the same task, obviously the common control statements must be specified separately for the compilation run and the object run. The control statements valid for the user program are described below.

5.4.1 Parameter transfer (ARGUMENT)

The ARGUMENT control statement permits the specification of a character string up to 253 characters in length, which is transferred unchanged (including blanks) to the parameter position of the MAIN procedure in the program. The parameter should be declared with

```
DCL name CHAR(n) VARYING;
```

in the MAIN procedure, where n ≤ 253.

Interpretation of the character string transferred is the responsibility of the user program. The character string is to be enclosed in single quotes (apostrophes). Quotes embedded in the character string must be represented by two quotes. Specification of a number of separate character strings is not practical, since only the last is transferred.

ARGUMENT = 'character string'

Default:

ARGUMENT = '' (empty or null character string)

5.4.2 Dump control (DUMP)

The DUMP control statement defines whether dumps and SNAP (the backtrace listing) are to be output at the end of the program.

```
DUMP = { specification          }
       { (specification,...)    }
```

Default: DUMP = (NO,C)

The following specifications can be used:

NO	corresponds to (NST,NA,NR,NSN)
ALL	corresponds to (ST,A,R(X'0',X'7FFFFF'),SN)
STACK NOSTACK	Dump the entire stack
AREA NOAREA	Dump the entire standard area
RANGE (a,e) NORANGE	Dump an area ranging from the start address a to the end address e. The addresses are to be specified in hexadecimal form (X'...').
SNAP NOSNAP	Output the current procedure nesting at the end of the program
COND UNCOND	All information required as a result of the control statements ST, A, R and SN is output at the end of the program, – in the case of an error only (C) – always (UC)

5.4.3 Output formats (FORMAT)

The FORMAT control statement determines the number of lines per page and the length of such lines for output to the printer and terminal. In addition it permits the specification of the language (German or English) in which the messages generated by the runtime system are to be output.

The individual specifications for the FORMAT control statement are described in section 3.5.4.

None of the specifications of the FORMAT control statement apply to PL/I program output resulting from the PUT, WRITE or DISPLAY statements.

5.4.4 List selection (LIST)

The LIST control statement defines which runtime system listings are to be output, and where they are to be output. This control does not affect output caused by PUT, WRITE or DISPLAY statements.

```
LIST = { specification          }
       {                        }
       { (specification,...)    }
```

Default: LIST = (NOP, NSM, NT)

Possible specifications are:

ALL — Corresponds to (OP,SM)

NO — Corresponds to (NOP,NSM)

OPTIONS
NOOPTIONS — Listing to SYSLST of all the enabled control statements for the object program.

SUMMARY
NOSUMMARY — Output to SYSLST of program statistics which contain:

- maximum occupancy of the stack by AUTOMATIC variables etc.
- maximum occupancy of the standard area (general storage for areas) as a result of ALLOCATE statements.

TERMINAL
NOTERMINAL — Additional output of the above listings to the terminal (SYSOUT) in interactive mode; does not apply to batch mode.

5.4.5 Additional message output (MESSAGE)

This control statement is used to determine whether runtime system messages are also to be output to SYSLST. The MESSAGE statement does not affect output from the user programs via the PUT, WRITE or DISPLAY statements, but it does apply to output from the ERROUT procedure.

```
MESSAGE = { SYSLST   }
          {          }
          { NOSYSLST }
```

Default: MSG = NS

SYSLST — In interactive mode messages from the runtime system are also output to SYSLST.

NOSYSLST — Messages are output to SYSOUT only.

5.4.6 Storage requirements (STORAGE)

The STORAGE control statement is used to control the storage management of the object program. The initial size and any increases in the size of the virtual memory for the standard area and the stack can be defined.

CONTROLLED and BASED variables as well as input/output areas are arranged in the standard area. The occupancy state is therefore determined by the program via the ALLOCATE/FREE and OPEN/CLOSE statements. The stack contains AUTOMATIC variables, auxiliary variables, and saved registers, and its occupancy state is mainly determined by the number of active blocks and the variables declared in these blocks. The storage statistics provide notes which enable the optimal sizes to be set for the standard area and the stack (LIST = SUMMARY).

```
STORAGE = { specification       }
          {                     }
          { (specification,...) }
```

Default: STORAGE = (AREA(16,16,),STACK(16,4))

Possible specifications:

AREA([q_1][,[q_2][,[q_3]]]):

A storage area of q_1 virtual pages (4 KB) is initially provided for the standard area. Any necessary extensions follow in increments of q_2 pages up to a maximum size of q_3 pages. The STORAGE condition is then set, if applicable.

According to the size of the permissible user address space, a storage requirement of up to 1500 pages (6 MB) is feasible for the standard area.

STACK([s_1][,s_2]):

The stack storage is reserved in segments each of s_1 virtual pages (4 KB). For error and end handling in the case of insufficient storage a reserve of s_2 pages is provided. The maximum value depends on the size of the allowable user address space.

5.4.7 System file assignment (SYSFILE)

The SYSFILE control statement defines which PL/I-TITLEs are to be mapped on to the BS2000 system files SYSDTA, SYSLST and SYSOUT. The default options for PAGESIZE and LINESIZE for these files can also be modified. Furthermore, it is defined which system files are to be assigned for the DISPLAY statement.

```
          ⎡specification       ⎤
SYSFILE = ⎨                    ⎬
          ⎣(specification,...) ⎦
```

Default:

```
SYSFILE = (SYSDTA(SYSIN),SYSLST(SYSPRINT),SYSOUT(SYSOUT),
          PAGESIZE(undef,undef),LINESIZE(undef,undef,undef),DISPLAY(SYSOUT))
```

Possible values are:

```
SYSDTA(title-1)  The files with the PL/I-TITLE title-1, title-2
SYSLST(title-2)  and title-3 are mapped onto system files SYSDTA,
SYSOUT(title-3)  SYSLST and SYSOUT.
```

PAGESIZE(p_1,p_2)

If values are specified for p_1 or p_2, then the program default options for PAGESIZE (e.g. setting via the OPEN statement) are overwritten for SYSLST or SYSOUT. p_1 is assigned to SYSLST, p_2 to SYSOUT. Specification of "0" for p_1 or p_2 has the same effect as "undef".

LINESIZE(l_1,l_2,l_3)

If values are specified for l_1, l_2, or l_3, then the program default options for LINESIZE (e.g. setting via the OPEN statement) are overwritten for SYSLST, SYSOUT or SYSDTA. Specification of "0" for l_1, l_2 or l_3 has the same effect as "undef".

dia = SYSOUT

In interactive mode, output for the DISPLAY statement is sent to the interactive terminal; input for DISPLAY REPLY is expected from the interactive terminal.

dia = SYSSON

In interactive mode, output for the DISPLAY statement is controlled by the operator console; input for DISPLAY REPLY is expected from the operator console.

dia = SYSDTA

In interactive mode, output for the DISPLAY statement is set to the interactive terminal, input for DISPLAY REPLY is read from the system file SYSDTA.

bat = SYSOUT

In batch mode, output for DISPLAY is sent to the system file SYSOUT; input for DISPLAY REPLY is accepted by the system file SYSDTA.

bat = SYSCON

In batch mode, output for the DISPLAY statement is controlled by the operator console; the input for DISPLAY REPLY is accepted from the operator console.

bat = SYSDTA

In batch mode, output for DISPLAY is put in the system file SYSOUT. Input for DISPLAY REPLY is accepted from the system file SYSDTA.

If bat is not specified, dia is assumed instead.

5.4.8 Trace control (TRACE)

This control statement enables or disables the TRACE outputs provided in the compiled program (see also the DEBUG control statement in section 3.6.2).

```
TRACE = [specification       ]
        {                    }
        [(specification,...) ]
```

Default: TRACE = (NO, T)

ALL Corresponds to (P,L,C,G,R)

NO Corresponds to (NP,NL,NC,NG,NR)

```
PROCTRACE
NOPROCTRACE

LABTRACE
NOLABTRACE          Enables or disables the trace for:

CALLTRACE           PROCEDURE entries
NOCALLTRACE         Labels encountered
                    CALL
GOTOTRACE           GOTO
NOGOTOTRACE         RETURN

RETURNTRACE
NORETURNTRACE
```

TERMINAL
NOTERMINAL

Additional TRACE output to the terminal (SYSOUT) in interactive mode. Does not apply to batch mode.

5.4.9 Setting tabs (TABULATOR)

Each data element output via the PUT LIST statement starts at the next tabulator position. The TABULATOR control statement is used to change the tab positions.

TABULATOR = (n,...)

Values for n must be integers ≥ 1. Each subsequent number must be greater than the last.

Default: TABULATOR = (1, 11, 21, 31, 41, etc.).

5.4.10 Alignment control (CONTROL)

This control statement defines the steps to be taken in the event of alignment errors.

```
CONTROL = { specification       }
          {                     }
          { (specification,...) }
```

Default: CONTROL = NOALIGN

The following specifications are permitted:

STD	Corresponds to NOALIGN
NO	Corresponds to NOALIGN
ALL	Corresponds to ALIGN
ALIGN	If alignment errors occur in the object program, the necessary alignment is simulated and the program continued as normal. *Note* Simulation uses a lot of computer time and should therefore only be used to make program tests easier.
NOALIGN	If alignment errors occur in the object program, the ERROR condition is set. The associated ONCODE value is 8098.

5.5 Check activation (ACTIVE)

This control statement is used to activate and enable checkpoints for the program run which are inserted via *COMOPT DEBUG = BREAKPOINT (x,...).

```
ACTIVE = { YES }
         {     }
         { NO  }
```

Default: ACT = YES

5.6 Program interrupt

A PL/I program run started at the interactive terminal can be interrupted by the user, by enabling the ESCAPE or BREAK functions at the interactive terminal (for Data Display Devices 8160 and 8161 via the K2 key, for example). The program remains loaded and a slash is output at the terminal to indicate that commands can be entered. The following commands are of special interest here:

- /RESUME
  The program run is continued at the interrupt point.
- /INTR text
  The "text" is transferred to the PL/I program. Specification of "/INTR * STOP" always causes an abnormal termination of the program run. If interrupt handling was not explicitly inserted at compile time, other texts are ignored and the program run continued at the interrupt point.
- /TCHNG
  Logical terminal characteristics can be changed.
- /EOF
  Terminate file input from the terminal.

Further details may be found in the Command Reference Manual [2].

If the program run was started in a DO procedure, the interrupt initiates an inquiry as to whether the procedure is to be terminated. If it is terminated the program remains loaded and processing can proceed as above. It should be observed, however, that the remaining commands of the DO procedure will not be executed.

In addition to the interrupt mechanism described above, it is possible to interrupt the PL/I program at given points in order to evaluate the text transferred via the command "/INTR text" in the PL/I program. Within the PL/I program the ATTENTION condition can be used to ascertain which measures should be taken as a result of the INTR (interrupt) command and any texts it may have supplied.

In order to evaluate the INTR command, the external PL/I procedure must be compiled with

```
*COMOPT OPTIONS = INTERRUPT
```

Then interrupt points are inserted in these external procedures to test for any pending INTR command. They are, positioned as follows:

- After every label
- Before every END statement
- Before every RETURN statement

If it is ascertained that an INTR command is pending at one of the interrupt points, the ATTENTION condition is raised. The text specified for the INTR command is also passed to the ON variable ONINTR, and is available to the user via the built-in function ONINTR. If no text is specified, invoking ONINTR results in a blank string. The text syntax is described in the INTR command [2].

A complete description of the ATTENTION condition and the built-in function ONINTR may be found in the PL/I language reference manual [1]. Control of the compiler via *COMOPT OPTIONS=INTERRUPT is explained in section 3.6 of this manual.

An overview of the program run in the case of an interrupt is shown in Fig. 5-1. An example of the ON unit may be found in the description of the ATTENTION condition.

Fig. 5-1 Program run interrupt by BREAK at the terminal and input of the INTR command

If the program run is interrupted because of /INTR *STOP, the following are listed at SYSOUT:

- an interrupt message
- register states
- instruction environment dump

The FINISH condition (not ERROR or ATTENTION) is set.

A message for the ATTENTION condition can be output via CALL ERROUT; no messages are output via the system unit.

5.7 Description of the operands of the SDF command START-PLI1-PROGRAM

5.7.1 Overview of the operands

Name of the operand	Purpose
FROM-FILE	Specifies the name and input source of the object module or load module
CPU-LIMIT	Maximum program runtime in CPU seconds
MONJV	Monitoring of program runtime with job variables
START-PARAMETERS	Parameter input for the MAIN program
LANGUAGE	Output of program messages in English or German
ASSIGN-SYSLST ASSIGN-SYSOUT ASSIGN-SYSDTA	Assigns PL/I-TITLE
LISTING	Controls list output by the runtime system
HEAP-ADMINISTRATION	Controls standard area storage management
STACK-ADMINISTRATION	Controls stack storage management
TABULATOR-POSITION	Tabulator positions for output with PUT LIST

5.7.2 Description of the individual operands

FROM-FILE operand

This operand assigns the name and the input source of the object that is to be executed.

```
FROM-FILE = <full-filename 1..54 without gen> / *MODULE(...) /
            *PHASE(...)

      *MODULE(...)
        |
        |  LIBRARY = *OMF / <full-filename 1..54 without gen>
        |
        | ,ELEMENT = *ALL / <full-filename 1..54 without gen-vers>

      *PHASE(..)
        |
        |  LIBRARY = <full-filename 1..54 without gen>
        |
        | ,ELEMENT = <full-filename 1..54 without gen-vers>(...)
        |   |
        |   |  VERSION = *HIGHEST-EXISTING / <alphanum-name 1..24>
```

FROM-FILE = <full-filename 1..54 without gen>
<full-filename> is the name of the cataloged file containing the load module generated with TSOSLNK .

FROM-FILE = *MODULE(...)
The parameters of the *MODULE structure serve to assign an object module generated by the compiler or a prelinked module produced by TSOSLNK. These modules are stored as R-type elements in PLAM libraries.

FROM-FILE = *PHASE(...)
The parameters of the *PHASE structure are used to assign a load module generated with TSOSLNK; this is stored as a C-type element in a PLAM library.

See also the START-PROGRAM command in the manuals "User Commands (SDF Format)" and "Binder-Loader-Starter".

CPU-LIMIT operand

This operand specifies the maximum CPU time in seconds allowed for the execution of the program.

```
CPU-LIMIT = JOB-REST / <integer 1..32767>
```

MONJV operand

This operand specifies the name of a job variable for monitoring the execution of the program.

```
MONJV = *NONE / <full-filename 1..54 without gen>
```

START-PARAMETERS operand

This operand can be used to transfer a character string to the MAIN program as a parameter.

```
START-PARAMETERS = ' ' / <c-string 1..253>
```

LANGUAGE operand

This operand determines the language in which the messages of the module/program are output.

```
LANGUAGE = ENGLISH / GERMAN
```

ASSIGN-SYSLST operand

This operand assigns PL/I-TITLE to the system file SYSLST.

```
ASSIGN-SYSLST = STD / PARAMETERS(...)

     PARAMETERS(...)
       |  TO-TITLE = SYSPRINT / <alphanum-name 1..8>
       | ,MAX-LINE-SIZE = 132 / <integer 1..256>
       | ,LINES-PER-PAGE = 60 / <integer 1..256>
```

ASSIGN-SYSOUT operand

This operand assigns PL/I-TITLE to the system file SYSOUT.

```
ASSIGN-SYSOUT = STD / PARAMETERS(...)

     PARAMETERS(...)
       |  TO-TITLE = SYSOUT / <alphanum-name 1..8>
       | ,MAX-LINE-SIZE = 120 / <integer 1..256>
       | ,LINES-PER-PAGE = 60 / <integer 1..256>
```

ASSIGN-SYSDTA operand

This operand assigns PL/I-TITLE to the system file SYSDTA.

```
ASSIGN-SYSDTA = STD / PARAMETERS(...)

     PARAMETERS(...)
       |  TO-TITLE = SYSIN / <alphanum-name 1..8>
       | ,MAX-LINE-SIZE = 120 / <integer 1..256>
```

TEST-SUPPORT operand

This operand controls the debugging aids.

```
TEST-SUPPORT = NONE / PARAMETERS(...)

     PARAMETERS(...)
       |
       |  TOOL-SUPPORT = NONE / AID
       |
       | ,TEST-POINT-INTERRUPT = NO / YES
       |
       | ,DUMP(DMP) = NONE / ON-ERROR(...) / ON-TERMINATION(...)
       |
       |      ON-ERROR(...)
       |        |  PROCEDURE-NEST = YES / NO
       |        | ,HEAP-STORAGE = NO / YES
       |        | ,STACK-STORAGE = NO / YES
       |
       |      ON-TERMINATION(...)
       |        |  PROCEDURE-NEST = YES / NO
       |        | ,HEAP-STORAGE = NO / YES
       |        | ,STACK-STORAGE = NO / YES
       |
       | ,TRACE(TRC) = NONE / PARAMETERS(...)
       |
       |      PARAMETERS(...)
       |        |  PROCEDURE-ENTRY = YES / NO
       |        | ,PROCEDURE-EXIT = YES / NO
       |        | ,PROCEDURE-CALL = YES / NO
       |        | ,LABELLED-STATEMENT = YES / NO
       |        | ,GOTO-STATEMENT = YES / NO
       |        | ,ADDITIONAL-OUTPUT = *NONE / TERMINAL
```

TEST-SUPPORT = NONE
No debugging aid support

TEST-SUPPORT = PARAMETERS

TOOL-SUPPORT = NONE / AID
The program is loaded without (NONE) or with (AID) LSD records.

TEST-POINT-INTERRUPT = NO / YES
Activation of test points

DUMP = NONE / ON-ERROR(...) / ON-TERMINATION(...)
Control of DUMP output

DUMP = NONE
No dump output

DUMP = ON-ERROR(...)
Dump output in the event of abnormal program termination only

PROCEDURE-NEST = YES / NO
Output of procedure nesting at end of program

HEAP-STORAGE = NO / YES
Dump of entire standard area

STACK-STORAGE = NO / YES
Dump of entire stack

DUMP = ON-TERMINATION(...)
Dump output on normal and abnormal termination of program.
For parameters of the ON-TERMINATION structure, see ON-ERROR structure.

LISTING operand

This operand controls list output by the runtime system.

```
LISTING(LST) = NONE / PARAMETERS(...)

     PARAMETERS(...)
       |  OPTIONS = NO / YES
       | ,SUMMARY = NO / YES
       | ,ADDITIONAL-OUTPUT = *NONE / *TERMINAL
```

LISTING = NONE
No list output

LISTING = PARAMETERS(...)

OPTIONS = NO / YES
YES: Output of the active control statements.

SUMMARY = NO / YES
YES: Output of program statistics

ADDITIONAL-OUTPUT = *NONE / *TERMINAL
*TERMINAL: Lists are additionally output on the terminal.

HEAP-ADMINISTRATION operand

This operand controls standard area storage management and allows optimization.

```
HEAP-ADMINISTRATION = STD / PARAMETERS(...)

     PARAMETERS(...)
       |  PRIMARY-ALLOCATION = 16 / <integer 1..524288>
       | ,SECONDARY-ALLOCATION = 16 / <integer 1..524288>
       | ,MAXIMAL-SIZE = 512 / <integer 1..524288>
```

HEAP-ADMINISTRATION = STD
The default values of the following PARAMETERS structure are to apply.

HEAP-ADMINISTRATION = PARAMETRS(...)

PRIMARY-ALLOCATION = 16 / <integer 1..524288>
Initial size

SECONDARY-ALLOCATION = 16 / <integer 1..524288>
Extension

MAXIMAL-SIZE = 512 / <integer 1..524288>
Maximum size

STACK-ADMINISTRATION operand

This operand controls stack storage management and allows optimization.

```
STACK-ADMINISTRATION = STD / PARAMETERS(...)

     PARAMETERS(...)
       |  PRIMARY-ALLOCATION = 16 / <integer 1..524288>
       | ,SECONDARY-ALLOCATION = 4 / <integer 1..524288>
```

STACK-ADMINISTRATION = STD
The default values of the following PARAMETERS structure are to apply.

STACK-ADMINISTRATION = PARAMETERS(...)

PRIMARY-ALLOCATION = 16 / <integer 1..524288>
Initial size

SECONDARY-ALLOCATION = 4 / <integer 1..524288>
Extension

TABULATOR-POSITION operand

This operand sets the tabulator positions for output of data elements using the PUT LIST statement. Tabulators are preset at increments of 10 (1, 10, 21, ...).

```
TABULATOR-POSITION = EVERY-TEN / list-poss: <integer 1..256>
```

5.7.3 Mapping of SDF operands to RUNOPT operands

SDF operand	**RUNOPT operand**
FROM-FILE	1)
CPU-LIMIT	1)
MONJV	1)
START-PARAMETERS	ARGUMENT
LANGUAGE	FORMAT
ASSIGN-SYSLST=STD	SYSFILE=SYSLST(SYSPRINT)
ASSIGN-SYSOUT=STD	SYSFILE=SYSLST(SYSOUT)
ASSIGN-SYSDTA=STD	SYSFILE=SYSLST(SYSIN)
TEST-SUPPORT	
TOOL-SUPPORT	1)
TEST-POINT-INTERRUPT	ACTIVE
DUMP PROCEDURE-NEST=YES HEAP-STORAGE=YES STACK-STORAGE=YES	DUMP= SNAP AREA STACK
TRACE PROCEDURE-ENTRY=YES PROCEDURE-EXIT=YES PROCEDURE-CALL=YES LABELLED-STATEMENT=YES GOTO-STATEMENT=YES ADDITIONAL-OUTPUT=*TERMINAL	TRACE= PROCTRACE RETURNTRACE CALLTRACE LABTRACE GOTO-TRACE TERMINAL
LISTING	LISTING=
SUMMARY=YES	SUMMARY
OPTIONS=YES	OPTIONS
ADDITIONAL-OUTPUT=*TERMINAL	TERMINAL
HEAP-ADMINISTRATION	STORAGE=AREA()
STACK-ADMINISTRATION	STORAGE=STACK()
TABULATOR-POSITION=()	TABULATOR=()

1) Entered in the EXECUTE command

6 File access by PL/I programs

6.1 General

This section discusses the BS2000 file characteristics and processing capabilities of significance to PL/I programs and explains the relationships with the appropriate PL/I language elements. A PL/I programmer should note that he can only use individual PL/I language elements if the referenced BS2000 file possesses certain characteristics. In particular, the set of data that constitutes the file must be structured/ organized in such a way that it meets the requirements of the program.

In section 6.2 there is a brief description of the files available in BS2000 and their characteristics. Section 6.3 gives an overview of the logical files of PL/I programs and describes the types of file organization and the environment. Section 6.4 deals with the problems involved in the assignment of PL/I files to BS2000 files.
Sections 6.5 and 6.6 examine important relationships involved in streamoriented and record-oriented input and output in connection with BS2000 types of file organization, and give information concerning the specifications in the /FILE command.

6.2 BS2000 files

In this section there is a brief description of the BS2000 file types in so far as they are of significance for PL/I programs. System files and user files will be dealt with here.

6.2.1 System files

System files are used by the system for specific functions. The following file types are of importance for PL/I programs and, correspondingly, for the compiler:

- SYSCMD
  This is the file from which the operating system takes all its commands. The batch mode default is the spoolin (card input) or a file activated by the ENTER command. The interactive mode default is the terminal. SYSCMD can be temporarily assigned to any file by the DO command.
- SYSDTA
  System file SYSDTA can only be used as input file to the compiler and the user programs. The batch mode default is for SYSDTA to supply the data records which immediately follow the EXEC command, no matter whether the task was activated by spoolin (card input) or by the ENTER command. The records are delimited by the next command, with the BREAK and EOF commands (see [2]) acting as special delimiters. In interactive mode, SYSDTA is defaulted to the user's terminal, from where records are usually fetched individually (one line at a time).

  If the EXEC command is part of a procedure file, and the records which follow are to be processed via SYSDTA, then SYSDTA and system file SYSCMD must be identical (see SYSFILE command in section 3.4.1 and [2]).
- SYSOUT
  Operating system messages (acknowledgments, errors etc.) and PL/I runtime system messages (also from the compiler) are output via the system file SYSOUT. Furthermore, output from the object programs can be directed to SYSOUT. In interactive mode, the information that is to be written to the system file SYSOUT is output on the appropriate terminal. In batch mode, the output is stored in a spool file and spooled out on the high-speed printer once the task is terminated. The file is then deleted.

- SYSLST
  Output from the object programs is generally directed to SYSLST if the file name SYSPRINT is being used in the PL/I program. All the compiler listings are also, by default, directed to this system file. The contents of this spool file are output on the high-speed printer once the task is terminated, and subsequently deleted.
- SYSOPT and SYSIPT
  The system files SYSOPT and SYSIPT are not supported by PLI1.

User files can be assigned to the system files SYSDTA and SYSLST by means of the SYSFILE command (see sections 3.4.1 and 3.5). The relevant input information is then taken from them or the relevant output information is written to them. Figure 6-1 illustrates the possibilities outlined above.

System file	Preset (PRIMARY)		Controlled by command:
	Batch	Interactive	
SYSCMD	Spoolin file	Terminal	DO;ENDP
SYSDTA			SYSFILE;(ENDP)
SYSLST	Spoolout file	Spoolout file	SYSFILE
SYSOUT		Terminal	-

Fig. 6-1 Assignment of system files

In addition, the relationship between the title or the file name in the program and the name of the system file is established by the PLI1 control statement SYSFILE (see section 5.4.7).

Within certain limits, the user can control which outputs are to be directed to SYSLST and SYSOUT. The possibilities here are described in chapters 3 and 5 (control statements MESSAGE and SYSFILE). In a batch task, SYSLST and SYSOUT are files for printer output. In an interactive task, the data destined for the printer is written to the system file SYSLST, while the output for the terminal is placed in the system file SYSOUT. It is a general rule that the system files SYSDTA, SYSOUT and SYSLST can only be processed sequentially, and that SYSDTA can only be used as an input file and SYSOUT and SYSLST as output files. System files can be used in connection with stream-oriented or record-oriented input/output.

6.2.2 User files

6.2.2.1 General

While a system file is created and deleted automatically, allowing the user no influence on the file characteristics (unless he uses the SYSFILE command), in the case of user files he can determine the main features of the file himself. There are many attributes and specifications possible for a file. The following groups of file characteristics and processing specific variables may be distinguished in connection with PL/I programs:

- file name
- file link name
- file organization
- record format
- volume
- access authorization
- file size
- other characteristic data

All the characteristics of a file can be specified with the aid of the FILE or CATALOG command. These commands are used for creating and cataloging a file or, in the case of existing files, for altering or adding specifications. With the FILE command, moreover, it is possible to determine processing-specific variables such as file link name, open mode etc.

Virtually none of the characteristics/specifications are transferred to the catalog until the file is closed (CLOSE). The file remains in existence until it is deleted by an ERASE command. In the case of already existing files, most of the characteristics have already been determined. With these files, the modification options are limited.

Note

Instead of using the FILE command, it is possible to specify some file characteristics in the PL/I program, using the ENVIRONMENT attribute.

6.2.2.2 File name

The physical files (data sets) are identified within BS2000 by means of a file name that can be up to 54 characters in length. When the user creates a file, BS2000 prefixes the file name of the created file with a user identification (userid), up to 10 characters in length.

The file name is the first parameter to be specified in the FILE command. The user can, under certain circumstances, access the files of another user by employing the latter's user ID.

The general file name format is this:

```
[$userid.]name1[[.name2]...[(⎧generation⎫)]
                            ⎩version   ⎭
```

$userid. is a user identification and need only be specified if the user wishes to access a file belonging to a different user.

name1 is the actual file name if ".name2..." does not follow. It is the name of a category of files if followed by further subnames. Some BS2000 commands permit the specification of file categories (/ERASE, for example), these commands, when called, process more than one file. File names for the PL/I compiler (source program) and file names in FILE or SYSFILE commands are always names of individual files, i.e. categories cannot be specified.

```
(⎧generation⎫)
 ⎩version   ⎭
```

This information allows differentiation of files having the same name. "generation" is an absolute or relative numeric specification. "version" is used for tape files and can be a name. A special point to note is that the version specification is not included in the label recorded on tape. See [7] for further details.

*DUMMY Symbol for a dummy file with the following characteristics:

- Input:
  ENDFILE is signaled during the first read operation.
- Output:
  The records to be written are only transferred to the buffer assigned to the file, and not to external media. Since there is no catalog entry for dummy files, all the necessary specifications must be made in the FILE command, as they are required when a file is to be newly created.

Examples of valid file names:

```
INPUT
RESULT.0001
$SMITH:RELEASE:TEXT
```

The last file name refers to the file RELEASE.TEXT, which must have been cataloged under the userid SMITH with SHARE=YES if access is to be attempted by another user, whereas the first two examples denote files in the user's own catalog.

6.2.2.3 File link name

The file link name is a name which is independent of the data set and serves to temporarily establish the connection between the PL/I file and the BS2000 file indicated by the FILE command. The file link name is specified in the LINK parameter of the FILE command. A PL/I file is linked to a BS2000 file via the TITLE entry in the PL/I program. The TITLE is indicated in the OPEN statement of the program. If the TITLE entry is missing, the TITLE is taken from the PL/I file name. A TITLE obtained in this way must comply with the rules governing BS2000 LINK names. BS2000 files can only be referenced by PL/I via the file link name. Further information can be found in section 6.4.

6.2.2.4 File organization (access method)

The file organization determines how the records are recorded on the volume and specifies which method is used internally to access the individual records. The following BS2000 access methods are supported by the PL/I input/output system:

```
SAM     ISAM   PAM (or UPAM)
```

The access methods are indicated in the FILE command by the specifications in the FCBTYPE parameter, but the relationship with the PL/I organization form defined in the ENVIRONMENT attribute must be taken into account (see section 6.4.2).

The Basic Tape Access Method (BTAM) is not used by PLI1 and will not therefore be mentioned again. It is possible to access tapes from within PL/I programs by means of the SAM access method. The Evanescent Access Method (EAM) is not supported by PLI1 for user programs. It is used only by the compiler in connection with object module generation.

- SAM (Sequential Access Method)
  The SAM access method enables records to be processed sequentially. SAM files are processed in PL/I using CONSECUTIVE organization.
- ISAM (Indexed Sequential Access Method)
  An ISAM file can be processed either sequentially, or directly by way of keys. The position of the first byte of the key must be specified via the KEYPOS parameter in the FILE command or via KEYLOC in the ENVIRONMENT attribute. This position is invariable in the records of one file. The key length is specified as a number of characters via the KEYLEN parameter in the FILE command or via KEYLENGTH in the ENVIRONMENT attribute. The length is constant for all the records in one file. In PL/I, ISAM files are usually processed using INDEXED organization.

  CONSECUTIVE organization can be used for ISAM files to enable the processing of files that are also destined to be processed by the utility routines EDT and EDOR or by other languages (e.g. ALGOL). Fixed key lengths and a predetermined method of key representation are applicable here. See section 6.6.1.3 for further details.

  For ISAM files, the PAD parameter in the FILE command is of importance; this ensures efficient distribution of storage space at initial creation of the file if the user wishes to subsequently insert records in existing files. If there is no PAD specification in the FILE command, PAD=0 takes effect for files processed by PLI1.

Note

Records in ISAM files are always prefixed with 4 additional characters by the Data Management System if the files were created with RECFORM = F. This must be taken into consideration when examining memory dumps since these files appear to have a record length field like V-type files. For the user, this fact is only of significance in calculating the file size.

- PAM (Primary Access Method)
  Direct access on the physical block level is possible with the PAM access method. The blocks have a fixed length (2048 bytes) and are also known as PAM blocks. PAM files are used for the PL/I organization forms REGIONAL(1) and REGIONAL(3). PAM files can also be processed using CONSECUTIVE organization.

6.2.2.5 Record structure

There are three different formats for records in files:

Fixed-length records:	F format (for all access methods)
Variable-length records:	V format (for SAM, ISAM)
Undefined records:	U format (for SAM, PAM)

One file can only contain records having the same format. The record format is defined by the RECFORM parameter in the FILE command, or in the ENVIRONMENT attribute. The transfer of records between main memory and external storage (data set) is carried out block-by-block. The relationship between record length and block size is explained below.

- Record Length
  The physical length of a record in the data set is specified in the RECSIZE parameter of the FILE command, or in the RECSIZE entry in the ENVIRONMENT attribute. The record length in the case of the input/output statements in the source program generally relates to the length of the logical record, while with the entry in the FILE command the additional administration information has to be taken into account. INDEXED files are a special case if KEYLOC = 0 is supplied in the ENVIRONMENT attribute (see section 6.3.3).

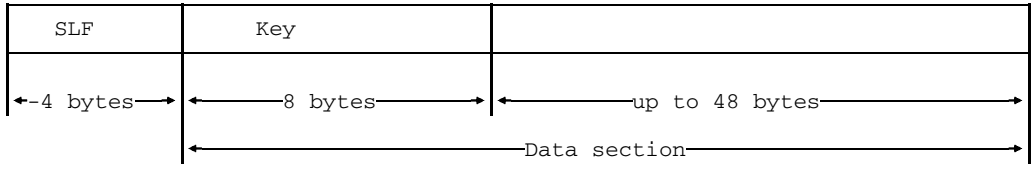
The administration information includes, for example, the length field of strings with the VARYING attribute if SCALARVARYING is specified for ENVIRONMENT (see section 6.3.5), the record length specifications when RECFORM = V, and the carriage control character for PRINT files.

If record format V is used, the information relating to the length of the individual record is given in the record length field (RLF), an information field prefixed to the beginning of the record. The two high-order bytes of the 4-byte record length field contain the length of the record (including the record length field). The two remaining bytes are reserved for file management purposes.

Example of a record with V format in an ISAM file:

```
/FILE DAT1,FCBTYPE=ISAM,RECSIZE=60,RECFORM=V,KEYPOS=5,KEYLEN=8
```

This record would have the following structure:



- BLOCK
  Data is not transferred between main memory (program) and external storage on a record-by-record basis; instead, the records are grouped into blocks and transferred block-by-block. The size of these blocks is indicated in the BLKSIZE parameter in the FILE command. The ENVIRONMENT attribute cannot be used to influence the size of the blocks. It is necessary to distinguish between the concepts of the physical and the logical block.

  A physical block is the unit of data transfer to and from an input/output device. This is 2048 bytes (one half-page) in length for all direct access volumes. We speak of a standard block, or of a PAM block. Files on magnetic tape can also be processed with nonstandard blocks, whose length is freely selectable, but must not exceed 32767 bytes.

  ISAM always uses standard blocks. SAM is capable of processing both standard and nonstandard blocks (with magnetic tape files). On direct access volumes, SAM operates with standard blocks.

  A logical block, however, may consist of several PAM blocks. The concept of the logical block enables the use of records that extend beyond the limits of the physical block. For this, the logical block size must be specified in the BLKSIZE parameter such that it is greater than or equal to the record length specification (RECSIZE). In the case of SAM files in the V format, a 4-byte block length field must also be taken into consideration. In addition, the Data Management System reserves a 12-byte control field for every logical block on PAM-key-free disk files in SAM and ISAM files.

  For records in the U format, only one record is transferred per block. With magnetic tape, only the current record length is transferred in this case. Where direct access volumes are used, one or more standard blocks always constitute the unit of transfer. The section of the main memory to which the operating system writes the data, or from which the data is read, is designated a buffer. In the case of nonstandard blocks, buffer and physical block are the same length. Otherwise, the buffer length corresponds to that of the logical block and can be a multiple of the standard block length (a maximum of 16 standard blocks).

In the case of record in the F and V format, one or more records are written to the buffer by PLI1. If, before a record is written, the free space remaining in the buffer is greater than or equal to the current record length, then the record is entered and the buffer contents are not immediately transferred to the data set. If there is no longer sufficient space available in the buffer, the buffer contents are transferred and the record is written to the "next" buffer. On account of the PAD specification in the FILE command, with ISAM files it is possible for the transfer to the data set to take place before the block is completely filled.

When the ISO code is used for tape files (CODE entry in the FILE command), the specifications in the block and record length fields appear as fourdigit decimal numbers (PIC'9999'), i.e. ISO code blocks in the variable format can have a length of up to 9999 bytes.

The following examples are designed to demonstrate the relationships between record, block and buffer:

Example 1

The following are assumed:

- SAM file
- Record format F: Record with 100 bytes
- Standard blocks
- Buffer size = 2 blocks ≙ 4096

Associated FILE command:

```
/FILE filename,LINK=title,FCBTYPE=SAM,RECFORM=F,
/    RECSIZE=100,BLKSIZE=(STD,2)
```

Alternatively:

```
ENVIRONMENT (... F,RECSIZE(100)...) and additionally,
/FILE filename,LINK=title,FCBTYPE=SAM,BLKSIZE=(STD,2)
```

The buffer is filled with 40 logical records; the rightmost 96 bytes are not used, but two complete blocks of 2048 bytes each are written. Note that record 21 of the buffer is contained in two blocks, the first 48 bytes in block 1 and the remaining 52 bytes in block 2.

Example 2

The following are assumed:

– SAM file
– Record format F: Record with 100 bytes
– Nonstandard blocks (for tape only)
– Buffer size = 2020

Associated FILE command:

```
/FILE filename,LINK=title,FCBTYPE=SAM,RECFORM=F,
/    RECSIZE=100,BLKSIZE=2020,DEVICE=TAPE,VOLUME=...
```

Alternatively:

```
ENVIRONMENT (...F(,100)...) together with
/FILE filename,LINK=title,FCBTYPE=SAM,BLKSIZE=2020,
/    DEVICE=TAPE,VOLUME=...
```

The buffer size is set at 2020 bytes. 20 records fit into the buffer, and a block with 2000 bytes is written.

Example 3

The following are assumed:

– SAM file
– Record format F: Record with 1500 bytes
– Standard blocks
– Buffer size = 1 block ≙ 2048 bytes

Associated FILE command:

```
/FILE filename,LINK=title,FCBTYPE=SAM,RECFORM=F,
/    RECSIZE=1500
```

Alternatively:

```
ENVIRONMENT (...F(,1500)...) together with
/FILE filename,LINK=title
```

This was an unsuitable choice because 548 bytes are wasted. A desirable buffer size would be 6144 bytes (3 blocks), whereby the buffer could accept 4 records, and 6000 of the 6144 bytes would be utilized. In this case, three blocks of 2048 bytes each would be written.

Example 4

The following are assumed:

– SAM file
– Record format V - with following record lengths (including RLF):
  100, 500, 300, 600, 200, 400, 700, 50, 100
– Nonstandard blocks (for tape only), BLKSIZE = 1000

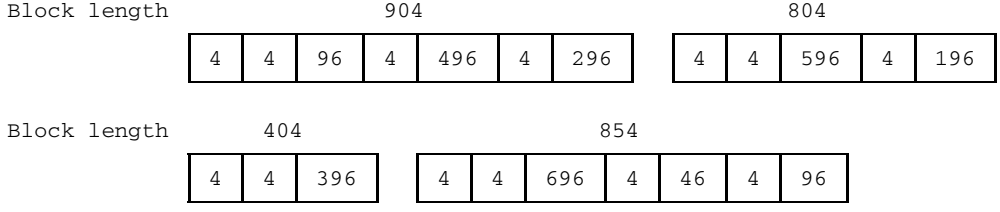
Associated FILE command:

```
/FILE filename,LINK=title,FCBTYPE=SAM,RECFORM=V,
/    RECSIZE=700,BLKSIZE=1OO0,DEVICE=TAPE,VOLUME=...
```

Alternatively:

```
ENVIRONMENT (... V RECSIZE(700)...) and additionally,
/FILE filename,LINK=title,FCBTYPE=SAM,BLKSIZE=1000,
/    DEVICE=TAPE,VOLUME=...
```

The following blocks are created:



Note

Nonstandard blocks containing variable-length records have a 4-byte length specification for each record and a 4-byte block length specification per block.

Example 5

The following are assumed:

– SAM file
– Record format F: Record with 8192 bytes
– Standard blocks
– Buffer size = 8 blocks = 16384 bytes

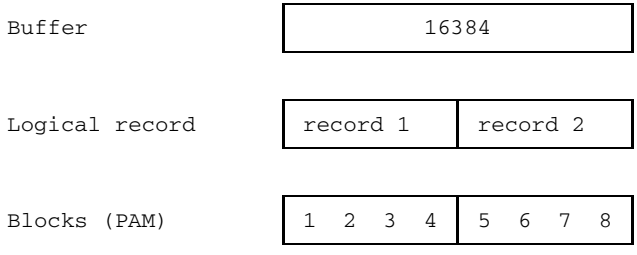
Associated FILE command:

```
/FILE filename,FCBTYPE=SAM,RECFORM=F,
/    RECSIZE=8192,BLKSIZE=(STD,8)
```

Alternatively:

```
ENVIRONMENT (...F(8192)...) together with
/FILE filename,LINK=title,FCBTYPE=SAM,BLKSIZE=(STD,8)
```

Each buffer contains two records, each record requires 4 standard blocks:



Example 6

The following are assumed:

- SAM file
- Record format V
- Maximum record length of 132 bytes
- Standard blocks

Associated FILE command:

```
/FILE filename,FCBTYPE=SAM,RECFORM=V,RECSIZE=132
```

Alternatively:

```
ENVIRONMENT (...V(, 132)...) together with
/FILE filename,LINK=title
```

A block has the same structure as illustrated in Example 4 and can accommodate at least (2048-4)/ 132 = 15 records of maximum length. Note that the filling of a block is terminated when the size of the current record exceeds the remaining free space. In the case in hand then, 131 bytes could remain unoccupied at the end of the block if the next record to be written were to require 132 characters. In this case, it would have to be entered in the next block.

6.2.2.6 Volumes

The external medium for data sets is defined by the DEVICE and VOLUME parameters in the FILE command.

The data media are divided into public and private volumes. Private volumes, which do not need to be permanently online while the task is executing, include all magnetic tapes and disk packs, as well as paper in the printer and card decks in the card reader ("paper devices"). A public volume is located on disk packs that are permanently connected to the system; it is preformatted ready to accept user files and is always online. The files stored on this medium can be available for use by more than one task at the same time.

The public volume is always addressed through the system defaults. If private volumes are to be used, then DEVICE and VOLUME must be specified. Paper devices can, if necessary, also be addressed via the SYSFILE command.

There is a connection between the method of organization of a file and the type of volume. Thus, for example, magnetic tape devices can only be processed with CONSECUTIVE organization, while the INDEXED, REGIONAL(1) and REGIONAL(3) types of organization, require a volume on a direct access device. In the PL/I program, the type of volume is normally of no significance since the typical file characteristics have already been established by other specifications. The only exceptions here are statements causing existing files to be overwritten; this is only possible in the case of direct access media, irrespective of the type of organization.

6.2.2.7 Access authorization

- Password
  By specifying a password, the user can protect his files against unauthorized use. There are read passwords and write passwords. The password consists of a constant of 1 to 4 characters in length which is stored in a special table in the system. If the file has a read password and a write password, then it is only necessary to specify the write password in order to be able to read the file also. The passwords are assigned to the files in the CATALOG command by means of the RDPASS and WRPASS parameters. If access to protected files is required, the PASSWORD command must be used in the relevant batch or interactive task.
- Access Protection
  The ACCESS parameter in the CATALOG command serves to determine whether a file may be used for reading only, or for both reading and writing. In this way it is possible to achieve protection against unintentional alteration of a data set. The SHARE parameter in the CATALOG command is used to specify whether only the owner can access the file, or whether the file can also be accessed by a job with a different user id.
- Temporary Write Protection
  If write protection is to apply only temporarily, this period can be stipulated in the RETPD parameter of the FILE or CATALOG command.
- Simultaneous Access (shared update)
  Using the SHARUPD parameter in the FILE command, it is possible to specify whether a file opened for writing can also be opened by other tasks for reading or writing (SHARUPD = YES), or whether more than one task can open the file if all require read access only (SHARUPD = NO).

  If, on the basis of the above rules, more than one task has been able to open the same file and a conflict situation occurs when two simultaneous attempts are made to access one record, one access is placed in a queue and the attempt repeated every 5 seconds. After 100 attempts, the TRANSMIT condition is set. Upon returning from the ON unit for the TRANSMIT condition, processing continues from the unit following the I/O statement.

  SHARUPD is only supported in the case of ISAM files.

6.2.2.8 File size

The size of a file, i.e. the amount of space it occupies, is specified by the parameter SPACE = (p,s) in the FILE command. If there is a SPACE option in the FILE command, then whenever the FILE command is executed

- the storage space is extended by p PAM pages (2048 bytes per page), with further pages being added if necessary to make the total number of pages a multiple of 3. If the file does not yet exist, this value is a primary allocation; if the file already exists, the value is an extension, or a reduction in the case of a negative value.
- Where specified, the value s (secondary allocation) is included in the file specification. It is evaluated whenever the PL/I program ascertains while writing records that there is insufficient storage space available. The storage space is then extended by s PAM pages, with further pages being added, if necessary, to make the total number of pages a multiple of 3. The value s has no significance in connection with REGIONAL.

If a new file is created by the FILE command and no SPACE option is made, then normally SPACE=(3,3) will be assumed unless a different value is selected at system generation.

The extension of storage space for files is time-consuming, so it is advisable to select the values such that extensions will be required as infrequently as possible.

Further details can be found in the description of the FILE command. The current storage occupancy level in PAM pages and the secondary allocation (value s from the SPACE parameter) can be ascertained by using the FSTATUS command.

If a file is opened in the PL/I program, the storage space requirement for the file and the value of the secondary allocation (i.e. the value s from the SPACE = (p,s) option) are already contained in the file specification. Depending on the attribute with which the file is opened, the following ensues:

INPUT	The values are not changed.
UPDATE	During the output of records, the storage space is extended by the secondary allocation, if necessary.
OUTPUT	During file opening, the current storage space and the secondary allocation value are corrected, if necessary (see below). During the output of records, the storage space is extended by the secondary allocation, if necessary.

In order to prevent an error occurring as a result of an insufficient SPACE value, when the file is opened with OUTPUT, the options in the file specification concerning the current storage space occupancy level and the secondary allocation are corrected, if necessary. The following applies in this connection:

SAM CONSECUTIVE

The current storage space and the secondary allocation value are increased, if necessary, such that they are exactly divisible by both 3 and the buffer size n specified by BLKSIZE=(STD,n) in the FILE command.

When extending a file (/FILE..., OPEN=EXTEND), the storage space is only corrected in the manner described above if the unused part is zero or does not satisfy the above conditions.

ISAM INDEXED
ISAM CONSECUTIVE

The current storage space is increased, if necessary, such that it is at least n + 1 pages, where n is the buffer size as specified by BLKSIZE=(STD,n) in the FILE command, and such that it is also divisible by 3.

PAM CONSECUTIVE

The values are not corrected.

PAM REGIONAL The storage space is increased, if necessary, such that it is at least r + 1 pages, where r is the size of a region, and such that it is also divisible by 3. The size of a region r is calculated as follows:

- with REGIONAL(1): from the record length d, as indicated in the file specification /FILE RECSIZE = x or ENV(RECSIZE(x)).
- with REGIONAL(3): from the buffer length b, as indicated in BLKSIZE=b in the FILE command.

6.2.2.9 Other file specifications

The PL/I user should also be aware of the following specifications of a user file:

- The OPEN mode for a file is defined in PL/I by the file attributes. On output, an additional control facility is available where required through the OPEN specification in the FILE command.
- For files opened with SEQUENTIAL in PL/I, it is possible to indicate that the new records are included in the case of an existing file through OPEN=EXTEND or, for PAM files, through OPEN=INOUT. Otherwise, the specifications of the old file are deleted and the old records are no longer available; the specifications are defined anew, as for a new file.

The effects of the FILE and CAT command parameters that are not dealt with in section 6.2.2 are described in [2]. They are, as a rule, of no special significance as far as PL/I is concerned.

6.2.3 Interfaces between the PL/I input/output system and BS2000

For accessing files, the input and output routines of PLI1 use some of the macros provided by the Data Management System (DMS) and the Executive of the operating system.

The following DMS macros are used:

OPEN	for opening a file
CLOSE	for closing a file
FCB	for defining a File Control Block
IDFCB	for providing an FCB with symbolic names
FSTAT	for supplying information on the status of a file
RDTFT	for reading the Task File Table

The transfer and access macros of the appropriate DMS access method.

The following Executive macros are used:

RDATA	for SYSDTA
WRLST	for SYSLST
WROUT	for SYSOUT
WROUT	for DISPLAY without REPLY
WRTRD	for DISPLAY with REPLY in terminal mode
TYPIO	for DISPLAY with REPLY in batch mode
WRTRD	for TRANSIENT file
WROUT	for TRANSIENT file

6.3 Program files and ENVIRONMENT attribute

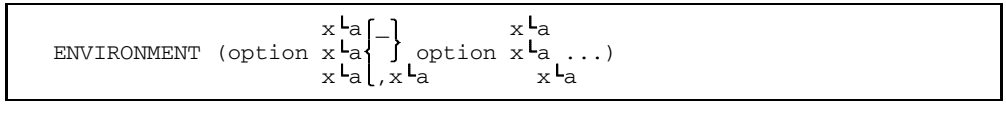
The PL/I language recognizes the data type FILE, which is used substitutionally for program-external data sets. The PL/I file is declared by means of the FILE attribute. The data sets are normally organized into records. However, PL/I implements two methods of transferring data between program and data set:

- the stream-oriented or character-by-character transfer of data (file attribute STREAM). In this case, the input/output system handles the data set as a continuous stream of characters and makes the data available to the program element-by-element, or arranges the output supplied by the program into a stream. See section 6.5 for details.
- the record-oriented transfer of data (file attribute RECORD). The input/ output system makes the data available to the user record-by-record, corresponding to the organization in the data set, or organizes the data set into records in such a way as they are supplied by the program. See section 6.6 for details. The stream-oriented and record-oriented transfer methods cannot both be used for the same file within a single OPEN/CLOSE cycle.

In the PL/I program, the files are referenced via their file names. In accordance with PL/I rules, the file name has a maximum length of 31 characters. This name need not be the same as the name of the data set (BS2000 filename). The contact is established by way of the TITLE specification on opening a file. See section 6.4.1.

When a file is declared in the program, information concerning the structure of the data set associated with this file can be specified with the aid of the ENVIRONMENT attribute. Some of this information corresponds to analogous options in the FILE command. If both are given, the option of the ENVIRONMENT attribute applies.

Format of the ENVIRONMENT attribute:



The possible options are given below in Figure 6-2. Further details can be found in the sections that follow:

Option	for	Meaning	
CONSECUTIVE INDEXED REGIONAL(1) REGIONAL(3)	DCL	Type of organization	
F F (b)[2] F ([b],r)[2]	DCL	Fixed record size	Corresponds to r: RECSIZE = r c: KEYPOS = p (for c ≠ 0) k: KEYLEN = k b: BLKSIZE = b F: RECFORM = F V: RECFORM = V U: RECFORM = U in the FILE command
V V (b)[2] V ([b],r)[2]	DCL	Variable record size	
U U (b)[2] U ([b]),r)[1)2]	DCL	Undefined record size	
RECSIZE (r)	DCL	Record size	
KEYLOC (c)	DCL	Start of key	
KEYLENGTH (k)	DCL	Key length	
BLKSIZE (b)[2]	DCL	Block size	
CTLASA CTLMACH	DCL	Carriage control character	CTLASA: RECFORM = (x,A) CTLMACH: RECFORM = (x,M)
SCALARVARYING	DCL	Input/output control with scalar VARYING variab.	
LEAVE	CLOSE	Do not rewind magnetic tape	
UNLOAD	CLOSE	Rewind and unload magnetic tape on close	
LIMCT (n)	DCL	REGIONAL (3): n + 1 regions are examined	
GENKEY	DCL	Group key in the case of INDEXED	
TERMINAL (m,e)	DCL	Device modes for TRANSIENT m: COMP|LINE l: ILCASE	In accordance with WRTRD macro operand: MODE = COMP | LINE ILCASE = YES

1) Block length option 'b' is ignored.

2) 'r', if omitted, is defaulted as follows:
for F: r = b
for V: r = b - 4

Fig. 6-2 ENVIRONMENT options

6.3.1 Types of organization

PL/I recognizes various ways of organizing records into a data set. There is a degree of interdependence between the PL/I organization and BS2000 access methods (see section 6.4.2). The type of organization cannot be specified in the FILE command.

Possible options for ENVIRONMENT:

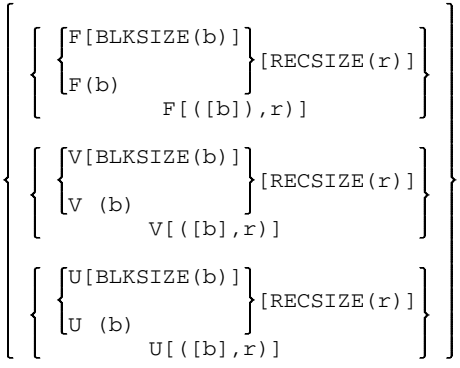
```
⎧CONSECUTIVE⎫
⎪INDEXED    ⎪
⎨REGIONAL(1)⎬
⎪REGIONAL(3)⎪
⎩           ⎭
```

- CONSECUTIVE
  The records are stored sequentially in a file and can only be processed sequentially. Data sets with CONSECUTIVE organization can be processed in stream- or record-oriented fashion.
- INDEXED
  The records are provided with a key (index). (The records are normally stored in ascending order of their keys.) Data sets with INDEXED organization can be processed sequentially or in direct access, but only in record-oriented fashion.
- REGIONAL(1)
  The file is made up of individual regions. One record is stored per region, the access key corresponding to the relative record number within the data set.
  Files with REGIONAL(1) organization can be processed sequentially or in direct access, but only in record-oriented fashion.
- REGIONAL(3)
  In the case of files having REGIONAL(3) organization, more than one record can be stored per region.
  For the purpose of addressing a record, a key which identifies both the region and the record within that region should be specified. Files with REGIONAL(3) organization can be processed sequentially or in direct access, but only in record-oriented fashion.

6.3.2 Record structure

The structure of the records in the data set can be described in either the FILE command or the ENVIRONMENT attribute.

Possible options for ENVIRONMENT:



If only 'b' is supplied, this option is developed as follows:

F (b) is equivalent to F(b,r) with r = b
V (b) is equivalent to V(b,r) with r = b-4

The same applies accordingly for BLKSIZE if RECSIZE is omitted.

- F|V|U
  This indicates whether the record length for all the records in the data set is fixed, variable or undefined. It corresponds to the RECFORM parameter of the FILE command. See section 6.2.2.5 for details.
- Option r
  This must be a positive integer constant. It corresponds exactly to the RECSIZE parameter of the FILE command. r is calculated from the length of the record to be transferred, taking into account additional information such as the record size field in V-type files and the keys in ISAM files.
  For illustration, refer to Figure 6-3.
  Details can be found in the following section and in 6.2.2.5.
- BLKSIZE and Option b
  A block size, when supplied in the ENVIRONMENT attribute for 'U', will be ignored by PLI1.

	Organization	File type	Rec. format	Key position		Record size
ENV	↓		↓	KEYLOC(c)		RECSIZE(r)
FILE		FCBTYPE =	RECFORM =		KEYPOS = p	RECSIZE = r
	CONSECUTIVE	SAM	F V U			Record 4 + record Record
		ISAM	F	1[1)]		Key + record
			V	5[1)]		4 + key + record
		PAM	F			Record
			V			4 + record
			U			Record
	INDEXED	ISAM	F	0	1	Key + record
				1...r		Record
			V	0	5	4 + key + record
				5...r		4 + record
	REGIONAL(1)	PAM	F			Record
	REGIONAL(3)	PAM	F			Record
			V			Record

Record: Length of PL I record in characters
Key: Length of key

Option not possible or unnecessary

1) Key length only 4: Key = FIXED BIN (31,0) (FORTRAN Key)
or 8: Key = PICTURE '(8)9'

Fig. 6-3 Determining the record size

6.3.3 Key options

For INDEXED and REGIONAL(3) definitions must be made concerning the record key.

Possible specifications for ENVIRONMENT:

```
{KEYLENGTH (k)        KEYLOC (c) }
```

- KEYLENGTH (k)
  This option indicates the key length as a number of characters. k must be a positive integer greater than zero. This option corresponds to the KEYLEN parameter in the FILE command.
- KEYLOC (c)
  This option is only of significance for INDEXED files; it indicates the position of the first character of the key. c must be a positive integer greater than or equal to zero. For values greater than zero, the option corresponds exactly to the KEYPOS parameter in the FILE command.

  When c = 0, PLI1 stores the key before the data record. The size of the record (useful information) plus the length of the key must then be specified as the record size. Moreover, in connection with KEYLOC(0), the record format (F or V) must be defined in the ENVIRONMENT attribute. If this is defined only in the FILE command, an UNDEFINEDFILE condition occurs on opening the file.

  Example

  ```
  ENVIRONMENT (INDEXED, F (,88), KEYLOC(0), KEYLENGTH(8))
  ```

  The records to be transferred must have a length of exactly 80 characters.

  Where KEYLOC or KEYPOS ≠ 0, and when information is written to a file, the key is always stored at the designated position in the record, i.e. any item of information in the program standing at this position in the record is overwritten in the file by the key.

6.3.4 Carriage control

Files intended for printing must have a carriage control character as the first character in each record. These control characters are generated automatically during stream-oriented output for files with the attribute PRINT. The coding of the control characters can be selected.

- CTLMACH

'00'B4	no line advance
'0n'B4	advance 'n' lines after print
'4n'B4	advance 'n' lines before print and one line after print if the output page is not empty.
'81'B4	advance to first line of new page after print
'8n'B4	advance according to channel 'n' after print
'C1'B4	advance to new page before print and if the output line is not empty, advance 1 line after print

  This option is not allowed for files with TITLE ('SYSPRINT') or TITLE ('SYSOUT') or for files which are to be redirected to system files SYSLST and SYSOUT.

Options supported for ENVIRONMENT:

$$\left\{\begin{matrix}\texttt{CTLASA}\\ \\ \texttt{CTLMACH}\end{matrix}\right\}$$

Default: CTLMACH for PRINT files;
otherwise, none (/FILE...,RECFORM = (x, N).

These options have their equivalents in the RECFORM parameter of the FILE command.

- CTLASA

  The following control characters are declared:

'4E'B4	(plus sign)	no line advance
'40'B4	(space)	advance 1 line before print
'F0'B4	(character 0)	advance 2 lines before print
'60'B4	(minus sign)	advance 3 lines before print
'F1'B4	(character 1)	advance to 1st line of new page

  'Cn'B4 advance according to channel 'n' before print and 1 line after print if the output line is not empty.

6.3.5 Controlling the VARYING variable

In the internal representation of string variables with the VARYING attribute, the storage space containing the value is prefixed with a halfword which accommodates the current length of the string. If the contents of a scalar variable with the VARYING attribute are entered as a record into a file, it is necessary to decide whether the length specification should also be output or not. If a record written in this way is read again, then it is necessary to indicate whether the length specification is contained in this record or not.

Control of this process is effected when declaring the file by specifying SCALARVARYING for the ENVIRONMENT ATTRIBUTE. The declaration has the following form:

```
DCL a FILE RECORD etc. ENVIRONMENT (SCALARVARYING)
```

The SCALARVARYING option is only evaluated for files that are opened with the RECORD attribute. The following relates to scalar variables with the VARYING attribute that are used in conjunction with data records. The following cases are dealt with:

- A record is read using READ INTO (a)
- A record is written using WRITE FROM (a) or REWRITE FROM (a)
- A record is read using READ SET
- A record is written using LOCATE SET.

The following subsections deal with character string variables (CHARACTER). The same applies analogously to bit string variables (BIT).

Furthermore, the length specification for record format V (RECFORM=V) is not covered by the discussions here, nor is any key that may be present. These must also be taken into consideration in accordance with the descriptions given in the appropriate sections.

6.3.5.1 Reading a record using READ INTO

If a record is read into the scalar target variable a using READ INTO (a) and this variable has the attribute VARYING, two cases are distinguished:

If SCALARVARYING is not specified in the ENVIRONMENT attribute of the file, it is assumed that the VARYING length specification is not contained in the record. The record is assigned to the target variable, the length specification in the VARYING variable being set to the correct value in the process, i.e. to the length of the record.

If SCALARVARYING is specified in the declaration of the file, it is assumed that the first two bytes of the record contain the length specification. The first two bytes are then transferred to the length specification for the variable, and the remaining characters of the record are placed in the storage area used for the target variable. If "length specification > record length - 2", this may lead to undefined results in a subsequent access to the variable.

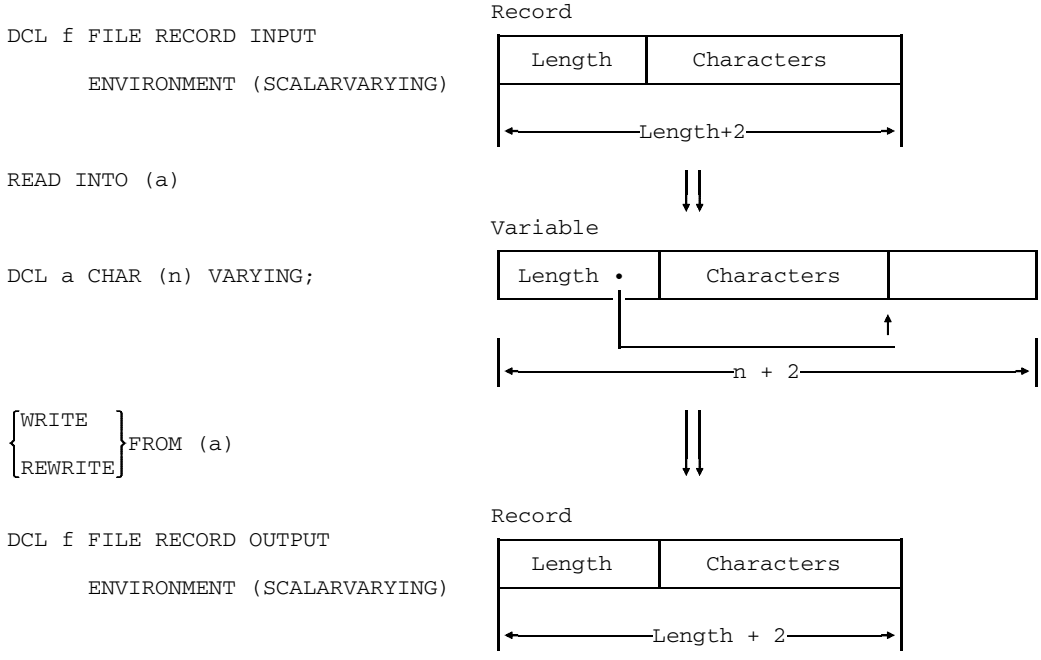


Fig. 6-4 Data transfer for a scalar character string variable of variable length with SCALARVARYING in the case of INTO or FROM

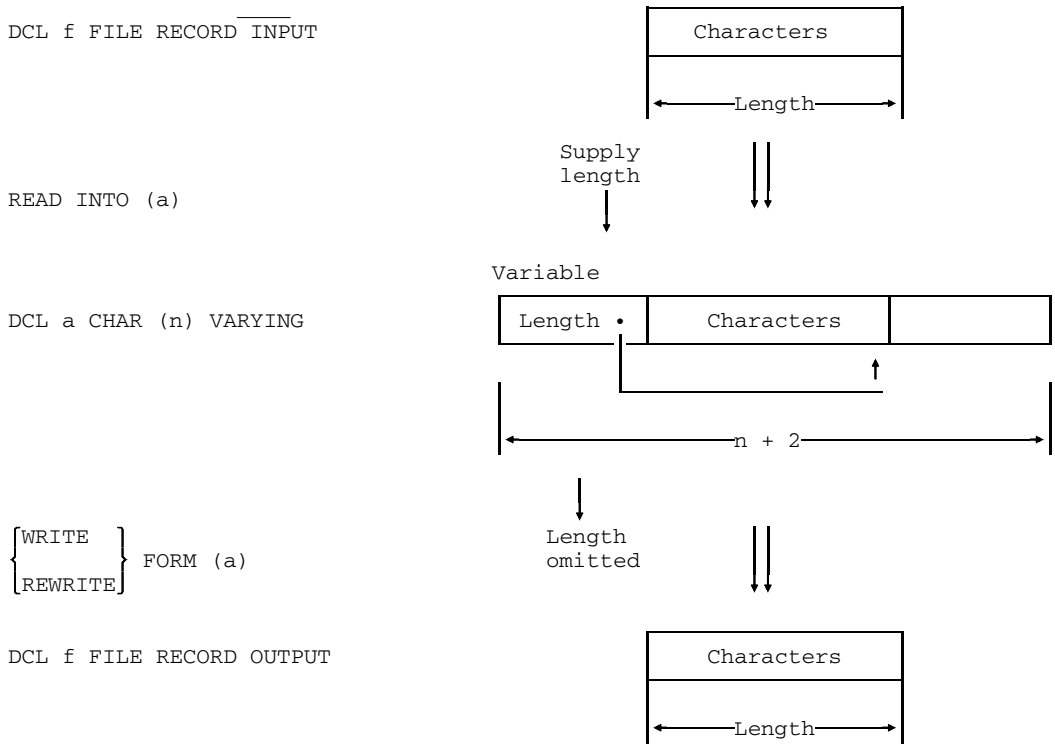


Fig. 6-5 Data transfer for a scalar character string variable of variable length without SCALARVARYING in the case of INTO or FROM

6.3.5.2 Writing a record using WRITE FROM

If a record is written from the scalar source variable a to a file using WRITE FROM (a) or REWWRITE FROM (a), and the variable has the attribute VARYING, two cases are distinguished.

If SCALARVARYING is not specified during the declaration of a file, then only the value, but not the length specification, of the source variable is output as a record. The length of the record is thus obtained from the current length of the source variable.

If SCALARVARYING is specified during the declaration of the file, then the length specification and the value of the source variable are output as a record. The length of the record is thus calculated from the current length of the variable + 2.

6.3.5.3 Reading a record using READ SET

If a record is read using READ SET, then there is no target variable to which it can be transferred; a pointer indicating the current record in the input buffer is simply provided.

When the READ statement is executed, the attributes of the variable which is to be used in accessing the record are not known.

SCALARVARYING is of no significance in this case.

It is then only possible to access the record with a variable having the attribute VARYING if the first two bytes contain a length specification. As with all access using BASED variables, the user himself is responsible for ensuring that the correct length specification is present in the record.

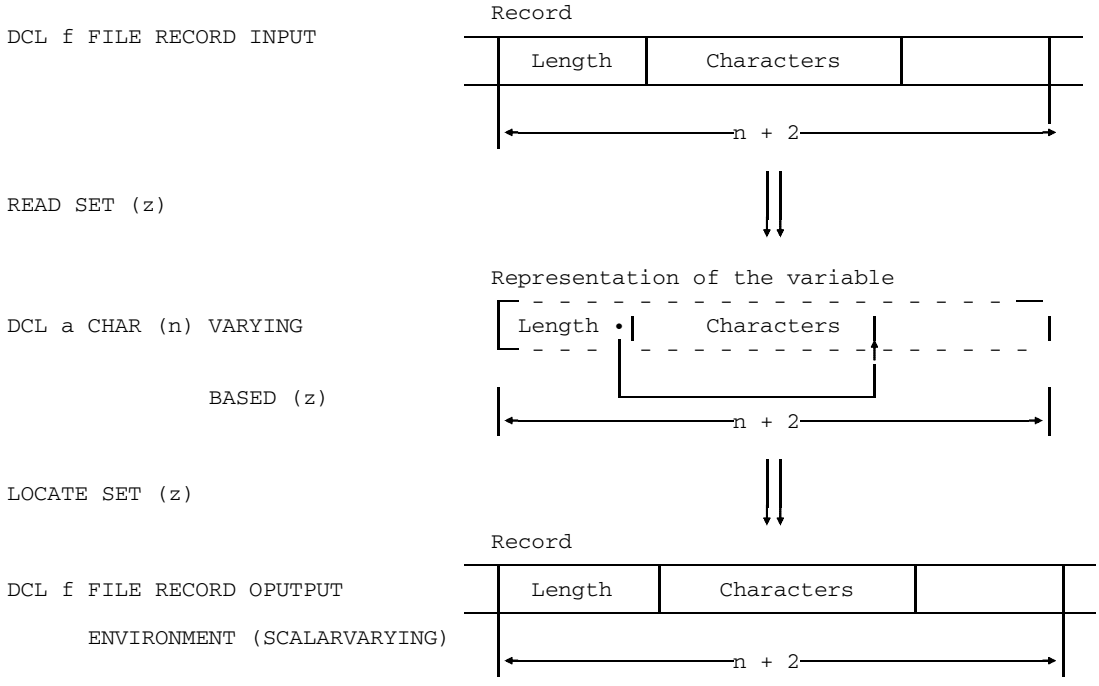


Fig. 6-6 Record transfer for a scalar character string variable of variable length in the case of LOCATE SET

6.3.5.4 Writing a record using LOCATE SET

If a record is written using LOCATE SET, the effect of this statement is simply to reserve storage space for a record in the output buffer. This storage space simultaneously constitutes the storage space for the variable. A pointer is set which points to the beginning of this storage space.

This storage space can be filled with information by way of a BASED variable; its contents will be placed in the file at some later time.

If SCALARVARYING is specified for the file, in addition to the storage space for the value a further 2 bytes for the length specification are reserved for a BASED variable with the attribute VARYING, by means of which a value may be assigned to the record. The record always has the maximum length + 2 bytes, irrespective of the current length of the values.

If SCALARVARYING is not specified for the file, during storage reservation for variables with the attribute VARYING no storage space is allowed for the length specification. In the case of assigning a value by way of a BASED variable with the attribute VARYING this may lead to the destruction of other data without this fact being recognized. Therefore, this mode of operation is not permitted. The responsibility lies, as with BASED variables in general, with the user.

6.3.6 Device control using TRANSIENT files

TRANSIENT-organized files control the terminal by means of the WRTRD and WROUT macros (see BS2000 Macro Calls [16]). The following device modes can be controlled via the ENVIRONMENT option TERMINAL:

```
ENV (TERMINAL (m,e))  where m = COMP|LINE
                      and   e = ILCASE.
```

The options correspond to the identical operands of the WROUT or WRTRD macro call and have the same effect.

The ILCASE option causes lowercase letters to be transferred as well when input is entered from the interactive terminal (corresponds to ILCASE=YES of the WRTRD macro).

The option COMP means compatible operating mode for interactive terminal input/output (corresponds to MODE = COMP of the WROUT or WRTRD macro).

The option LINE means LINE mode for interactive terminal input/output (corresponds to MODE = LINE of the WROUT or WRTRD macro). The LINE mode permits input or output to be composed of more than one line so that formatting is possible during input and output. The following control characters may be contained in the information to be transferred:

X’15’: New line
X’0C’: New (i.e. previously deleted) screen
X’1D’: Start of italics or flashing text
X’1E’: End of italics or flashing text
X’0E’: Start of second character set on interactive terminal
X’0F’: End of second characters set on interactive terminal

The appropriate ENVIRONMENT option in conjunction with RECSIZE, for example, enables the complete screen contents to be read or written with one READ or WRITE statement using the above device control characters. The ENVIRONMENT options TERMINAL are only effective for the interactive terminals 816x and 9750.

6.3.7 Other options

Further options are described in the following sections:

LEAVE, UNLOAD	6.7.4
LINCNT	6.6.4.1
GENKEY	6.6.2.1

6.4 Assigning program files to BS2000 files

The relationship between a PL/I file and a BS2000 file (assignment) is established via the file link name. On the BS2000 side this concerns the LINK specification in the FILE or CHANGE command; on the PL/I side, the TITLE specification in the OPEN statement.

If there is no TITLE specification in the OPEN statement, the file name specified in the definition is used to form the TITLE. Special rules apply to the system files; these are contained in the following section. Assigning PL/I files to BS2000 files is only practical when the type of organization and the access method are compatible.

6.4.1 Rules governing the assignment of file names

The connection between the PL/I program and a BS2000 file is effected either explicitly by the OPEN statement or implicitly during the OPEN process. In the case of a missing TITLE specification or the implicit opening of files, the file name provides the TITLE. The TITLE is interpreted as the BS2000 link name when the PL/I file is assigned to the BS2000 file. For this reason, the TITLE must comply with the rules governing link names. These rules permit a character string of up to 8 characters. If the PL/I TITLE or the PL/I file name used as a TITLE is longer than 8 characters, only the first 8 characters are taken. Also in the TITLE all "_" characters are always replaced by "$".

Note

If the PL/I file is defined with the attribute EXTERNAL, the file name is abbreviated by PL/I in accordance with the conventions for external names to 7 characters using the 4 : 3 rule (the first four and the last three characters of the name). This should be borne in mind when examining the linkage editor listings.

The original name (i.e. not the one abbreviated to 7 characters) is available when the TITLE has to be formed from the file name. Likewise, the full file name appears in the 'ONFILE specification' with the dummy variables and in error messages.

The following assignments predetermined by the control statement SYSFILE (see section 5.4.7) apply to the system files SYSDTA, SYSLST and SYSOUT:

- If 'SYSIN' is determined as the TITLE in the PL/I program, this refers to the BS2000 system file SYSDTA.
- If 'SYSPRINT' is determined as the TITLE in the PL/I program, this refers to the BS2000 system file SYSLST.
- If 'SYSOUT' is determined as the TITLE in the PL/I program, this refers to the BS2000 system file SYSOUT.

Note

With the GET statement, PL/I always replaces a missing FILE specification with SYSIN, while SYSPRINT is used for the FILE specification when the corresponding situation occurs with the PUT statement.

The assignments between TITLE and system file given above can be changed by the control statement SYSFILE (see section 5.4.7). If some other assignment is stipulated at the start of a PL/I program, any PL/I TITLEs desired may be connected with the system files. Note also that by using the SYSFILE command the system files can be temporarily reassigned to user files. It is also possible to direct other outputs (e.g. error messages) to SYSPRINT or SYSOUT. Control can be exercised over these outputs by means of the control statements MESSAGE, FORMAT and DIAGNOST (see chapters 3 and 5).

When using system files, note also that only sequential processing is possible; however, record or stream-oriented operation may be employed. The record and stream-oriented processing modes must not occur together in a single OPEN/CLOSE cycle for the same system file.

Note further when using system files that exact chronological coordination between the input/output system and other services that require the same system file cannot always be guaranteed. Thus, for example, a trace output can be written to SYSLST with complete disregard to the fact that the input/output system has previously prepared an output for SYSLST but not yet executed the output operation.

Another special case concerns the use of the DISPLAY statement. Depending on the RUNOPT option SYSFILE=DISPLAY (see section 5.4.7) the DISPLAY statement with REPLY writes to or reads from the interactive terminal, the operator console, or to the system file SYSOUT or from the system file SYSDTA.

Summary

The following steps are involved in assigning PL/I files to the physical files in BS2000:

- Determining the TITLE of the file from the TITLE specification in the OPEN statement, or alternatively from the PL/I file name in accordance with the above rules. Note:

GET without FILE or STRING PUT without FILE or STRING	GET FILE (SYSIN) PUT FILE (SYSPRINT)
OPEN without TITLE	OPEN TITLE('filename')

- Checking whether this TITLE is assigned to a system file by means of the control statement SYSFILE. The following are preset:

Title	System file
SYSIN SYSPRINT SYSOUT	SYSDTA SYSLST SYSOUT

  Note that the system files SYSDTA and SYSLST can be reassigned to a user file by means of the command SYSFILE (see section 3.4.1 and [2]).

- If the TITLE is not assigned to a system file by means of the control statement SYSFILE, there must be in existence a user file with the identical LINK name.
- If the TITLE cannot be identified during these search operations, an UNDEFINEDFILE condition will be reported.

The effect of a PL/I program may differ depending on the result of the above chain of decisions. For if the TITLE is found in the list for the control statement SYSFILE, the file is processed using BS2000 Executive macros. Otherwise, DMS macros are used. DMS macros take into consideration the characteristics of the user files such as record length, block size etc. However, if PLI1 decides in favor of Executive macros, the options applicable to the system files take effect (see section 6.5.1, for example, in this connection). In particular, there may be changes in characteristics such as RECSIZE and BLKSIZE if a user file is processed like a system file by way of the SYSFILE command.

If the intention is to process PL/I files having the TITLE 'SYSIN', 'SYSPRINT' or 'SYSOUT' not with Executive macros but with DMS macros the assignment defined by the control statement *RUNOPT SYSFILE or the preset assignment, as the case may be, must be canceled. For example, any names which do not occur in the program may be used for this (e.g. *RUNOPT SYSFILE = SYSDTA (DUMMY)).

6.4.2 Rules governing the assignment of organization methods

Use of the BS2000 access methods employed in the input/output system of the PL/I programs depends on the file organization specified in the program and on the physical file. Determining factors here are the file attributes specified at file definition time such as SEQUENTIAL, DIRECT etc., and the specification of the organization method in the ENVIRONMENT attribute. The following table gives an overview of the permissible combinations.

PL/I file Access	File organization	BS2000 Access method
SEQUENTIAL (without KEYED)	CONSECUTIVE	SAM[1)] PAM ISAM[1)] System files[1)]
	INDEXED	ISAM
	REGIONAL(1)	PAM
	REGIONAL(3)	PAM
SEQUENTIAL KEYED	INDEXED	ISAM
	REGIONAL(1)	PAM
	REGIONAL(3)	PAM
DIRECT KEYED	INDEXED	ISAM
	REGIONAL(1)	PAM
	REGIONAL(3)	PAM

1) Only these combinations can be used for stream-oriented processing

There are other rules that determine the permissibility of certain forms of input/output operations for the above combinations. These are to be found under the particular file organization method in section 6.6.

The following are preset:

- If no organization method is defined by the ENVIRONMENT attribute, the file is processed as a CONSECUTIVE file.
- If the access method was not defined by the FILE command, PLI1 defines the following:
  - SAM for CONSECUTIVE files
  - ISAM for INDEXED files
  - PAM for REGIONAL files

6.4.3 Relationship between FILE command and ENVIRONMENT specification

Section 6.3 discussed which file characteristics can be defined in the program with the ENVIRONMENT attribute. Certain of these characteristics may only be defined in this manner; with others the definition may be contained in the program or in the FILE command.
A third group of characteristics can be defined only in the FILE command. Figure 6-7 gives an overview. Furthermore, certain contextual defaults may become effective.

		ENVIRONMENT specific.	FILE command
Organization		CONSECUTIVE INDEXED REGIONAL(1) REGIONAL(3)	
File type			FCBTYPE = {SAM | ISAM | PAM}
Record	Format	F V U	RECFORM = {F | V | U}
	Length	RECSIZE (r)	RECSIZE = r
Key	Location	KEYLOC (c) for c = 0	KEYPOS = 1|5 [2]
		KEYLOC (c) for c ≠ 0	KEYPOS = c
	Length	KEYLENGTH (k)	KEYLEN = k
Block size		BLKSIZE (b) [3]	BLKSIZE = b
Combinations		F (b) | F (b,r) [3] V (b) | V (b,r) [3] U (b) | U (b,r) [1]	RECFORM = F|V|U BLKSIZE = b RECSIZE = r
Storage space requirement			SPACE = s
Carriage control characters		CTLMACH	RECFORM (x,M)
		CTLASA	RECFORM (x,A)
LOCATE and READ SET control		SCALARVARYING	

1) Block size b is ignored
2) 1: for format F
   5: for format V
3) 'r', if omitted, is defaulted as follows:
   for F: r = b
   for V: r = b - 4

Fig. 6-7 Relationship between ENVIRONMENT attribute and FILE command

6.4.4 Determining the file characteristics

Every file in BS2000 has associated with it certain characteristics which describe the type and structure of the file. In the case of an existing file these characteristics have already been defined, while for a new file they must first be compiled from the information supplied by the user.

If a file is opened with OPEN INPUT or OPEN UPDATE in a PL/I program, then it is assumed that an associated BS2000 file already exists and is entered in the file catalog. This presupposes also that the characteristics of the BS2000 file are compatible with the set of attributes of the PL/I file and with the ENVIRONMENT options. Appropriate input/output statements can be used to add, modify or erase records in the file, but the characteristics of the file cannot be changed (except, for example, SPACE).

If a file is opened with OPEN OUTPUT, then it is normally assumed that no corresponding file exists and that a new file is being created. However, if a corresponding file does already exist, it is assumed that this file is to be replaced by a new file, i.e. the old file will be erased.

Under certain conditions when using the OPEN operand of the FILE command it is possible to ensure that the old file remains in existence. In this case the process is similar to that for OPEN UPDATE, i.e. the old records are retained and the new records are added.

When a new file is created, the file characteristics must be laid down. The user can specify the desired values

- in the PL/I program through the ENVIRONMENT specification or
- in the FILE command.

The possible specifications are explained in the preceding sections. Should the characteristics be incomplete at this stage, then - insofar as is possible and practical - preset values are inserted by the PL/I system which are referred to here as the system default values for files. The file characteristics thus determined for the BS2000 file must be compatible with the set of attributes of the PL/I file.

When the file is closed, the characteristics are placed in the file catalog. The file is thus cataloged and the characteristics are associated with it.

Figure 6-8 gives an overview of the processes described here. A detailed description is contained in the following subsections.

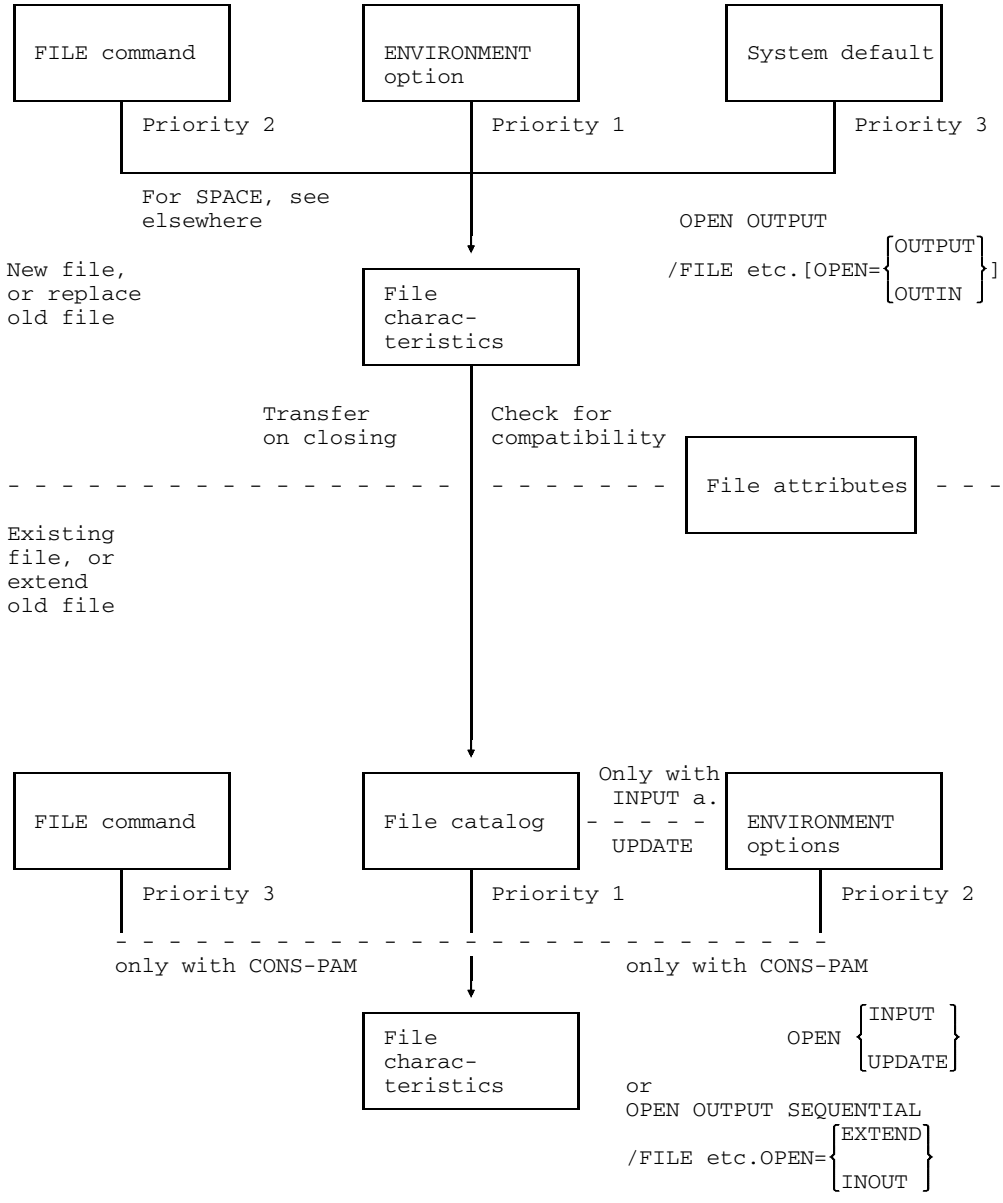


Fig. 6-8 Determining the file characteristics (order of priority indicated)

6.4.4.1 Characteristics for a new file

When a file is opened in the PL/I program with the file attribute OUTPUT, the following cases are distinguished::

- File does not yet exist.
  If no file yet exists for the corresponding file name, a new file will be created.
- File already exists.
  If a file already exists for the corresponding file name, then there are two possible courses of action:
  - If the OPEN parameter in the FILE command stipulates that the file is to be extended and if this operation is possible in accordance with the conditions described in the following section, the file is retained and processing proceeds in a similar fashion to opening with OPEN UPDATE (see following sections).
  - Otherwise, if the above does not apply, processing takes place as if the file does not exist, i.e. a new file is subsequently created.

  Exception

  If with OPEN OUTPUT to an existing file null parameters (e.g. RECSIZE) are specified, the corresponding FILE parameters from the catalog are entered.

If a new file is created, the characteristics for it are determined in the following order:

1. Firstly, the values from the FILE command are incorporated into the file characteristics, with the exception of SPACE= which at this time has already been processed and is already contained in the file characteristics.
2. If certain required information is still missing from the file characteristics, then if available it will be subsequently taken from the ENVIRONMENT specification in the PL/I program.
3. Finally, any values that are still missing are inserted as system defaults. These values are listed in Figure 6-9.

RECSIZE(0) or RECSIYE = 0 in Figure 6-9 means that the value is undefined and that an appropriate buffer (currently 2048 bytes) will be set up by the Data Management System. This will then be the maximum permitted length for a record. In the case of SAM files, 4 bytes are required for management purposes with the result that the maximum possible length of the record is reduced by these 4 bytes. In addition, the Data Management System reserves a 12-byte control field for every logical block on PAM-key-free disk files in SAM and ISAM files.

Condition	System default	
	ENVIRONMENT option	FILE command
	CONSECUTIVE	
CONSECUTIVE INDEXED REGIONAL		FCBTYPE = SAM FCBTYPE = ISAM FCBTYPE = PAM
{CONSECUTIVE INDEXED REGIONAL(3)} REGIONAL(1)	V	RECFORM = V RECFORM = F
{V U} F	RECSIZE(0) 3) RECSIZE(r) 1)	RECSIZE = 0 3) RECSIZE = r 1)
V {F U}	KEYLOC(5) KEYLOC(1)	KEYPOS = 5 KEYPOS = 1
	KEYLENGTH(8)	KEYLEN = 8
DEVICE=tape otherwise:		BLKSIZE = b 1) BLKSIZE = (STD,n) 2)
OPEN...STREAM PRINT	CTLMACH	RECFORM (x,M)

1) no default; specification mandatory

2) n: chosen so that the record (according to RECSIZE) fits exactly

3) defaulted by Data Management System (currently 2048, or SAM 2044 or 2032 for SAM) (see text)

Fig. 6-9 System default for the file characteristics when creating new files, taken from top to bottom

Once the file characteristics have been laid down in this way, a check is made as to whether they are compatible with the file attributes specified in the program. If this is not the case, an UNDEFINEDFILE condition is raised.

If a CLOSE statement is explicitly issued for a newly created file or if the file is closed implicitly at the end of the program run (which is normally the case even when the program run is errored), then the characteristics for the file are placed in the file catalog and thus retained for the file. The file is then considered to exist.

Exception

With PAM files, RECFORM and RECSIZE are not included in the catalog and must therefore be specified whenever the file is subsequently opened. This applies only to CONSECUTIVE organization; with REGIONAL files this data is recorded in a management header for the file.

6.4.4.2 Extend old output file (EXTEND)

When a file is opened by OPEN OUTPUT, a new file will normally be created. If a corresponding file is already present, processing takes place as if this file did not exist: the characteristics and records of the old file are then no longer available. The size of the current storage space allocation and the secondary allocation as per "/FILE filename, SPACE=" are retained.

However, it is possible by using the parameter OPEN in the FILE command to stipulate that the old file be retained along with its characteristics and records, and that new records be added to it. The file must have been opened with the file attribute RECORD. So the following conditions must be satisfied in order to extend an existing output file:

- the file is opened with OPEN OUTPUT [SEQUENTIAL]
- the FILE command contains the specification:
  OPEN = EXTEND for SAM or ISAM
  OPEN = INOUT for PAM

With regard to the characteristic data, the same operations are performed as are described in the following section for files which are opened with UPDATE but no check is made on compatibility with the ENVIRONMENT options.

On opening CONSECUTIVE files, positioning is effected to the end of the file so that new records can be added to the end of the file in accordance with the general rules for output statements. With INDEXED and REGIONAL files, positioning is also performed to end of file; the new records must then be supplied with keys which are greater then all existing keys.

6.4.4.3 Characteristics for existing files

If a file is opened with the file attribute INPUT or UPDATE, then the corresponding file must already exist. The same applies if a file is to be extended as described in the previous section. The characteristics for this file are then already contained in the file catalog and are taken from there.

For files cataloged as /FILE STATE = FOREIGN, the characteristics are obtained from the labels unless they are specified explicitly.

Exception

In the case of PAM files, RECFORM and RECSIZE are not recorded in the catalog and must therefore be specified in the FILE command or under ENVIRONMENT, where a specification in ENVIRONMENT has priority. This applies only to CONSECUTIVE file organization.

If the PL/I program contains ENVIRONMENT options, a check is made as to whether these are compatible with the characteristics of the file. A further check determines whether the characteristics are compatible with the file attributes specified at file opening. If either of these checks proves negative, an UNDEFINEDFILE condition is set.

Exception

If a file is opened with OUTPUT and extended as described in the previous section, no check is made on compatibility with the ENVIRONMENT options.

Opening (INPUT or UPDATE) a cataloged, empty file causes the condition ENDFILE to be raised as soon as an attempt is made to read the file.

6.5 Stream-oriented input and output

Stream-oriented input and output (STREAM) is the most convenient method for the user of transferring data between PL/I program and external data media. STREAM is also very largely independent of differing computing systems and the data sets can be transferred almost at will between different computers. On input, the external data is regarded as a continuous stream and processed consecutively character by character. The characters are edited in accordance with the user's wishes by the input system and converted to the internal form of value representation. On output, the internal form of representation is converted into character form in accordance with the user's specifications and output as a continuous stream of characters. In doing this the user can select various levels of control, either simple LIST or DATA controlled editing where conversion to the external form of representation is determined by PL/I, or EDIT controlled editing where the user himself can determine the external representation, even at character level.

Where output is to be printed, the data stream can be subdivided into lines and pages; section 6.5.2 discusses this in more detail.

6.5.1 Rules governing stream-oriented input and output

Files to be involved in stream-oriented input/output must be organized on a CONSECUTIVE basis. Other forms or organization must not be used.

The BS2000 access methods SAM and ISAM and the system files can be used for stream-oriented input/output. When using ISAM, the same special rules concerning key location and key length apply as for record-oriented processing with CONSECUTIVE (key location only 1 or 5, key length only 4 or 8). Permitted record formats are F, V and U. With the U format precisely one block is transferred per record, where the maximum possible record length is given by BLKSIZE. Under certain circumstances use of the U format can lead to high wastage.

As far as the FILE command is concerned, the same specifications apply as are given in section 6.6.1.7; however, FCBTYPE = PAM is not permitted.

For file opening, writing, reading and closing operations the same rules apply as for record-oriented processing. These rules are given under section 6.6.1.

The following table shows the file attributes which are permitted in PL/I for STREAM files and the statements which may be used:

Attributes			Statement	Organization
STREAM	INPUT 1)		GET [SKIP]	CONSECUTIVE
	OUTPUT	LINESIZE	PUT [SKIP]	
	OUTPUT PRINT	PAGESIZE LINESIZE	[PAGE,LINE] PUT[ ] [SKIP]	

1) not applicable to empty file

The length of any particular record is determined primarily by the use of SKIP in the GET or PUT statement, or in the format specification in the case of EDIT control. However, if SKIP is not used before the maximum record length is reached, the transition to the new record takes place automatically in accordance with the stream definition. The maximum record length is determined by:

1. LINESIZE attribute with OPEN (only for OUTPUT)
2. RECSIZE/BLKSIZE in the FILE command or ENVIRONMENT attribute
3. SYSFILE control statement (for system files)
4. Default (see Figure 6-10).

For system files, the above options are examined in the order shown, until a value is found.

All other files are subject to the following additional rules:

- Specifications of RECSIZE (BLKSIZE) and device characteristics always have priority if these specifications are less than LINESIZE with OPEN in the SYSFILE control statement.
- If RECSIZE > LINESIZE in the case of files in F format, then during writing the character stream is filled with blanks up to the specified record length.

File	Interactive task	Batch task
SYSLST	120	120
SYSOUT	Line length of device[1)]	132
SYSDTA	120	120
File with RECFORM=F or U	RECSIZE[2)] or 120	
File with RECFORM=V	RECSIZE[2)] or 120	

1) less 1 character for carriage return, where applicable
2) with PRINT files, a further character is deducted for carriage control

Fig. 6-10 Default values for LINESIZE with stream-oriented input/output

The limit values for system files also apply to SYSLST and SYSDTA if these files were assigned to a user file by means of a SYSFILE command. With these files the user has the option of changing the record length during the program run by closing the file (CLOSE), reopening it and giving a new LINESIZE specification in the OPEN statement.

In the case of EDIT controlled transfer, automatic transition to the next record takes place precisely at the relevant limit selected. On output, therefore, values are broken up where necessary, while on input, successive lines are chained.

With DATA or LIST controlled output, the individual output values are not broken up at the end of the record but are written to the next record if there is no longer sufficient space for them to be contained in their entirety in the current record; i.e. if LINESIZE were to be exceeded. Output values are only broken up if the element itself is greater than LINESIZE.

6.5.2 PRINT files

Files defined implicitly or explicitly with the attributes STREAM OUTPUT PRINT are known as print (PRINT) files. They can be referenced only by the PUT statement. The PUT statement causes a character string to be generated into which are inserted the carriage control characters for page and line feed operations. These control characters subdivide the character string into lines which are each assigned to one record in the external file. This control character is automatically prefixed to every record in a PRINT file. The coding of this character can be influenced by the ENVIRONMENT options CTLMACH and CTLASA (see section 6.3.4).

On output to the system files SYSLST and SYSOUT, carriage control characters in accordance with CTLMACH are always generated. Any specification other than this will be ignored.

On output to a file created with /FILE...RECFORM=(x,N), carriage control characters are generated as with RECFORM=(x,M).

If the PRINT file is not output on a printer, the control character has no effect and it is treated as part of the record.

A new record is always created whenever a SKIP or PAGE specification is processed in the output statement or the data stream reaches the record length defined explicitly or implicitly by LINESIZE (see section 6.5.1). With DATA and LIST controlled output, the new record is commenced as soon as the current output value in the current record can no longer be taken up in its entirety.

For the default record lengths, refer to the values defined in section 6.5.1. Note, however, that space for the control character must be taken into consideration in each case.

The page size for a PRINT file may be declared as follows:

1. PAGESIZE option in the OPEN statement.
2. *RUNOPT SYSFILE = PAGESIZE(x,y,z) control statement.
3. Defaults:
   for SYSOUT in interactive mode (display terminal): PAGESIZE(1)
   otherwise: PAGESIZE(60)

These options are examined in the order shown above until a value is found. When the end of a page is reached, the ENDPAGE condition is set.

If a PRINT file was stored on an external volume, i.e. not printed automatically via SYSLST, the file can be output to printer at any time using the PRINT (SPACE=E) command.

	Printer (batch mode)	Interactive device	
		Line mode 1)	Page mode
PAGESIZE(n)		n=1 1)	n>1
LINESIZE(n)			n≤ line length of device for device with carriage return n≤ line length of device -1
SKIP option	Output of line	Output of line	Line to page buffer
LINE(n)		As SKIP(1)	
PAGE		As SKIP(1)	Output page buffer
ENDPAGE		is never set	
LINENO		increases without limit	
PAGENO		always results in value 1	

1) Default: Line mode

Fig. 6-11 Special considerations for files with STREAM PRINT for the system file SYSOUT

6.5.3 Stream-oriented input/output to interactive device

To facilitate the use of STREAM input/output for interactive devices (system file, SYSOUT, SYSDTA), certain deviations from the PL/I rules are allowed in this case:

- If the last character output to the interactive terminal is a colon (:), then the output is additionally followed with a SKIP (as if it were followed by a statement PUT FILE(a) SKIP;).

  This is true of all output directed to system file SYSOUT, irrespective of the way this file is referenced in PL/I:

  – PUT FILE (SYSOUT)...

  – OPEN FILE(a) TITLE ('SYSOUT')
    PUT FILE(a)...

  – *RUNOPT SYSFILE = TITLE ('b')
    OPEN FILE(a) TITLE('b');
    PUT FILE(a)...

- Strings (like PRINT files) are output without the enclosing quotes.

6.5.3.1 Input from interactive device (SYSDTA)

For STREAM input from SYSDTA, the following deviations apply, unlike input from files:

- For LIST and DATA input, the end-of-line acts as separator for the items involved. If any of the separators blank, comma, or semicolon precedes the end-of-line, the whole is treated as one separator. Thus, the following applies for the limit values:

  – empty input: ignored
  – line consisting of all blanks: ignored
  – blank..., blank..., end-of-line together are one separator
  – a line consisting of one separator only: empty element

- For EDIT input, the end-of-line is the end of the string being entered. The input string, if too short, is padded with blanks to the required length.

- If the last character of the input string is a hyphen (-), then the separator function of the end-of-line is canceled for the particular line: The input string is continued on the next line. The hyphen is removed from the input string.

6.5.3.2 Output to interactive device (PRINT file to SYSOUT)

When a file with the attributes STREAM PRINT is output to the system file SYSOUT, certain special considerations apply to interactive operation which do not apply to batch mode.

With regard to output on an interactive device, a distinction must be made between line mode and page mode.

In line mode the text for output is output a line at a time to the interactive device. Line mode is selected by default. It can be explicitly selected by setting the page size to the value 1 (PAGESIZE(1)).

The following differences then apply:

- The LINE functions acts like SKIP(1).
- The PAGE function acts like SKIP(1).
- The ENDPAGE condition is never set.
- SKIP(n) where n > 3 is reduced to SKIP(3).
- The line number which can be maintained by the built-in function LINENO increases without limit.
- The page number which can be maintained by the built-in function PAGENO always has the value 1.

Page mode is selected by setting the page size to a value greater than 1. In this mode, the lines of a page are temporarily stored in a page buffer. An implicit or explicit page feed (PAGE) causes the contents of the page buffer to be output on the interactive device. Where a VDU is in use, the entire screen is first cleared.

The following differences apply:

- The value for the line length (LINESIZE) must not be greater than the line length on the output device; otherwise an UNDEFINEDFILE condition will be raised.

  For output devices having a carriage return, the line length (LINESIZE) must be 1 less than the line length on the output device.
- When a new line is commenced, the old line is not output to the output device but is stored in the page buffer.
- SKIP(n) where n > 3 is reduced to SKIP(3).

- The PAGE function causes the contents of the page buffer to be output to the output device.
- The value for the page size (PAGESIZE) for VDUs must not be greater than the number of lines less 1 permitted for the screen; otherwise an UNDEFINEDFILE condition will be raised.
- If the selected page size is exceeded, the PAGE function will be implicitly executed.

6.6 Record-oriented input and output

Record-oriented input and output is used for those files which have the RECORD attribute in the PL/I program. Either the programmer can assign this attribute directly to the file or it is assumed implicitly on the basis of other file attributes (e.g. UPDATE, DIRECT etc.). On this, see also chapter 8 of the PL/I language reference manual [1]. With record-oriented input/output, data is transferred in the internal form of representation. This therefore restricts the degree of interchangeability between different computer systems of files created in this mode of operation.

In record-oriented transmission, the unit of transfer from the viewpoint of the PL/I program is a record in the external file; i.e. precisely one record is transferred in each READ, REWRITE, LOCATE or WRITE operation. All PL/I types of organization are permitted for record-oriented input/ output.

This section is therefore subdivided according to the organization methods

```
CONSECUTIVE
INDEXED
REGIONAL(1) and
REGIONAL(3)
```

Figure 6-12 gives an overview of the permitted input/output statements for RECORD files in relation to the file attributes and type of organization. Certain exceptions to these combinations are discussed in the following sections.

Attributes			Statement	Organization		
RECORD	SEQUENTIAL	OUTPUT	WRITE FROM LOCATE [SET]	CONS		
		INPUT	READ {INTO / SET / IGNORE}		IND	REG
		UPDATE	READ {INTO / SET / IGNORE} REWRITE [FROM]			
			DELETE			
	SEQUENTIAL KEYED	OUTPUT	WRITE FROM KEYFROM LOCATE [SET] KEYFROM		IND	REG
		INPUT	READ {INTO / SET} [KEY]			REG (1)
			READ {INTO / SET} [KEYTO] READ IGNORE			REG
		UPDATE	READ {INTO / SET} [KEY]			REG (1)
			READ {INTO / SET} [KEYTO] READ IGNORE			REG
			WRITE FROM KEYFROM			
			REWRITE [FROM]			REG
			DELETE [KEY]			1)
	DIRECT KEYED	OUTPUT	WRITE FROM KEYFROM			REG
		INPUT	READ INTO KEY		IND	
		UPDATE	READ INTO KEY WRITE FROM KEYFROM REWRITE FROM KEY DELETE KEY			

1) see text for exceptions

not possible

FCBTYPE	SAM*	ISAM	PAM
RECFORM	FVU	FV	FV

Fig. 6-12 Permitted I/O statements with various file attributes and types of organization

A matter requiring special consideration in record-oriented data transfer is the SCALARVARYING specification in the ENVIRONMENT attribute. Writing and reading scalar strings with the attribute VARYING in the LOCATE mode (LOCATE, READ SET) are only practical with SCALARVARYING. See section 6.3.5 for details.

With record-oriented output it is also possible to create data sets which are to be subsequently printed out using the PRINT command. If carriage control is desired (SPACE=E in the PRINT command), the following points must be noted:

- Organization must be CONSECUTIVE.
- The carriage control character must be generated by the program and output as the first character of the record.
- The codings for carriage control characters are contained in section 6.3.4. The type of carriage control (esp. ASA) should be indicated using the FILE command or the ENVIRONMENT attribute.

6.6.1 Rules governing CONSECUTIVE organization

CONSECUTIVE files are organized sequentially in the order of access or of their keys. A CONSECUTIVE file may only be processed with the file attribute SEQUENTIAL. CONSECUTIVE can be applied to the access methods SAM, PAM, and ISAM; keys are generated automatically for ISAM. Depending on the access method, CONSECUTIVE files may contain records of fixed, variable or undefined length. The possible input/output statements are summarized in Figure 6-13.

Organization	Attributes			Statement
CONSECUTIVE SAM (ISAM) (PAM) F V U	RECORD	SEQUENTIAL	OUTPUT	WRITE FROM LOCATE
			INPUT	READ{INTO | SET | IGNORE}
			UPDATE	READ{INTO | SET | IGNORE} REWRITE [FROM]

Fig. 6-13 I/O statements for CONSECUTIVE organization

6.6.1.1 Opening CONSECUTIVE files

When PAM, SAM or ISAM files are opened with the file attribute OUTPUT, positioning is effected to the beginning of the file. Records contained in the file are lost. However, if

```
/FILE...OPEN=INOUT or
/FILE...OPEN=EXTEND
```

is specified in the FILE command, then positioning is effected to the end of the file and the new records are added to the file.

Opening with the file attribute UPDATE is only permitted for files on direct access volumes. The use of OPEN UPDATE assumes that the file has already been created. After OPEN the current record position is at the beginning of the file.

If the file is an ISAM file, the KEYLEN/KEYLENGTH must be 4 or 8, and KEYPOS/KEYLOC must be 1 or 5. Otherwise an UNDEFINEDFILE condition will occur.

6.6.1.2 Closing CONSECUTIVE files

Closing a file is done either explicitly by specifying the CLOSE statement or implicitly at program termination. Should an output buffer still be present, this will be output to the file. The assignment of a PL/I file to the BS2000 file is canceled. The buffers allocated to the file in the PL/I program are released.

6.6.1.3 Writing to a CONSECUTIVE file

The file must have been opened with the OPEN attribute OUTPUT. The WRITE statement causes one record to be output. The LOCATE statement sets a pointer (locator) to the next free location in the I/O buffer. If RECSIZE and the length of the variable specified in the WRITE or LOCATE statement are incompatible, a RECORD condition is raised.

If the file is an ISAM file, the key is generated automatically by the input/output system. In this case only 4 or 8 may be specified for KEYLENGTH/ KEYLEN, and KEYPOS must be 1 or 5. If KEYLEN=4, then binary keys (FIXED BIN(31,0) with an increment of 1 are generated (FORTRAN form); if KEYLEN=8 is specified, the key will be generated as an 8-digit decimal number (PICTURE ’(8)9’) with an increment of 1 (compatible with EDT and EDOR).

6.6.1.4 Reading from a CONSECUTIVE file

The OPEN attribute must be INPUT or UPDATE. The statement READ INTO causes a record to be read from the input buffer. READ SET sets the pointer to the record in the input buffer. The statement READ IGNORE(n) causes the number of records specified by n to be read but not transferred. If there is incompatibility between the current record length for the file and the variable specified in the READ statement, a RECORD condition is raised; a transfer error will result in a TRANSMIT condition; and an ENDFILE condition is signaled upon reaching the end of the file. The key of an ISAM file is not supplied to the program.

6.6.1.5 Overwriting records in a CONSECUTIVE file

The OPEN attribute must be UPDATE. The record successfully read immediately before is overwritten; that is, no READ error message must have occurred nor a preceding READ IGNORE (n). RECORD condition is reported if the record and variable length are incompatible or for the SAM and PAM access methods with variable-length records, if an attempt is made to rewrite shortened or extended records, which is allowed for ISAM. Transfer errors lead to the TRANSMIT condition.

6.6.1.6 Deleting records in a CONSECUTIVE file

It is not possible to delete the records of a CONSECUTIVE file.

6.6.1.7 FILE command for CONSECUTIVE files

If neither the file is already cataloged nor appropriate information supplied in the ENVIRONMENT attribute, the FILE command should provide the following parameters:

```
/FILE     filename,    (Name of the data set)
 LINK   = PL/I-title,
          ⎧SAM ⎫
 FCBTYPE=⎨ISAM⎬,
          ⎩PAM ⎭

 RECSIZE= r,           (Number of characters incl. management information)

          ⎧F|V|U                ⎫
 RECFORM=⎨                      ⎬ ,
          ⎩({F|V|U}[,{A|M}])    ⎭

          ⎧STD     ⎫     Buffer size = 1 PAM block
 BLKSIZE=⎨(STD,n)  ⎬,    Buffer size in n PAM blocks up to 16
          ⎩m       ⎭     Buffer size in number of characters
                         (required for tape files)

          ⎧1⎫
 KEYPOS =⎨ ⎬,           for RECFORM = F       ⎫
          ⎩5⎭           for RECFORM = V       ⎪
                                              ⎪
          ⎧4⎫                                 ⎬      (for ISAM only)
 KEYLEN =⎨ ⎬,                                 ⎪
          ⎩8⎭                                 ⎭

          ⎧p       ⎫    p:  primary allocation/extension/reduction
 SPACE  =⎨        ⎬
          ⎩(p[,s]) ⎭    s:  increment

          ⎧EXTEND⎫
 OPEN   =⎨      ⎬,
          ⎩INOUT ⎭
```

Rules governing parameters:

- For RECSIZE, the physical record length must be entered in number of characters; i.e. record length field, carriage control character, record key etc. must be included in the record length as required.
- For RECFORM, specifying U is meaningful only in connection with tape files. Specification of A or M is only required if the file is to be subsequently printed out using the PRINT command. See here also section 6.5.2.
- For BLKSIZE, with the exception of SAM tape files the buffer length must be specified in PAM blocks (up to 16 PAM blocks). The maximum permissible size that may be specified for tape files is 32762 characters.

- The values for the SPACE parameter in the FILE command must be specified as a number of PAM blocks in accordance with the quantity of data to be entered; in particular, they must be compatible with the value of BLKSIZE. In the case of SAM, the primary and secondary allocation specifications should in addition be divisible both by 3 and by the number of PAM blocks per logical block. In the case of ISAM, space should additionally be provided for the index area. Thus, at least 1 PAM block more should be specified than given for BLKSIZE. If an invalid entry is supplied, its value is set to the next higher allowable value when the file is opened.

Example

```
/FILE RESULT,LINK=OUTPUT,FCBTYPE=SAM,RECFORM=V,RECSIZE=84,      -
             BLKSIZE=STD,SPACE=(3,3)
```

Taking the system defaults into account, this is equivalent to:

```
/FILE RESULT,LINK=OUTPUT,RECSIZE=84
```

Further examples of the FILE command are given in section 6.2.2.5.

6.6.2 Rules governing INDEXED organization

In the case of a file having INDEXED organization, the records are provided with a key. In the data set this key is always part of the associated record. As far as the PL/I program is concerned, the key may also precede the record. It is possible to access the records directly or sequentially according to the sequence of the keys. The file must reside on a direct access volume.

A file having INDEXED organization can be processed in the following modes:

```
          SEQUENTIAL
or        SEQUENTIAL KEYED
or        DIRECT KEYED
```

INDEXED organization is compatible only with the ISAM access method. The files may contain records of variable or fixed length.

The possible input/output statements are summarized in Figure 6-14.

Organization	Attributes			Statement
INDEXED ISAM F V	RECORD	SEQUENTIAL	INPUT	READ {INTO | SET | IGNORE}
			UPDATE	READ {INTO | SET | IGNORE} REWRITE [FROM] DELETE
		SEQUENTIAL KEYED	OUTPUT	WRITE FROM KEYFROM LOCATE KEYFROM
			INPUT	READ {INTO | SET} {[KEY] | [ ] | [KEYTO]} READ IGNORE
			UPDATE	READ {INTO | SET} {[KEY] | [ ] | [KEYTO]} READ IGNORE WRITE FROM KEYFROM REWRITE [FROM] DELETE [KEY]
		DIRECT KEYED	INPUT	READ INTO KEY
			UPDATE	READ INTO KEY WRITE FROM KEYFROM REWRITE FROM KEY DELETE KEY

Fig. 6-14 I/O statement for INDEXED organization

6.6.2.1 Key specification

The key specified in the program under KEY or KEYFROM is converted to the data type CHARACTER where necessary. The character string forming the key may have a maximum length of 255.

During a write operation to a file, the key supplied by the program is entered in the area of the record as defined by KEYPOS and KEYLEN or KEYLOC and KEYLENGTH. When this is done, the key overwrites the corresponding part of the information supplied by the record variable. Note, however, the special effect of ENV (KEYLOC(0)) as described in section 6.3.3.

If the key supplied by the program is shorter than the specification in KEYLEN or KEYLENGTH, it is padded out on the right with blanks. If the supplied key is longer, it is truncated on the right.

During a read operation, the key contained in the record is placed in the variable named under KEYTO. If this variable has the attribute CHARACTER and the length of this string is not consistent with KEYLEN or KEYLENGTH, it will be truncated or padded out with blanks on the right. If the variable is not a CHARACTER string, any of the conditions may occur that also occur when assignments are made.

For files opened with

```
                          [INPUT ]
RECORD SEQUENTIAL KEYED  {       }
                          [UPDATE]
ENVIRONMENT (INDEXED GENKEY)
```

specifying GENKEY indicates that a key specified by READ KEY (key) may also be shorter than is declared for the file. Then the first record whose key begins with the specified character string is read; if no such record is present, the condition KEY is set.

If there is more than one key beginning with the specified character string, it is possible by using READ KEY (key) to read only the first record (the one with the lowest key value); the others can be accessed by sequential reading (READ or READ KEYTO). If the key is not contained within the record (KEY-LOC(0)), then the key of the first record cannot be accessed since KEY and KEYTO are not permitted together in a single statement.

6.6.2.2 Opening an INDEXED file

Files with INDEXED organization may be opened as SEQUENTIAL, SEQUENTIAL KEYED or DIRECT (KEYED). The attributes DIRECT or SEQL (without KEYED) may only be used with the OPEN attributes INPUT or UPDATE. DIRECT or SEQUENTIAL without KEYED are not therefore permitted in conjunction with the attribute OUTPUT. The initial creation of files which are to be processed with the attribute DIRECT must be performed with the attribute SEQUENTIAL KEYED.

When a file is opened with the file attribute INPUT or UPDATE, then in the case of DIRECT KEYED the current record position is undefined after opening while with SEQUENTIAL or SEQUENTIAL KEYED it is at the beginning of the file. The data set concerned must contain at least one record.

The access to files declared "shared update" is supported for files that are opened with the UPDATE attribute. The shareability of files is indicated by the option SHARUPD=YES in the FILE command. See BS2000 DMS Reference Manual [7] for the shareability of files.

6.6.2.3 Closing an INDEXED file

Closing a file is done either explicitly by specifying the CLOSE statement or implicitly at program termination.

Any remaining buffers of files opened with OUTPUT or UPDATE are output. The buffers associated with the file in the PL/I program are released and the assignment to the BS2000 file is canceled.

6.6.2.4 Writing to an INDEXED file

For writing to a file which has INDEXED organization, OUTPUT or UPDATE can be specified as the OPEN attribute. WRITE causes the record to be written; the LOCATE statement provides the pointer to the reserved location in the output buffer. WRITE requires that the file was opened with OUTPUT SEQUENTIAL KEYED or UPDATE DIRECT KEYED. The LOCATE statement requires an OPEN with OUTPUT SEQUENTIAL KEYED.

A KEY condition is raised when a record with the same key as the one supplied already exists. The KEY condition is also reported if there is no further space in the file for the transferred record. Similarly the KEY condition is reported in the case of OUTPUT if the keys are not in ascending order. There are different ONCODE values for the individual cases; these are given in chapter 14.

RECORD condition is raised if the record and variable lengths are incompatible. A TRANSMIT condition is reported in the event of transmission errors.

If a file is opened with OPEN = EXTEND (see 6.4.4.1), the new records must be supplied with keys greater than all existing ones.

6.6.2.5 Reading from an INDEXED file

For reading from a file which has INDEXED organization, INPUT or UPDATE can be specified as the OPEN attribute.

Incompatibility between record length and variable length will result in a RECORD condition.

If no record having the specified key is found, this will result in a KEY condition.

A TRANSMIT condition is raised in the event of transmission errors.

When reading is performed from files declared shared update, the block of files opened with UPDATE and containing the record to be read, is locked to other users. This lokkout condition remains unaffected until a READ, REWRITE or DELETE statement is given.

6.6.2.6 Overwriting records in an INDEXED file

When using the REWRITE statement to overwrite records in an INDEXED file, the OPEN attribute must be UPDATE. The record whose key is specified in the REWRITE statement will be overwritten (in the case of DIRECT); or the record that has just been read (in the case SEQUENTIAL). A KEY condition is raised if no record having the specified key exists.

When records are rewritten in shared update files, the block in which the record is to be rewritten is released after writing in order to restore shareability.

6.6.2.7 Deleting records in an INDEXED file

When using the DELETE statement to delete records in an INDEXED file, the OPEN attribute must be UPDATE. The record whose key was given in the KEY specification will be deleted, or the record that has just been read (in the case of SEQUENTIAL UPDATE). If no record having this key exists in the file, a KEY condition will be raised.

When records are deleted in shared update files, the block containing the record to be deleted is released after deletion to restore shareability.

6.6.2.8 FILE command for INDEXED files

If neither the file is already cataloged nor appropriate information supplied in the ENVIRONMENT attribute, the FILE command should provide the following parameters:

```
/FILE      filename,          (Name of the data set)
 LINK    = PL/I-title,
 FCBTYPE = ISAM,
 RECSIZE = r                 (Number of characters)

           ⎧F⎫
 RECFORM = ⎨ ⎬
           ⎩V⎭

           ⎧STD     ⎫        (Buffer size = 1 PAM block)
 BLKSIZE = ⎨        ⎬,
           ⎩(STD,n) ⎭        (Buffer size in n PAM blocks up to 16)

 KEYPOS  = 1≤ c ≤ (r-k),     (Position of start of key)

 KEYLEN  = k,                (Number of characters)

           ⎧p       ⎫       p:  primary allocation
 SPACE   = ⎨        ⎬
           ⎩(p[,s]) ⎭       s:  secondary allocation/extension

 OPEN    = EXTEND
```

Rules governing parameters:

- For RECSIZE, the record length must be entered as a number of characters. With format V files, the record length field (4 characters) must be taken into consideration; in the case of ENV (KEYLOC(0)), the key length must be added to the record length.
- For KEYPOS, the position of the leftmost character of the key is specified in the record. The minimum possible value is 1 when RECFORM=F, or 5 when RECFORM=V.
- The values for the SPACE parameter in the FILE command must be specified in accordance with the data sets to be entered; in particular, they must be compatible with the BLKSIZE value. To accommodate the index area, at least 1 PAM block more must be specified than the value for BLKSIZE.

An UNDEFINEDFILE condition may be raised if specifications are omitted.

Example

```
/FILE SET,LINK=TYP,FCBTYPE=ISAM,RECSIZE=90,     -
/          RECFORM=V,BLKSIZE=STD,KEYPOS=5,      -
/          KEYLEN=6,SPACE=(18,6)
```

Further examples are contained in sections 2.2.6 and 6.2.2.5.

6.6.3 Rules governing REGIONAL(1) organization

A REGIONAL(1) file is divided into regions, each of which is assigned a numerical key. REGIONAL(1) files are implemented using the PAM access method. A REGIONAL(1) file can contain only fixed-length records. Each region of the file contains only one record. Each region number therefore corresponds to a relative record position in the file. The record key (KEY) is identical to the region number and is not recorded in the file. The records can be accessed sequentially or directly.

The records are arranged in close succession, like a continuous stream; consequently one PAM block can contain one or more records or it may contain part of one record only. From the region number (KEY) and the record length, the input/output system determines the block number and the relative position of a record within this block.

The I/O statements permitted for REGIONAL(1) are summarized in Figure 6-15.

Organization	Attributes			Statement
REGIONAL PAM F V 2)	RECORD	SEQUENTIAL	INPUT	READ {INTO / SET / IGNORE}
			UPDATE	READ {INTO / SET / IGNORE} REWRITE [FROM] DELETE
		SEQUENTIAL KEYED	OUTPUT	WRITE FROM KEYFROM LOCATE KEYFROM
			INPUT	READ {INTO / SET}[KEYTO]1) READ IGNORE
			UPDATE	READ {INTO / SET}[KEYTO]1) READ IGNORE REWRITE [FROM] DELETE
		DIRECT KEYED	OUTPUT	WRITE FROM KEYFROM
			INPUT	READ INTO KEY
			UPDATE	READ INTO KEY WRITE FROM KEYFROM REWRITE FROM KEY DELETE KEY

1) for REGIONAL(1) alternatively with KEY
2) for REGIONAL(3)

Fig. 6-15 I/O statements for REGIONAL(1) and REGIONAL(3) organization

6.6.3.1 Key specification

The key provided by the program through KEY/KEYFROM must be a character string consisting solely of numerics (0 thru 9) and blanks in place of leading zeros. The length should not exceed 8 characters. Leading blanks are interpreted as zeros. If more than 8 numerics are specified, only the rightmost 8 numerics are used; if less than 8 numerics are specified, padding blanks are added to the left.
Interpretation of the key starts from the left within the 8 characters. Blanks embedded between numerics delimit the key; i.e. a blank to the right of a numeric along with any other characters to the right of the blank will be ignored. If only blanks are found, the key will receive the value zero. A KEY condition will result if the above rules are not observed.

The key thus obtained from the KEY or KEYFROM specification serves as the region number and is not recorded in the file. The lowest key possible is 0.

6.6.3.2 Dummy records

A REGIONAL(1) file contains valid records and/or dummy records. When a REGIONAL(1) file is created, all the regions are preset to contain dummy records if file opening is effected with DIRECT OUTPUT. If opening is performed with SEQUENTIAL OUTPUT; the entry of dummy records is carried out in conjunction with write operations.

A dummy record is identified by ’FF’B4 in the first character, the rest of the record is undefined. The user can access dummy records at any time and must be able to recognize these himself. The input/output system does not check for dummy records, i.e. no KEY condition will result when dummy records are accessed.

6.6.3.3 Opening a REGIONAL(1) file

A REGIONAL(1) file can be opened in any of the following ways:

- Initial open (OUTPUT)
- Extend (OUTPUT, UPDATE)
- Edit (INPUT, UPDATE)

A REGIONAL(1) file can be opened initially with either DIRECT OUTPUT or SEQUENTIAL OUTPUT.

For the initial open in connection with DIRECT OUTPUT, on OPEN the entire primary storage space is preformatted with dummy records by the input/output system in accordance with the SPACE specification in the FILE command. Secondary SPACE specifications are not taken into consideration.

If a file is initially opened for SEQUENTIAL OUTPUT, the records must be supplied in ascending order of their region numbers. Any region that is skipped is assigned a dummy record by the input/output system.

An existing file can only be extended if OPEN = EXTEND (or OPEN = INOUT) and SPACE = primary are specified in the FILE command. Otherwise, an existing file is deleted and a new file is "initially opened". Extending means to once add "primary" PAM pages, where "primary" is rounded as necessary to a multiple of 3 and at a minimum, to the value of BLKSIZE.

The following options are supported for Extend:

- SEQUENTIAL KEYED OUTPUT
  The keys of the newly supplied records must be greater than all existing ones.
- SEQUENTIAL KEYED UPDATE
- SEQUENTIAL DIRECT OUTPUT or UPDATE

A REGIONAL(1) file, once created, can be opened with the attributes DIRECT INPUT or DIRECT UPDATE, or SEQUENTIAL INPUT or SEQUENTIAL UPDATE.

6.6.3.4 Closing a REGIONAL(1) file

Closing a file is done either explicitly by specifying the CLOSE statement or implicitly at program termination.

Apart from the usual actions normally performed during the CLOSE process, a REGIONAL(1) file opened with OUTPUT SEQUENTIAL causes the input/output system to pad out the storage area with dummy records from immediately following the current position to the end of the file.

6.6.3.5 Writing to a REGIONAL(1) file

The WRITE statement causes the specified variable to be written to the output buffer as a record; the LOCATE statement provides a pointer to the reserved location in the output buffer. No check is made as to whether the record is already present, and it is not possible for a KEY condition to occur as a result of a key already existing. If in SEQUENTIAL processing the record does not immediately follow its predecessor, all the skipped regions are supplied with dummy records.

A KEY condition is raised if the value of the key is greater than the number of the last region permitted on the basis of the file characteristics or if the key is syntactically errored.

6.6.3.6 Reading from a REGIONAL(1) file

For reading from a file which has REGIONAL(1) organization, INPUT or UPDATE must be specified as the OPEN attribute.

The READ statement causes a record to be read into the variable specified in the statement. READ SET sets the pointer to the beginning of the current record in the input buffer. Note that the input/output system also delivers dummy records. The user must detect these himself; no KEY condition is set if dummy records are encountered. KEY condition is reported if the value of the key is greater than the number of the last region generated at file creation time or if the key is syntactically wrong.

6.6.3.7 Overwriting records in a REGIONAL(1) file

If a record is to be overwritten using the REWRITE statement, then as above there is no prior check to ascertain whether or not the record is a dummy record. If the file was opened with the attribute SEQUENTIAL, the REWRITE statement must have been preceded by a successful READ. Thus, no KEY condition can result unless the key addresses a region which is not contained within the file or is syntactically errored.

6.6.3.8 Deleted records in a REGIONAL(1) file

When a record in a REGIONAL(1) file is deleted using the DELETE statement, the record concerned is designated a dummy record. The first character of a dummy record contains (8)'1'B. As during writing, no check is made as to whether the record exists, and no KEY condition is raised unless the key addresses a region which is not contained within the file or is syntactically errored. DELETE can only be used for a file that was opened with UPDATE. If a file was opened with the attributes SEQUENTIAL UPDATE, a READ statement must be given beforehand.

6.6.3.9 FILE command for REGIONAL(1) files

If the file is not already cataloged and/or if the ENVIRONMENT attribute does not contain corresponding options, the FILE command must contain a minimum of the following parameters:

```
/FILE          filename,    (Name of the data set)
 LINK      =   PL/I-title
[FCBTYPE   =   PAM,]        (Default)
 RECSIZE   =   r,           (Number of characters)
 RECFORM   =   F,
 BLKSIZE   =   (STD,n),
 SPACE     =   p,           (Primary allocation for OUTPUT/
                             secondary allocation for EXTEND or INOUT)
```

Rules for the FILE command:

- For RECSIZE, only the record length is entered, as a number of characters. The key does not affect the length of a record and does not appear in the FILE command.
- SPACE must allow 1 PAM page for management information.
- RECSIZE simultaneously defines the buffer size. No BLKSIZE specification is required.
- SPACE defines the storage space requirement as a number of PAM blocks, and must be consistent with RECSIZE. If a secondary allocation is specified, this will be ignored.

Example

File for 2000 regions

```
/FILE TELEPHONE,LINK=NO,FCBTYPE=PAM,RECSIZE=200,       -
/               RECFORM=F,SPACE=198
```

6.6.4 Rules governing REGIONAL(3) organization

A REGIONAL(3) file must reside on a direct access volume. It allows fixed or variable-length records which are prefixed by a key. Processing is possible in either sequential or direct modes. The PAM access method is used.

A REGIONAL(3) file is divided into regions of n PAM pages each where $1 \leq n \leq 16$. The region size is determined by the BLKSIZE parameter in the FILE command. Each region has a region number, the number of the first region being 0, the second 1 etc. The first PAM page of the file is used to store administration information, with the result that the region with the region number r begins at PAM page r . n + 2.

Each region contains one or more records, depending on the current length of the records. A key is recorded before each record and the length of this key is not taken into consideration in RECSIZE. The key preceding each record is known as the region-specific record key. The I/O statements permitted for REGIONAL(3) files are summarized in Figure 6-15.

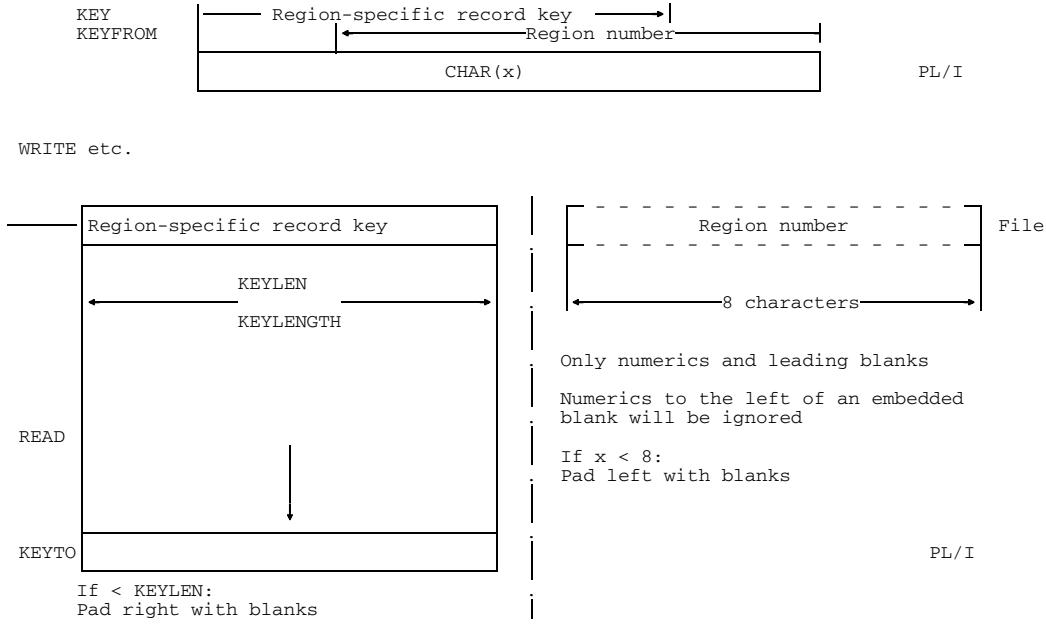


Fig. 6-16 Organization of the key on access and transfer on reading for REGIONAL(3) files

6.6.4.1 Key specification

The key specified in the source program is referred to as the source key. It is regarded as a combination of the region number, and a key valid within the region and recorded there. The first 8 characters from the right in the source key are used for the region number; these characters (up to 8) may comprise only numerics (0 thru 9) and leading blanks. Leading blanks are interpreted as zeros. As with REGIONAL(1), the first embedded blank terminates the region number. If there are no numeric characters or if there are invalid characters in the source key, a KEY condition is raised. The number obtained is used for addressing the referenced region (fetching or writing back the n PAM blocks concerned).

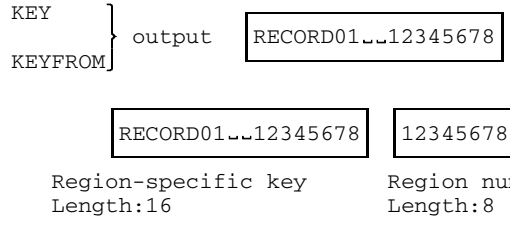
As many characters are taken for the region-specific record key from the left of the source key as are specified in KEYLEN in the FILE command or in KEYLENGTH in the ENVIRONMENT attribute. If the source key is shorter than the KEYLEN specification, the region-specific record key will be padded with blanks on the right. In this case in particular it will also contain all the numerics of the region number. The region-specific record key is prefixed to the relevant record as a character string. See also Figure 6-16.

Example 1

```
PL/I specificat.:  KEY = ('RECORD01␣␣12345678')
FILE command:      KEYLEN = 6
Interpretations:   Source key                    ≙RECORD01␣␣12345678
                   Region-specific record key    ≙RECORD01
                   Region number                 ≙12345678

  KEY
        } output   RECORD01␣␣12345678
  KEYFROM

                   RECORD01    12345678

     Region-specific key      Region number
     Length:6                 Length:8
```

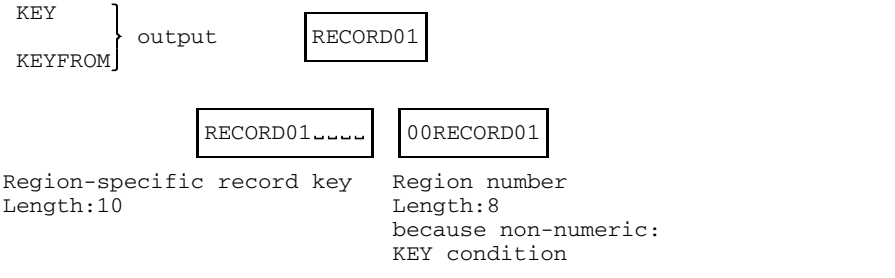
Example 2

```
PL/I specificat.:  KEY = ('RECORD01␣␣12345678')
FILE command:      KEYLEN = 16
Interpretations:   Source key                        ≙RECORD01␣␣12345678
                   Region-specific record key        ≙RECORD02␣␣12345678
                   Region number                     ≙12345678
```



Example 3

```
PL/I specificat.:  KEY = ('RECORD01')
FILE command:      KEYLEN = 10
Interpretations:   Source key                        ≙RECORD01
                   Region-specific record key        ≙RECORD01␣␣␣␣
                   Region number                     ≙KEY condition
```



Example 4

```
PL/I specificat.:  KEY = ('RECORD␣A␣REGION␣B')
FILE command:      KEYLEN = 6
Interpretations:   Source key                          ≙RECORD␣A␣REGION␣B
                   Region-specific record key          ≙RECORD␣A
                   Region number                       ≙KEY condition

  KEY      ┐
           } output   ┌──────────────┐
  KEYFROM  ┘          │RECORD␣A␣REGION␣B│
                      └──────────────┘

                      ┌────────┐ ┌─────────┐
                      │RECORD␣A│ │0000000B │
                      └────────┘ └─────────┘
  Region-specific record key  Region number
  Length:6                    Length:8
                              because non-numeric:
                              KEY condition
```

Note on duplicate keys:

Since for WRITE no check is made on the region-specific record key to ascertain whether a record having this key already exists in the region specified, records with duplicate keys may occur.
In DIRECT access these records will only be retrieved when all the preceding records having the same key in the region have been deleted.

Normally, whenever no space is found for a new record in the region specified or whenever a record to be read does not exist in the region specified, a KEY condition will be raised.

By specifying

```
ENVIRONMENT (LIMCT(n))
```

it is possible to stipulate that not only the region specified in the key is to be examined but also n others. A KEY condition will only be raised if, in n + 1 regions no space is found for a record (output) or the record is not found (input). If the end of the file is encountered during this process, wraparound to the beginning of the file occurs.

The default is LIMCT(0).

6.6.4.2 Dummy records

With OPEN OUTPUT DIRECT, the entire file is preformatted region by region with dummy records. In these the first character of each key field is filled 'FF'B4. The user himself is responsible for ensuring that his keys do not begin with the character 'FF'B4. If the file is opened with OUTPUT SEQUENTIAL, the entry of dummy records takes place in conjunction with the write operation.

6.6.4.3 Opening a REGIONAL(3) file

In the case of a file with REGIONAL(3) organization, the following takes place during the OPEN process:

With OPEN OUTPUT DIRECT, the entire primary storage space allocated to the file in the SPACE parameter of the FILE command is regarded as being divided into regions, and each region is preformatted with dummy records. Any secondary allocation is ignored.

With OPEN OUTPUT SEQUENTIAL KEYED, no preformatting takes place. However, OPEN OUTPUT SEQUENTIAL KEYED requires that with the subsequent WRITE statements the records be supplied in ascending order of region numbers. The region-specific record keys are not checked to ensure that a sequence is maintained; this means that the records within a region are not sorted according to the region-specific record keys. Regions whose numbers are skipped are preformatted with dummy records, as also is any space remaining free in a region. If a REGIONAL(3) file is extended, the same applies by analogy for REGIONAL(1) files as described in section 6.6.3.3.

6.6.4.4 Closing a REGIONAL(3) file

Closing a file is done either explicitly by specifying the CLOSE statement or implicitly at program termination.

Where an OPEN OUTPUT SEQUENTIAL has taken place, all the regions will be filled with dummy records that are not yet preformatted or filled with significant records i.e. in the event of subsequent INPUT or UPDATE processing there will be no undefined regions in existence.

6.6.4.5 Writing to a REGIONAL(3) file

The region number is obtained from the source key as previously described, and the first free dummy record is sought in this region. Into this dummy record is entered the new record (and its region-specific record key) or, in the case of LOCATE, the pointer pointing to the beginning of the record is supplied to the program and the region-specific record key is entered. If in SEQUENTIAL processing the record is not intended for the same region as its predecessor, the skipped area is filled with dummy records. See also section 6.6.4.3.
A KEY condition is raised if the region number is too large or if there is no free record in the existing region. See, however, 6.6.4.1 (LIMCNT).

6.6.4.6 Reading from a REGIONAL(3) file

A distinction is made between DIRECT and SEQUENTIAL processing. With DIRECT, the region number and the region-specific record key are obtained from the source key. The region is read into the I/O buffer (or a KEY condition is raised in the event of key out of bounds) and a sequential search is performed from the beginning to the end of the region for the first record having the region-specific record key. If such a key is not found, a KEY condition is reported; see, however, section 6.6.4.1, LIMCNT. Otherwise, the record is supplied to the program or, in the case of READ SET, the pointer is set to the beginning of the record. With SEQUENTIAL, the next record defined is sought and supplied or, in the case of READ SET, the pointer will point to the record. With READ IGNORE, as many defined records as specified in IGNORE will be skipped. An ENDFILE condition may be reported, where applicable. Dummy records are regarded as non-defined records and are not supplied. If KEYTO is specified, only the region-specific record key will be assigned to the variable specified there. See also Figure 6-16.

6.6.4.7 Overwriting a REGIONAL(3) file

For SEQUENTIAL UPDATE, the REWRITE must be preceded by a successful READ.

The record specified in the source key (with DIRECT) must be present, otherwise a KEY condition will result. Searching takes place as in the case of reading from REGIONAL(3) file. Once found, the record will be overwritten with the new contents. The first record having the desired key is overwritten in the region specified.

6.6.4.8 Deleting a record in a REGIONAL(3) file

The record is addressed on the basis of the source key and converted into a dummy record. No KEY condition is raised if the record is not found.

A KEY condition occurs if the region number is too large.

6.6.4.9 FILE command for REGIONAL(3) files

If the file is not already cataloged or if the ENVIRONMENT attribute contains corresponding options, the FILE command must contain a minimum of the following parameters:

```
/FILE      filename     (name of the data set)
 LINK    = PLI-title
[FCBTYPE = PAM,]        (default)
 RECSIZE = r,           (maximum length of the record)
 RECFORM = {F}
           {V},

           {STD     }   buffer size = 1 PAM block
 BLKSIZE = {      },
           {(STD,n) }   buffer size in 'n' PAM blocks (max. 16)
 KEYLEN  = k,           (key length of the recorded key as a number
                        of characters) (1 to 255)

 SPACE   = p            (primary allocation for OUTPUT/
                        secondary allocation for EXTEND or INOUT)
```

Rules for the FILE command:

- In RECSIZE, key length 'k' is not taken into account whereas it must be included in calculating the number of records that can be entered into one region.
- The size of region is determined by BLKSIZE, and must be such that it can accommodate at least one record, incl. key.
- Each region can accommodate one or more records, depending on the relationship between BLKSIZE and RECSIZE and on KEYLEN.
- For KEYLEN, the length of the region-specific record key is specified. $1 \leq k \leq 255$.
- In the SPACE specification, 1 PAM page must be allowed for administration information.

- The SPACE specification should be consistent with RECSIZE and KEYLEN. It is not possible to automatically extend the storage space allocated at file creation. Any increment specified will therefore be ignored.

  Specify the following: p = a * n + 1 where
  a = number of regions required
  n = BLKSIZE specification ($1 \leq n \leq 16$)

Example

File for 1000 regions, up to 100 bytes per record, up to 17 records per region, key-length 18 bytes

```
/FILE ARTICLE,LINK=NO,FCBTYPE=PAM,RECSIZE=100,                -
              RECFORM=F,BLKSIZE=STD,KEYLEN=18,SPACE=1002
```

6.7 Magnetic tape

The processing of magnetic tapes is discussed in detail in the manual "Data Management System (DMS), Tape Processing" [7]. With the exception of the attribute BACKWARDS there are no special language elements in the PL/I language for tape files; uniform language facilities are provided for all files. Due to the physical characteristics of the magnetic tape storage medium there are certain limitations and certain additional control options which are explained in the following.

6.7.1 Access methods for tape files

The only access method permitted for tape files is SAM:

```
/FILE...FCBTYPE=SAM (For exception see 6.7.4)
```

6.7.2 File attributes

The following sets of attributes can be used when declaring a tape file or when opening a tape file:

```
⎡                  ⎧INPUT ⎫             ⎤
⎢STREAM[PRINT]     ⎨      ⎬             ⎥
⎢                  ⎩OUTPUT⎭             ⎥
⎨                                       ⎬
⎢                  ⎧INPUT[BACKWARDS]⎫   ⎥
⎢RECORD SEQUENTIAL ⎨                ⎬   ⎥
⎣                  ⎩OUTPUT          ⎭   ⎦
```

The file must have CONSECUTIVE ORGANIZATION. It can be specified in the declaration with

```
[ENVIRONMENT (CONSECUTIVE)]
```

This method of organization is the default and need not therefore be specified.

6.7.3 Accessing

A magnetic tape is a storage medium that can only be written to or read sequentially. This corresponds to the CONSECUTIVE method of organization. Of the access methods normally permitted for CONSECUTIVE organization, UPDATE is not possible for magnetic tape. Figure 6-17 gives an overview of the permitted sets of attributes and the permitted statements for reading and writing records.

Organization	Attributes			Statement
CONSECUTIVE SAM F V U	STREAM	PRINT	INPUT	GET [SKIP]
			OUTPUT	PUT [SKIP]
				PUT [PAGE,LINE] [SKIP]
	RECORD	SEQUENTIAL	OUTPUT	WRITE FROM LOCATE
			INPUT [BACK-WARDS]	READ {INTO, SET, IGNORE}
			UPDATE	not possible with magnetic tape

Fig. 6-17 Attributes and accessing for magnetic tape

Details on accessing may be found in sections 6.5 and 6.6.1. In addition, it is possible when reading to begin with the last record in the file and to end with the first. This can only be done using the attribute set

```
RECORD SEQUENTIAL INPUT BACKWARDS
```

6.7.4 Closing the file

When closing a magnetic tape file, it is also possible to determine where the tape will be positioned to. The following may be specified:

```
CLOSE FILE (tapefile) [ENVIRONMENT ( { UNLOAD } )]
                                     {       }
                                     { LEAVE }
```

The ENVIRONMENT specifications have the following meaning:

none	The tape is rewound to the beginning.
UNLOAD	The tape is rewound and unloaded. The device remains assigned to the task.
LEAVE	If the tape is read forwards, it is positioned to the end of the file, or to the end of the tape if the file is continued on another tape reel. If the tape is read BACKWARDS, it is positioned to the beginning of the file, or to the beginning of the tape if the file begins on an different reel.

Note

LEAVE will have the desired effect only if the tape is closed with CLOSE LEAVE and subsequently opened for reading with OPEN SINOUT and

```
/FILE...,FCBTYPE=BTAM
```

has been declared.

7 Procedure interface

The PL/I language permits individual sections of a whole program to be compiled separately. These are the external procedures. An external procedure can be invoked from another external procedure. The PL/I facilities available for this purpose are described in detail in chapter 6 of the language reference manual [1].

Section 7.1 explains how procedures are called in machine-oriented form, and provides all the information necessary to ensure that the user fully understands this. In addition, the following control methods are dealt with in greater detail:

- Type of parameter passing (VARIABLE, ASSEMBLER)
- Calling of library modules (LIBRARY)
- Linkage editor control (WXTRN)

Knowledge of the way in which internal calls are executed is important if a memory dump is to be interpreted or if the user intends to set up an assembler procedure that has the same structure as a PL/I procedure.

Another section describes the conditions under which linkage between PL/I procedures and procedures written in assembler language can be achieved.

7.1 PL/I interfaces

The following subsections explain how an external PL/I procedure is invoked and, where applicable, how arguments are passed and a result is returned, and how a return is made to the invoking block. To understand this section, it is necessary to have knowledge of the internal representation of data elements and of the assembly language.

The information given here is relevant to Version 3.xx of the compiler. Subject to alterations.

7.1.1 Invocation interface

If a procedure is called, then certain operations are initiated first of all on the invoking side. This is dealt with in section 7.1.1.1.

Further operations are required on the invoked side. The initial processing necessary here is dealt with in detail in section 7.1.1.2.

7.1.1.1 Invoking procedure

When a block is invoked, the address at which the program to be invoked should be started is entered in register 15. Using the instruction

```
BALR 14,15
```

the address immediately following the instruction is stored in register 14 as the return address and then there is a branch to the target address specified in register 15.

The start address of the activation record of the calling block is in register 13.

Register 12 contains a reference to the dummy register vector. This register must not be changed. See also Figure 7-1.

Register	Contents
1-4	Specifications for the first four parameters if available Further parameter specifications are in the activation record
12	Internal information; must not be changed
13	Start address of the activation record
14	Return address
15	Destination address

Fig. 7-1 Register contents for the invocation of an external PL/I procedure or an assembler module in accordance with PLI1 conventions

7.1.1.2 Prolog

In the prolog of the invoked procedure, certain operations are performed which consist, basically, of saving the current values of the registers in the activation record of the invoking procedure and setting up the own activation record.

Specifically, the following functions are performed (see also the general example in Figure 7-2).

```
Program         CSECT                          Entry to assembly program
                USING Programm, 15             For constants
                B   Start                      Skip constants
                DS  OF
                DC  FL1'n'                     }
                                               } Own name of length n
                DC  CL7'Program'               }

Length          DC  F'96'                      }
                                               } For own activation record
Header          DC  X'00010000'                }
Start address   DC  A (Start)                  For USING
                USING activation record,13     For foreign activ. record
Start           STM 14,12, Register            Save registers 14 thru 12
                L   10,Start address          }
                                               } For instructions
                USING Start, 10                }
                L    9,Header                  Intermediate storage
                L    8,Tempend                 End of predecessor act.rec.
                L    7,Length                  Activ.record length
                ALR  7,8                       End  of own act. record

                LH   6,Tempsegmentno           } Only if: *COMOPT OPTIONS=XS
                ICM  6,1,Tempsegmentno         } adapt segment numbers
                CLM  6,2,4072(12)              } current segment number
                BNE  Request storage           } identical

                CL   7,444(,12)                Check whether space in
                BNH  Further
Request space   EQU  *
                L    15,432(,12)               } Request more space
                BALR 14,15                     } for stack
Further         EQU  *
                DROP 13
                USING Activation record,8      For own activation record
                STH   6,Tempsegmentno          Only if: *COMOPT OPTIONS
                                               = XS
                LR    6,7                      }
                STM   6,7 Tempend              }
                ST    9, Header word           } Record
                ST   13, Predecessor           }
                LA   13, Activation record     Start address of own
                DROP  8                        activation record for
                USING activation record, 13    own activation record
```

Fig. 7-2 Flow chart for a prolog

- The first instruction is a branch instruction, which skips the constants at the beginning. It must begin at word boundary.

- After the branch instruction mentioned above there are two fullwords containing the name of the assembler procedure under which it is called.

  It has the following format:

  – 1 byte number of significant characters in the name
  – 7 bytes name, rightmost bits filled with blanks if necessary

```
rel. address  | Activation record  DSECT
         +0   | Header word        DS   1F
         +4   | Predecessor        DS   1F
         +8   |                    DS   1F
         +12  | Register           DS  14F
         +68  |                    DS   2F
         +76  | Temporary end      DS   1F
         +80  | Permanent end      DS   1F
         +84  |                    DS   2F
         +92  | Tempsegmentno      DS   CL1
         +93  | Permsegmentno      DS   CL1
         +94  |                    DS   CL2
         +96  |
```

Fig. 7-3 Structure of the activation record for the program in Fig. 7-2

- The contents of registers 14, 15 and 0 thru 12 are saved in the activation record of the invoking procedure.
  The start address of the activation record is contained in register 13.
- In the invoked procedure an own activation record is created; this should be set up immediately after the activation record of the invoking procedure. For this, there is a check to ascertain whether there is still enough space available for the activation record. If not, more space is requested. Register 12 is needed for this purpose. Registers 7 and 8 are used for the passing and return of the values. With the exception of registers 7, 8, 14 and 15, none of the registers is altered.
- The following values are entered in the own activation record (see also the description of the activation record in chapter 10):

  – Block type (+1)

  – Label (+2)

  – Predecessor activation record (+4)
    The value of register 13 is entered.

  – Permanent end (+80)
    The first address after the own activation record is stored. This value is usually calculated by adding the equivalent field of the predecessor activation record to the length of the own activation record.
    If, however, there is insufficient space available after the predecessor activation record, this field is modified appropriately as more space is requested.

  - Temporary end (+76)
    The value that is stored for the temporary end is the same as that stored for the permanent end (+80).

- The start address of the own activation record is loaded in register 13. This register is not reset until just before the return to the calling procedure.
- The contents of register 12 must not be modified.

A detailed explanation of the internal structure of the activation record can be found in chapter 10. Figure 7-3 illustrates the structure of the activation record as can be used for the assembler program in Figure 7-2.

7.1.2 Passing of parameters

The values of any parameters specified where a procedure is invoked must be made available to the invoked procedure. This is normally done by passing, for each parameter, a pointer indicating the beginning of the location at which the value is stored.

Note the following when modules are compiled with "*COMOPT OPTIONS = XS". For parameters of type BIT UNAL, the passed pointer points to an absolute bit pointer that points to the parameter value.

In certain cases, the data description associated with the parameter is required in addition to the parameter value.

There are several ways of passing parameters. For external procedures, the user can control them by specifying OPTIONS:

```
DCL procedure ENTRY etc. or OPTIONS (option)
```

The external procedure must be capable of processing the parameters appropriately. Details are given in the subsections that follow.

Section 7.1.4 explains how, under certain conditions, the result of a function reference is also handled as an additional parameter.

If an argument is passed not "by reference" but "by assignment" (see also section 6.2.4.2 in the language reference manual [1]), then the passed pointer indicates the storage location of the generated auxiliary variable and not the storage location of the argument. Accordingly, the data description for the auxiliary variable is also passed, if necessary.

7.1.2.1 Normal case (PL/I)

The general rules set out below apply for procedures where, during the declaration of external procedure via a DECLARE statement, OPTIONS is not specified for the control of parameters. By adhering to these rules, the user will produce the optimum conditions with regard to time and storage space economy.

For each parameter a pointer is passed; this indicates the start of the storage location at which the parameter value is stored.

The data descriptions for the parameters are, where applicable, passed after the parameter values.

The following rules apply with regard to the transfer of the data descriptions for the parameters:

- A pointer to the data description is passed,
  - if the parameter contains an * (AREA, BIT, CHARACTER),
  - and if the DIMENSION or STRUCTURE attribute is present.
- A length specification is passed
  - if there is an * in the parameter (AREA, BIT, CHARACTER),
  - and if the parameter is scalar.
- An undefined value is passed,
  - if the AREA, BIT, CHARACTER or DIMENSION attribute is present
  - but there is no * in the parameter.
- In all other cases the data descriptions are not passed.

If the data description of the parameter is not known by the invoking side, it is assumed that it contains an *.

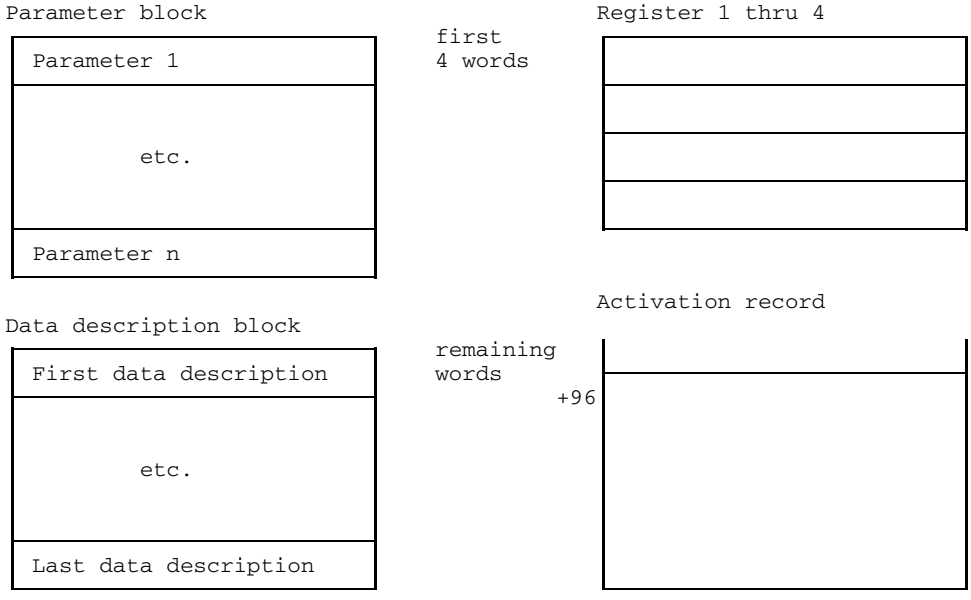


Fig. 7-4 Passing of parameters (Normal Case)

If the pointers to the parameter values are listed in the order in which they are written in the source program and if, subsequently, the values passed for the data description are listed in the same order, they are passed to the called procedure in the following way (see Figure 7-4).

- the first four words in registers 1 thru 4 and
- all the remaining words after relative address +96 in the activation block of the invoking procedure.

The start address of the activation record is transferred into register 13.

There is a detailed explanation of the data description in chapter 10.

In certain cases, if a procedure is invoked by means of a function reference, an additional parameter can be passed for the return of a result. This is described in section 7.1.4.

Using the PARAMETER (INPUT) attribute (see PL/1 Beschreibung, section 4.2, PARAMETER) it is possible to specify that this parameter only passes values to the called procedure, but not to the calling procedure (passing by statement; passing by value; see PL/1 Beschreibung section 6.2.4.2). If the parameter satisfies the following conditions, the value itself is passed instead of the pointer to the parameter value:

- scalar parameter
- no * specification
- not greater than a fullword.

If the value occupies less than a fullword, entities with REAL BINARY attributes are stored right-justified; all other ones, left-justified in the fullword.

As regards parameters of external procedures, the specification must be given both on the calling and on the called side of the same parameter:

Example

Calling side

```
DCL  Entry      ENTRY (CHAR(1),         PARAMETER (INPUT),
                       FIXED BINARY(15) PARAMETER (INPUT),
                       CHAR(30);
```

Called side

```
Entry:    PROCEDURE (A,B,C);

DCL     A       CHAR(1)            PARAMETER (INPUT),

        B       FIXED BINARY(15)   PARAMETER (INPUT),

        C       CHAR(30)           PARAMETER;
```

As regards parameters for internal procedures the entry as given for the called side will do.

The PARAMETER (UPDATE) and PARAMETER (OUTPUT) specifications do not effect a change in parameter passing.

7.1.2.2 General assembler convention (VARIABLE)

In special cases, it may be necessary to pass the data descriptions for all the parameters during parameter passing. This can be achieved by specifying OPTIONS (VARIABLE) when declaring the entry. The following should be declared:

```
DCL entry ENTRY etc.
          OPTIONS (VARIABLE);
```

This form of parameter passing is designed for use in the invocation of assembler procedures. It cannot be used for PL/I procedures.

If OPTIONS (VARIABLE) is specified, then the parameters are not passed directly, but are combined with the data descriptions in an input block and a pointer to this input block is transferred into register 1. The structure of the input block is as follows (see also Figure 7-5):

- The first word is a header word, the rightmost 16 bits of which contain the number of parameters passed. The leftmost 16 bits are undefined.
- There then follows a pointer for each argument, in the order in which they are written in the source program; the pointer refers to the address at which the argument value is stored.
- If the procedure in question has been referenced by means of a function reference, the result is returned via an additional parameter, as described in section 7.1.4.
- For each parameter and, where applicable, for the result, a pointer then follows indicating the data description belonging to the parameters and, if appropriate, to the result.
- For data descriptions having the PICTURE attribute, data type X’00’ or X’01, and not data type X’3A’ or X’3B’, is transferred; the former has an extended data description since it also contains a picture description. The data description and the picture description are dealt with in detail in section 10.6.

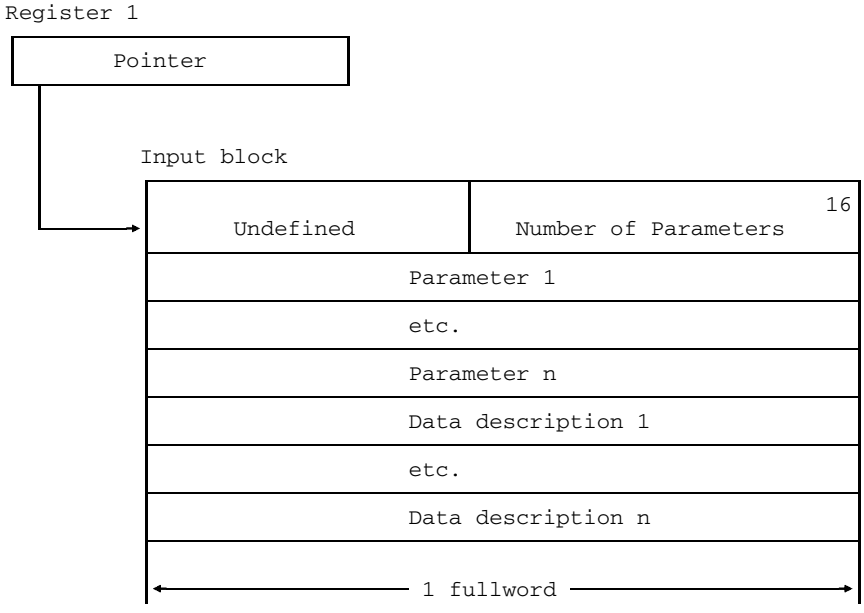


Fig. 7-5 Passing of parameters in the input block when OPTIONS (VARIABLE) is specified

7.1.2.3 Standard assembler convention (ASSEMBLER)

If OPTIONS (ASSEMBLER) is specified, then the parameters are passed in accordance with Industry Standard compatible assembler conventions. The declaration is then as follows:

```
DCL entry ENTRY etc. OPTIONS ( { ASSEMBLER } [INTER]);
                               { ASM       }
```

All entries shown above following OPTIONS mean the same.

An input block is set up for the transfer and its address is passed in register 1. The structure of the input block is as follows (see also Figure 7-6):

- A pointer is specified for each argument, in the order in which they are written in the source program; the pointer refers to the address at which the argument value is stored. The leftmost 8 bytes all have the value ’0’B.
- For the last parameter, the leftmost bit has the value ’1’B, and not ’0’B.
- If no parameters are passed, then all the bits in register 1 have the value ’0’B.

Data descriptions are not passed.

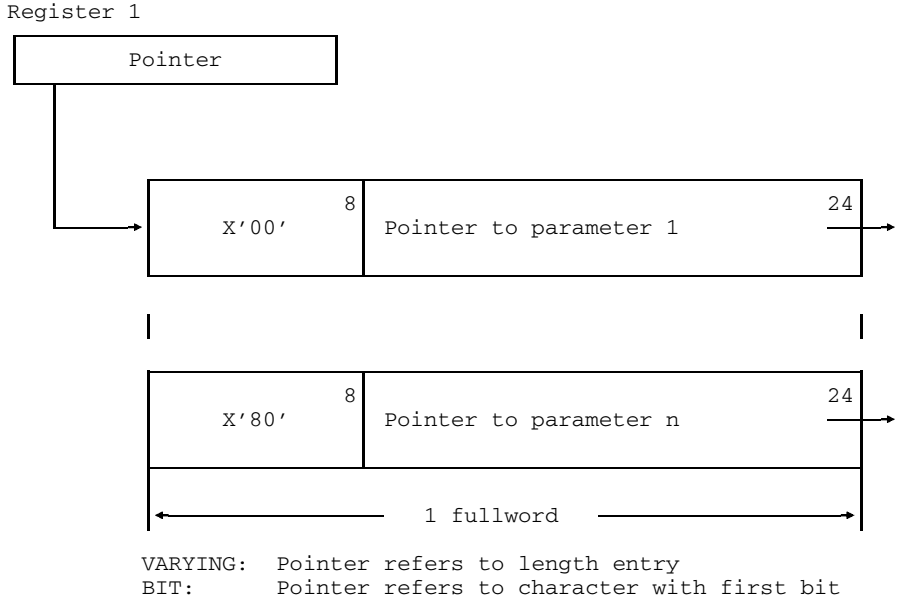


Fig. 7-6 Passing of parameters in the input block when OPTIONS (ASSEMBLER[INTER]) is specified

The following exceptional feature should be noted:

- If a parameter has the VARYING attribute, the pointer indicates the length specification before the data, and not the beginning of data. See also the internal representation in chapter 10.
- Only a byte address can be passed as a pointer. This must be taken into consideration if the parameter has the BIT attribute and the first bit does not begin at a byte boundary.

7.1.3 Problem processing

After the prolog comes the actual solution of the problem. Here,

- register 13 always contains the start address of the own activation record and
- the contents of register 12 are never modified.

Register	Contents
12	internal information; is never modified
13	start address of the own activation record (reset in call phase or return phase)

Fig. 7-7 Registers that always have specific contents if PL/I blocks are used

7.1.4 Return of the result

If a procedure is invoked via a function reference then, on return, a result is returned to the invoking procedure. The result is returned in the following manner:

- A real scalar floating point value is returned in floating point registers F0 thru F3.
- A value with a maximum length of 1 fullword is returned in register 1.
- All other values are returned via an additional parameter; one returned value is processed with an * as a special case.

See also the following subsections.

7.1.4.1 Return in register 1

The value of the result is returned in register 1 if the item is scalar and one of the following data types is given:

- BIT(n) NONVARYING where n ≤ 32; the value is stored in register 1 and is right-justified.
- REAL FIXED BINARY (always like PREC(31,x))
- POINTER
- OFFSET

7.1.4.2 Return in floating point registers

The value of the result is returned in floating point registers if the item is scalar and the following data types are present:

- REAL FLOAT BINARY PRECISION (g)
- REAL FLOAT DECIMAL PRECISION (g)

The registers are used as follows:

• Register F0 left half	for DECIMAL	g = 1 thru 6
	for BINARY	g = 1 thru 21
• Register F0	for DECIMAL	g = 7 thru 16
	for BINARY	g = 22 thru 53
• Register F0 and F2	for DECIMAL	g = 17 thru 33
	for BINARY	g = 54 thru 109

7.1.4.3 Return via parameters

If none of the cases described in sections 7.1.4.1 and 7.1.4.2 apply, then the value is returned via an additional parameter, the result parameter.

The result parameter is the last parameter in the parameter list. On invocation the pointer is defined, but the value of the parameter is not. The called program stores the result in the storage area indicated by the pointer of the result parameter. The structure of the parameter and the allocation of the data description are the same as for other parameters and can be found in section 7.1.2.

If no parameters have been declared for the procedure, then the result parameter is the only parameter.

If the result is a value with an *, it is returned in accordance with the rules described in section 7.1.4.4.

7.1.4.4 Return when * is specified

If an * is specified in the RETURNS option in the PROCEDURE or ENTRY statement, then, as explained above, another parameter, the result parameter, is added to the list of parameters; this is used for the result.

There is always a data description for this result parameter.

During parameter transfer, this result parameter is treated as a normal parameter. Its value is undefined when the procedure is called; it is defined later in the called procedure.

The result parameter consists of a pointer (indirect pointer), which indicates a pointer in the activation record of the calling procedure (direct pointer). The value of the direct pointer is undefined when the procedure is called.

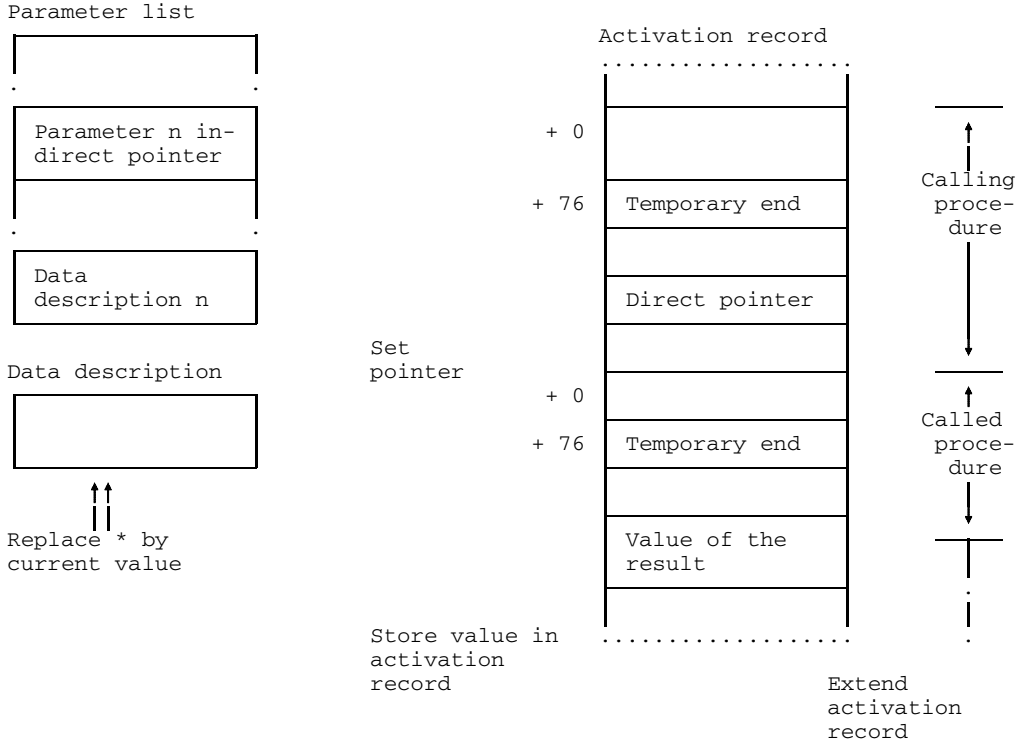


Fig. 7-8 Return of a value when * is specified

The following functions are executed by the invoked procedure (see also Figure 7-8):

- In the data description, the * is replaced by the current value (Pos. 1 in Figure 7-8).
- The current activation record is temporarily extended by the amount of storage space required for the result and the result is stored there Pos. 2).
- A pointer to the storage location occupied by the result is stored as a direct pointer in the storage cell indicated by the indirect pointer passed as a parameter. The storage cell of the direct pointer is located in the activation record of the invoking procedure (Pos. 3).

  In contrast to the general rule, in the case of variables having the VARYING attribute, the pointer does not indicate the length specification, but the first byte after the length specification.
- When there is a return from the invoked procedure, its (extended) activation record is not freed (as in all other cases), but is added to the activation record of the invoking procedure. It is then freed by the invoking procedure after the result of the function reference has been processed (Pos. 4).

All these interrelationships are illustrated in Figure 7-8. The structure of the activation record is described in chapter 10 and the passing of parameters is dealt with in section 7.1.2.

7.1.5 Terminating a procedure

There are two ways of leaving the invoked procedure.

- Return
  There is a return to the place from which the procedure was invoked (RETURN, END).
- Branch
  There is a branch to any position of a dynamically preceding (calling) procedure, via a GOTO statement.

7.1.5.1 Return

If the procedure returns to the point of invocation, the following environment is established:

- The start address of the predecessor activation record is loaded in register 13. It is located in the own activation record (+4).
- Registers 2 thru 11 are reset to the values they had when the procedure was invoked.
- If no result is returned in register 1, it is set to the value it had when the procedure was invoked.

The return address was stored in register 14 on invocation and was saved in the activation record of the predecessor (+12) in the prolog. If register 14 is also reset to its former value, then a return can be made via the BR 14 instruction.

If the invocation is a function reference, a result is returned. This is dealt with in section 7.1.4.

Register	Contents
0	any
1	functional value (result) or unchanged
2-11	unchanged
12	unchanged; cannot be temporarily altered either
13	unchanged (pointer to the activation record of the calling procedure)
14-15	any (register 14 usually contains the return address)

Fig. 7-9 Register contents for the return to a PL/I procedure

```
Return      EQU *
            USING Activation record, 13   for own activation record
            L   13,Predecessor            address of predecessor activ.
                                          record
            LM  1,11,Register+12 1)       restore register 1 thru 11 1)
            LM  2,11,Register+16 1)       restore register 2 thru 11 1)
            L   14,Register+0             restore register 14
            BR  14                        branch
```

1) Do not restore register 1 during a function reference.

Fig. 7-10 Example of a return to the invoking point

7.1.5.2 Branch

A procedure can be left via a GOTO statement. In this case the branch destination must be in a dynamically preceding block.

The following functions are executed in the invoked procedure to permit the branch:

- The third full word of the label value contains the address of the activation record that belongs to the branch destination. This address is entered in register 13 thus establishing the environment of the target procedure.
- Registers 3 thru 11 are reset at their former values, which were saved in the activation record of the branch destination (relative 32).
- The base address for the branch destination is transferred to register 10. It is located in the second fullword of the label value.
- A branch is made to the address of the branch destination, which is located in the first fullword of the label value.

See also Figures 7-11 and 7-12.

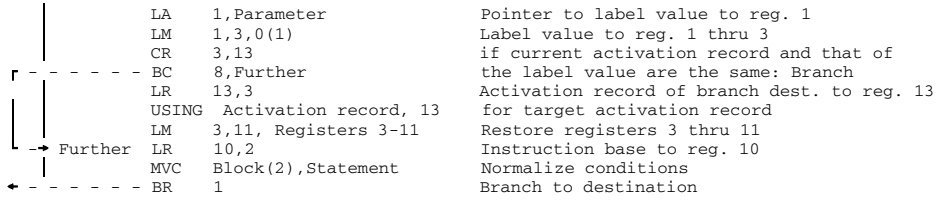


Fig. 7-11 Flow diagram for a branch to a label passed as parameter

```
      Activation record DSECT
+0                      DS 8F
+32   Registers 3 - 11  DS 9F       Registers 3 - 11
+68                     DS 4F
+84   Block conditions  DS CL2   }
                                 } Bits for enabled conditions
+86   Statement         DS CL2   }
```

Fig. 7-12 Structure of the activation record for Figure 7-11

7.1.6 Library procedure (LIBRARY)

Certain services are available to the user in the form of supplied and ready-compiled external procedures. The scope of the services provided by these procedures is dealt with, separately for each procedure, in chapter 11. The general interrelationships involved are to be found in the appropriate sections.

If one of these procedures is invoked, the entry must be declared in the invoking procedure via the DECLARE statement, as specified in the description of the procedure.

For some of the procedure

```
OPTIONS (LIBRARY) or
OPTIONS (LIB)
```

must be specified when the entry is declared. This means that as many number characters (#) as are needed to make the name 8 characters long are added to the entry name, which, in accordance with the general rules for external names, can consist of a maximum of 7 characters. The procedure is contained under this name in the PLI1 runtime system.

This means that the user can also use the same name for his own external procedures, as long as he adheres to the PL/I rules governing the use of names. See also the example in Figure 7-13.

The OPTIONS (LIBRARY) specification should not be used if the entry identifies an external procedure that has been generated and compiled by the user.

```
DCL ERROUT ENTRY OPTIONS (LIBRARY);
CALL ERROUT; ..........................  call entry ERROUT##
BEGIN;                                   in the library
   DCL ERROUT ENTRY;
   CALL ERROUT; .......................  invokes own external
                                         procedure ERROUT
```

Fig. 7-13 Example of the same entry name with and without OPTIONS LIBRARY

7.1.7 (WXTRN) Linkage

If, in an external procedure, the entry of another external procedure is declared, then the latter is automatically included at link-edit time; an explicit specification is not required here.

If automatic linkage is not desirable, it can be prohibited by specifying OPTIONS (WXTRN) during entry declaration:

```
DCL entry ENTRY or OPTIONS (WXTRN);
```

Such an entry is only bound into the program if this is explicitly requested via the INCLUDE specification at link-edit time or if it is automatically linked as a result of another external procedure belonging to the program. See also section 3.3.4.

7.2 Assembler procedures

In certain cases it may be desirable, or necessary, to use a procedure written in assembler language rather than an external PL/I procedure. This can be for any of the following reasons:

- The functions of an already existing assembler-language program are to be used in a PL/I program.
- Functions that can only be implemented using assembler language are to be used in a PL/I program.
- A PL/I procedure is to be replaced by an equivalent, but more efficient, assembler procedure.

There are two ways of achieving this:

- An assembler program having the same characteristics as a PL/I procedure is created.
- An assembler procedure that conforms to standard assembler conventions is written (see section 7.1.2.3).
- If data descriptions of the parameters are required, the general assembler convention (see section 7.1.2.2) can be used.

In chapter 13 of this manual ASSEMBLER macros are described which simplify the connection of ASSEMBLER programs to PL/I programs and vice versa.

This is further explained in the following subsections:

7.2.1 Assembler procedure conforming to PLI1 conventions

This method of using assembler procedures in PL/I programs is particularly suitable if the assembler procedures are newly written and specially designed for use in PL/I programs. Then they can be adapted most efficiently.

In section 7.1 there is a detailed explanation of how PL/I procedures are invoked on the assembler level. An assembler procedure called by a PL/I program must behave exactly like a PL/I procedure in terms of initial processing (prolog) following the call, acceptance of parameters, result return, and control transfer.

If an assembler procedure called by a PL/I program is to invoke a PL/I procedure in turn, then the invocation, parameter supply, and where applicable the acceptance of results must also be carried out as with PL/I procedures.

When a PL/I program run is terminated, there are still certain concluding operations to be performed, for example the contents of output buffers are output and open files are closed etc. This is known as termination processing. It can be compared with the "ON FINISH ON unit" facility available in the PL/I language.

If the assembler procedure is set up in accordance with PL/I conventions, then the PL/I termination processing is also valid for it.

If an error occurs in the assembler procedure, then the appropriate PL/I error processing facility is activated and, if applicable, the appropriate ON unit is called.

To ensure that the error handling routine of the PL/I runtime system is also available for any errors that may occur in the assembler procedure, it is good practice to declare the entry point of the assembler program as follows in the calling PL/I procedure:

```
DCL entry or OPTIONS (PLI1)
```

Then register R13, which is important for error handling, is saved on every call and reset on return.

7.2.2 Assembler procedures conforming to standard assembler conventions

If assembler programs generated in accordance with Standard Assembler Conventions are invoked by PL/I procedures, then, during the declaration of the assembler procedure entry in the PL/I program,

```
OPTIONS (⎧ASSEMBLER⎫[INTER])
         ⎨         ⎬
         ⎩ASM      ⎭
```

must be specified. The parameters are then passed in the form described in section 7.1.2.3. The INTER option has no additional meaning.

As a prerequisite, however, the assembler subroutine must observe certain restrictions:

- no own interrupt handling
- no modification of register 12
- no calling of PL/I procedures
- no occurrence of conditions
- Assembler subroutine cannot be called by function reference

7.2.3 Assembler procedures conforming to general assembler conventions (VARIABLE)

All details of 7.2.2 apply accordingly. Declaration is by:

```
OPTIONS (VARIABLE)
```

The passing of parameters is described in 7.1.2.2. Unlike OPTIONS (ASSEMBLER), returns of function values are permitted, though they are then treated as additional parameters. A special benefit of this type of parameter supply is that a corresponding PL/I descriptor is generated or evaluated for every parameter and that due to the separate calculation of the number of parameters, the BIT UNALIGNED parameter type may be used also.

7.2.4 Invocation of PL/I procedures from assembly-language programs

If PL/I procedures are to be invoked from assembly-language programs, the entry to a PL/I procedure must be declared as follows:

```
Entry:  {PROCEDURE}
        {         } OPTIONS(ASSEMBLER)
        {ENTRY    }
```

Note the following when writing the declaration:

- The specifications ASSEMBLER and MAIN cannot be used together.
- The RETURNS specification is not permissible.
- The PL/I program interrupt handling feature must have been activated when the PL/I procedure is running.
- When a PL/I procedure is invoked the following action is to be taken in the invoking assembly-language program:
  The address of the parameter list is expected in register 1. The parameter list is to be created in accordance with standard assembler conventions (see section 7.1.2.3).
  The address of the save area is to be entered in register 13. The save area must start on word boundary and have a length of 18 words.
  The address of the PL/I procedure that is to be called must be contained in register 15.
  The invocation must be effected with the instruction BALR R14, R15. Return from the PL/I procedure is made via register 14.

7.3 FORTRAN and COBOL procedures

7.3.1 General

The PLI1 interlanguage facilities permit communication, at execution time, with FORTRAN and COBOL procedures. The foreign procedures should have already been generated by:

the FORTRAN compiler	FOR1 from V1.40 or
the COBOL compiler	COB1 from V1.21.

Communication between PLI1 and the foreign procedure is handled as follows:

- in the case of FORTRAN via parameters and EXTERNAL data
- in the case of COBOL via parameters only.

A PLI1 procedure references a COBOL procedure by a CALL statement, a FORTRAN procedure by a CALL statement or as a function procedure. Parameters are supplied with the call; function values are returned through functions.

A COMMON block in FORTRAN and a PLI1 variable with the STATIC EXTERNAL attribute are stored in the static memory. If both have the same name, then they are on top of one another and they have the effect of two identical STATIC EXTERNAL variables in PLI1: if one of the variables is assigned a value, then the other is simultaneously assigned the same value.

There is no comparable capability in COBOL; only parameters can occupy the same storage location in PLI1 and COBOL.

The whole language interface is implemented by the PLI1 compiler using OPTIONS entries. Existing FORTRAN and COBOL procedures need, in general, not be changed or recompiled and new procedures can be generated without being considered for interlanguage communication.

In COBOL, a procedure is a subroutine, but in FORTRAN it is subroutine or a function. The conventions of the language in question are not affected by the language interface.

The processing of a file must be terminated with a CLOSE statement before calling a foreign procedure if the same file is to be processed in this foreign procedure.

REGIONAL files can be processed in PLI1 procedures only.

In case of CONSECUTIVE access to ISAM files PLI1 will only process 4-byte BINARY keys or 8-byte CHARACTER keys.

7.3.2 Matching the data

A detailed knowledge of the COBOL and FORTRAN languages is not necessary for interlanguage communication, but knowledge of the data representation is. The internal representation of the argument in PL/I and that of the corresponding parameter in FORTRAN or COBOL must be compatible. As is the case with the passing of parameters between external PL/I procedures, so too here the compatibility cannot be chekked by the compiler. This is the responsibility of the user.

If a PL/I data type of a parameter has no equivalent data type in FORTRAN or COBOL, the compiler issues a warning. Results at runtime are then unpredictable and the program may terminate abnormally. PL/I has more data types than FORTRAN or COBOL and not all of them have equivalents in either of the two other languages.

Besides the data types, the alignment of the data in memory is also important. The following combinations are valid here:

	aligned	unaligned
PL/I	ALIGNED	UNALIGNED
FORTRAN	Normal case	-
COBOL	SYNCHRONIZED	not SYNCHRONIZED

The alignment of an argument is deduced, like the data type, from the parameter description or from the argument itself. Only ALIGNED arguments can be passed to SYNCHRONIZED COBOL parameters, or to FORTRAN parameters. Both ALIGNED and UNALIGNED parameters can be passed to COBOL parameters that have not been declared with SYNCHRONIZED. The user himself is responsible for the consistency of arguments and parameters.

FORTRAN

The data types which are compatible between PL/I and FORTRAN are listed in Figure 7-14.

FORTRAN	PL/I only ALIGNED data	
INTEGER * 2	FIXED BINARY (p,q)	where 0 < p ≤ 15, q=0
INTEGER * 4	FIXED BINARY (p,q)	where 15 < p ≤ 31, q=0
REAL * 4	FLOAT BINARY (p) FLOAT DECIMAL(p)	where p ≤ 21 where p ≤ 6
REAL * 8	FLOAT BINARY (p) FLOAT DECIMAL(p)	where 21 < p ≤ 53 where 6 < p ≤ 16
REAL * 16	FLOAT BINARY (p) FLOAT DECIMAL(p)	where 53 < p ≤ 109 where 16 < p ≤ 33
COMPLEX * 8	COMPLEX FLOAT BIN(p) COMPLEX FLOAT DEC(p)	where p ≤ 21 where p ≤ 6
COMPLEX * 16	COMPLEX FLOAT BIN(p) COMPLEX FLOAT DEC(p)	where 21 < p ≤ 53 where 6 < p ≤ 16
COMPLEX * 32	COMPLEX FLOAT BIN(p) COMPLEX FLOAT DEC(p)	where 53 < p ≤ 109 where 16 < p ≤ 33
LOGICAL * 1	BIT(8)	
LOGICAL * 4	BIT(32)	
CHARACTER n fixed length	CHARACTER (n) NONVARYING	
CHARACTER n variable length	CHARACTER (n) VARYING	

Fig. 7-14 Compatible PL/I and FORTRAN data types

PL/I arrays (DIMENSION), providing they have connected memory, can be passed to FORTRAN fields. If the elements are character strings, they must have the NONVARYING attribute. Fields cannot be returned as a functional value.

In contrast with arrays in PL/I, which are stored page by page, multidimensional fields in FORTRAN are stored column by column. By iSUB defining of an array it is possible to access the same element in PL/I, e.g. using B (i,j,k), as is accessed in FORTRAN, using A (i,j,k). (See section 4.2 under DEFINED attributes: iSUB defining in the PL/I reference manual.)

Example

1. PL/I routine calls FORTRAN subroutine, with parameter passing:

```
PLIROUT:        PROC;

       DCL    FORUP ENTRY (DIM(4,5,6)...)OPTIONS(FORTRAN);
       DCL A  DIMENSION (4,5,6)...;
       DCL B  DIMENSION (6,5,4)DEF A (3SUB,2SUB,1SUB)...;

       /* IN PLI B IS USED */
       CALL FORUP (A);

END;
```

2. PL/I subroutine is invoked by a FORTRAN routine, with parameter passing:

```
PLIUP:    PROC (A)   OPTIONS (FORTRAN);
            .

       DCL A  DIMENSION (4,5,6) PARAMETER...;
       DCL B  DIM (6,5,4) DEF A (3SUB,2SUB,1SUB)...;

       /* IN PLI B IS USED */
            .
END;
```

In both cases, in FORTRAN, the fields are declared with

```
       DIMENSION A (6,5,4)
```

Note

DEFINED variables with iSUB option cannot be used with GET DATA and PUT DATA.

Declarations with an * are also allowed in permissible places when parameters are passed to FORTRAN. There is then a check to ensure that current values are used in the data descriptions to be passed.

COBOL

The compatible PL/I and COBOL data types are listed in Figure 7-15.

Concerning alignment, the following options are compatible:

```
ALIGNED   and SYNCHRONIZED
UNALIGNED and no SYNCHRONIZED option.
```

If arrays are passed, a warning is issued, since they are only compatible to a limited extent with a COBOL table defined via the OCCURS clause.

COB1	PLI1
COMPUTATIONAL 1-4 digit positions	FIXED BINARY (p,0), where 0 < p ≤ 15
5-9 digit positions	FIXED BINARY (p,0), where 15 < p ≤ 31
COMPUTATIONAL-1	FLOAT DECIMAL (p), where p ≤ 6 FLOAT BINARY (p), where p ≤ 21
COMPUTATIONAL-2	FLOAT DECIMAL (p), where 6 < p ≤ 16 FLOAT BINARY (p), where 21 < p ≤ 53
COMPUTATIONAL-3 (n)	FIXED DECIMAL (n)
DISPLAY	CHARACTER (n)

SYNCHRONYZED	ALIGNED
without this option	UNALIGNED

Fig. 7-15 Compatible PL/I and COBOL data types

Structures can be passed (on the COBOL side they correspond to logical data records organized by means of level numbers) if there is a guarantee that the internal data structure in PL/I matches that in COBOL. This is generally the case up to level number 2. Chapter 10 contains details of the internal representation for PL/I. In the case of ALIGNED structures with aligned data types (BINARY, FLOAT etc.), a certain incompatibility may arise.

7.3.3 Declaration, call

The entry to a FORTRAN or COBOL procedure must be declared via

```
DCL entry ENTRY etc.

          ⎧FORTRAN [INTER]⎫
OPTIONS (⎨                ⎬)
          ⎩COBOL          ⎭
```

The options have the following meanings (see also 7.3.4):

FORTRAN	A FORTRAN procedure is called. Program interrupts occurring during the execution of the FORTRAN procedure are handled by PL/I.
FORTRAN INTER	A FORTRAN procedure is called. Program interrupts occurring during the execution of the FORTRAN procedure are handled by FORTRAN.
COBOL	A COBOL procedure is called.

The following should be taken into account when declaring the entry:

- In COBOL there are no function procedures; consequently, the RETURNS attribute cannot be specified.
- The OPTIONS option is also permitted in the case of entry variables.
- An entry, as defined above, can also be used in the case of GENERIC.
- The NOMAPIN, NOMAPOUT and ARGi options permitted with OPTIONS in programs conforming to Industry Standard are ignored. The course of action is as for NOMAP.

7.3.4 Interrupt handling

Only a few of the interrupts that can occur at execution time are handled in FORTRAN. The PL/I-FORTRAN language interface makes it possible for any interrupts that have not been handled to be passed on to PL/I. For this, the user must specify OPTIONS (FORTRAN INTER). By doing this, the interrupts that are not handled by the invoked FORTRAN procedure are handled in PL/I by means of an ON unit, or by the PL/I system's ON unit. If INTER has not been specified, any hardware or system interrupt that occurs is only handled by PL/I.

The following currently applies:

- The INTER option has been specified:
  Before each FORTRAN procedure invocation, the interrupt handling facility of FOR1 is activated. The PLI1 error processing facility is deactivated. There is, however, a guarantee that PLI1 termination processing will still be performed if an abnormal termination in a FORTRAN procedure occurs.

  When the procedure returns from the FORTRAN procedure, the PLI1 interrupt handling facility is reactivated.

  Due to the not insubstantial amount of computer time that is used up in switching around the interrupt handling facility for every invocation (several thousand machine instructions), this should only be implemented in the case of relatively large FORTRAN procedures and, if necessary, in the test phase, for the purpose of improving error diagnosis.
- The INTER option has not been specified:
  All interrupts are processed by the PL/I interrupt handler. There is no switching around between different interrupt handling facilities, that is
  - all errors detected by the FOR1 system are handled by the FOR1 interrupt handler and then passed on to PLI1 termination processing.
  - the interrupts reported by the hardware of the operating system (STXIT handling) are handled directly by the PLI1 error processor; this may require that the object listing and the memory dump be analyzed for the purpose of assigning the interrupt to the statement that caused it to occur.

    Remedy: Repeat the program run with a recompiled program having the OPTIONS (FORTRAN INTER) specification and/or use the IDA debugging aid in the FORTRAN procedure.

In COBOL, INTER has no effect as there is no COBOL interrupt handling by STXIT. The same applies as for OPTIONS (FORTRAN), i.e. without INTER.

Exception
After errors detected by the COBOL system have been handled, control does not return to PL/I. For remedy, see section 7.3.5.

7.3.5 Program termination

- FORTRAN
  - Action required at the end of the PL/I program
    A user writing a PL/I program with language transfer need not take any special measures to end the FORTRAN calls at the end of the PL/I program.
  - Action required at the end of the FORTRAN procedure
    PLI1 termination processing is guaranteed to be activated when the FORTRAN procedure is terminated via STOP or abnormal termination. It may even be possible to continue the PL/I program run via the FINISH condition.
- COBOL

  In the COB1 runtime system it is not possible to declare termination processing. If the COBOL procedure terminates abnormally, as a result of a STOP RUN statement or a program error detected by the COBOL system for example, the PL/I program will not close properly. Data for certain files may be lost.

7.3.6 Invocation of PL/I procedures from FORTRAN and COBOL programs

If PL/I procedures are to be invoked from FORTRAN or COBOL programs, the entry to a PL/I procedure must be declared as follows:

```
Entry:{PROCEDURE} OPTIONS({FORTRAN})
      {         }       ({      })
      {ENTRY    }       {COBOL  }
```

The options have the following meanings:

FORTRAN	The PL/I procedure is invoked by a FORTRAN program.
COBOL	The PL/I procedure is invoked by a COBOL program.

The following should be noted when writing a declaration:

- The COBOL and FORTRAN options are mutually exclusive. They cannot be used together on one entry even if the MAIN option is specified.
- The RETURNS option cannot be used together with the COBOL option, since there are no COBOL function procedures.
- For parameters with the attributes AREA, BIT, CHAR or DIMENSION, only integer constants are allowed as length and dimension specifications.
- The NOMAP (p), NOMAPIN and NOMAPOUT options specified in accordance with PLI1 conventions are provided with a warning and are ignored. For all parameters the presetting NOMAP holds good.
- Ensure that the PL/I program interrupt handling is activated when the PL/I procedure is running.
- In all other cases the provisions for interrupt handling stated under section 7.3.4 apply.
- At end of program the procedure described in section 7.3.5 is followed.

7.4 ILCS procedures

7.4.1 General

The PLI1 language interfacing facilities also permit program communication at runtime in accordance with the conventions of the Inter Language Communication Services (ILCS). These conventions enable any combination of programs written in different programming languages to be linked. Further details are given in the ILCS release notice. The ILCS capability of the various language compilers is given in their documentation.

Communication between ILCS-capable routines is handled by means of arguments 'by reference', i.e. only references to arguments are passed. The routines can be invoked as procedures or as functions (in so far as this is permitted in the respective language).

The ILCS interlanguage facilities are enabled on the PLI1 side by means of OPTIONS entries. The ILCS statements in other programming languages can be found in their documentation.

7.4.2 Start handling

Each ILCS-capable main program first calls the ILCS start handling routine. This, in turn, calls all participating language-specific start handling routines to ensure that all the language environments concerned are set up and operative.

7.4.3 Declaration, call

An ILCS entry to a PLI1 procedure or a PLI1 function is identified by means of the OPTIONS option as follows:

```
n : { PROCEDURE } [(p)] OPTIONS (ILCS) [ RETURNS ( { FIXED } ) ]
    { ENTRY     }                                  { FLOAT }
```

Invoking an ILCS procedure or ILCS function from a PLI1 procedure is declared in the following manner using the OPTIONS option:

```
DCL n ENTRY [ ( p ) ] OPTIONS (ILCS) [ RETURNS ( { FIXED } ) ]
                                               { FLOAT }
```

Where:
- n is the name of the procedure or function to be invoked.
- p is the list of arguments of the procedure or function to be invoked.

An ILCS procedure or ILCS function is invoked from a PLI1 procedure in the same way as a PLI1 procedure is invoked using the CALL statement.

7.4.4 Mapping of files

ILCS conventions support only the following parameter types:

BINARY FIXED (31)
BINARY FLOAT (21)
BINARY FLOAT (53)
DECIMAL FLOAT (6)
DECIMAL FLOAT (16)
CHARACTER (i)

The arguments must have normal alignment (fullword or byte boundary, depending on type), i.e. on the PLI1 side they must have the attribute ALIGNED. For the corresponding declarations in other languages, refer to the relevant literature.

As the ILCS language interface does not recognize descriptors for arguments, it is not possible to check the arguments in the calling and invoked routines. It is the user's responsibility to ensure that the arguments in the calling routine correspond to the parameters in the invoked routine as regards data type and mapping.

7.4.5 Interrupt handling

For interfacing ILCS programs with PLI1 programs, the ILCS interrupt handling is deactivated on entering the PLI1 program and PLI1 interrupt handling is activated. If errors should occur in the PLI1 program, PLI1 error handling is performed. On quitting the PLI1 program, PLI1 interrupt handling is deactivated and ICLS interrupt handling is reactivated.

7.4.6 Termination handling

If a PLI1 program invoked by an ILCS program runs to end-of-program, the ILCS termination handling routine is called. This performs termination processing for all the participating ILCS programs and consequently also invokes the PLI1 termination handling routine. Finally, it terminates the program itself.

8 Optimization facilities

Section 8.1 gives an overview of the various types of optimization provided by the compiler and its libraries.

Section 8.2 contains suggestions on enhancing efficiency by reorganizing the program. These suggestions appear in the order of increasing programming effort required to effect improvements. This section, which also contains information on programming virtual memory, concludes with a discussion of the advantages of modular programming.

Section 8.3 explains in some details when to implement a particular operation by the insertion of an inline code or through library calls. This information often helps to avoid the additional effort of a library call.

Section 8.4 explains the various ways of optimization, carried out by the compiler when the control statement "COMOPT OPTIMIZE =" is requested. In addition the user is instructed how to proceed in order to get the full benefit from this optimization. This should be of particular interest to scientific programmers.

Section 8.5 informs the user how to control the optimization performed by the compiler.

Finally section 8.6 contains some advice on programming which may help the beginner.

8.1 Overview

8.1.1 Compiler

The main purpose of optimization is the generation of object programs whose runtimes are kept to a minimum and whose storage space requirements are as low as possible. This implies not only the generation of the most suitable code for PL/I statements, but also that, wherever appropriate, the sequence of statements is altered in such a way as to improve efficiency without affecting the result.

The following types of optimization are carried out by the compiler:

- Elimination of common expressions
- Transfer of invariant expressions out of DO loops
- Reduction of linear expressions in DO loops
- Elimination of common expressions and transfer of invariant expressions in connection with reducible functions
- Simplification of expressions
- In-line code for conversions
- In-line code for string processing
- In-line code for most of the built-in functions
- Special code for array and structure assignments
- Register and address optimization. This implies keeping values in registers for as long as possible and generating an efficient address arithmetic on the basis of an analysis of the program execution and the reference count.
- Elimination of common constants and common data descriptions to obtain efficient utilization of storage capacity.
- Special code for invoking internal procedures and for returning from these procedures.

Some of these types of optimization are performed even when they have not been requested by the control statement "COMOPT OPTIMIZE =", but others will only be carried out on request.

8.1.2 Runtime system

The PLI1 runtime system consists of a large number of modules which have been designed according to logical considerations. They are combined into two ready linked modules, which can be used jointly by all PL/I programs.

When establishing the linkage one has the option between using the ready linked modules or individual modules.

In the former case dynamic loading of the ready linked modules does not occur until runtime. The System Administrator may declare them shareable. In this way they will be shared by all PL/I programs, thus saving memory space when several PL/I programs are simultaneously executed. It saves time at the start of the program, too.

In the latter case the compiler determines which of the individual modules will be used by selecting a minimum subset of modules to be linked with the generated object module.

8.2 Manual optimization

Owing to the modularity of the PL/I libraries and the extensive optimization performed by the compiler, the efficiency of many user programs will be satisfactory and no special tuning of programs will be required.

Measures for improving the efficiency of programs, other than those referred to above, are described in this section. The measures mentioned in section 8.2.1 require less effort than those given in section 8.2.2.

It is assumed that the problems regarding the system (e.g. the organization of the PL/I libraries) have been solved and that the reader is familiar with the control statements for compiling (COMOPT) and execution (RUNOPT).

8.2.1 Running a program - stage 1

Remove all debugging aids from the program. It is obvious that in some cases the use of debugging aids causes additional overhead, because of their tendency to produce large quantities of output. Although some debugging aids, such as the conditions SUBSCRIPTRANGE and STRINGRANGE, produce output only when an error has actually occurred, they require considerably more time and memory for testing.

PUT DATA statements should also be removed from the program, particularly those for which no data list is specified. These statements require information concerning variables and conversion modules of the library. This takes up additional memory space.

The debugging aid "COMOPT DEBUG = STMT" does not increase runtime but adds approximately 8 bytes to the memory space required for each statement.

8.2.2 Tuning a program - stage 2

In PL/I there are often several ways of solving a problem. Generally one of them will be superior to all others. Which one this is, depends on the method of the implementation of the language. The difference may amount to one or several machine instructions, but it could also be several hundred.

The second stage of tuning a program concerns those language elements which demand a great deal of work from the compiler and for which alternative language elements exist. Attention to these facts may result in considerable savings when a program has to be frequently executed.

It should however be realized that the use of a certain language element is not necessarily wrong, just because it is less efficient than another one. Here it is also necessary to consider what the program is currently asked to do and what might be expected from it in the future.

Some examples of language elements which are very inefficient are given below:

1. The use of self-defining structures (REFER) permits very compact representation of data. This can however lead to representations which fall short of optimization. In the example

   ```
   DCL 1  structures   BASED (pointer),
          2 length,
          2 before,
          2 table      DIM (x REFER (length)),
          2 after;
   ```

   access to the variable "before" needs one instruction and about 4 instructions are needed for access to the variable "after".

   The best way of dealing with a self-defining structure is to arrange all members whose lengths are known before those whose length is adjustable. Among the members of adjustable length, those accessed most frequently should be furthest in front. In the example considered here it is preferable to place the variable "after" ahead of the variable "table".

2. Unnecessary structuring of programs should be avoided. Procedures, BEGIN blocks and ON units all need additional time and additional memory space for calls, management and return.

   In cases where a DO group (DO;...;END;) is satisfactory, it is preferable to using a BEGIN block (BEGIN;...;END;).

   These recommendations should be considered together with suggestions on modular programming later on in this section.

3. The following measures apply to conversions, which should be reduced to a minimum as a matter of principle.

   a) Information concerning conversions to be performed in-line is given in section 8.3.

   b) Conversions may be avoided by using additional variables. While in the example

```
DCL string CHAR(8);
string = string + 1;
```

   two conversions, with one of them necessitating a library call, are needed, the equivalent example

```
DCL string  CHAR(8);
DCL counter DECIMAL FIXED;
counter = counter + 1;
string  = counter;
```

   requires only a single conversion without library call.

   c) When data have to be transferred from one structure to another, these structures should match in order to allow the data to be transferred in bulk.

   d) In arithmetic expressions the use of differing data types should be avoided. Character strings are particularly unsuitable here.

   e) Picture strings (PICTURE) are preferable to character strings (CHAR). If, for instance, an input item is to consist of three decimal characters and neither ONSOURCE nor ONCHAR is used, the program section

```
DCL  string  CHAR(3),
     number   FIXED DECIMAL(5,0);
ON CONVERSION GOTO error;
number = string;
```

   is less efficient than the equivalent program section

```
DCL  string   CHAR(3),
     value    PIC '999' DEFINED string,
     number   FIXED DECIMAL(5,0);
IF   VERIFY   (string, '123456780')   = 0
     THEN GOTO error;
number = value;
```

   f) Internal switches, counters and variables, used for subscripting array elements, should be declared with FIXED BINARY. Data destined for output should be DECIMAL.

   g) Declaration of the precision for variables used in expressions requires some care. Different precisions may lead to additional instructions for the formation of intermediate values.

4. The following measures apply to strings:

   a) Information on in-line operations is given in section 8.3.

   b) Bit strings should be declared ALIGNED, unless there are special reasons for aiming at maximum storage density.

      Where close packing of memory is necessary and an aggregate containing only bit strings with the attribute UNALIGNED and having the attribute BASED or PARAMETER, care is advisable. Wherever possible one should make sure that such aggregates begin at a byte boundary by declaring at least one of their members with the attribute ALIGNED. Otherwise the compiler assumes that the aggregate may begin at an arbitrary bit boundary. This requires additional instructions.

      Thus, instead of the declaration

```
DCL 1   structure  BASED,
        2 type     BIT(1),
        2 filler   BIT(7),
        2 bit      BIT(1);
```

      it would be better to write

```
DCL 1   structure  BASED,
        2 type     BIT(1) ALIGNED,
        2 bit      BIT(1);
```

   c) Note that concatenations of bit strings are time consuming. It is better to use the pseudo variable SUBSTR.

   d) Strings having the attribute NONVARYING are disadvantageous if their length is not known at compile time, e.g.

```
DCL      string  CHAR(length).
```

   e) The built-in function DATE is time consuming. It should only be used once in a program.

5. The following measures apply to input/output:

   a) Large block sizes (BLKSIZE) are less time consuming than small ones.

   b) The SPACE option in the FILE command should preferably be specified to be as large as the anticipated size of the file in order to avoid subsequent requests for additional space.

   c) In stream input/output a long data list is preferable to several short ones.

d) Input and output of character strings can be simplified by the use of overlay. In the example

```
DCL 1   input,
        2 type     CHAR(2),
        2 record,
          3 field 1  CHAR(5),
          3 field 2  CHAR(7),
          3 field 3  CHAR(66);
GET EDIT (input) (A(2), A(5), A(7), A(66));
```

4 fields are being processed for each input. It is better to read in the whole structure as one field and to overlay the structure on this field:

```
DCL  field CHAR(80)  DEFINED (input);
GET  EDIT(field) (A(80));
```

A READ statement would be even better.

8.2.3 Tuning a program for virtual storage

The output of the PLI1 compiler is well adapted to virtual storage requirements. Executive code and constants are write protected and separated from the data. Generally, it is hardly advisable to tune a program simply to reduce the amount of paging. Where this is essential, various courses of action are available. Their effect is, however, usually marginal.

The purpose of tailoring a program to a virtual storage system is the reduction of paging, which means that the number of data transfers from the paging device into main storage and vice versa is minimized. This can be realized by storing contiguously fields that are addressed concurrently and by creating as few pages as possible that may be changed.

This can be realized by writing the source program so as to enable the compiler to generate from it the most suitable adaptation to a virtual storage system. The most effective contribution is obtainable from the control options, available when linking the compiled modules, by assigning definite modules to certain pages.

Further tailoring to a virtual storage system can be obtained in the design and programming stages of modular programs.

For the compiler to produce optimum results, careful consideration in writing the source program and in declaring the data is required.

In data declarations particular attention should be paid to aggregates taking up substantially more than one page. Items within one aggregate that are accessed together should be placed together. In this case the choice between an array whose elements are structures and a structure whose elements are arrays, may also be of importance.

In the example

```
DCL 1 structure   DIMENSION (3000),
      2 name      CHAR (20),
      2 number    FIXED BINARY;
```

name and number are contiguously stored for each array element and can therefore be readily accessed together. In the equivalent declaration

```
DCL 1 structure,
      2 name     CHAR (20)    DIMENSION (3000),
      2 number   FIXED BINARY DIMENSION (3000);
```

the names are stored contiguously, while the name and the number belonging to it are a large distance apart.

The choice between storage classes STATIC INTERNAL and AUTOMATIC has little effect on paging. This does not apply to storage classes CONTROLLED and BASED.

Complete control over the positioning of memory locations for variables can be achieved by using BASED variables and placing them contiguously in one area (AREA). All variables within the area will be held in contiguous storage.

Further improvement is possible by reducing the number of non-write protected modules. For this purpose the following facts must be known:

Variables which the compiler recognizes as not requiring alterations are considered as constants and are stored in a write protected module. These are

- Variables with attribute STATIC (CONSTANT) INTERNAL INITIAL (option)
- Variables with attribute STATIC INTERNAL INITIAL (option), satisfying the following conditions:
  - not target variable in an assignment
  - not argument in a call
  - not target variable for input
  - not argument for the call of a built-in function

The compiler generates for each external procedure the following modules:

- A write protected module with CSECT, containing the executable statements, all constants, and the variables which are considered as constants (see above).
- An optional non-write protected module for
  - STATIC variables, CONTROLLED variables, file constants, etc.

These modules are the units which are further processed by the linkage editor. Normally the modules are stored in the random sequence in which they occur, which may lead to unnecessary wastage of and increased demand for memory space.

The linkage editor may however be directed by means of control statements to place certain modules contiguously or to put a particular module at the beginning of the page. For further details the reference manual of the linkage editor should be consulted.

When a program has been suitably split up into modules, it is then possible to analyze the use of the modules and arrange them in such a way that efficient paging is obtained. It must however be pointed out that this is a difficult and time consuming job.

8.2.4 Modular programming

Although it is possible to write a program consisting of only one external procedure, it is more sensible to divide a program into modules. In PL/I the basic units of modularity are the procedure and the BEGIN block.

The benefits of modular programming and the point of view taken in structural programming are well known, but there are also considerations relating to the efficiency of programs.

Some of the general advantages of modular programming are listed below:

- The time and space required for the compilation depends on the size of the program. Generally, the compilation time will increase more than linearly with program size. Moreover changes in small source procedures require less recompilation time than changes in large ones.
- A procedure dedicated to a simple function needs only those data which are required for this function. Due to the properties of the AUTOMATIC variables, the risk that this will destroy data for other functions is less high.
- If a procedure is meant to perform only a single function, it is much simpler to replace this function by a different version. It may well be that such a function is also suitable for other applications.
- Allocation of memory space for all AUTOMATIC variables of a procedure takes place when the procedure is invoked at any of its entry points. Reducing the number of functions performed by a procedure often enables the number of variables declared in the procedure to be reduced. This in turn may reduce the total storage space requirement for AUTOMATIC variables.

With regard to programming efficiency, even higher priority attaches to the following considerations:

1. If the static CSECT and the activation block exceed 4096 bytes, the compiler has to insert additional code in order to reference the remote memory.
2. When the code generated for a procedure exceeds 4096 bytes, the base register has to be converted more frequently.

While it is true that extra invocation of procedures prolong execution time, modular programming can compensate for this by having to execute considerably fewer instructions.

8.3 In-line operations

Many operations are handled in-line. The user will therefore find it worthwhile to ascertain which of the operations are performed in-line and which of them require a library call, and to modify his program such that the least possible number of library calls is used. Most in-line operations deal with conversions and string handling.

8.3.1 Data conversion

The data conversions which are performed in-line are listed in Figure 8-1. Any conversion outside the specified area or not fulfilling the specified conditions will be performed as a library call.

Arithmetic picture strings will be converted in-line if the picture specification does not contain any characters other than the following:

V and 9
non-drifting characters + - S $
drifting characters +... -... S... $...
suppression characters * Z
insertion characters , . / B

As far as in-line conversion is concerned, pictures which do not contain any characters other than those given above may be divided into three groups.

Type 1: Pictures containing 9 only with one optional V and one leading or trailing sign.
Examples: ’99V99’, ’99’ ’S99V9’, ’99V+’, ’S999’

Type 2: Pictures with suppression characters, one sign character and one insertion character or pictures of type 1 with insertion characters.
Examples: ’ZZZ’, ’**/**9’, ’ZZZ9V.99’, ’+ZZZ.ZZ’, ’S///99’, ’9.9’

Type 3: Pictures with drifting characters, insertion characters and sign characters.
Examples: ’$$$$’, ’-,--9’, ’S/SS/S9’, ’+++9V.9’, ’$$$9-’

Reasons why, in the cases stated above, conversion may not be performed in-line are given below.

- Condition SIZE is enabled and could be raised.
- The digit positions of source and target are not overlapping. As an example DECIMAL (6,8) or DECIMAL (5,-3) will not be converted in-line to PIC ’999V99’.

- The picture contains a certain combination which is difficult to convert in-line; e.g.:
  - There is no V between drifting characters * or Z and the first 9, as for instance in ’ZZ.99’.
  - Drifting characters or suppression characters appear to the right of a decimal point, as for instance in ’ZZZV.ZZ’, ’++V++’.

The compiler issues an information message for every generated branch in the runtime system if *COMOPT DIAGNOST = INFORMATION was specified.

Conversion		Condition and comment	
Source	Target		
FIXED BINARY	FIXED BINARY		
	FIXED DECIMAL		
	FLOAT	Target: fullword or doubleword	
	BIT (n)	n ≤ 32,NONVARYING n ≤ 2088,NONVARYING ALIGNED STRINGSIZE disabled	
	CHARACTER (n)	n ≤ 256; STRINGSIZE disabled; via FIXED DECIMAL	
	PICTURE numeric	Length ≤ 256 via FIXED DECIMAL; Pict. type 1, 2 and 3; SIZE disabled	
FIXED DECIMAL (g,k)	FIXED BINARY		
	FIXED DECIMAL		
	FLOAT	p+q≤75; Target: fullword or doublew.	
	BIT (n)	Length ≤ 2088;NONVARYING ALIGNED; STRINGSIZE disabled	
	CHARACTER (n)	n ≤ 256; if g = k then integer STRINGSIZE disabled	
	PICTURE numeric	Picture type 1, 2 and 3	
FLOAT Fullword or doubleword	FIXED BINARY		
	FIXED DECIMAL	k ≤ 80;SIZE disabled	
	FLOAT	Target, source: fullword or doublew.	
	BIT (n)	n ≤ 2088;NONVARYING ALIGNED STRINGSIZE disabled	
BIT (n)	FIXED BINARY	n ≤ 32	Source: NONVARYING
	FIXED DECIMAL	n ≤ 32	Source: NONVARYING ALIGNED
	FLOAT		
	CHARACTER	n = 1	

Fig. 8-1 Implicit conversions performed in-line (part 1)

Conversion		Condition and comment
Source	Target	
CHARACTER (n)	BIT	n = 1; CONVERSION disabled
	FIXED DECIMAL	
	FLOAT	
	FIXED BINARY	
PICTURE alphanumeric	CHARACTER (n)	n ≤ 256; NONVARYING
	PICTURE alphanumeric	Pictures must be identical
PICTURE numeric picture type 1, 2 and 3 except $...	FIXED BINARY	via FIXED DECIMAL; SIZE disabled
	FIXED DECIMAL	Picture type 2 without * or insertion charact. /B SIZE disabled
	FLOAT	via FIXED DECIMAL; SIZE disabled
	PICTURE	Picture type 1, 2 or 3; SIZE disabled
LABEL	LABEL	
Locator	Locator	

Fig. 8-1 Implicit conversions performed in-line (part 2)

String operation	Condition for operands
Assignment	CHARACTER
	BIT of constant length ≤ 32
	BIT of constant length beginning at byte boundary
Boolean operations	as in assignment
Comparison	as in assignment
Concatenation	CHARACTER

Fig. 8-2 String operations performed in-line under the specified conditions

String function	Conditions
AFTER	never
BEFORE	never
BIT	always
BOOL (a, b, c)	if c constant then as in Boolean operations
CHAR	always
COLLATE	always
COPY (a, b)	if a = string fixed length≤4096 and b = constant≤4095
DECAT	never
HIGH (a)	if a = constant ≤ 4095
INDEX	CHARACTER
LENGTH	always
LOW (a)	if a = constant ≤ 4095
REPEAT (a, b)	if a = fixed length ≤ 4096 and b = constant ≤ 4095
REVERSE (a)	only in conjunction with INDEX (REVERSE (a), b) if a = CHARACTER and b = CHAR (1) or b = string constant of length ≤ 256
SEARCH	never
STRING	if fill characters are present due to VARYING or BIT ALIGNED or noncontiguous storage then as in concatenation otherwise as in assignment
SUBSTR	if STRINGRANGE disabled
TRANSLATE (a,b,c)	if a string of constant maximum length ≤ 256 and b and c fixed length ≤ 256
UNSPEC	always
VALID	never
VERIFY	CHARACTER

Fig. 8-3 Built-in STRING functions performed in-line under specified conditions

8.3.2 String handling

The functions and operations, which are performed in-line, are listed in Figs. 8-2 and 8-3. However, in certain contexts, some of these functions will be carried out as library calls. If, for instance, the expressions in the BIT or CHAR built-in functions require an implicit conversion which can not be executed in-line the appropriate library procedure will be called.

8.4 Global optimization features

8.4.1 Common expressions

The term "common expressions" is used to describe expressions like "B * C", as for instance in

```
A = B * C + D1
  .
  .
  .
D = B * C + D2
```

where the variables B and C are not altered between the occurrences of the two expressions. In this case the expression "B * C" need not be evaluated more than once.

The technique of avoiding repeated evaluation of such expressions is called "common expression elimination".

An important application of common expression elimination is found in statements containing subscripted variables in which the same subscript value is used for several variables.

Example

```
COST (ARTICLE) = QUANTITY (ARTICLE) * PRICE (ARTICLE)
```

The value of the subscript ARTICLE is computed once only. Should the computation of the subscript yield a decimal value, conversion to a binary value is carried out automatically.

8.4.1.1 Interrupt handling

The order of most operations in a PL/I statement depends on the priority of the operators involved. However, the order of evaluating the subexpressions, whose results determine the operands of the operator of lower priority, is only defined inasmuch as an operand is completely evaluated before its value is used in a further operation. Examples of such subexpressions are subscript expressions, locator qualifier expressions, or function references.

Owing to the reason given above, the sequence in which ON units, referenced in conjunction with sub-expressions, are called, may be unpredictable. Consequently evaluation of an expression may yield different values.

The result may depend on the sequence in which the ON units are called and on the statements which are being executed there. If an ON unit for a computational interrupt is called, the following applies:

1. All variables contain that value which has been assigned to them in the execution of preceding statements. These values can be used in the ON unit. Thus the statement PUT DATA can, for instance, be used to show the value a variable possesses at entry into the ON unit.
2. If, in an ON unit which has been called by setting a computational condition, a value is assigned to a variable, then this value will be the current value in each subsequent part of the program.

Whenever variables might be changed due to a computational condition interrupt, either in the corresponding ON unit or as the result of a branch from the ON unit, common expression elimination is inhibited. A relevant example is given below:

```
ON   ZERODIVIDE B,C = 1;
  .
  .
  .
X = A * B + B/C;
Y = A * B + D;
```

Although the sub-expression A * B is common to both assignments, it is not being eliminated since, should the condition ZERODIVIDE occur, the same sub-expression may have a value in the second assignment which is different to that in the first one.

The rules given above are only valid if the option ORDER is either explicitly or implicitly specified. If the restrictions mentioned above are undesirable, the option REORDER can be specified; which will provide unrestricted elimination of common expressions.

ORDER and REORDER are described in later sections.

8.4.2 Transfer of invariant expressions or statements out of DO loops

If an expression occurs in a loop and the compiler is able to recognize that each time the loop is executed the result is the same, we are dealing with an invariant expression. This applies also to statements.

An invariant expression or an invariant statement can be moved out of the loop and placed before the loop, so that instead of being evaluated each time the loop is executed it is evaluated once only before the loop is entered. This is illustrated by the following example:

```
DO  I = 1 TO N:
   .
   .
   .
J = 3;
   .
   .
   .
END;
```

The assignment J = 3 may be invariant under certain conditions and can then be moved out of the loop. In certain cases it may also be helpful to move it backward or forward within the loop.

If the removal of invariant parts is desired, the option REORDER must be explicitly or implicitly present in a block comprising the loop.

8.4.3 Reduction of linear expressions in DO loops

Multiplication of a control variable by a constant is the simplest form of a linear expression. Wherever this is feasible a multiplication is converted to an addition, which is executed more efficiently.

Example

```
DO  I = M TO N BY 2;
    .
    .
    .
    A(I) = I * 4;
    .
    .
    .
    END;
```

If in this example the variable I is not altered within the loop, a code is generated which corresponds to the following program section:

```
       I = M;
       IF I > N THEN GOTO End;
       TEMP = 4 * I;

Start: .
       .
       .
       I = I + 2:
       TEMP = TEMP + 8;
       IF I < N THEN GOTO start;
End:   .
       .
       .
```

What this example does not show is that the address computation for the subscripted reference (A) is also optimized at the same time. This is in fact the reason why this type of optimization is especially powerful.

If the type of optimization described in this section is desired, REORDER must be specified.

8.4.4 ORDER and REORDER options

The options ORDER and REORDER are used for optimization control. They can be specified in PROCEDURE and BEGIN statements and apply to the entire block. The options are inherited by all contained blocks, unless an explicit option is provided there. The default is ORDER.

8.4.4.1 ORDER options

If, in ON units called because of computational condition interrupts, a variable must always contain the last value assigned to it in the block, the option ORDER has to be specified for the procedure block or the BEGIN block, as the case may be.

Common expressions may also be eliminated from a block with the option ORDER. In this case computational interrupts may occur less frequently than if common expressions had not been eliminated. However, if a computational condition interrupt occurs in a block with the option ORDER, the value of the variables in statements preceding the point where the condition was set, is always the last which has been assigned to the variable, provided the ON unit contains a reference to that variable.

In a block with the option ORDER other types of optimization are also permitted, with the exception of the movement of expressions which can cause an interrupt. Since this can only be avoided by disabling all relevant computational conditions, the use of ORDER means in practice that move-out of invariant parts from loops is not carried out.

8.4.4.2 REORDER option

If REORDER is specified, the compiler will perform all the optimizations, which do not affect the logic of the program as written down in the source program, so long as no errors appear during the execution of the program. Move-out of invariant parts of a loop is carried out, so that they are performed once only before or after the loop.

The time taken for the execution of loops may be reduced, if the values of variables, which are very frequently changed within the loop, are held in registers. During error free program execution values may be kept in registers and the transfer of data to and from the memory is dispensed with, resulting in a considerable saving of time. If the last value of the variable is required after the loop has been executed, the value of the variable is assigned to its location in memory when leaving the loop.

Allocations to registers may be performed more efficiently if REORDER is specified. There is, however, no guarantee that variables which are altered in the block will have their last value when a computational condition occurs, since the latest value may be held in a register and not in memory. This is the reason why an ON unit, which has been called as the result of a computational interrupt, should not contain a reference to a variable which is changed in the REORDER block. The use of built-in functions ONSOURCE and ONCHAR is, however, valid in this context.

A program is errored if a computational condition or an ERROR or ATTENTION condition occurs in a block with REORDER option at runtime, thereby using a variable whose value is not ensured.

Since these restrictions preclude the correction of erroneous data, with the exception of the correction through ONSOURCE and ONCHAR in an ON unit for the condition CONVERSION, the user has to resort to the system unit for the condition, i.e. stopping the execution of the program or using the ON unit for correcting the errors and restarting the relevant section of the program with new data.

An example of this is given below:

```
ON OVERFLOW  PUT DATA;
DO J = 1  TO M;
   DO I = 1 TO N;
      X(I,J) = Y(I) + Z(J) * L + SQRT(W);
      P = I * J;
      END;
   END;
```

If the above sequence of instructions occurs in a block with the option REORDER, the compiled program section corresponds to the following statements:

```
ON OVERFLOW  PUT DATA;
Temp1 =SQRT(W);
DO J = 1 TO M;
   Temp2  =J;
   DO I = 1 TO N;
      X(I,J) = Y(I) + Z(J) * L + Temp1;
      P = Temp2;
      Temp2 =Temp2 + J;
      END;
   END;
```

Temp1 and Temp2 are temporary variables taking the value of expressions which have been moved forwards out of the loop. The multiplication I * J is reduced to the addition (TEMP2 + J). The assignment P = I * J could be replaced by P = N * M outside the loop.

Let us, for instance, assume that the condition OVERFLOW occurs. In this case it can not be guaranteed that the variables will have their current value, since they may be held in registers rather than in the memory location of the variables which can be referenced in the ON unit.

Although the example given above does not show it, the subscript calculations for X(I,J), Y(I) and Z(I) are also optimized (I,J,P).

8.4.5 Elimination of side effects / reducible functions

A function is reducible if its call causes no side effects and if the value it returns depends on the values of the arguments alone.

Every change of a variable known outside the function in which it has been changed, as well as every I/O process, constitutes a side effect. A function is regarded as reducible by the compiler if, and only if, it is declared with the attribute REDUCIBLE. When handling calls to reducible functions common expressions are eliminated and invariant expressions transferred as explained above. Quite independent of this it is useful to declare reducible functions REDUCIBLE since the compiler is capable of better optimization if side effects are excluded.

8.4.6 Optimization of Boolean expressions

Complex Boolean expressions contained in a condition for a branch will be resolved into a number of branches with simple Boolean expressions. Of these Boolean expressions only as many as are necessary for uniquely determining the result will be evaluated. The rest are ignored. Irreducible functions will however be called in every case. Thus, the statement

```
IF A < B | C > D THEN GOTO L;
```

is handled as if the following statements had been written

```
IF A < B THEN GOTO L;
IF C > D THEN GOTO L;
```

The code for the statement

```
IF A < B & C > D THEN X = Y;
```

corresponds to the statements

```
   IF A ≥ B THEN GOTO L;
   IF C ≤ D THEN GOTO L;
   X = Y;
L: ...
```

8.4.7 Expression simplification

To simplify expressions means to modify them without changing the expected effect in order to generate a more effective mode. One major simplification concerns expressions containing arithmetic constants.

Constant expressions of the form $C_1 + C_2$, $C_1 - C_2$, $C_1 * C_2$, where C_1 and C_2 are integer constants, will be replaced by equivalent constants. As an example the expression 2 + 5 is replaced by the constant 7.

The compiler attempts to extract the constant parts from expressions arising in addressing of array elements and elementary members of structures.

Expressions of the form

C_1 * (exp + C_2), C_1 * (exp - C_2), C_1 * (C_2 + exp)

are transformed into equivalent expressions of the form

C_3 * exp + C_4

where C_1, C_2, C_3, C_4 are constant integers and exp is an integer expression.

In this way the following is achieved:

```
DCL A DIMENSION (20,20);
A (C1 * I + C2, C3 * K + C4)
```

will be addressed just as efficiently as A (I,K).

8.4.8 Initialization of aggregates

This type of optimization applies to aggregates which are declared AUTOMATIC, BASED or CONTROLLED.

If all the elements of an array are initialized to have the same value, a code is generated which initializes the first element and performs the remaining initialization by a single transfer instruction, or in the case of variable bounds, by means of a special subroutine.

Example

```
DCL A DIMENSION (20,20) FIXED BINARY INIT ((400)0);
```

Array A is initialized as described. This type of optimization is not applied to arrays whose elements are declared CHAR VARYING or BIT VARYING.

This type of optimization is also used when an element value is assigned to an array, if the array has connected storage.

If the aggregate to be initialized, whether it be an array or a structure, consists of fixed length elements with constant initial values, an initialization constant is put into memory for the whole aggregate, which can then be initialized by a single transfer instruction. This type of optimization is inapplicable when the storage space required for the initialization constant exceeds that for the general initialization code.

Example

```
DCL A DIMENSION (3) BINARY FIXED INIT (1,2,3);
```

Array A is initialized in this way.

8.4.9 Special code for aggregate assignment

Wherever feasible the compiler will implement assignments to arrays and structures by means of a single transfer of data. This is the case when source aggregate and target aggregate have data descriptions which, apart from storage classes, are identical and have connected storage.

8.4.10 Utilization of registers in DO statements

As far as is practicable, the relevant values for the formation of a DO loop are held in registers while the loop is being executed. For example in handling the statement.

DO control variable = initial value BY increment TO final value;

the compiler will attempt to hold the values for control variable, increment, and final value in registers. The optimization thus achieved is extremely effective.

8.4.11 Internal procedure calls

The compiler tries to recognize those internal procedures which can only be called from their encompassing block, cannot be used recursively, and have an automatic storage area of fixed size.

For such procedures code strings for call and return can be generated which are shorter than in the general case.

The same applies to BEGIN blocks whose automatic storage area is of fixed size.

8.4.12 Utilization of global optimization

This section contains particulars regarding the dos and don’ts of coding practices which should be borne in mind in order to get the full benefit of global optimization which the control statement * COMOPT OPTIMIZE = TIME is capable of providing.

8.4.12.1 Common expression elimination

Common expression elimination is inhibited in the following cases:

1. In expressions containing variables whose values are modified either in an I/O condition or in a computational condition.
2. If a BASED variable is overlaid on a variable used in common expressions and a new value is assigned to the BASED variable between the equivalent expressions, optimization is inhibited.

   Thus, in the following example of a program section, the equivalent expression X + Z is not eliminated because the BASED variable A is overlaid on the variable X (P = ADDR(X)) and a value is assigned to the variable A between the two equivalent expressions.

```
DCL A BASED (P);
P = ADDR (X);
  .
  .
  .
P = ADDR (Y);
  .
  .
  .
B = X + Z;
P -> A = 2
C = X + Z;
```

3. When using aliased variables. An aliased variable is a variable whose value may be changed by reference to an identifier other than its own identifier. Examples of such variables are DEFINED variables and their associated base variable, arguments, parameters and BASED variables, and the variables overlaid by them.

   Variables whose addresses are known to an external procedure via pointers, and which are used either as external variables or as arguments, are also regarded as aliased variables. An aliased variable, while not completely preventing the elimination of common (equivalent) expressions, imposes restrictions on it.

   If a common expression contains an aliased variable, the flow paths, in which common expressions may possibly occur, are searched for assignments in which either this variable itself or one of its aliased variables is used as the target variable.

   If a program contains an external pointer variable, it is assumed that this pointer can be set to all variables whose addresses are known to external procedures. This means that all variables which are addressed by external pointers, or by other pointers to which the value of an external pointer has been allocated, may refer to external variables.

4. When the form of an expression is modified. If the partial expression B + C is treated as a common expression, the compiler would be incapable of recognizing it as a common expression in the following statement:

   ```
   D = A + B + C;
   ```

   Because the compiler processes this expression from left to right, it recognizes the expressions A + B and (A + B) + C. If, however, the call is coded as D = A + (B + C), the user may take it for granted that B + C will be regarded as a common expression, since the compiler is bound to tackle the expression of highest priority first.

5. Dependence on the scope of common expressions. In order to recognize common expressions, the program is analyzed and flow units are determined. A flow unit is a section of program which can only be entered at the beginning and from which exit is always made at the end. One flow unit may contain several PL/I statements and, conversely, one PL/I statement may contain several flow units.

   Common expressions will be recognized across several flow units. However, if processing flow paths between the flow units becomes complex, recognition of common expressions beyond the bounds of flow units will be inhibited.

Common expression elimination will be facilitated by noting the following points:

1. Variables in expressions should neither be external nor be associated with external pointers and should not be used as parameters of the built-in function ADDR.
2. External procedures, external label variables and label constants which are known in external procedures should not be used in the source program.
3. Variables in expressions should neither be altered nor accessed in ON units.
4. Specify REORDER for the block.
5. Declare reducible functions with REDUCIBLE.

8.4.12.2 Transfer of invariant expressions

Transfer of invariant expressions out of loops is prevented by:

1. ORDER specification for the block. However, transfer is not prevented by the ORDER option in every case, but only for operations which are capable of setting a computational condition.
2. Use of variables whose value is set by or used in input/output statements.
3. Use of variables which can be set in ON units for I/O conditions or computational conditions, or which are aliased variables.
4. Complex program flow involving external procedures, external variables, or label constants.

Transfer of invariant expressions out of loops is facilitated by:

1. Specification of REORDER for the block.
2. Avoidance of points 2 to 4 given above.
3. Declaring reducible functions REDUCIBLE.

8.4.12.3 Reduction of linear expressions in loops

This type of optimization is only permitted if the control variable is not changed inside a loop. The same conditions as those described above for the transfer of invariant expressions apply.

8.4.12.4 Register and address optimization

The conditions for this type of optimization are the same as those described above for common expressions elimination.

8.4.12.5 Use of registers in DO statements

Here the conditions for reducing linear expressions in loops dealt with earlier also apply. In addition the following conditions must be fulfilled:

- Referencing of the control variable in an ON unit or in a procedure is not permitted during execution of the loop.
- Exit from loops is forbidden.

8.5 Optimization control (OPTIMIZE)

The control statement

```
*COMOPT OPTIMIZE = specification
```

can be used to govern the way in which the compiler performs the optimization. The specifications are explained in the following subsections. These specifications may also be supplied for PROCEDURE OPTIONS (option).

8.5.1 Time optimization (TIME)

All the optimizations described in section 8.1.1 will be carried out.

8.5.2 Change enabling of conditions (ENABLING)

This specification causes presettings for all computational conditions, with the exception of OVERFLOW, UNDERFLOW and ZERODIVIDE, to be "disabled". Thereafter only conditions which do not require the compiler to generate special testing instructions will remain preset to "enabled", since the testing will be performed by the machine instructions themselves.

8.5.3 Sequence of statements modifiable (REORDER)

This specification causes the system default to change over from ORDER to REORDER.

It is advisable to use OPTIMIZE = (TIME, ENABLING, REORDER) for all production runs and, in the source procedure where it is necessary, to set the corresponding condition prefix before the statement and to specify ORDER explicitly.

8.5.4 Overlapping (OVERLAP)

When this option is specified, the compiler assumes that target and source area of an assignment neither overlap nor are identical unless it can definitely recognize an overlap, since movements between non-overlapping data are faster. In this case the compiler issues a warning. In the absence of this specification, the compiler decides in doubtful cases for "overlap" and generates a safe code. This causes an increase in runtime.

This specification has no bearing on the case in which there is an expression on the right hand side, except when this is SUBSTR, UNSPEC or STRING with a variable reference.

8.6 Programming notes

This section deals with some of the mistakes and pitfalls commonly encountered in writing a program. They are caused by misinterpretation or oversight of rules or by lack of attention to conventions and constraints on implementation.

8.6.1 Source program and general syntax

1. When a source program has been written manually, entry of the source text by another person may result in spelling mistakes if the following characters are not clearly represented.

```
1 (numeral), I (letter), | (vertical line)
/ (slash), ' (apostrophe)
7 (numeral), > (greater than)
L (letter), < (less than)
O (letter), 0 (numeral)
S (letter), 5 (numeral)
Z (letter), 2 (numeral)
_ (underline), - (minus sign)
```

2. Care should be taken to make sure that only the area specified by

```
COMOPT MARGINS = TEXT (a, b)
```

   is used for writing source lines. Default is TEXT (2,72).

3. Omission of certain characters may lead to errors which are often hard to detect. Some of these are:

   – The closing apostrophe of a literal is missing.

   – The closing parenthesis is missing or there are more opening parentheses than closing ones.

   – The delimiters */ of a comment have been omitted or have been written wrongly as /*.

   Literals and comments extending over more than one line are highlighted in the compiler listing by an asterisk after the line number.

4. Reserved keywords in the 48-character set (e.g. GT, CAT) must always be preceded and followed by a blank or comment.

   Caution

   Inadvertant misuse of these keywords may result in error messages which are hard to understand, e.g. "DCL NL" is interpreted as syntax error.

5. Care should be taken to ensure that there are enough END statements for the termination of a group, a procedure and a BEGIN block. This is particularly important when the form "END label;" is selected. The relevant rules are such that the compiler is not always in a position to recognize if one END statement is not enough.
6. In some contexts it may not at all be obvious that a specification has to be enclosed between parentheses. In particular, expressions after the keywords WHILE and RETURN are cases in point.

8.6.2 Program control

The main procedure, i.e. the procedure at which the program is to be started, must include the following entry:

```
PROCEDURE OPTIONS (MAIN)
```

If several external procedures have specification MAIN, the linkage editor will use the procedure it reaches first.

8.6.3 Declarations and attributes

1. The memory location for variables with attribute AUTOMATIC is allocated when entry is made into the block. If this involves calculations in which variables are used, their values must have already been determined before entry to the block.

   Example

   ```
   Name:  PROCEDURE;
          length = 4;
          DCL string CHAR (length);
   ```

   This example leads to an error because allocation of memory space precedes execution of the assignment.

2. Missing commas in DECLARE statements are a source of errors. For example, in a structure declaration each member of the structure must be terminated by a comma.

3. The length of external identifiers (EXTERNAL) should not exceed 7 characters. Otherwise they will be truncated to the first four and the last three characters.

4. In a declaration with attribute PICTURE the picture character V separates the integer from the fraction. It does not occupy a character position, nor does it produce output of the separator character period or comma. This can only be done by means of the picture character period (.) or comma (,). However, these two picture characters are pure insertion characters and do not give any indication as to which digits belong to the integer and fractional part, respectively.

   Example

   ```
   DCL  A  PIC '99,9',
        B  PIC '99V9',
        C  PIC '99,V9';
   A,B,C = 45.6;
   PUT LIST (A,B,C);
   ```

   On the basis of the example given above the following numbers are issued:

   ```
   04,5      456    45,6
   ```

   If these values are read in again by GET LIST (A,B,C), variables A, B and C will become

   ```
   Character string  '04,5'    '560'    '45,6'
   Decimal value        45       56      45.6
   ```

If the values of the variables are output once more through PUT LIST (A,B,C), one obtains

```
04,5        560      45,6
```

5. Separate declarations for the same identifier with attribute EXTERNAL must yield the same set of attributes, after supplementation by default, as otherwise errors may arise. Any conflict cannot be detected by the compiler.

   An INITIAL attribute need only be specified once. If there are several specifications of it, then all of them must yield the same value.

6. Within its scope an identifier can be used for only one single purpose. The following example is in error, since identifier X has more than one meaning.

```
   PUT  FILE (X) LIST (A)   X is file
   X = Y + Z;               X is variable
X: M = N;                   X is label
```

7. When constants are being transferred to external procedures as parameters, attention must be paid to precision.

   Example

```
           DCL  procedure ENTRY EXTERNAL;
           CALL procedure (6);     Precision (1,0)
Procedure: PROCEDURE (x);
           DCL x FIXED DECIMAL;    Precision (5,0)
```

   Here argument and parameter do not have the same precision.

8. Depending on certain conditions, a parameter is either overlaid on the argument (transfer by reference) or it is given its own auxiliary memory location, to which the value of the argument is assigned (transfer by value). In the latter case a change in the parameter will no longer change the argument. Return of a value is not possible in this case.

   Example

```
     DCL  A    FIXED BIN,
          B    FLOAT;
     CALL P    (A,B);
P:   PROCEDURE  (X,Y);
     DCL (X,Y) FIXED BIN;
     X = 3;                /* A is changed */
     Y = 4;                /* B is not changed */
```

9. If in the declaration for an identifier not all attributes have been specified, those which have been omitted will be supplied by default. Particular attention should here be paid to the following rules.

   REAL FLOAT DECIMAL (6) is assumed for the arithmetic variable, provided the identifier does not begin with an alphabetic character between I and N or is not compiled according to PL/I standard (ISO). In these cases FIXED BINARY (15,0) will be provided.

   If one of the attributes determining the data type is specified, a missing attribute will be taken from the list REAL/FLOAT/DECIMAL and, if compiling according to PL/I standard, from the list REAL/FIXED/ BINARY.

   This is illustrated by the following example (NOISO):

```
DCL I;            completed:  REAL FIXED BINARY  (15,0)
DCL J REAL;       completed:  REAL FLOAT DECIMAL (6)
DCL K STATIC;     completed:  REAL FIXED BINARY  (15,0)
DCL L FIXED;      completed:  REAL FIXED DECIMAL ( 5,0)
```

10. The precision of a complex expression is not readily apparent. It follows the rules for expression evaluation. For example, for the expression

```
1 * 2I
```

    the attributes are COMPLEX DECIMAL PRECISION (2,0).

11. In a procedure having several entry points with different parameters, it is necessary to make sure that only those parameters are referenced which are associated with the current entry point.

    Example

```
A:  PROCEDURE (P,Q);
    P = Q + 8; RETURN;
B:  ENTRY (R,S);
    R = P + S;
    END;
```

    In the assignment R = P + S the reference to P is wrong, since the value of P is not defined when entering B.

8.6.4 Assignment and initialization

1. When a variable is accessed, it is assumed that its value is consistent with the attributes of the variable. If this is not the case, either the program may go ahead with the wrong value or else one of the conditions is set. Such an error in the value may be caused by the fact that up to this point no value has been assigned to the variable, so that its value is undefined. Alternatively an error in the value of the variable may have been assigned to it e.g. through one of the following operations:

   a) by the built-in function UNSPEC

   b) by input of a record (RECORD)

   c) by overlaying a PICTURE on a CHARACTER string with an assignment to the character string and then accessing the picture

   d) transfer of parameter to another external procedure when the attributes of argument and parameter do not match

   e) assignment of a value to a based variable and access to this value via a based variable with different attributes.

   If a variable does not have a value assigned to it, then its value is undefined and so is the further execution of the program. It is wrong to assume that in this case the variable has the value 0.

   When the value of a subscript has not been defined the omission can be detected by enabling the condition SUBSCRIPTRANGE, provided the subscript does not happen to be consistent with the declared range.

2. An attempt to output a variable whose value is undefined may lead to an interrupt. In the example

```
DCL A   DIM(10)  FIXED DECIMAL;
A(1) = 13;
PUT LIST (A);
```

   leads to an error, which can be avoided by presetting the array to value 0, for instance, by writing

```
A = 0;
```

3. Note the difference between the assignment symbol = and the comparison operator =.

   The statement

   ```
   A = B = C;
   ```

   signifies that variables B and C are tested for equality and that the result ('0'B or '1'B) is assigned to variable A.

4. If, in the initialization or assignment of a string variable of fixed length, a string is allocated which is shorter than the target, blanks will be added to the right in the case of CHAR, and bits with the value '0'B will be added to right in the case of BIT. In the example

   ```
   DCL A   CHAR(6),
       B   CHAR(3)  INIT ('CR');
   A = B;
   ```

   the variables will contain the following values after assignment:

   ```
   A = 'CR____' and B = 'CR_'
   ```

5. If the condition SIZE is disabled and such a condition occurs, the result is unpredictable.

   FIXED DECIMAL: Digits may be suppressed without causing an interrupt. If the target precision is even, a byte may be inserted in the most significant position.

8.6.5 Arithmetic expressions, Boolean expressions and conversions

1. The rules for expression evaluation, particularly the sequence of operations, should be strictly observed. Typical errors which may be encountered here are shown

```
X > Y | Z       corresponds to         (X > Y) | Z
                does not correspond to X > Y | X > Z

X > Y > Z       corresponds to         (X > Y) > Z
                does not correspond to X > Y & Y > Z
```

   All operations of equal priority are performed from left to right with the exception of **, prefix operator +, prefix operator - and, which are performed from right to left.

   The example

```
A = B ** - C ** D;
```

   is equivalent to

```
A=B**(-(C**D))
```

   The purpose of parentheses is to modify the stipulated rules. Addition of redundant parentheses may, however, give added reassurance and enhance legibility.

2. Conversion is covered by comprehensive rules, which should be carefully studied in order to obviate unnecessary frustration. The following example illustrates this point:

   a) DECIMAL FIXED to BINARY FIXED may lead to unexpected results when fractions are involved:

```
DCL  I FIXED BIN(31,5) INIT(1);
I = I + 0.1;
```

   Accordingly, the value of I is now 1.0625, because 0.1 is converted to FIXED BINARY (5,4), thus taking the binary value 0.0001B (without significance), which gives the decimal value 0.0625. This rounding error can be reduced by specifying higher precision (e.g. 0.1000) for the constant.

   b) In arithmetic operations involving character strings, intermediate values are held in dummy variables having the attributes FIXED DECIMAL (15,0) i.e. without fractional positions. In the example

```
DCL  A  CHAR(6) INIT('123.45'),
     B  FIXED(5,2);

B = A;        /* value of B 123.45 */
B = A + A;    /* value of B 246.00 */
```

   this suppression of fractional positions thus occurs in the second statement.

c) The rules for the conversion of an arithmetic value to BIT affect the assignment to a bit variable from a decimal constant.

Example

```
DCL  A  BIT (1),
     D  BIT (5);

A = 1;          /* value of A:  '0'B     */
D = 1;          /* value of D:  '00010'B */
D = '1'B;       /* value of D:  '10000'B */
```

d) The rules for conversion may sometimes lead to unexpected results when an arithmetic value is assigned to a character string (CHAR).

Example 1

```
DCL  A  CHAR(4),
     B  CHAR(7);

A = '0';          /* value of A: '0___'    */
A = 0;            /* value of A: '___0'    */
B = 1234567;      /* value of B: '___1234' */
A = -0.7;         /* value of A: '-0.7'    */
```

The three character positions in the second and third assignment will only be filled with characters if there is a negative prefix operator, a decimal point and a single zero before the decimal point (see fourth assignment).

Example 2

```
DCL    Number CHAR(8) INIT('0');
DO I = 1 TO 100;
   Number = Number + 1;
   END;
```

The example given above shows how a conversion error arises as the following operations are executed during assignment:

– The initial value '0_____' is converted to FIXED DECIMAL (15,0) and yields value 0.

– The decimal constant 1 has the attributes FIXED DECIMAL (1,0).

– During addition, a dummy variable with attribute FIXED DECIMAL (15,0) having value 1 is formed.

– This value is converted to CHAR(18); the resulting character string consists of 17 blanks and the digit 1.

– As the target value has the CHAR(8) attribute, only the first 8 characters are assigned. These are 8 blanks, which when converted in the next pass (see above) lead to a conversion error.

e) In divisions involving FIXED items, unexpected truncation of significant digits or the condition FIXEDOVERFLOW may occur.

Example

```
25 + 1/3
```

This expression will be evaluated as follows:

```
 Operand            Precision            Result

   1                 (1,0)                 1
   3                 (1,0)                 3
  1/3               (15,14)                0.333...
  25                 (2,0)                25
 25 + 1/3           (15,14)               2 5.333...
```

Because of the precision, computed according to the rules, the digit 2 is truncated, but if the condition FIXEDOVERFLOW is enabled, it will be retained.

Here it is necessary to enlarge the integer part of the result of the division. This may, for instance, be achieved by

```
25 + 01/3
25 + PREC (1/3, 15,13)
25 + DIVIDE (1, 3, 15, 13)
```

or by assigning the result of the division to a separate variable which has been declared with the desired precision.

f) The value of a picture string (PICTURE) is only checked for accuracy during assignment.

Example

```
DCL A  PIC '999999',
    B  CHAR(6)      DEFINED A,
    C  CHAR(6);

B = 'ABCDEF';          /* value also in A  */
C = A;                 /* no CONVERSION    */
A = C;                 /* CONVERSION       */

A = 123456;            /* value of A: 123456    */
                       /* value of B: '123456'  */
C = 123456;            /* value of C: '___ 123' */
C = A;                 /* value of C: '123456'  */
```

g) An item with the attributes FIXED DECIMAL PRECISION (g,k) has an internal representation with precision g + 1, if g is even. In order to be certain that the precision g is not exceeded in this case, the condition SIZE can be enabled.

Example

```
DCL  (A,B,C) FIXED DECIMAL PRECISION (6,0) INIT (500000);

(SIZE): A = B + C;
```

The SIZE condition occurs. The value of A is, however, correct (1 000 000), since DECIMAL FIXED (6,0) has the same internal representation as DECIMAL FIXED (7,0).

8.6.6 DO groups

1. If a condition prefix is specified before a DO group, it applies to the DO statement only and not to the whole of the group.
2. If the condition for leaving the DO group is already satisfied at the first entry to the group, the group will not be executed.

   Example

   ```
   I = 6;
   DO J = I TO 4;
      X = X + J;
      END;
   ```

   In this example the loop will not be executed, because the control variable already has the value 6 at the first entry into the group, thus exceeding the final value 4.
3. Expressions in a DO statement are put into a dummy variable. Its representation follows from the rules for the evaluation of the expression. Comparison with control variables will possibly necessitate further conversions.

   Example

   ```
   DCL A  DECIMAL FIXED (5,0);
   A = 10;
   DO I = 1 TO A/2;
      PUT LIST (I);
      END;
   ```

   In the above example there is an error in the loop, arising from the following operations.

   ```
   Expression          Attribute           Value

    A                  DEC  (5,0)           10
    A/2                DEC (15,10)           5
    A/2 converted      BIN (31,34)         SIZE condition
   ```

   For the comparison with the control variable I the value A/2 must be converted to BINARY. This raises the condition SIZE. If this condition is disabled, the program will continue with an undefined value.

   The loop will be executed 5 times if it is modified as follows:

   ```
        dummy = A/2
        DO I  = 1 TO dummy;
   or
        DO I = 1 TO PREC (A/2,5,0);
   ```

4. A group cannot be used as ON unit. If more than one statement has to be specified for an ON unit, a BEGIN block must be used. A DO group may be included in this block.

5. In the DO statement

```
DO x = a BY b TO c
```

   expressions a, b and c are only evaluated at entry into the group and the values obtained are stored in a dummy variable. If variables occurring in these expressions are changed within the group, the values of a, b and c will not be affected.

   Example

```
increment = 1;
end       = 5;
DO  I     = 1 BY increment TO end;
    increment = 100;
    End       =   0;
    END;
```

   This loop will be executed exactly 5 times. In contrast, the loop in the following example is executed only once.

```
DO I= 1 BY 1 TO 5;
   I= 5;
   END;
```

6. For a group with a control variable and a WHILE option to be executed repeatedly, the control variable must explicitly say so. For instance, in the example

```
DO I  = 1 WHILE (X > Y);
   .
   .
   .
   END;
```

   the loop will only be executed once if the condition X > Y is fulfilled; otherwise it will not be executed at all.

7. In DO loops the identifier I is very often used without declaration as control variable.

   Example

   ```
   DO I    = 1 TO 10;
   ```

   Within the scope of variable I, the same name may be implicitly given to another variable.

   Example

   ```
   DCL X BASED (I);
   ```

   These two statements contradict each other and an error message will be generated. If I is a pointer, its application in a DO group is restricted to the following uses:

   ```
   a) DCL  (I, IA, IB, IC) POINTER;
      .
      .
      .
      DO I  = IA, IB, IC;

   b) DCL  (I, IA) POINTER;
      .
      .
      .
      DO WHILE (I = IA);
   ```

8. When the control variable of a DO group is used as a subscript, care must be taken that its value does not exceed the bounds of the array.

   Example

   ```
   DCL A  DIM(10);
   DO I = 1 TO N;
      A(I) = X;
      END;
   ```

   If N is greater than 10, the value of other variables may be destroyed by the assignment. Such an error is hard to find, especially when object code is destroyed. The error can be traced by enabling the condition SUBSCRIPTRANGE.

8.6.7 Aggregates

1. When array expressions are used in an assignment and an element of the target item is used also on the right side of the assignment, it should be remembered that according to PL/I standard all elements on the right side are evaluated first and the value thus obtained is assigned to a dummy variable. It is only thereafter that the value of the dummy variable is assigned to the target variable.

   Example

   ```
   DCL A DIM (10,20);
   A = A + A (1,1);

   The effect corresponds to the following statements.

   DCL A     DIM (10,20),
       dummy DIM (10,20);
   dummy = A + A (1,1);
   A = dummy;
   ```

   In this way all elements of array A are assigned the value of element A (1,1).

   Note that the following section of program leads to a different result:

   ```
   DCL A  DIM(10,20);
   DO  I = 1 TO 10;
       DO J = 1 TO 20;
          A (I,J) = A (I,J) + A (1,1);
          END;
       END;
   ```

   In the above the value of element A (1,1) is doubled and this doubled value is then added to the value of the other elements (except A (1,1)).

   If the compiler is capable of recognizing that there is no such overlap between the two sides of the assignment, no dummy variable will be created and direct transfer to the target variable is made.

2. When two arrays are multiplied the multiplication is confined to elements having the same subscripts.

   Example

   ```
   DCL (A,B,C)  Dimension(10,10);
   A = B * C;

   This corresponds to the following statements:

   DO  I = 1 TO 10;
       DO J = 1 TO 10;
          A(I,J) =B(I,J) * C(I,J);
          END
       END;
   ```

8.6.8 Strings

1. The assignment of a value to a string variable with attribute VARYING by means of the pseudo variable SUBSTR, has no affect on the length of the string accessed. If the length of this variable is not defined or if parts of the variable outside its defined length are being referenced, the result will be undefined. An error like this can be traced if the condition STRINGRANGE is enabled.
2. It should be noted that even in intermediate results the length of a bit string or a character string must not be allowed to exceed 32767 bits and characters respectively. This cannot be tested by the object program.

8.6.9 Functions and pseudo variables

If the pseudo variable UNSPEC is used as the target variable in an assignment, it has the attribute BIT and the expression standing on the right is converted to BIT. It is important to make sure that this conversion is possible.

8.6.10 Conditions and ON units

1. Attention should be paid to the correct positioning of the ON unit. If an ON unit is to be executed when a certain condition occurs, the ON unit must have been executed at a time prior to that of the occurrence of the condition. In the example

   ```
   GET FILE (file) LIST (a,b,c);
   ON ENDFILE (file)  GOTO end of file;
   ```

   the execution of the program would be terminated with an error if the file were empty; passing control to "end of file" would not be executed, since at initial access time the ON statement has not yet been carried out. Moreover execution of the ON statement is repeated after each execution of the GET statement, always with the same result.

2. An ON unit is executed either when the associated condition occurs or when the SIGNAL statement is used. Passing control to an ON unit is not permitted.

3. ON units for the condition CONVERSION are left either through a GOTO statement or by correcting the erroneous characters by means of the pseudo variables ONSOURCE or ONCHAR. This requirement does not apply if the ON unit has been entered through SIGNAL.

4. At normal exit from the ON unit to the AREA condition, renewed allocation is tried, which leads to an infinite loop, unless provision is made in the ON unit. The ON unit may exploit the fact that in renewing the allocation the size of the memory space requirement as well as the reference to the area are newly calculated and modify the area, or it can provide for sufficient space in the original area.

5. ON units should not be employed to replace sections of programs. Their use should rather be confined to those cases where one wishes to intercept an unexpected exception condition, because the use of ON units is not very efficient. In all other cases suitable tests should be programmed instead of using the error detection facilities of the PL/I compiler for this purpose.

   For instance, in a program in which keys are used for referencing a file, the problem of omitting some of the keys, because they do not fit into the limits of the file, could be solved by means of the KEY condition. It is however preferable to detect the keys by suitable testing operations and to omit access to the file altogether in such a case.

8.6.11 Input/output

1. The condition UNDEFINEDFILE is not only raised when the attributes provided by PL/I are incompatible, but also in the following cases:

   a) Block length smaller than record length.

   b) KEYLENGTH zero or undefined for INDEXED and REGIONAL files.

   c) When the sum of the values of KEYLENGTH and KEYLOC for INDEXED files exceeds the record length.

   d) When the logical record length for the record format VB is not at least 4 bytes smaller than the block length.

2. When a file is simultaneously used for input and output, it must not be declared with the attribute INPUT or OUTPUT. This can be specified in the OPEN statement.

3. Input/output lists must be enclosed by parentheses. This applies also to the iteration list. Thus, in the following example two pairs of external parentheses have to be used:

   ```
   GET LIST ((A(I) DO I = 1 TO N));
   ```

4. The last 8 bytes of the key for referencing a REGIONAL file must be a character string representing a decimal integer. When this key is created, the rules for conversion from arithmetic to CHAR should be observed. Accordingly, the following example is an incorrect section of a program.

   ```
   DCL key   CHAR (8);
   DO I = 1 TO 10;
      key = I;
      WRITE FILE (file) FROM (variable) KEYFROM (key);
      END;
   ```

   The default for I is FIXED BINARY (15,0). In the conversion this gives rise to a string of 9 characters. Consequently the numeric on the right is cut off in the assignment to "key".

5. If an I/O statement contains one of the options KEY, KEYFROM or KEYTO, the corresponding file must have the attribute KEYED.

6. The names SYSIN and SYSPRINT are only implied in the statement GET and PUT respectively. In all other cases, such as for instance in an ON statement and in other I/O statements, these names have to be explicitly specified.

7. PAGESIZE and LINESIZE are not file attributes, i.e. they cannot be used in file declarations but only in OPEN statements.

8. If in GET EDIT or PUT EDIT processing all elements of a data list are processed, no further elements from the format list will be processed, even when these do not require a data element.

   Example

   ```
   GET EDIT (A,B) (F(5), F(5), X(70));
   PUT EDIT (A,B) (A(3), F(5), SKIP);
   ```

   Format X(70) will not be processed. For example, if you wish to skip the remaining 70 positions of a punch card, it is necessary to begin the next statement with X(70) or SKIP. In the case of the PUT statement the SKIP format will not be carried out. It has to be specified in the next PUT statement.

9. If an array or a structure is specified in the data list, then each element of the array and each elementary member of the structure constitutes an element of the data list, thus also requesting an element of the format list.

   Example

   ```
   DCL 1   A,
           2   B   CHAR(5),
           2   C   FIXED(5,2);
   PUT EDIT (A)    (A(5), F(5,2));
   ```

   Elementary members B and C have format A(5) and format F(5,2) respectively.

10. Array elements are processed in the sequence in which the right subscript changes most rapidly:

    ```
    A(1,1), A(1,2), A(1,3), A(2,1) etc.
    ```

11. Character strings read in by GET LIST or GET DATA must be enclosed in apostrophes.

12. The representation of the semicolon in the 48-character set (.,) is not recognized as a separator semicolon when input is made through GET DATA.

13. If a PUT DATA statement is used without a data list, all the data names known at this stage on the basis of the rules for the validity of names will be supplemented. The corresponding scope is determined statically.

    The same applies when PUT DATA is used in an ON unit, except that the ON unit is used dynamically. If the point where the ON unit is invoked in a block is parallel and subordinate to the ON unit, data that is valid only within that block will not be output.

It may thus be useful if during the testing stage the statement

```
ON ERROR PUT DATA;
```

is initially repeated in all internal blocks.

Note

If the ERROR condition returns in an ON unit for the ERROR condition, an infinite loop results.

"PUT DATA;" outputs even those variables whose values are still undefined. As a result, the CONVERSION condition may be set, in its turn resulting in the ERROR condition.

An infinite loop can be avoided by using the following statements:

```
ON ERROR  SNAP BEGIN;
            ON ERROR SNAP SYSTEM;
            PUT DATA;
            END;
```

14. A pointer set by READ SET or LOCATE SET is only valid until the next statement for the same file. In output files the statements WRITE and LOCATE may be used in any combination.

8.6.12 Procedure functions with several entries

On a procedure function reference, a procedure may be entered by all entries with RETURNS specified and may be left again via all RETURNS statements for which a value has been supplied. For ’n’ entries with RETURNS and ’m’ RETURN statements, there are ’n x m’ theoretical ways of converting the value of the RETURN expression to the attribute set according to the RETURNS option. Which of the conversions is required depends on the run of the program and cannot be identified by the compiler but only at the time the RETURN statement is executed, i.e. dynamically. Therefore, when using several RETURNS options with different attribute sets and several RETURN statements with different attribute sets of the expressions, precaution is taken for any cases that may arise. The resulting overhead is a marked increase in memory requirements and computer time.

```
                                                                  Results
PUT SKIP LIST ( CHARACTER ()   );                                     3
PUT SKIP LIST ( FIXED-POINT () );                                     3.0
PUT SKIP LIST ( BIT ()         );                                   '0011'B

CHARACTER:  PROCEDURE  RETURNS ( CHAR (*)  );
            .
            .                                                    3 entries
            .
            RETURN ('7');
FIXED-      ENTRY      RETURNS ( FIXED(5,2) );                   3 return
POINT:      .                                                      points
            .
            .
            .
            RETURN (7);                                          = 3 x 3 con-
BIT:        ENTRY      RETURNS ( BIT(*) );                        versions
            .
            .
            .                                                     Dynamic
            RETURN ('111'B);                                      decision on
            END CHARACTER;                                        the type of
                                                                  conversion
```

Attribute set RETURNS statement		Theoretically required conversions	Attribute set RETURNS option
'7'	CHAR (1) NONVARYING		CHAR (*) NONVARYING
7	REAL FIXED DEC (1,0)		REAL FIXED BIN (5,2)
'111'B	BIT (3) NONVARYING		BIT (*) NONVARYING

Fig. 8-4 Procedure with several entries and RETURNS option and several RETURNS statements with expression

8.6.13 Variable length entry

If variable values or expressions rather than constant values are used to declare lengths (AREA, BIT, CHAR) or DIMENSIONs, access to such items, esp. in connection with structures and arrays, may involve a heavy overhead. They should be avoided whenever possible. It may be a good idea to check whether the desired effect cannot also be achieved via precompiler variables acting as constants.

8.6.14 Passing of parameters

If a parameter satisfies the following conditions:

- Only value passing to the called procedure and not back to the calling procedure
- Scalar
- No * option
- Memory requirement ≤ 1 fullword,

then PARAMETER (INPUT) specification will effect a favorable passing of the value. Not the pointer to the value, but the value itself is then passed. See section 7.1.2.1.

8.6.15 Absolute bit pointer for XS

If the address space above 16 Mbytes (XS) is to be used, some rules must be observed regarding absolute pointers where bit precision is required. These absolute bit pointers may be found in the following cases:

- when ADDR refers to a bit string
- when a bit string is passed as a parameter

In view of the general rules for alignment, absolute pointers only appear if all of the following conditions are met simultaneously:

- Declaration of a bit string
- Declaration within a structure
- Within the structure, a declaration immediately precedes which also meets all of these conditions and whose length is not a multiple of 8.
- Declaration with UNALIGNED

How an absolute bit pointer is declared, is described in chapter 4.

It should be borne in mind that an absolute bit pointer requires a larger storage location than an absolute pointer: if it is used within structures subsequent fields will shift. Check whether conditions are imposed on the internal arrangement of fields of the structure (see chapter 10) that do not permit this.

9 Debugging aids

The purpose of debugging aids is to facilitate or indeed to make possible the detection of errors in a program by means of suitable methods of testing.

The semantics of a program should not be affected by the inclusion of debugging aids.

Control over all the available debugging facilities is exerted by control statements for the compiler (*COMOPT) and for the program (*RUNOPT), as well as via test statements in response to check points.

Experience has shown that the fullest use of debugging aids will only be made if they can be conveniently handled.

The following control options are available:

- Control statements for the compiler
- Control statements for the program
- Setting of breakpoints
- Test statements when a breakpoint is reached.

The following types of debugging aids are available:

- Trace for
  - Entry points to procedures (PROCEDURE)
  - Procedure calls (CALL)
  - Branching (GOTO)
  - Labels
  - Return (RETURN)
- Snap (SNAP)
- Binary dump
- Debugging AID interface

9.1 Compiler control

The control statement

```
*COMOPT DEBUG = specification
```

tells the PLI1 compiler to incorporate certain debugging aids into the program.

The following specifications are possible:

PROCTRACE	Output of a log line: on entry to a procedure
CALLTRACE	on execution of a CALL statement or of a function reference
GOTOTRACE	on execution of a GOTO statement
LABTRACE	on reaching a label
RETURNTRACE	on returning from a procedure
BREAKPOINT(a,...) BREAKPOINT	Breakpoints are set before the lines a, ...
STMT	Runtime errors are displayed with the number of the source line in the error text.

The BREAKPOINT line entry 'a' is supplied in the following format:

```
[i-] z [:s]
```

where

i: consecutive number of the INCLUDE statement
decimal integer 0...256
value 0: text not from INCLUDE file; default

z: consecutive line number of the source listing
decimal integer < 10^7

s: consecutive number of the statement on the source line
decimal integer < 32
default: 1

```
/EXEC     $PLI1
          *COMOPT DEBUG = CALLTRACE
          *END
```

```
/EXEC     $PLI1
          *COMOPT DEBUG = BREAKPOINT (50.2)
          *END
Breakpoint at line 50 before second statement.
```

Fig. 9-1 Examples of compiler control

9.2 Program control

The following control statements are available for controlling debugging during execution time:

```
*RUNOPT DUMP  = Specification
*RUNOPT TRACE = Specification
```

For dynamic control of debugging aids during execution time, control statements may be specified on arrival at an inserted breakpoint.

For activation and deactivation of a checkpoint the following control statement is available:

```
*RUNOPT ACTIVE = YES or NO
```

On the arrival at a breakpoint the possible options for control statements are the same as the specifications for DUMP, TRACE and ACTIVE mentioned above.

When an active breakpoint is reached during execution time, the following message is output:

```
* BKPT/procedure/[i-]z[:s]
```

where the items have the following meanings:

procedure:	Name of internal or external procedure containing the statement
i:	Number of the INCLUDE file; omitted for number 0
z:	Number of source line in source listing
s:	Number (consecutive) of statement in source line omitted for value 1

An input statement is then expected. This may be

- a null statement
- a control statement

It will be executed immediately. If it contains errors, the "ERROR" message together with the part of the control statement not yet executed is output to SYSOUT and SYSLST. This part must be reentered after correction.

The following options may be supplied for *RUNOPT or as control statements:

- TRACE =
  - PROCTRACE, NOPROCTRACE
  - CALLTRACE, NOCALLTRACE
  - GOTOTRACE, NOGOTOTRACE
  - LABELTRACE, NOLABELTRACE
  - RETURNTRACE, NORETURNTRACE
  - TERMINAL, NOTERMINAL
  - ALL, NO

  For further details see section 9.1 and chapter 5.
- ACTIVE = YES or NO
  All breakpoints are activated or deactivated.
- SYSTEM
  Command mode is entered (breakpoint); meaningful only as a control statement.
- DUMP =
  - STACK stack
  - AREA standard area
  - RANGE(a,e) area of hexadecimal addresses a to e
  - SNAP nesting of calls
  - COND only in case of error

  Further details are given in chapter 5.

```
/EXEC     Program
          *RUNOPT TRACE = CALLTRACE
          *END
```

```
*BKPT     /Program/50.2       Output to SYSOUT
TRACE = CALLTRACE             Input from SYSDTA
```

The breakpoint at the second statement in line 50 has signaled. Trace for procedure calls will be activated.

Fig. 9-2 Example for control of debugging aids during execution time

9.3 Trace output

Trace information is output to SYSLST (if TRACE = TERMINAL also to SYSOUT). The lines for the individual specifications have the following structure:

```
PROCTRACE:     *P: p LEVEL: n r
CALLTRACE:     *C: p [i-] z [:s]
GOTOTRACE:     *G: l [i-] z [:s]
LABTRACE:      *L: l [i-] z [:s]
RETURNTRACE:   *R: e [i-] z [:s]
```

where:

p: Procedure name or entry name or name of entry variable

n: Depth of call nesting

r: Contents of registers R1 to R4, R14, R15

i: Number of INCLUDE file or blank

z: Number of source line in source listing

s: Number of statement in source line

l: Name of label, including subscript where appropriate

e: Primary entry name

9.4 Activation of check points

The control statement *RUNOPT ACTIVE = YES or NO or the control statement ACTIVE = YES or NO causes breakpoints to be set to active and passive respectively.

9.5 Program interrupt

Program execution can be interrupted by

- the control statement SYSTEM at a breakpoint.
- the break/escape key on the visual display unit. (Its design varies according to the particular terminal).
- the command /BREAK, if /SYSFILE SYSDTA = (SYSCMD) applies (in DO procedures as well as in batch operations), provided the program reads data from SYSDTA.

The program can subsequently be restarted by /RESUME or by /INTR, where /INTR sets the condition ATTENTION.

If the control statements for the program are terminated by

```
*END/
```

control passes to the command mode. SYSDTA, for example, can then be reassigned. The command /RESUME causes resumption of the program.

9.6 Dump

A dump can be requested by the control statement *RUNOPT DUMP =, by calling one of the procedures described in chapter 11 or by the control statement DUMP. In these cases the output, obtained in the interactive mode, is

```
%IN 45 DUMP DESIRED? REPLY (Y = YES; N = NO)?
```

If the reply is Y, the dump will be output to SYSLST.

When DUMP = STACK and DUMP = AREA the above enquiry will first be issued for the output of the pseudo-register vectors and then for the desired area of memory.

9.7 SNAP

Output takes place to both SYSLST and SYSOUT. The output lines have the following format:

name type address source-line

The source line option will only be displayed if the compilation was performed by DEBUG = STMT. Specifically, 'name' is defined as follows, depending on 'type':

TYP	meaning of the name
ENTRY	entry name of a procedure
SYSTEM	entry name of a library procedure
BEGIN	number (consecutive) of a BEGIN block. This number appears in the address table; cf. LIST = MAP (section 3.8.8)
ON	name of condition

Names whose lengths exceeds 7 characters are shortened to the first 4 and the last 3 characters.

The source line entry is of the form

```
[i-] z [:s]
```

See also section 9.2

```
***************************** SNAP *****************************

START OF PRINTING OF NESTED SUBROUTINES

CALLED FROM    TYPE      ON ADDRESS      INC-#   LINE-# STMT

  SNAP         SYSTEM      0020D2
  FEHLER       ENTRY       000360                   26   1
  ZEROIDE      ON          000374                   14   1
  ER$INTR      SYSTEM      0107EA
  ER$PUB       SYSTEM      010EAC
  SR$STXT      SYSTEM      00FD04
  ##00004      BEGIN       00045A                   20   1
  X97          ENTRY       000254                   12   1
```

Fig. 9-3 SNAP output (sample)

*COMOPT OPTIMIZE = TIME provides the capability for coding certain internal procedures and blocks in a simplified form (see 8.4.11 and 8.5.1). These procedures and blocks are identified by the entry "(QUICK)" in the storage map listing (*COMOPT LIST = MAP). They do not appear on the SNAP listing.

9.8 Interface to the AID debugger

For symbolic debugging with the debugger AID, additional information must be generated when the program is compiled. This can be effected by means of the compiler option "SYMTEST=ALL".

The debugging information is then either transferred statically, during linking/loading, to the loaded program if "SYMTEST=ALL" is specified, or it may be loaded dynamically by AID if the object module is stored in a PLAM library.

For detailed information about debugging with AID, see the manual "AID - Debugging of PL/I Programs" [18].

10 Internal Representation

Generally the user of the PL/I programming language does not need to concern himself with the internal representation of his data in the computer storage. It may, however, be useful to consider the internal representation of the data in the following contexts:

- For error analysis in the object program the current values of variables can be examined, e.g. via
  - UNSPEC (a)
  - HEXDEC (UNSPEC (a))
  - CALL ADUMP or SDUMP or RDUMP

  This makes is possible to determine their status in a bit-by-bit analysis.
- When records are output the record length may be ascertained from a data portion and a management portion. The length of the data portion depends on the type of the source variables and its current contents. How this is ascertained is described in this section. For example, it can be attained using the built-in function SIZE. Whether an additional management portion is required depends on the target file, described in chapter 6.
- When records are read from files generated by a foreign program, the user needs to know whether the structure from which the records were once written and that of the target variables are compatible. For the data portion this can be ascertained using the methods described in these sections. For the management portion this is explained in chapter 6.

There are certain requirements relating to the arrangement of data in storage for the internal representation of scalar variables, array elements, and structure elements as well as for the alignment of structures. This affects the amount of storage required and also alignment on a certain addressing boundary.

This is explained in the following subsections. The following terms are used for the storage requirements:

- Bits
  The specified number of bits is required. The variable can begin at any bit address.

- Bytes
  The specified number of bytes (characters) is required. The variable can begin at any byte address.
- Halfwords
  The specified number of halfwords is required. Each halfword consists of 2 bytes. The variable can begin at any even byte address.
- Fullwords
  The specified number of fullwords is required. One word comprises 4 bytes. The variable can begin at any byte address divisible by 4.
- Doublewords
  The specified number of doublewords is required. Each doubleword comprises 8 bytes. The variable can begin at any byte address divisible by 8.

If, for example, 2 fullwords are required for a variable, the storage space begins at a fullword address. If, however, 8 bytes are required, the storage space begins at a byte address, which may also be a fullword address. The length of the storage area is the same in both cases (8 bytes and 2 fullwords).

Storage space for scalar variables, complete arrays and main structures begins at least on a byte boundary. This does not apply if the variable is a result of overlay defining (e.g. via the DEFINED or BASED attributes) and the part to be overlayed is a subarray or a substructure which is aligned on a bit boundary.
If storage space is allocated for a variable with the attribute CONTROLLED or BASED, it is always in multiples of doublewords, in which case the storage space always begins on a doubleword boundary.

10.1 Arithmetic variables

A brief summary of the

- Default and maximum value of precision (PRECISION)
- Storage requirements depending on precision g
- Addressing boundary depending on alignment (ALIGNED/UNALIGNED)

is provided for arithmetic variables in Figure 10-1.

Detailed specifications may be found in the following subsections.

Pictured arithmetic character strings are included in the character strings in section 10.2.

Data type			PRECISION		Storage requirements		
Mode	Scale	Base	Default	Maximum	for g	ALIGNED	UNALIGNED
REAL	FIXED	DECIMAL	(5,0)	(15,k)	CEIL ((g + 1)/2) Bytes		
		BINARY	(15,0)	(31,k)	1...15	1 halfword	2 bytes
					16...31	1 fullword	4 bytes
	FLOAT	DECIMAL	(6)	(33)	1...6	1 fullword	4 bytes
					7...16	1 double-word	8 bytes
					17...33	2 double-words	16 bytes
		BINARY	(21)	(109)	1...21	1 fullword	4 bytes
					22...53	1 double-word	8 bytes
					54...109	2 double-words	16 bytes
COMPLEX	Real and imaginary portions as in REAL; portion first						

k:+127...-128

Fig. 10-1 Summary of storage requirements and addressing boundaries for arithmetic variables

10.1.1 Fixed binary variables (FIXED BINARY)

Fixed binary variables have the attribute FIXED BINARY PRECISION (g,k), where g specifies the minimum precision in binary digits, and k the position of the binary point. For internal representation

16 bits for g = 1 to 15	which is 2 bytes or 1 halfword
32 bits for g = 16 to 32	which is 4 bytes or 1 fullword

are used. All internal computations are performed to this precision, rounded up if necessary. The storage area begins

- on a half- or fullword boundary for ALIGNED
- on a byte boundary for UNALIGNED.

If k = 0, the implied binary point required for ascertaining the value is located to the right of the rightmost bit. If k is positive, it moves k binary places to the left; if negative, |k| binary places to the right.

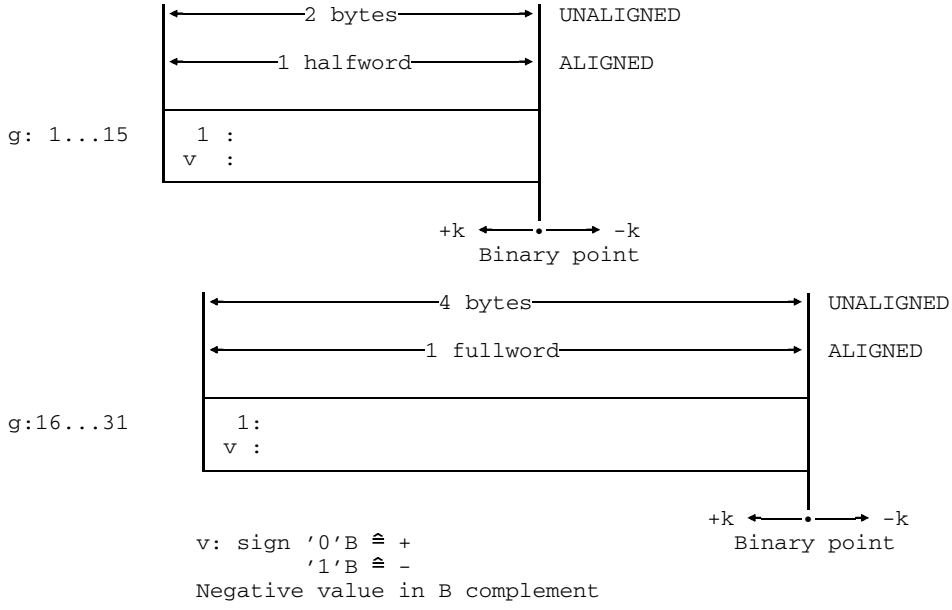


Fig. 10-2 Internal representation of variables with the attribute REAL FIXED BINARY PRECISION (g,k)

```
DCL A   FIXED BINARY PRECISION(10,0)  ALIGNED;
DCL B   FIXED BINARY PRECISION(15,0)  ALIGNED;
DCL C   FIXED BINARY PRECISION(25,0)  ALIGNED;
DCL D   FIXED BINARY PRECISION(31,0)  ALIGNED;

DCL M   FIXED BINARY PRECISION(10,0)  UNALIGNED;
DCL N   FIXED BINARY PRECISION(15,0)  UNALIGNED;
DCL O   FIXED BINARY PRECISION(25,0)  UNALIGNED;
DCL P   FIXED BINARY PRECISION(31,0)  UNALIGNED;
```

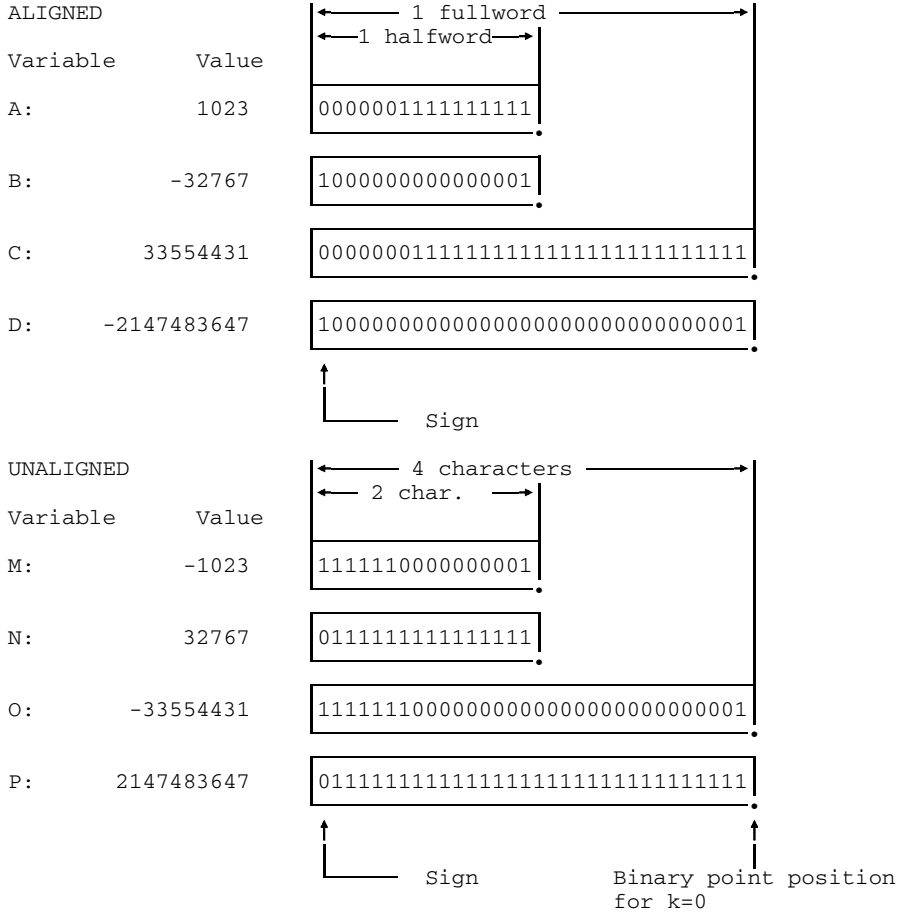


Fig. 10-3 Internal representation of fixed point variables with varying alignement and precision according to the maximum value

The leftmost bit indicates whether the value is positive (’0’B) or negative (’1’B). Negative values are represented in binary complement. The following steps can be taken to ascertain the amount of a negative value:

- All bits, including sign bits, are inverted.
- A bit ’1’B is added on the extreme right, and any overflow bit in the sign bit is ignored.

The resulting value represents the (positive) value. Figure 10-4 shows some examples.

Negative value Inverted Inverted + 1	1 0101=-11 0 1010 0 1011=11	1 1111=-1 0 0000 0 0001=1	1 0000=-0 0 1111 0 0000=0 1

Fig. 10-4 Examples of ascertaining the positive value for k = 0 with negativ values

10.1.2 Fixed decimal variables (FIXED DECIMAL)

Fixed decimal variables have the attribute FIXED DECIMAL PRECISION (g,k), where g specifies the minimum precision in decimal places and k determines the position of the decimal point. For internal representation a sign position is added on the right and, if this results in an odd number of places, a digit position is added to the left. Digits and signs are stored in 1/2-bytes. The storage requirement is therefore

$$\text{CEIL}\left(\frac{g+1}{2}\right) \text{ bytes}$$

All internal computations are performed to this precision, rounded as necessary. For both ALIGNED and UNALIGNED, the storage area begins

- on a byte boundary

If k = 0, the implied decimal point required for ascertaining the value is located to the right of the rightmost digit position. If k is positive it moves K decimal places to the left; if negative, |k| decimal places to the right.

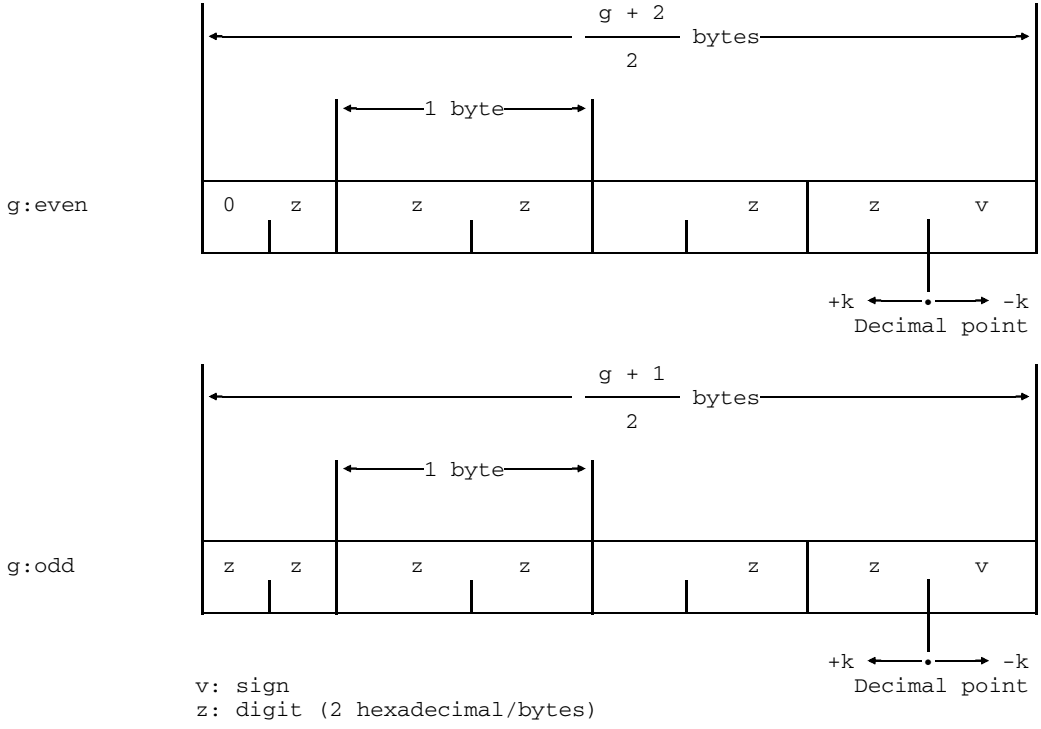


Fig. 10-5 Internal representation of variables with the attribute REAL FIXED DECIMAL PRECISION (g,k)

```
DCL A   FIXED DECIMAL PRECISION(02,0)   ALIGNED;
DCL B   FIXED DECIMAL PRECISION(05,0)   ALIGNED;
DCL C   FIXED DECIMAL PRECISION(10,0)   ALIGNED;
DCL D   FIXED DECIMAL PRECISION(15,0)   ALIGNED;

DCL M   FIXED DECIMAL PRECISION(02,0)   UNALIGNED;
DCL N   FIXED DECIMAL PRECISION(05,0)   UNALIGNED;
DCL O   FIXED DECIMAL PRECISION(10,0)   UNALIGNED;
DCL P   FIXED DECIMAL PRECISION(15,0)   UNALIGNED;
```

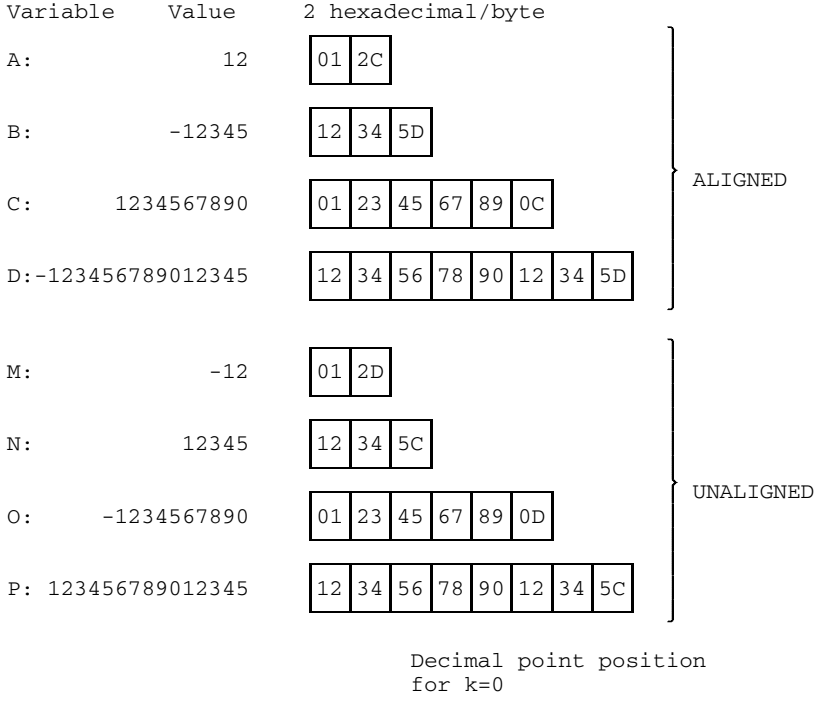


Fig. 10-6 Example of the internal representation of decimal fixed point variables

The internal representation of decimal digits and the sign is shown in Figure 10-7.

Sign	Binary	Hexadecimal
+	1010	A
	1100	C
	1110	E
	1111	F
-	1011	B
	1101	D

Digit	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9

Fig. 10-7 Internal representation of signs and digits for decimal fixed point-variables in 4 bits (binary and hexadecimal)

10.1.3 Floating point variables (FLOAT)

Floating point variables have the attribute FLOAT PRECISION (g), where g is the minimum precision of the mantissa, in binary places for BINARY and in decimal places for DECIMAL. Internally the representation

$m \cdot 16^e$

is used in both cases, with the mantissa m being a true binary fraction (binary point at the extreme left). The exponent is a binary integer (binary point at the extreme right) related to base 16.

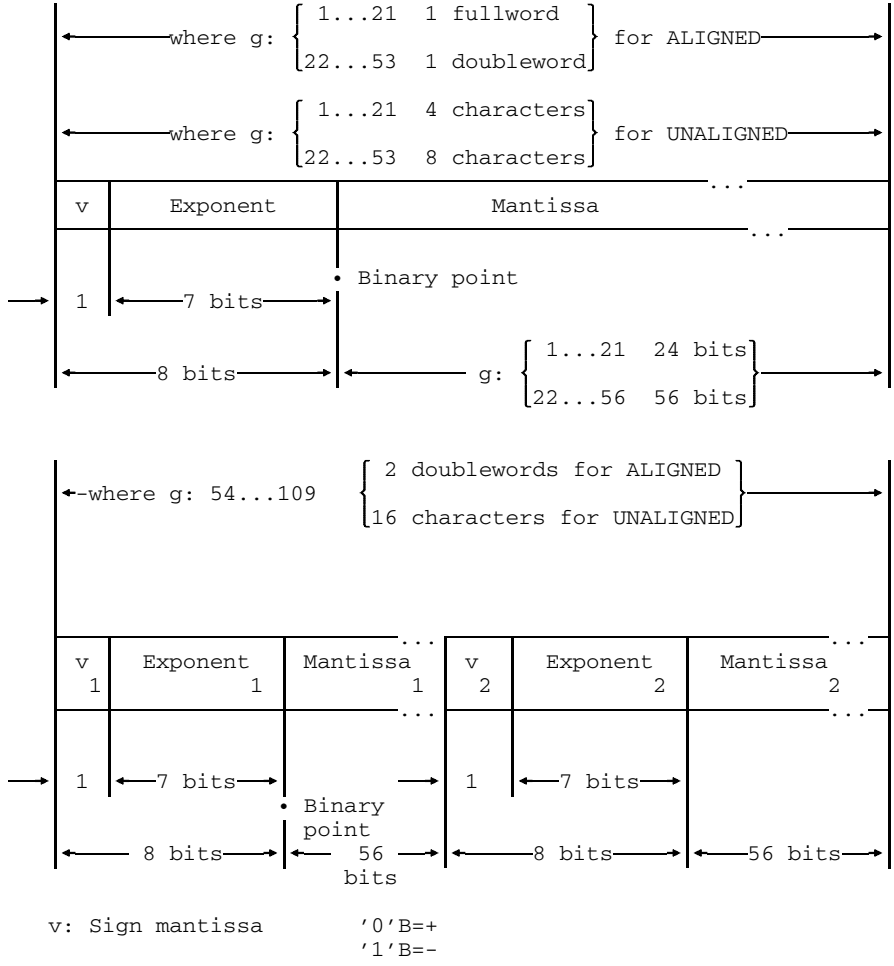


Fig. 10-8 Internal representation of variables with the attribute REAL FLOAT BINARY PRECISION (g)

The leftmost bit is the mantissa sign, where ’0’B represents a positive value and ’1’B a negative value. The following 7 bits represent the value of the exponent. To ascertain the value of the exponent the value 64 must be subtracted from the binary value of the bit pattern. Examples are shown in Figure 10-9.

Binary	Decimal	Exponent
0 000000	0	-64
0 000001	1	-63
0 000010	2	-62
0 111110	62	- 2
0 111111	63	- 1
1 000000	64	0
1 000001	65	+ 1
1 000010	66	+ 2
1 111101	125	+61
1 111110	126	+62
1 111111	127	+63

Exponent = Decimal -64

Fig. 10-9 Examples of the internal representation of the exponent for floating point variables

The length of the mantissa depends on the precision as follows:

	BINARY (g)	DECIMAL (g)	Mantissa	Total
Short	1...21	1...6	24 bits	32 bits
Long	22...53	7...16	56 bits	64 bits
Extended	54...109	17...33	112 bits	128 bits

With the extended floating point number, the representation of the long floating point number is used twice consecutively; the sign bit and exponent bits in the second part have no relevance for the value.

The internal representation of the mantissa is the same as for a binary fixed point variable with the binary point to the left of the leftmost binary digit.

Floating point values are always normalized, i.e. after the appropriate correction of the exponent, the mantissa is shifted in 4 bit positions until at least one of the 4 leftmost bits differs from ’0’B.

Internal computations are performed with the precision required by the internal representation.

```
DCL A   FLOAT BINARY PRECISION(10)   ALIGNED;
DCL B   FLOAT BINARY PRECISION(21)   ALIGNED;
DCL C   FLOAT BINARY PRECISION(53)   ALIGNED;
DCL D   FLOAT BINARY PRECISION(109)  ALIGNED;

DCL M   FLOAT BINARY PRECISION(10)   UNALIGNED;
DCL N   FLOAT BINARY PRECISION(21)   UNALIGNED;
DCL O   FLOAT BINARY PRECISION(53)   UNALIGNED;
DCL P   FLOAT BINARY PRECISION(109)  UNALIGNED;
```

```
Internal representation            Value
(hexadecimal)

A:42100000                          1.600E+01
B:FFFFFFFF                         -7.237005E+75
C:7FFFFFFFFFFFFFF                   7.237005577332260E+75
D:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF  -7.23700557733226221397318656304297E+75

M:40100000                          6.250E-02
N:00FFFFFF                          8.636168E-78
O:80FFFFFFFFFFFFF                  -8.636168555094444E-78
P:00FFFFFFFFFFFFFFFFFFFFFFFFFFFFF   8.63616855509444462538635186280038E-78
```

Fig. 10-10 Example of the internal representation of binary floating point variables

```
DCL A   FLOAT DECIMAL PRECISION(03)   ALIGNED;
DCL B   FLOAT DECIMAL PRECISION(06)   ALIGNED;
DCL C   FLOAT DECIMAL PRECISION(16)   ALIGNED;
DCL D   FLOAT DECIMAL PRECISION(33)   ALIGNED;

DCL M   FLOAT DECIMAL PRECISION(03)   UNALIGNED;
DCL N   FLOAT DECIMAL PRECISION(06)   UNALIGNED;
DCL O   FLOAT DECIMAL PRECISION(16)   UNALIGNED;
DCL P   FLOAT DECIMAL PRECISION(33)   UNALIGNED;
```

```
Internal representation            Value
(hexadecimal)

A:42100000                          1.60E+01
B:FFFFFFFF                         -7.23700E+75
C:7FFFFFFFFFFFFFF                   7.237005577332260E+75
D:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF  -7.23700557733226221397318656304297E+75

M:40100000                          6.25E-02
N:00FFFFFF                          8.63616E-78
O:80FFFFFFFFFFFFF                  -8.636168555094444E-78
P:00FFFFFFFFFFFFFFFFFFFFFFFFFFFFF   8.63616855509444462538635186280038E-78
```

Fig. 10-11 Examples of the internal representation of decimal floating point variables

10.1.4 Picture character string variable (PICTURE)

Arithmetic character string variables with the PICTURE attribute are always represented internally as character strings, as described in section 10.2.3.

10.1.5 Complex variables

Complex variables always consist of a real portion and an imaginary portion, both of which are represented in the same way. The imaginary portion immediately follows the real portion.

10.2 String variables

Figure 10-12 provides a brief summary of

- default and maximum value of the length specification for character strings (CHARACTER) and bit strings (BIT).
- storage requirements
- addressing boundaries

for string variables.

Further details may be found in the following subsections.

String		UNALIGNED	ALIGNED
CHAR (n)	NONVARYING	n bytes	
	VARYING	2 + n bytes	1 halfword + n bytes
BIT (n)	NONVARYING	n bits	$\frac{n}{8}$ CEIL (-) bytes
	VARYING	$\frac{n}{8}$ CEIL (2+-)bytes	1 halfword + $\frac{n}{8}$ CEIL (-) bytes
PIC'...'	As for CHAR (m) NONVARYING m=number of symbols without V,F,K		
PIC'...' COMPLEX	Real and imaginary portions as for PIC'...'; real portion first		

n: default = 1
maximum = 32767

m: numeric max. 255
alphanumeric max. 511

Fig. 10-12 Summary of the storage requirements and addressing boundaries of string variables

10.2.1 Bit string variables (BIT)

The following storage space (x = 0 to 7) is required for bit string variables with the attribute BIT(n):

```
NONVARYING UNALIGNED                n bits
NONVARYING   ALIGNED                n bits + x filler bits
VARYING    UNALIGNED  2 bytes     + n bits + x filler bits
VARYING      ALIGNED  1 halfword + n bits + x filler bits
```

x thus comprises as many filler bits as are needed to make the total number of bits (n + x) divisible by 8, so that they fill an exact number of bytes.

The storage area begins on

```
a bit boundary for       NONVARYING UNALIGNED
a byte boundary for      NONVARYING   ALIGNED
a byte boundary for      VARYING    UNALIGNED
a halfword boundary for  VARYING      ALIGNED
```

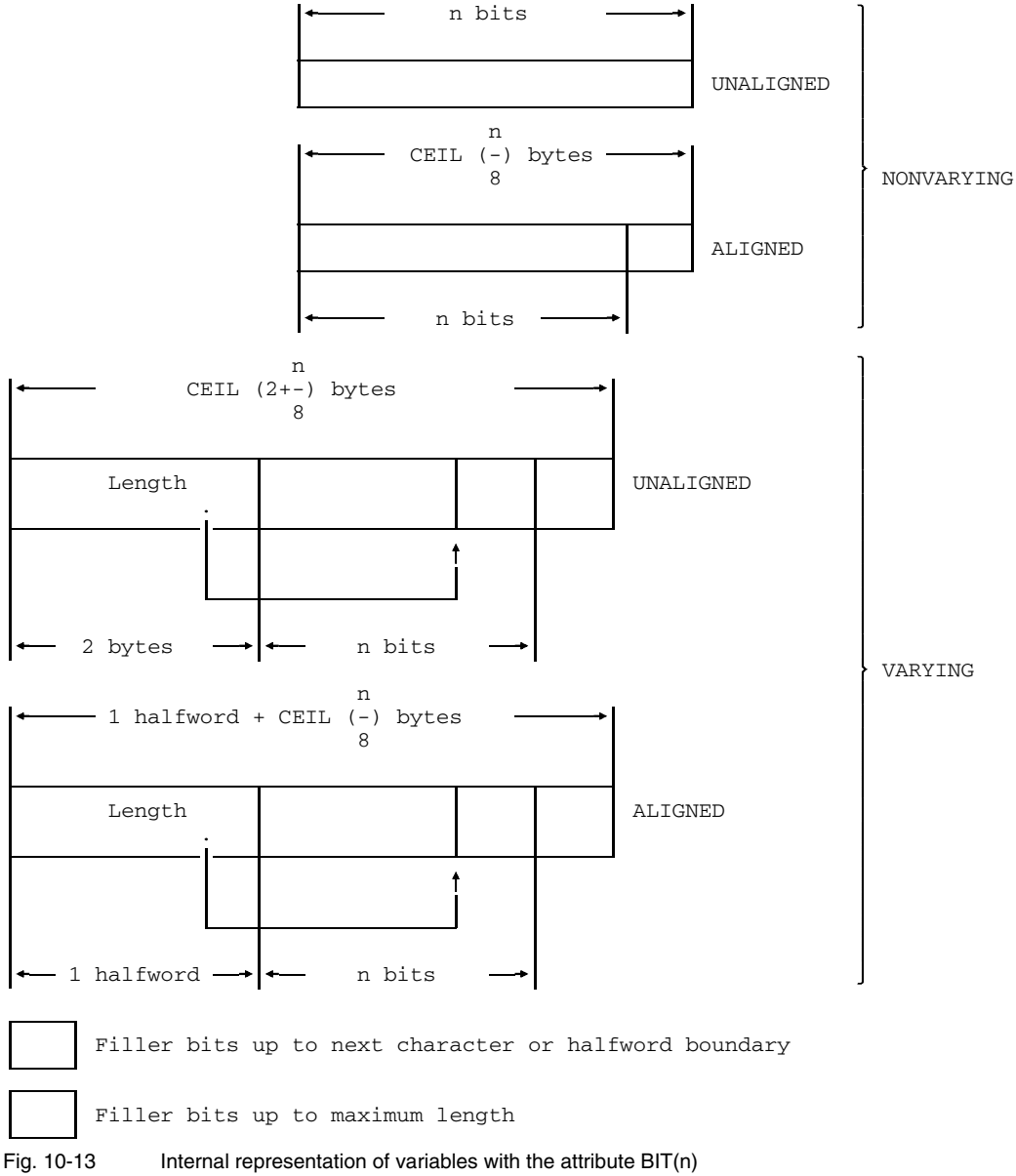


Fig. 10-13 Internal representation of variables with the attribute BIT(n)

```
DCL A   BIT(4)   NONVARYING   ALIGNED;
DCL B   BIT(4)   NONVARYING   UNALIGNED;
DCL C   BIT(8)   NONVARYING   ALIGNED;
DCL D   BIT(8)   NONVARYING   UNALIGNED;

DCL M   BIT(4)   VARYING      UNALIGNED;
DCL N   BIT(4)   VARYING      ALIGNED;
DCL O   BIT(8)   VARYING      UNALIGNED;
DCL P   BIT(8)   VARYING      ALIGNED;
```

```
Value            Internal representation (binary)

A: 1100          1100
B: 1111          1111
C: 11111100      11111100
D: 11111111      11111111

M: 11            0000000000000001011

N: 1111          00000000000001001111

O: 11111         000000000000110111111

P: 11111111      0000000000000100011111111

                    Length specification
```

Filler bits to maximum length

Fig. 10-14 Examples of the internal representation of bit strings

For the record output of a scalar variable with the attribute VARYING, only as many characters are output as required to ensure that the number of bits defined in the length specification can be obtained in the record. The last character of the record can therefore contain a further 0 to 7 filler bits. The length specification is only output if the option ENVIRONMENT (SCALARVARYING) is specified for the file. The same applies to record input. See section 6.3.5 for a detailed explanation.

Scalar variables, main structures and arrays begin at least on a byte boundary. This point is of particular importance for the record output of variables with the attribute BIT NONVARYING UNALIGNED. If such variables are located at the beginning or end of a record, an unexpected record alignment may occur during output. When the record is read in again adjacent records may be inadvertently modified. See also section 10.5.5.

10.2.2 Character string variables (CHARACTER)

The following storage space is required for character strings with the attribute CHAR(n):

```
NONVARYING                          n bytes
VARYING    UNALIGNED 2 bytes    + n bytes
VARYING      ALIGNED 1 halfword + n bytes
```

The storage area begins on

```
a halfword boundary for VARYING ALIGNED
a byte boundary for all other cases.
```

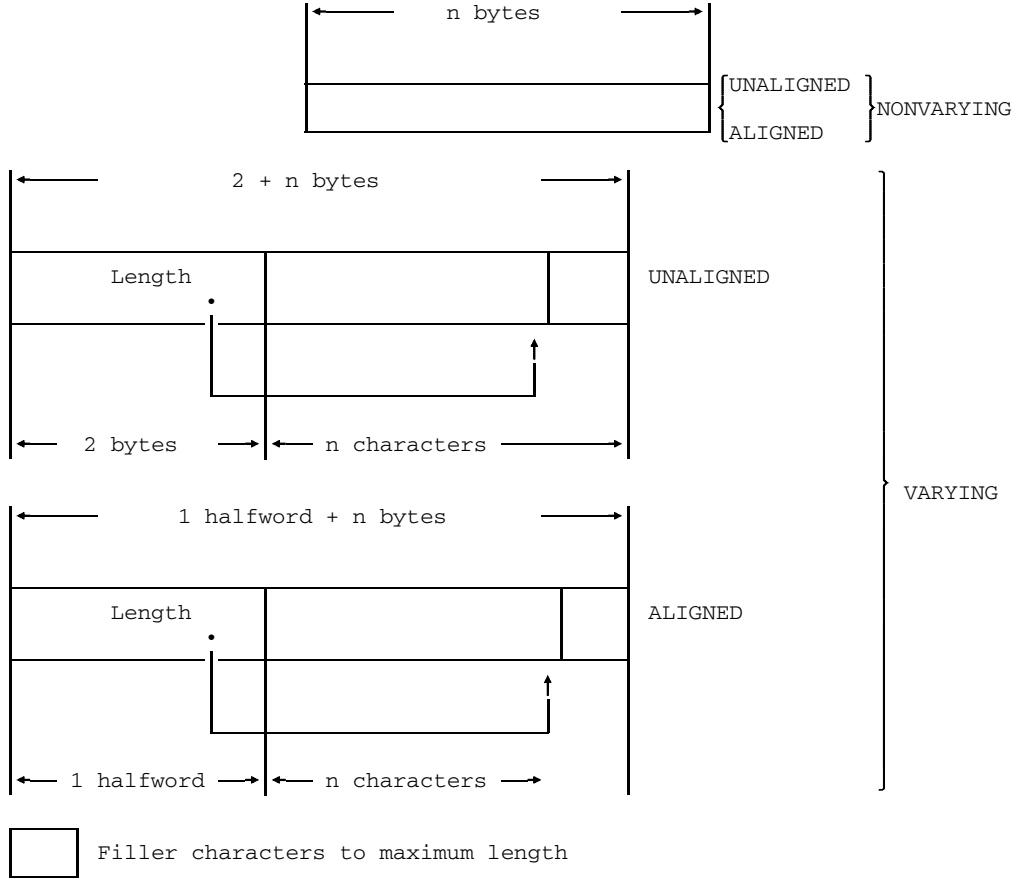


Fig. 10-15 Internal representation of variables with the attribute CHARCTER(n)

```
DCL A   CHARACTER(05)   NONVARYING   ALIGNED;
DCL B   CHARACTER(05)   NONVARYING   UNALIGNED;
DCL C   CHARACTER(11)   NONVARYING   ALIGNED;
DCL D   CHARACTER(11)   NONVARYING   UNALIGNED;

DCL M   CHARACTER(05)   VARYING      ALIGNED;
DCL N   CHARACTER(05)   VARYING      UNALIGNED;
DCL O   CHARACTER(11)   VARYING      ALIGNED;
DCL P   CHARACTER(11)   VARYING      UNALIGNED;
```

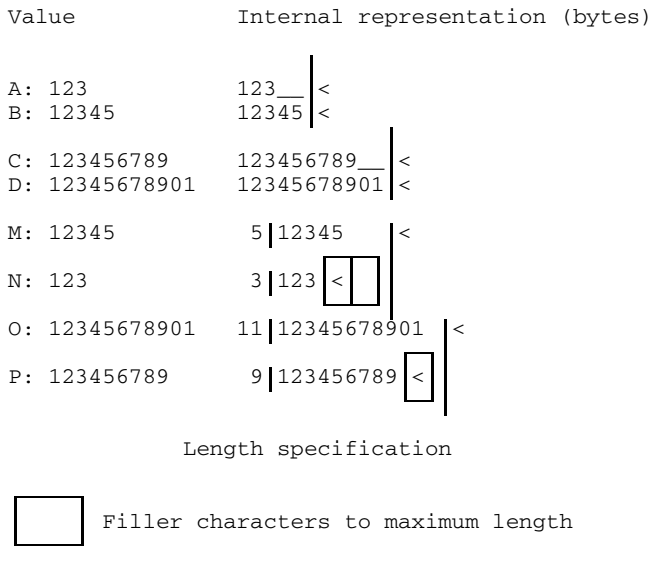


Fig. 10-16 Examples of the internal representation of character strings

For the record output of a scalar variable with the VARYING attribute, only the number of characters specified in the current length are output. The length specification is not output unless the option ENVIRONMENT (SCALARVARYING) is specified for the file. The same applies to record input. See section 6.3.5 for further details.

10.2.3 Picture variable (PICTURE)

In a variable with the PICTURE attribute the values are always represented as strings as for CHAR(n) NONVARYING. The length n of the character string results from the number of picture symbols after the factors have been evaluated, but does not include the picture specifications V, K and F.

Specification of the COMPLEX attribute indicates that the variable consists of a real and an imaginary portion, both of which have the same internal representation. The imaginary portion immediately follows the real portion.

```
DCL A   PIC'AX9XXXXX';
DCL B   PIC'*999.V9999';              Internal representation
DCL C   PIC'S999ES99';

A = 'A:5A-B';                         A:5A-B

B = 123.456;                          *123.4560

C = 12E73;                            +120E+72
```

Fig. 10-17 Internal representation of picture character string variables

10.3 Program control variables

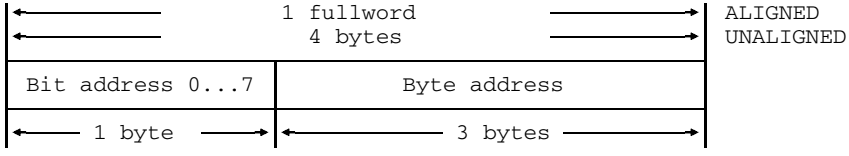
10.3.1 Pointer (POINTER, OFFSET)

The contents of these variables is a pointer that points to a storage location.

10.3.1.1 Pointer if "*COMOPT OPTIONS = NOXS"

After compilation with "*COMOPT OPTIONS = NOXS", relative and absolute pointers have the same internal representation. A pointer requires either one fullword or four bytes of storage. The three rightmost bytes contain the absolute or the relative storage address. The leftmost byte contains the bit address 0 through 7; the five leftmost bits are always ’0’B.

A null pointer has the hexadecimal value ’FFFEFFF8’




```
Null pointer (hexadecimal):
```

FFFEFFF8

Fig. 10-18 Internal representation of pointers with "*COMOPT OPTIONS = NOXS"

```
DCL A   POINTER   ALIGNED;
DCL B   POINTER   UNALIGNED;
DCL C   OFFSET(M)  ALIGNED;
DCL D   OFFSET(U)  UNALIGNED;
```

```
                                    internal representation (hex.)

                                        A: 00|0461D8
                                        B: 00|046200
                                        C: 00|000020
                                        D: 00|000020

                                  Bit address    Byte address
```

Fig. 10-19 Examples of the internal representation of pointer variables with "*COMOPT OPTIONS = NOXS"

10.3.1.2 Pointer with "*COMOPT OPTIONS = XS"

If compilation was performed with "*COMOPT OPTIONS = XS", then relative pointer and null pointer have the same internal representation as described in section 10.3.1.1.
The absolute pointer has two new internal representation forms:
The absolute pointer which can only point to a storage location with byte precision. It requires 1 fullword or 4 bytes storage space containing the absolute storage address. A bit address is not present.

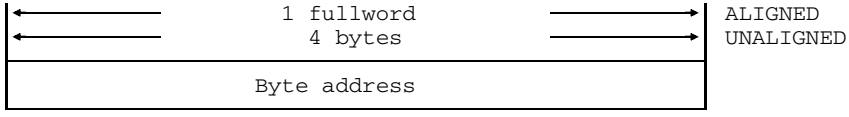


Fig. 10-19a Internal representation of the absolute bit pointer if "*COMOPT OPTIONS = XS"

The absolute bit pointer which can point to a storage location with bit precision. It requires 1 fullword plus 1 halfword, or 6 bytes. The four leftmost bytes contain the absolute storage address. The two rightmost bytes contain the bit address 0 through 7.

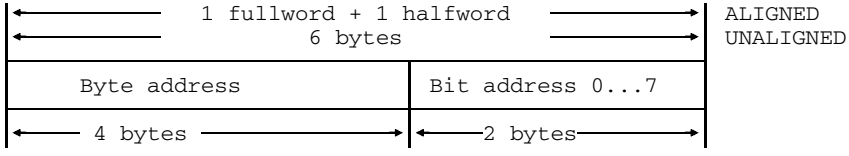


Fig. 10-19b Internal representation of the absolute bit pointer if "*COMOPT OPTIONS = XS"

```
DCL A   POINTER   ALIGNED;
DCL B   POINTER   UNALIGNED;
DCL C   POINTER(BITPTR) ALIGNED;
DCL D   POINTER (BITPTR) UNALIGNED;
```

```
                                        internal representation (hex.)

                                             A: 100461D8
                                             B: 00046200
                                             C: 100461D8|0000
                                             D: 00046200|0000

                                      Byte address   Bit address
```

Fig. 10-19c Examples of the internal representation of absolute pointer and absolute bit pointer variables if *COMOPT OPTIONS = XS"

10.3.2 Area (AREA)

The storage area required for an area variable with the AREA(n) attribute amounts to:

```
          n
CEIL (3 + -) doublewords
          8
```

The first three doublewords contain management information, the remainder are available for the assignment of BASED variables. The storage area always begins on a doubleword boundary.

If BASED variables are assigned, the area is always reserved in doubleword portions, which means that filler information may result at the end of the BASED variables.

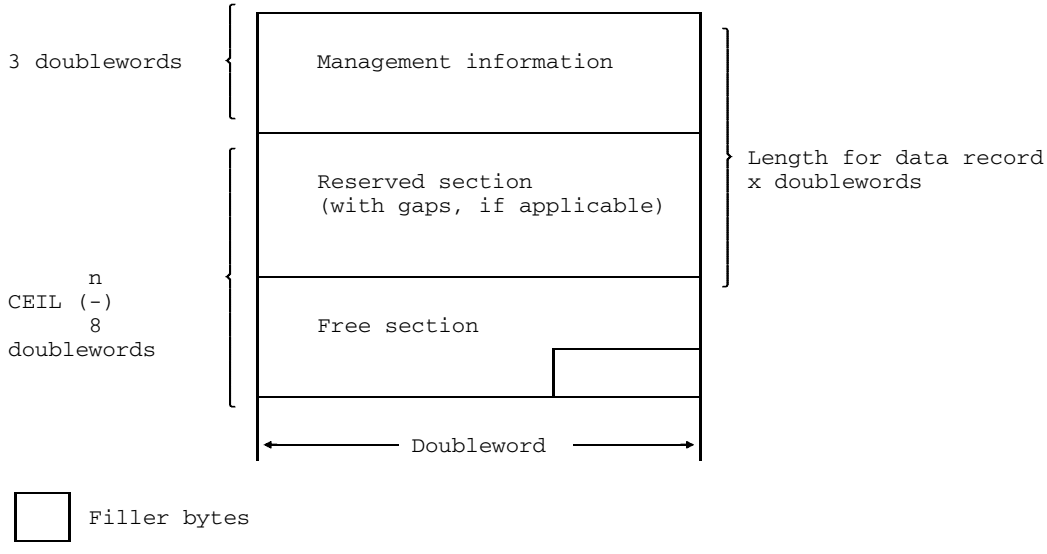


Fig. 10-20 Internal representation of an area with the attribute AREA(n)

For record output, the three doublewords of management information and the doublewords of the reserved section are output. The record length is therefore always a number of doublewords (i.e. multiple of 8 bytes). The reserved section may contain gaps, which are also transferred; output does not involve the elimination of the gaps. The management information remains unchanged during both input and output.

A detailed description of the internal structure of the area and the management strategy may be found in section 10.7.4.

```
DCL BEREICH      AREA(40);

DCL VARIABLE     CHAR(6) BASED   INIT('------');

DO I=1 TO 5;
   ALLOCATE VARIABLE IN(BEREICH) SET (Z(I));
   END;                                                    Record length

FREE Z(2)->VARIABLE IN(BEREICH);
WRITE FILE(DATEI) FROM(BEREICH);                           64 bytes
FREE Z(5)->VARIABLE IN(BEREICH);
WRITE FILE(DATEI) FROM(BEREICH);                           56 bytes
```

Fig. 10-21 Example of the record length of an area

10.3.3 Label (LABEL)

The storage requirement of a label variable amounts to 3 fullwords or 12 bytes.

The 4 rightmost bytes contain the block activation record address, which may be of special importance for recursive calls. The middle 4 bytes contain the base address and the 4 leftmost bytes the label, the target address. The storage address begins on a full-word boundary for ALIGNED, on a byte boundary for UNALIGNED

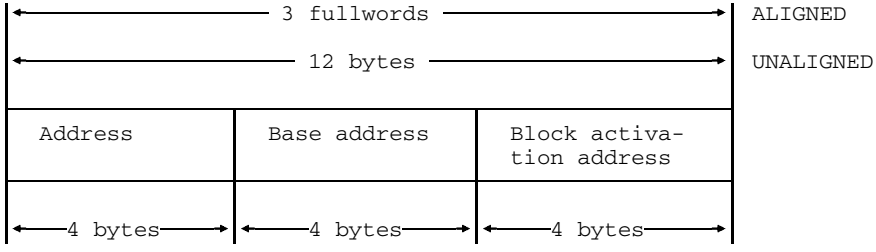


Fig. 10-22 Internal representation of variables with the LABEL attribute

```
DCL MARKE   LABEL VARIABLE;
```

```
Internal representation (hexadecimal)

00000190|00000000|00046120
00000194|00000000|00046120
00000198|00000000|00046120
0000019C|00000000|00046120

Statement          Block activation
address            record address

          Base address
```

Fig. 10-23 Examples of the internal representation of label variables

10.3.4 Format (FORMAT)

The storage requirement for a format variable amounts to 2 fullwords or 8 bytes.

The four rightmost bytes contain the block activation address, which may be of special importance for recursive calls. The 4 leftmost bytes contain the address of the format list. The storage address begins on a fullword boundary for ALIGNED, on a byte boundary for UNALIGNED.

With OPTIONS = NOISO a format value can be assigned to a label variable, in which case the format value occupies the 8 leftmost bytes and the 4 rightmost bytes are undefined.

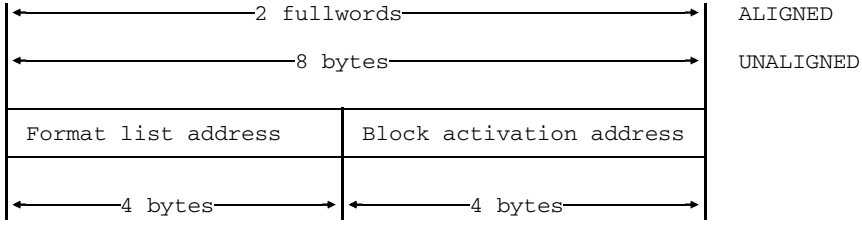


Fig. 10-24 Internal representation of variables with the FORMAT attribute

```
DCL FORMAT   FORMAT VARIABLE;
```

```
                Internal representation (hexadecimal)

                     00000468|00046120
                     00000448|00046120
                     00000428|00046120
                             |
                Format list  |    Block activation
                address      |    address
```

Fig. 10-25 Examples of the internal representation of format variables

10.3.5 Entry (ENTRY)

The storage requirement for an entry variable amounts to 2 fullwords or 8 bytes.

The 4 leftmost bytes contain the entry-point address, and the 4 rightmost bytes the address of the activation record for a given activation of the static parent block. The storage area begins on a fullword boundary for ALIGNED, on a byte boundary for UNALIGNED

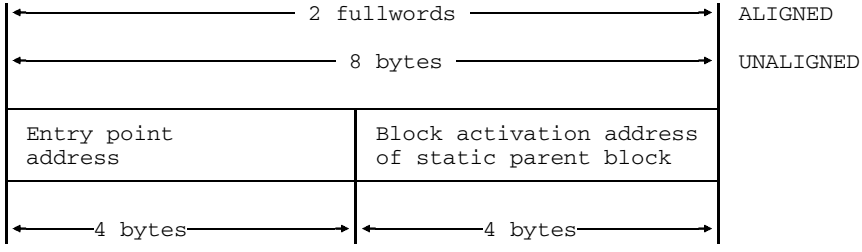


Fig. 10-26 Internal representation of variables with the ENTRY attribute

```
DCL EINGANG   ENTRY VARIABLE;
```

```
                Internal representation (hexadecimal)

                     00000120|00028120
                     00000180|00028120
                     000001D4|00028120
                     00000234|00028120

               Entry-point   |    Block activation record
               address       |    address of static parent block
```

Fig. 10-27 Examples of the internal representation of entry variables

10.3.6 File (FILE)

The storage requirement for a file variable amounts to 2 fullwords or 8 bytes.

The 4 leftmost bytes contain the address of the file-attribute block, and the 4 rightmost bytes the address of the file-status block. The storage address begins at a fullword address for ALIGNED, at a byte address for UNALIGNED

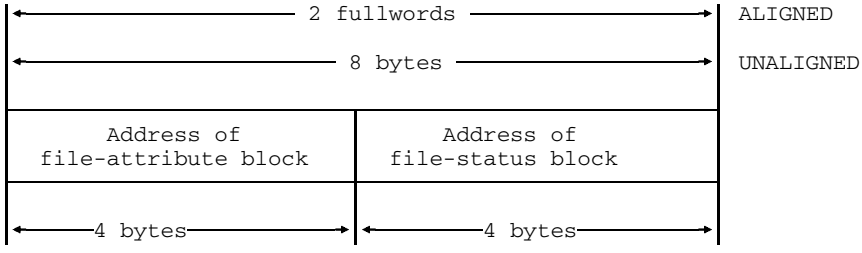


Fig. 10-28 Internal representation of variables with the FILE attribute

```
DCL VAR   FILE VARIABLE;
```

```
                Internal representation (hexadecimal)

                     000004F0|00033758
                     00000548|00033B00
                     000004B0|000333B0

        Address of           | Address of
        file-attribute block | file-status block
```

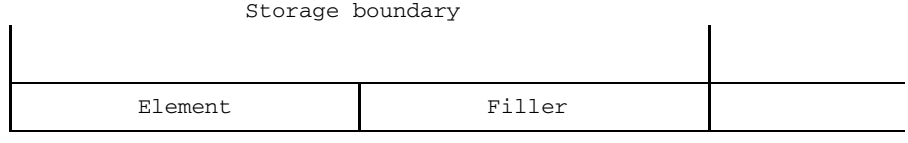
Fig. 10-29 Example of the internal representation of file variables

10.4 Array (DIMENSION)

A variable with the attribute DIMENSION consists of a number of array elements located one after the other in storage. The number of elements is determined via the option in the DIMENSION attribute.

An array element can be either a scalar variable or a structure. All the elements within an array have the same structure and are represented in the same way internally. The internal representation, i.e. storage requirements and addressing boundaries for an array element, is described in section 10.3 for a scalar element and in section 10.5 for structures.

The arrangement in storage must comply with the storage space requirements and addressing boundaries for each element. This may result in filler information between the elements, necessitated by the arrangement of the elements within the array. In such cases the filler information between the elements also appears after the last element of the array. The storage space for this is included in the storage requirement of the array. See Figure 10-30.




```
Storage requirement = n (e + f)

where e = storage required for an element
      f = storage space up to next storage boundary
      n = number of array elements
```

Fig. 10-30 Storage requirements of an array

The elements of an array with more than one dimension are arranged in such a way that the rightmost index with respect to the beginning of the storage area is the one that changes most quickly, e.g.

```
A(1,1) A(1,2) A(1,3) A(2,1) A(2,2) A(2,3)
```

Special notice should be taken if BIT UNALIGNED is used in conjunction with records. See section 10.5.5.

The record output. The filler information at the end of an array becomes part of the record. If, however, a single element (scalar element or structure) is output, the filler information at the end of the element does not become part of the record. See example below.

```
                                                    Implicit

DCL 1  ARRAY    DIMENSION(1),
       2 FEST      FIXED BINARY,                    ALIGNED
       2 ZEICHEN   CHARACTER(1);                    UNALIGNED

                                                    Results
WRITE FILE(DATEI) FROM(ARRAY);                      Record length = 4 bytes
WRITE FILE(DATEI) FROM(ARRAY(1));                   Record length = 3 bytes
```

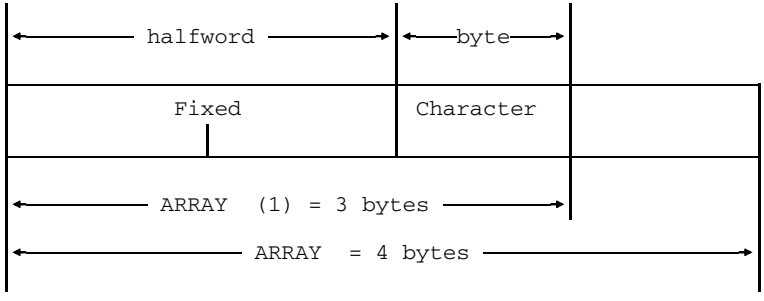


Fig. 10-31 Example of filler information at the end of arrays

10.5 Structure (STRUCTURE)

The storage requirements of structures and their alignment on storage boundaries is described in section 10.5.1. The appropriate consequences in particular areas of application are discussed in subsequent sections.

10.5.1 Storage requirements

The operation to ascertain the storage requirements of a structure begins with the lowest structure levels and continues with the next higher structure level until the main structure is reached. Thus, the following steps are performed for each substructure and the main structure.

- The storage boundary on which a structure level begins is determined from the highest alignment required by its members.
- Beginning with the storage boundary ascertained in this way, the storage space required is allocated for each member of the structure in accordance with the required alignment.
- The end of the storage space for the last member marks the end of the total storage space, i.e. a filler field at the end of a structure is not included in the length of the structure.
- In this way the storage boundary and storage requirements for a substructure are determined. If a higher structure level is present, this structure and its storage requirements become part of the higher structure.

Filler fields are inserted whenever the end of the previous element falls on a storage boundary that is different from that required for the next element. The contents of this filler field are undefined.

When a structure is output as a record, this is always done in multiples of bytes. In connection with an element which has the BIT NONVARYING, UNALIGNED attributes, this may lead to errors. See section 10.5.5.

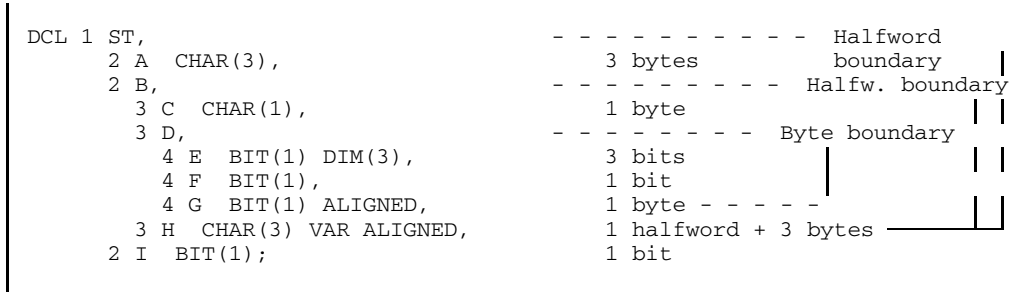


Fig. 10-32 Example of the internal representation of a structure

Fig. 10-33 Internal representation of the structure in accordance with Fig. 10-32

10.5.2 Aliased variables

The term "aliased variable" generally means that storage space can be referenced by two or more variables, each of which has its own data description. Aliased variables can occur in the following instances:

1. Using the DEFINED attribute
2. Via the built-in functions STRING and SUBSTR
3. Via variables with the BASED attribute
4. On parameter transfer
5. When records are input
   - via the READ statement
   - via the READ SET statement
6. With EXTERNAL variables of the same name.

For aliased variables it is essential that the variables referring to the same storage space match. When records are entered, the variables must accord with the record structure, and when ascertaining whether they match, it can be assumed that the record has the same data description as the variable from which it is derived (i.e. the variable which was output).

For aliased variables according to points 1, 2 and 4 the match is checked by the compiler, and a message output, if applicable. For point 4, this check is only valid within an external procedure. In the case of overlay defining via BASED variables and record input according to points 3 and 5, a check by the compiler is not possible and the user must ensure that the variables match. As long as he complies with the rules that govern overlay defining via DEFINED, the match is approved. In all other cases the user must check the match on the basis of the internal representation.

The internal structure representation described in section 10.5.1 supports the overlay defining of main and substructures permitted in PL/I if both have the same data description. An example of this is shown in Fig. 10-34. Main structure Y and substructure X.B have the same data description and may be overlayed.

```
                                                    Implicit          Storage req.

DCL 1 X,
      2 A       CHAR(2),                            ALIGNED           2 bytes
      2 B,
        3 C     BIT(1),                           UNALIGNED           1 bit
        3 D     CHAR(3),                          UNALIGNED           3 bytes
        3 E     FIXED BINARY (31,0);                ALIGNED           1 fullword

DCL 1 Y,
      2 C       BIT(1),                           UNALIGNED           1 bit
      2 D       CHAR(3),                          UNALIGNED           3 bytes
      2 E       FIXED BINARY (31,0);                ALIGNED           1 fullword


PUT SKIP(2) LIST (SIZE(X));                        12 }
PUT SKIP(1) LIST (SIZE(Y));                         8 } Result


WRITE FILE(DATEI) FROM(X.B);                       } Identical record lengths
WRITE FILE(DATEI) FROM(Y);                         } = 2 fullwords
```

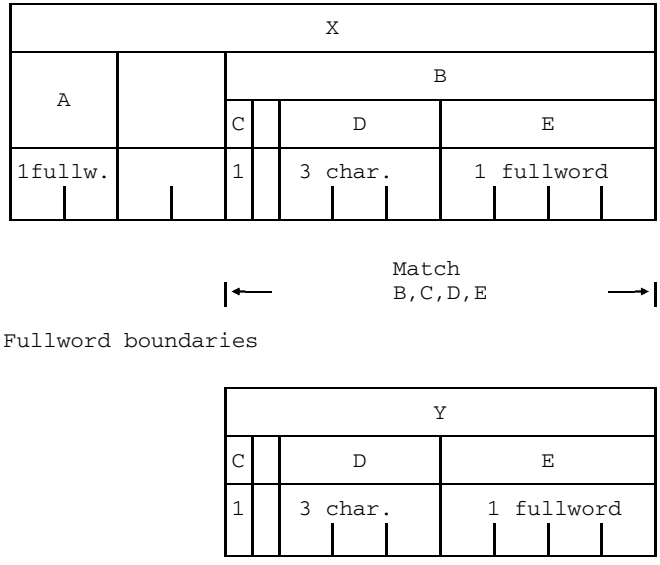


Fig. 10-34 Example of the compatibility of main and substructures with the same data description

10.5.3 Matching at the beginning

The internal representation of structures described produces a match at the beginning of the structure as long as the same data description is given for the structures on logical level 2 and their members. This match does not apply to any following structures of level 2 or their members.

An example is illustrated in Fig. 10-35. In both main structures X and Y the members X.F and Y.F have differing descriptions, which means that structures X.C and Y.C of level 2 do not match. The only sections of the structure that match are those which precede X.C or Y.C. The sections of the structure which follow X.C and Y.C (to the end of the main structure) no longer match. This also applies to elements D and E, even though they have the same data descriptions. This can be seen very clearly from the graphic representation of the storage area in Fig. 10-35.

```
                                                    Implicit          Storage req.

DCL 1 X,
      2 A       CHAR(3),                            UNALIGNED         3 bytes
      2 B       CHAR(1),                            UNALIGNED         1 byte
      2 C,
        3 D     BIT(1),                             UNALIGNED         1 bit
        3 E     CHAR(3),                            UNALIGNED         3 bytes
        3 F     FIXED BINARY (31,0);                  ALIGNED         1 fullword

DCL 1 Y                  BASED(Z),
      2 A       CHAR(3),                            UNALIGNED         3 bytes
      2 B       CHAR(1),                            UNALIGNED         1 byte
      2 C,
        3 D     BIT(1),                             UNALIGNED         1 bit
        3 E     CHAR(3),                            UNALIGNED         3 bytes
        3 F     FLOAT BINARY (51);                    ALIGNED         1 doubleword
```

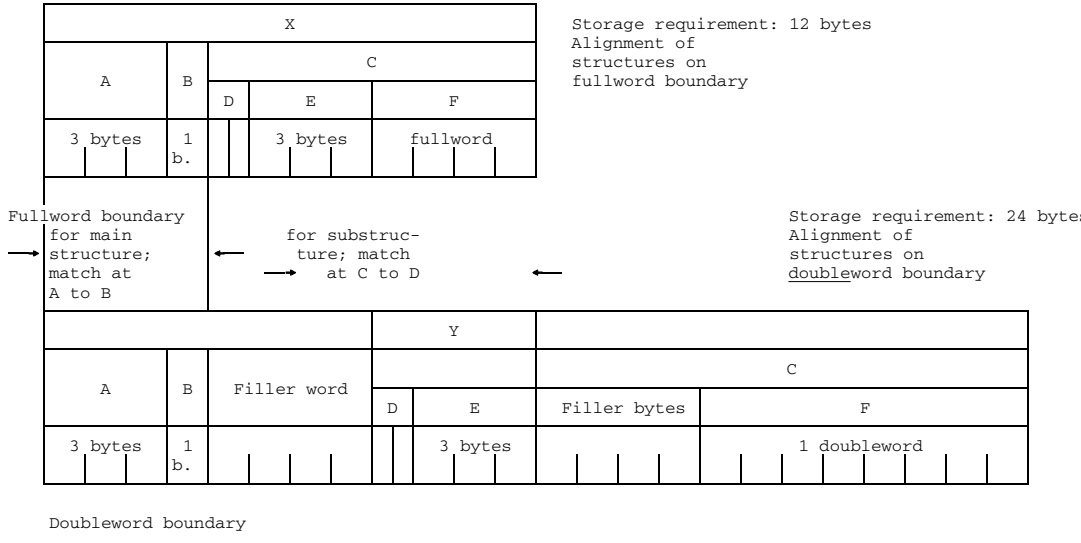


Fig. 10-35 Example of the match at the beginning of a structure

If we consider substructures X.C and Y.C in Fig. 10-35 separately from the main structures, then elements D and E are located on logical level 2 and have the same data descriptions. Structures X.C and Y.C therefore match in elements D and E.

```
If, for example, records were output and entered via

WRITE FROM (X)
READ  INTO (Y)

then reference to Y.A and Y.B would be permitted, and

WRITE FROM (X.C)
READ  INTO (Y.C)
```

would permit reference to Y.D and Y.E.

10.5.4 Self-defining structures

Self-defining structures are those in which REFER declares for one or more elementary members that the current length or current dimension is stored in another elementary member.

In this case, instead of a number of variables accessing one storage location, one variable can access a number of storage locations of different length. Information from which the length can be determined is stored in the structure itself so that the result will always be correct even for accesses to storage locations of different length.

```
                                                    Implicit      Storage req.

  DCL 1 S    BASED,
      2 A    CHAR(2),          INIT('——'),        UNALIGNED    2 bytes
      2 B,
        3 C  FIXED BINARY (31,0),                   ALIGNED    1 fullword
        3 D  CHAR(N REFER (C)) INIT((8)'='),      UNALIGNED    n bytes
        3 E  FLOAT BINARY (51) INIT(0.5);           ALIGNED    1 doubleword


                                                    Result
N = 4;    ALLOCATE S SET(Z4);
          PUT SKIP(2) LIST (SIZE(S));               24  ┐
N = 8;    ALLOCATE S SET(Z8);                           } Length in bytes
          PUT SKIP(2) LIST (SIZE(S));               32  ┘
N = 0;

                                                    Record lengths

WRITE FILE(DATEI) FROM(Z4->S);                      24 bytes
WRITE FILE(DATEI) FROM(Z8->S);                      32 bytes

CLOSE FILE(DATEI);                                  ┐
READ FILE(DATEI) SET(Z);                            } Access via Z->S
READ FILE(DATEI) SET(Z);                            ┘
```

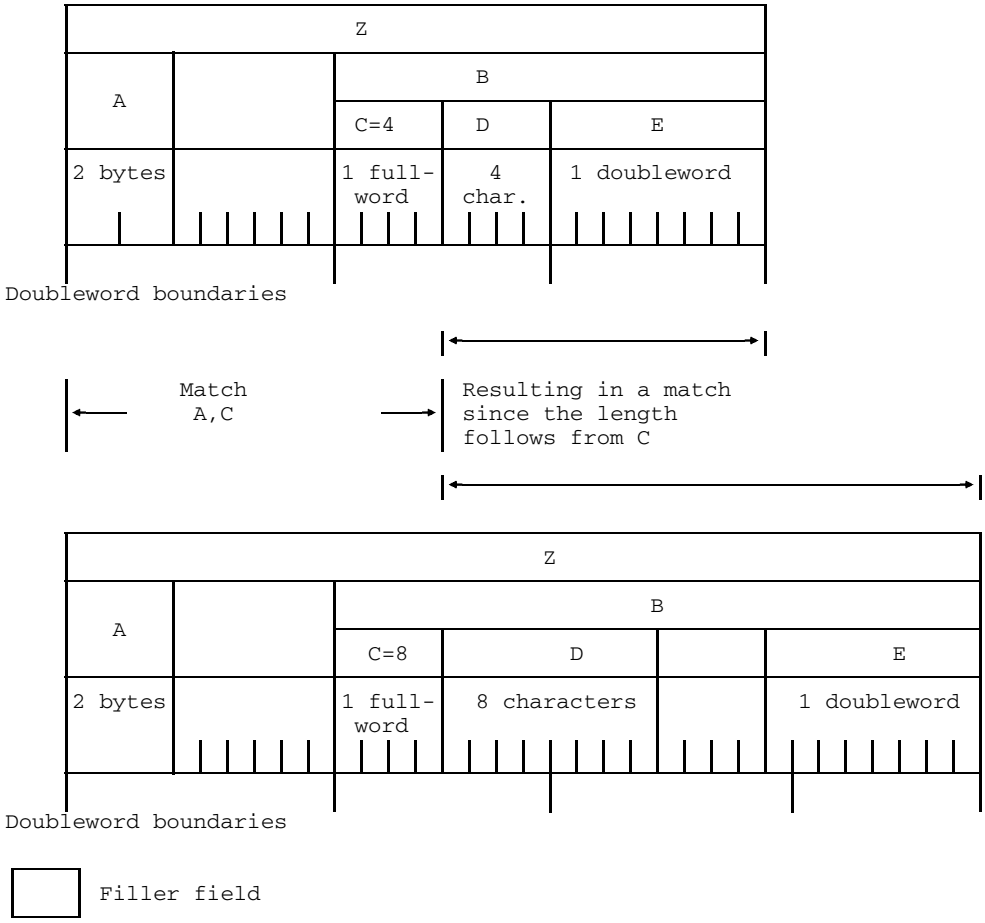


Fig. 10-36 Example of a self-defining structure

When records are output, a self-defining structure can give rise to records of varying lengths. The length of each record is calculated from the current values when WRITE or LOCATE is executed.

When records are entered using READ SET, every record can be referenced with the same variable that was used to write the record, independently of its current length. For input using READ INTO, the variable must be allocated with sufficient storage space to accommodate the longest record. Then, the RECORD condition may occur for short records, which can be ignored with "ON RECORD FILE(a);" allowing the variable to be accessed in a correct manner.

10.5.5 Record

When records are output via WRITE FROM, a copy of the contents of the storage location is output as a record. The record length is always a multiple of a byte, and results from the specifications made in sections 10.1 to 10.5. Filler information may only occur at the end of structure if the last element is an array, a scalar bit string with the UNALIGNED attribute or an area (AREA), and this element itself contains filler information.

A record written in this way has no explicit description. It can, however, be assumed that implicitly it has the same description as the variable from which it is derived.

In the case of record input with READ INTO, a copy of the record is stored in the storage location of the target variable, regardless of the data description of the target variable. This may result in the RECORD condition. In an access operation the data description of the variable is virtually overlaid on the record kept in the storage location. The variable description and the description implicitly assigned to the record on generation must match according to the conventions.

The same applies to record output via LOCATE and record input via READ SET, although in this case the storage area of the variable is located in an I/O buffer of the file. For this reason only variables with the BASED attribute can be used.

If files generated on another data processing system are processed, the internal representation may differ from that described here, and it is then necessary to ascertain the internal representation in the file and check that it matches that described here.

Warning

If substructures possessing elementary members with the attribute BIT UNALIGNED are output as records, filler information up to the byte boundary is also output, if applicable. When entered again into a similar substructure such a record is transferred into the variable by bytes, not by bits. This means that the contents of adjacent variables of the target structure can be destroyed or unintelligible information can arise.

```
DCL 1 ST,                          }
      2 A     BIT(1),              }
      2 B,                         }  Implicit
        3 C   BIT(1),              }  UNALIGNED
        3 D   BIT(1),              }
      2 E     BIT(1);              }
```

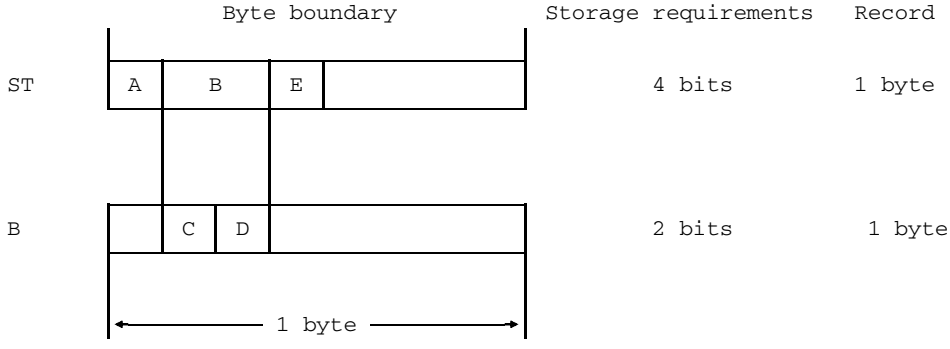


Fig. 10-37 Example of a substructure record with BIT UNALIGNED

If, in the example in Fig. 10-37, a record was written with WRITE FROM (ST.B) and entered again with READ INTO (ST.B), then the variables ST.A and ST.E would be overwritten and the current contents destroyed.

If the record was entered into a main structure with the same structure as ST.B, the access operation would produce invalid information. The same would apply if the target variable was scalar and declared with BIT (2).

It should be observed that such substructures can also occur via overlay defining in conjunction with DEFINED, BASED or PARAMETER. The case described above can also occur with subscripted variables.

10.6 Description of the data type

The data description is determined by the data attributes. In some cases the data description is required at runtime for internal processing and it is therefore stored in an internal format. The structure of this data description is described in section 10.6.1.

An internal representation of the picture specification - the picture description - is sometimes required for the data type declared with the PICTURE attribute. Its structure is described in section 10.6.2.

10.6.1 Data description

The internal representation of the data description can be of significance when parameters are passed. It consists of:

- a header word, for scalar variables
- a header word followed by 3 further words per dimension, for arrays
- a data description for the main structure, for substructures and for the elements, in the sequence in which they were entered in the source program, where structures are concerned. It should be noted here that the DIMENSION option is passed on to lower levels which then behave as arrays.

The format of a header word is illustrated in Fig. 10-38. The individual fields signify the following:

- Type BIT (7)
  The values should be taken from the table in Fig. 10-39. Further subdivision of the header words depends on the type.
- u BIT (1)
  This bit indicates whether ALIGNED ('0'B) or UNALIGNED ('1'B) is specified for the data type.
- Dimension BIT (8)
  This field indicates the number of dimensions declared. If DIMENSION is not specified, then the value of this field is (8)'0'B.
- Length specification
  The format and contents are dependent on the type. Specifications for this are given in Fig. 10-39. The values are the same as those specified in the source program, i.e.
  - precision g
  - scaling k; leftmost bit = sign bit (Binary complement)
  - length or maximum length
  - number of immediate members of a structure

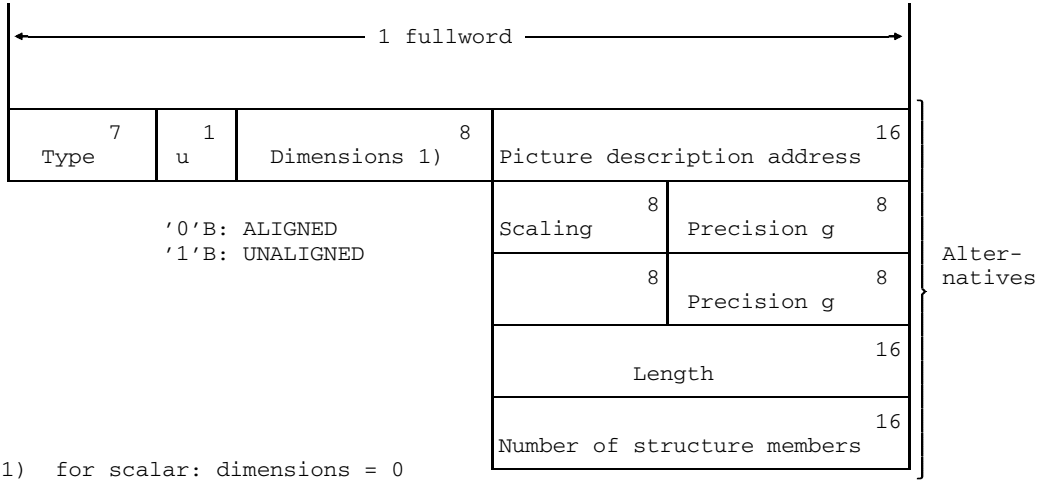


Fig. 10-38 Data description

For type 58, another fullword follows, which contains the address of the picture description (see section 10.6.2).

Type with u (hex.)		Attribute			PRECISION	Length specification	
ALIGNED	UNAL						
00	01	PICTURE expanded (see also Type = 3A or 3B)					
04	05	BINARY	REAL	FIXED	2 bytes	k	g
08	09				4 bytes		
0A	0B			FLOAT	4 bytes		g
0C	0D				8 bytes		
0E	0F				16 bytes		
12	13		COMPLEX	FIXED	2 x 2 bytes	k	g
16	17				2 x 4 bytes		
18	19			FLOAT	2 x 4 bytes		g
1A	1B				2 x 8 bytes		
1C	1D				2 x 16 bytes		

Fig. 10-39 Data types in the data description (Part 1)

Type with u (hex.)		Attribute			PRECISION	Length specification	
ALIGNED	UNAL						
1E	1F	DECIMAL	REAL	FIXED		k	g
26	27			FLOAT	4 bytes		g
28	29				8 bytes		
2A	2B				16 bytes		
2C	2D		COMPLEX	FIXED		k	g
34	35			FLOAT	2 x 4 bytes		g
36	37				2 x 8 bytes		
38	39				2 x 16 bytes		
3A	3B	PICTURE (see also Type = 00 or 01)				Characters	
3C	3D	CHARACTER		NONVARYING		Characters	
3E	3F			VARYING		max. char. 1)	
40	41	BIT		NONVARYING		Bits	
42	43			VARYING		max. bits 1)	
44	45	POINTER				0	
48	49	OFFSET					
4C	4D	LABEL					
4E	4F	FORMAT					
50	51	ENTRY					
54	55	FILE					
56	57	AREA				Characters	
58	59	STRUCTURE				Members	

1) without header word (length specification)

k with sign

Fig. 10-39 Data types in the data description (Part 2)

If a dimension is specified for an item, the number of dimensions is specified in the "Dimension" field. Three further words, which contain the specification below, follow each dimension:

- lower bound according to specification in source program
- upper bound according to specification in source program
- address spacing between elements:

```
for BIT NONVAR UNAL:    spacing in bits
otherwise:              spacing in bytes.
```

The format is shown in Fig. 10-40. If an array contains a structure, this specification is made for the structure and its members. See also the example in Fig. 10-41.

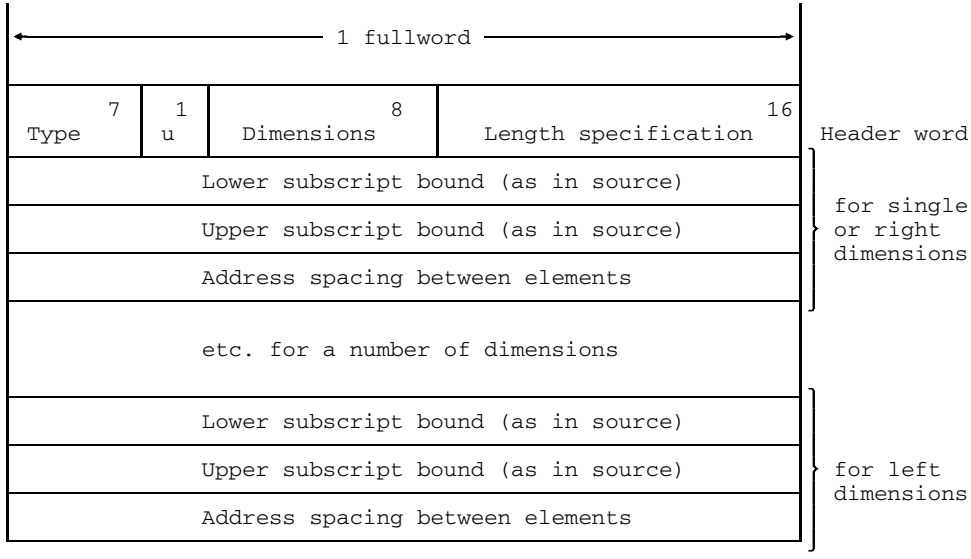


Fig. 10-40 Data description for arrays

```
DCL  1 Main,

       2 Sub 1,
       2 Sub 2  DIM (3,4),

         3 Element,

       2 Sub 3;
```

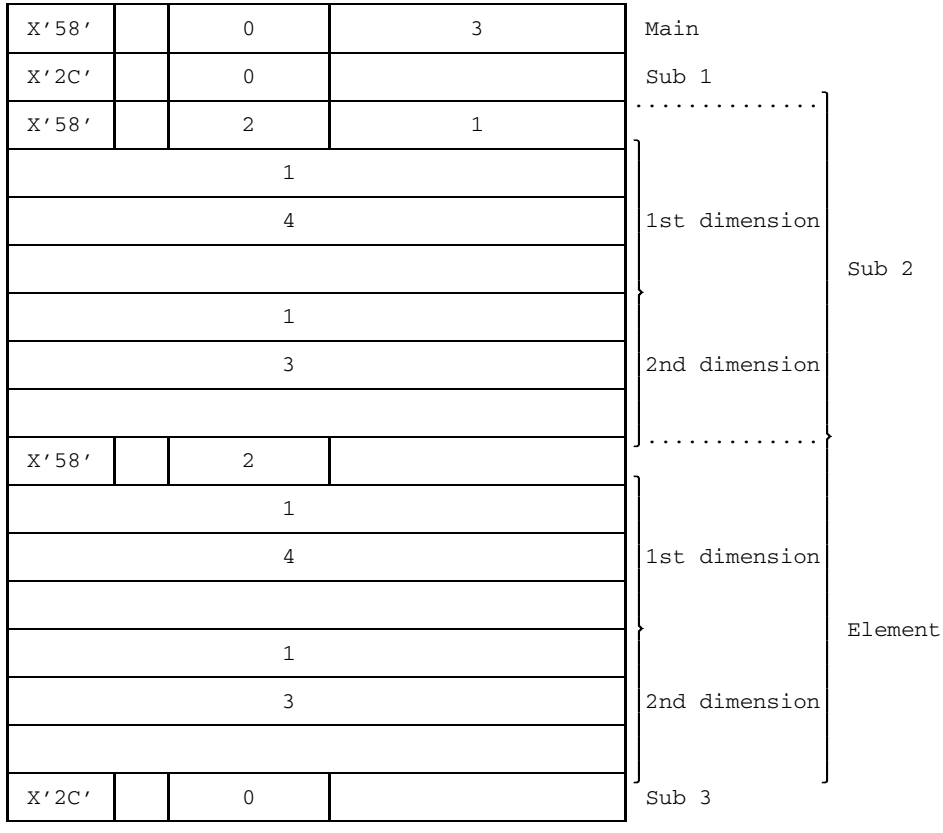
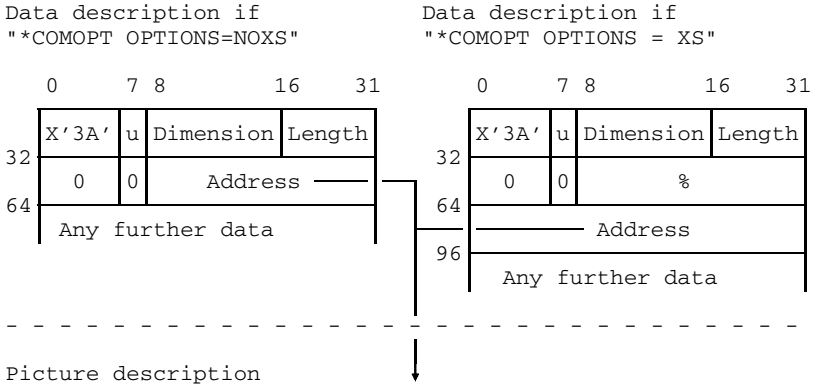


Fig. 10-41 Example showing how the data description of a structure is represented

10.6.2 Picture description

Type 58 in the data description indicates that the address of a picture description follows in the next fullword if "*COMOPT OPTIONS = NOXS" and in the second fullword if "*COMOPT OPTIONS = XS". The structure of a picture description is illustrated in Fig. 10-42. The elements signify the following:




```
DCL 1  Picture description UNALIGNED,
       2  Data type         BIT (7),
       2  Alignment         BIT (1),
       2  Picture type      BIT (8),
       2  Scaling           BIT (8),
       2  Picture length    FIXED BIN (15,0),
       2  Length            BIT (16),
       2  Storage length    BIT (16),
       2  Drifting character,
          3  Mantissa       BIT (8),
          3  Exponent       BIT (8),
       2  Picture           CHAR (x REFER Picture length);
```

Fig. 10-42 Data description for data type 0 and the associated picture description

- Data Type
  The attribute list associated with the picture is represented here:

```
15  REAL DECIMAL FIXED
19  REAL DECIMAL FLOAT where g: 1 to 6
20  REAL DECIMAL FLOAT where g: 7 to 16
21  REAL DECIMAL FLOAT where g: 17 to 33
22  COMPLEX DECIMAL FIXED
26  COMPLEX DECIMAL FLOAT where g: 1 to 6
27  COMPLEX DECIMAL FLOAT where g: 7 to 16
28  COMPLEX DECIMAL FLOAT where g: 17 to 33
30  CHARACTER NONVARYING
```

- Alignment

```
'0'B: ALIGNED
'1'B: UNALIGNED
```

- Picture type

  X'01': Picture characters 9 V S + - $
  X'02': Picture characters as X'01' and Z * , . / B
  X'03': Picture characters as X'02' and S... +... -... $...
  X'04': Picture character X only

- Scaling
  Scaling resulting from the digit positions to the right of picture character V and option F(n)

- Picture length
  Length of the picture after resolution of the factors; however, DB and CR both count as one character.

- Length
  Length of precision given by the attribute list associated with the picture.

- Storage length
  Length of the internal representation of the variable in number of characters.

- Drifting characters
  When set ('1'B), the bits in the Mantissa and Exponent field have the following meaning (bit 1 = leftmost bit):

  Bit 1: not used
  Bit 2: S... present
  Bit 3: +... present
  Bit 4: -... present
  Bit 5: $... present
  Bit 6: only Z or * present
  Bit 7: * present
  Bit 8: not used

- Picture
  This field contains the picture after the factors have been resolved. The length of this field is given in the picture length field. The picture characters are represented in coded form.

  Numeric pictures:

```
(t) terminal:    last character in drifting portion
(d) drifting:    character in drifting portion
(s) static  :    non-drifting (static) character

X'00'  9      X'54'     $(t)      X'30'     S(t)
X'04'  Y      X'58'     $(d)      X'34'     S(d)
X'08'  Z      X'5C'     $(s)      X'38'     S(s)
X'0C'  *      X'60'     /(t)      X'3C'     +(t)
X'10'  E      X'64'     /(d)      X'40'     +(d)
X'14'  K      X'68'     /(s)      X'44'     +(s)
X'18'  T      X'6C'     .(t)      X'48'     -(t)
X'1C'  I      X'70'     .(d)      X'4C'     -(d)
X'20'  R      X'74'     .(s)      X'50'     -(s)
X'24'  CR     X'78'     ,(t)      X'84'     V
X'28'  DB     X'7C'     ,(d)
X'2C'  B      X'80'     ,(s)
```

  Alphanumeric pictures:

```
X'00'    A      X'04'     9         X'08'     X
```

10.7 Storage management

The variables declared explicitly or implicitly in a source program require storage space. The storage attribute determines when and in which storage area they are to be assigned storage space.

The internal structure and the management of the various storage areas are described in the following subsections.

Storage class		Allocation	Stack	Release at	Stack
STATIC	EXTERNAL / INTERNAL	Start of program		End of program	
AUTOMATIC	INTERNAL	Block activation	+1	Block deactivation	-1
CONTROLLED	EXTERNAL / INTERNAL	ALLOCATE	+1	FREE	-1
BASED(z)	INTERNAL	ALLOCATE		FREE	
		ALLOCATE IN area		FREE IN area	
				FREE area	
		LOCATE SET (x)		LOCATE WRITE, CLOSE	
		READ SET (x)		READ,CLOSE, REWRITE	
PARAMETER	INTERNAL	Invocation		Return	
DEFINED(x)	INTERNAL	same as x			
MEMBER	INTERNAL	same as main structure			

Fig. 10-43 Summary of the allocation and release of storage space

10.7.1 Static variables (STATIC)

Variables with the STATIC attribute are assigned their storage area at compile time. Special storage management is not necessary in this case.

10.7.2 Activation records (stack, AUTOMATIC)

The storage area for activation records (stack) takes on one activation record for each block activation. This contains all the items which must be stored for managing a block activation. The storage space is assigned automatically on activation of a new block, without control by the user. If a return branch is made from the block, the storage space is released. The structure of the storage area is shown in Fig. 10-44.

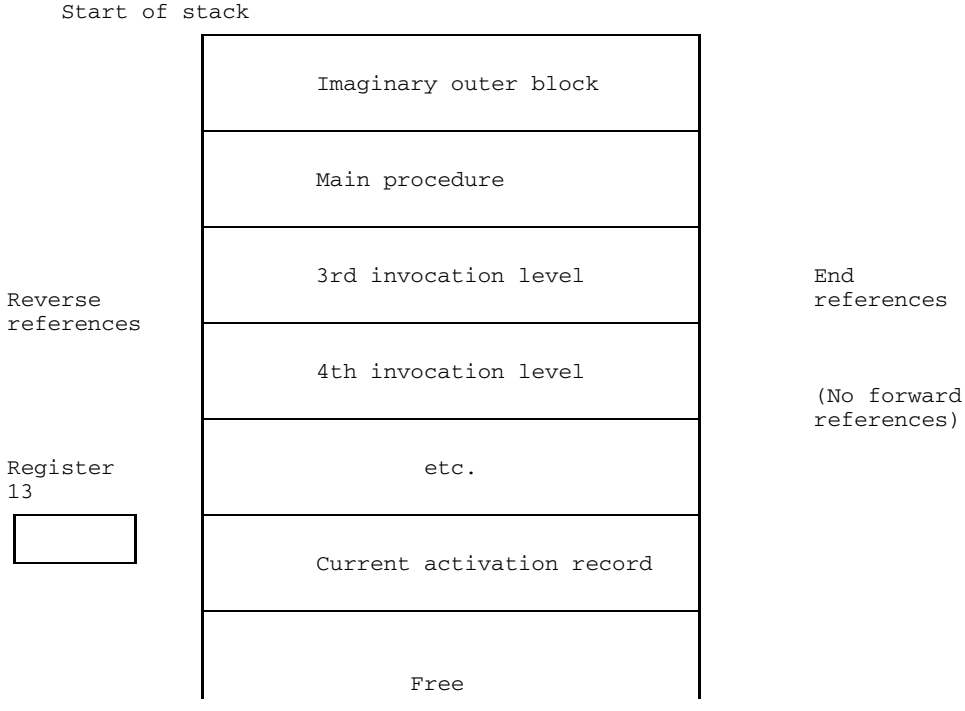


Fig. 10-44 Structure of the storage area for block activations

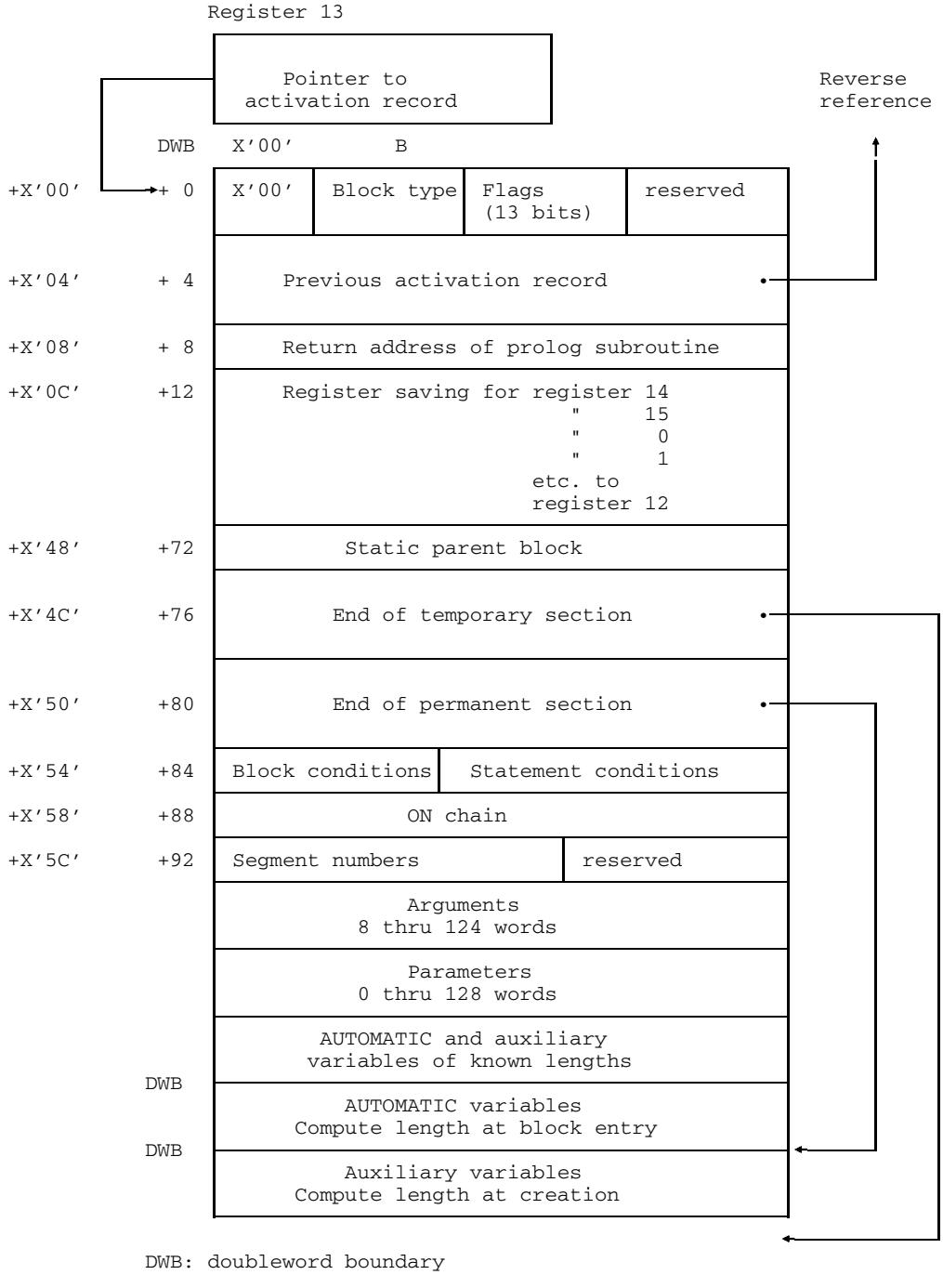


Fig. 10-45 Activation record of a block

An activation record for PL/I block is structured as shown in Fig. 10-45. It always begins on a doubleword boundary. The meaning of the fields is as follows:

- Block Type

  X'01': external procedure or PL/I-compatible assembler procedure
  X'02': internal procedure
  X'03': BEGIN block
  X'04': ON unit

- Flags
  A set bit ('1'B) signifies the following (bit 1 = leftmost bit):

  Bit 1: initialization activation of the PL/I runtime system monitor
  Bit 2: library procedure (OPTIONS (LIBRARY))
  Bit 3: activation due to a condition
  Bit 4: the block contains ON units
  Bit 5: the block contains a statement with a condition prefix
  Bit 6: activations with special save area (e.g. error handling)
  Bit 7: initial calling of conversion routines
  Bit 8: a list of statement names is available
  Bit 9: *COMOPT OPTION=ISO was used in compilation
  Bit 10: *COMOPT OPTIMIZE=ENABLING was used in compilation
  Bits 11-13 reserved

- Predecessor Activation Record
  The address indicates the activation record of the dynamically preceding block.

- Return Address for Prolog Subroutine
  For the initialization of AUTOMATIC variables if a number of entries are present.

- Register Saving
  The values of registers 14 and 15 and 0 thru 12, existing when the successor was called, are stored by the dynamically following block.

- Static Parent Block
  The address refers to the activation of the block which is in the source program the statically superordinate block (for accessing items declared there).

- End of Temporary Section
  The address refers to the word following the activation record, which is the next free word or the next activation record. Any temporary variables (auxiliary variables) present are included.

  If "*COMOPT OPTIONS = NOXS" was used in compilation the left byte will contain the the segment number of the temporary section.

- End of Permanent Section
  As for "temporary end", but the temporary variables are excluded. If no temporary variables are present, then "temporary end" and "permanent end" contain identical addresses.

  If "*COMOPT OPTIONS = NOXS" was used in compilation the left byte will contain the segment number of the permanent section.

- Block Conditions
  This field is only relevant if the Flags field (+2) contains in bit 5 = ’1’B. A set bit (’1’B) indicates that the condition specified for the block is enabled (bit 1 = leftmost bit):

Bit 1 to 4:	reserved
Bit 5:	UNDERFLOW
Bit 6:	OVERFLOW
Bit 7:	ZERODIVIDE
Bit 8:	FIXEDOVERFLOW
Bit 9:	CONVERSION
Bit 10:	SIZE
Bit 11:	SUBSCRIPTRANGE
Bit 12:	STRINGRANGE
Bit 13:	STRINGSIZE
Bit 14 to 16:	reserved

- Statement Conditions
  This field contains the conditions enabled for the current statement. On entry of a block it contains the same values as the "Block Conditions" field. If explicit options specify deviations for the statement, this field is changed to the appropriate state for the execution of the statement and restored to its original state afterwards.

- ON Chain
  This field is only relevant if in the Flags field (+2) bit 4 = ’1’B. It contains a pointer to the chain of management elements for ON units.

- Segment numbers
  This field is only relevant if "*COMOPT OPTIONS = XS" was used in compilation. Otherwise this field is reserved (bit 1 = leftmost bit):

  Bits 1-8: Segment number of the temporary section
  Bits 9-16: Segment number of the permanent section
  Bits 17-24: reserved

- Arguments
  Any parameter options for invoking procedures are entered in this field. The invocation is described under "Passing of Parameters", and the length is determined by the invocation with the most extensive argument option. To cover implicit invocations, this field has a minimum length of 8 words and a maximum length of 124 words. (Registers 1 thru 4 are used for the first four arguments).

- Parameters
  This field is only available if parameters are present for the block. The options in this field correspond to those in the preceding "Argument" field, but the first four parameters are also contained in this field.

- AUTOMATIC Variables and Auxiliary Variables of Known Length
  The storage locations for the AUTOMATIC variables of the block and auxiliary variables required in the block are located here if their lengths are known at compile time. Generally the AUTOMATIC variables are allocated first and then the auxiliary variables, but for optimization purposes they may be allocated differently.

  Furthermore, this field contains pointers to the AUTOMATIC and auxiliary variables located in the two following areas.

- AUTOMATIC Variables
  This field contains the storage locations for AUTOMATIC variables whose length cannot be ascertained until the block is entered. Pointers to these variables are stored with the AUTOMATIC variables of known length.

- Auxiliary Variables
  This field contains the storage locations for auxiliary variables whose length cannot be ascertained until creation. Pointers to these variables are stored with the auxiliary variables of known length.

10.7.3 Standard area (CONTROLLED, BASED)

The standard area accepts the following items:

- Variables with the CONTROLLED attribute
- Variables with the BASED attribute if these are not assigned to a named area (AREA)
- Auxiliary items which are allocated storage space during the object run, e.g. input/output buffers.

The allocation of storage for variables with the CONTROLLED or BASED attributes is controlled exclusively by the user via the ALLOCATE and FREE statements. Released storage may be located between reserved space. If long enough, such portions of free storage gaps may be reserved in a subsequent allocation.

The standard area consists of at least one initial portion defined at the beginning of the program. The first doubleword is undefined. The next 4 doublewords accept management information for the standard area, and the remaining portion is available for allocations.

If more storage is required, further portions are requested automatically. The first two fullwords of each extension portion contain the absolute addresses of the beginning and end of the predecessor portion. See Figure 10-46.

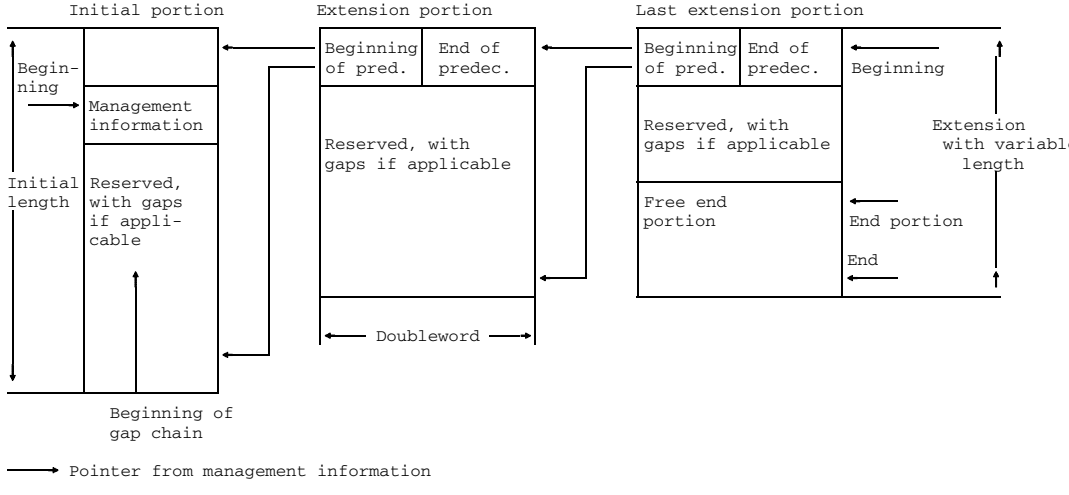


Fig. 10-46 Principle of the structure of the standard area

The storage of all portions, without the first doubleword of each portion and without the management information, together form the storage available in the standard area for allocations. Once allocated, portions remain so until the end of the object run.

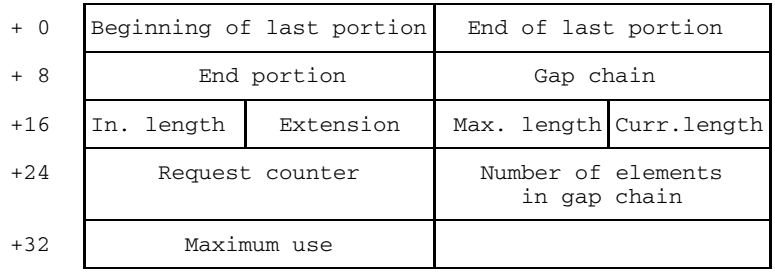


Fig. 10-47 Structure of management information in the standard area

Management information in the standard area has the following meaning (see Figure 10-47):

- Beginning and End of the Last Portion
  Absolute addresses of the last portion to be assigned. If only the first portion is present, it is this.
- End portion
  Absolute address of storage as yet unreserved at the end of the standard area. The end portion does not form part of the gap chain and can begin in any portion.
- Gap Chain
  Absolute address of the first gap in the gap chain. The gap chain is always arranged by ascending addresses, consolidating adjacent gaps where applicable.

  The length of a gap is always a multiple of a doubleword (8 bytes). It is structured as follows:

  – first fullword: absolute address of the next gap or ’0’B
  – second fullword: length of the gap in bytes, including the above two fullwords of management information for this gap
  – remainder: undefined
- Initial length, Extension, Max. length
  These specifications are taken from the control statement "*RUNOPT STORAGE = AREA (initial length, extension, max. length)" and have the following meaning:

initial length: number of pages desired for the initial portion
extension: number of pages desired for an extension
max. length: maximum length.

The value specified for "Extension" can be exceeded if more storage is currently required and can be reduced if less storage is available and is currently sufficient.

- Request Counter
  Each time a new portion is requested this counter is incremented by 1. If the control statement "*RUNOPT LIST = SUMMARY" is specified, this value is output at the end of the program.
- Number of Elements in the Gap Chain
  The number of elements in the gap chain is retained here, and is used for error-checking purposes.
- Maximum use
  Maximum standard area length used in pages (4K bytes). If the "*RUNOPT LIST = SUMMARY" control statement is specified, this value is displayed at the end of the program, under "STANDARD-AREA:".

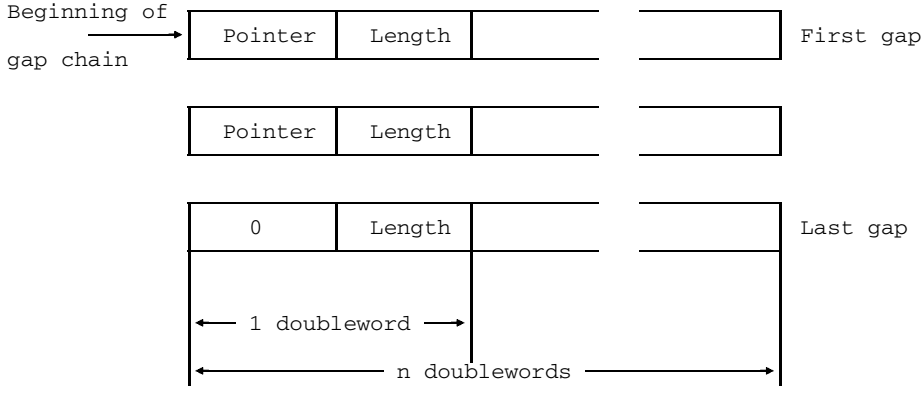


Fig. 10-48 Principle of the structure of the gap chain

Free storage located at the end of the standard area is designated the end portion and forms the end gap. It is not part of the gap chain, and its start address is contained in the management information.

If the ALLOCATE statement is to be used to allocate storage in the standard area, the following steps are attempted, in the order shown:

1. If there is still storage in the end portion, the allocation can be made from this.

2. If less than 3/4 of the storage space has been allocated to the standard area out of the maximum amount provided, a further portion containing the number of pages specified in "Extension" is requested. If 3/4 of the standard area has been allocated, the system checks for the presence of a gap chain.
3. If a gap chain is present, a check is made to see whether the storage request can be satisfied from the gap chain, sorted by ascending addresses. If a sufficiently large gap is found, storage is allocated. If the gap is exactly as requested, it is removed from the gap list; if it is larger, it is reduced accordingly.
4. If there is no gap chain, another portion containing the number of pages as specified by "Extension" is requested. If this is impossible, the number of pages required for the allocation is requested. If any of these attempts succeeds, space is allocated from the now larger end portion.
5. If all attempts are unsuccessful, the STORAGE condition is set.

If storage space in the standard area is to be freed via the FREE statement, processing proceeds as follows:

1. Storage space and end portion, when adjacent, are combined into one.
2. The released storage space is inserted, in sorted form, as a gap into the gap chain and united with adjacent gaps where applicable.

By means of the ADUMP function the contents of the standard area can be printed out (see chapter 11).

The "*RUNOPT LIST = SUMMARY" control statement displays, at the end of the program run, the maximum used length (x) of the standard area in pages and number of portions (y):

```
STANDARD AREA: x PAGES; SYSTEM CALLS: y REQM
```

10.7.4 Named area (AREA)

A named area is an area variable declared by the user with the AREA attribute. At the same time the storage attribute is used to declare the storage area in which this area is to be located.

Via the ALLOCATE statement, variables with the BASED attribute can be assigned storage space in an area. The storage space can also be an area, which can result in area nesting.

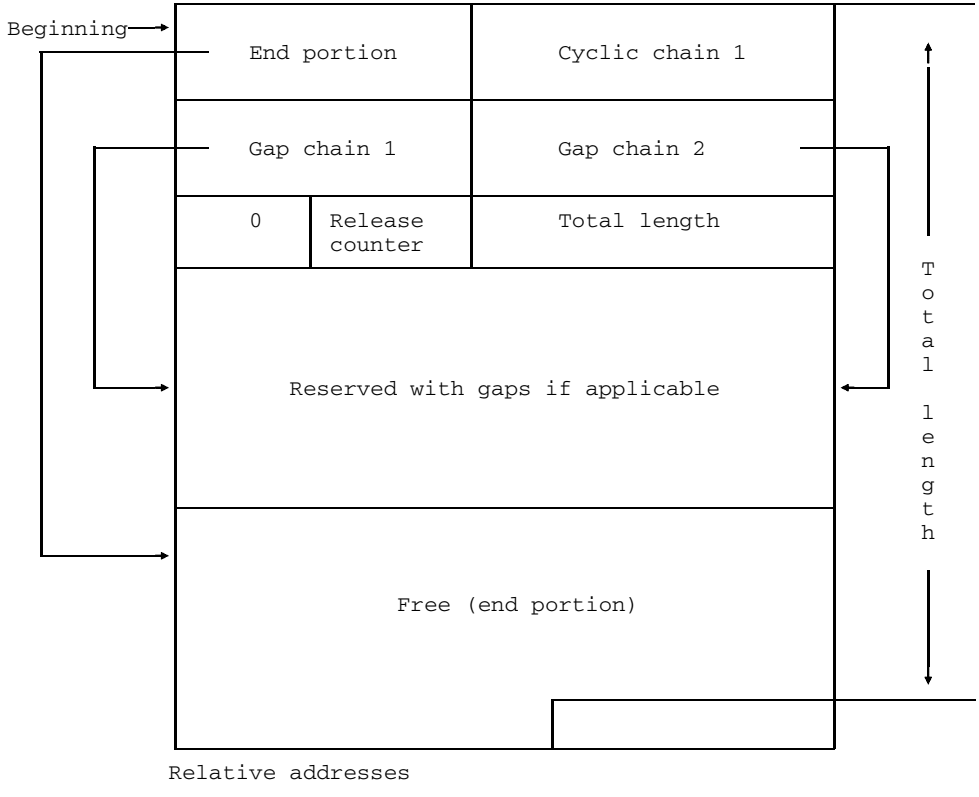


Fig. 10-49 Structure of a named area

The length of an area may be ascertained from the length specification in the AREA(n) attribute in bytes, plus 24 bytes for management information. Storage is always allocated in multiples of a doubleword. The fields of the area have the following meaning (see Figure 10-49):

- End Portion
  Relative start address of the end portion
- Cyclic Chain 1
  Searching of gap chain 1 does not commence at the beginning, but at the last position where storage was allocated. Thus the address of the predecessor to this gap is retained and the chain searched cyclically. This represents an optimization of the search process.
- Gap Chain 1
  Relative start address of gap chain 1, which contains all gaps not included in gap chain 2, and is sorted in ascending order of addresses.
- Gap Chain 2
  Relative start address of gap chain 2 which contains the last gap to occur and other gaps of the same length, if applicable. The first gap is the most recent gap.
- Release Counter
  This counter is incremented by 1 each time storage is freed. If an AREA condition occurs the count is retained, and after the return from the AREA condition a check is performed to ascertain whether the counter has been incremented (i.e. whether storage has been freed). If the count is unchanged the ERROR condition (prevention of endless loops) is raised.
- Total Length
  The total length is n + 24 bytes, where n is the specification from the AREA attribute.
- Remaining Area
  The remainder of the area is available for storage allocation to variables, and can be made up of a reserved and a free section. The reserved section may contain gaps. The free section (end portion) does not form part of a gap chain.

Storage is always allocated in multiples of 1 doubleword, which means gaps are also doubleword multiples. Each gap is contained in either gap chain 1 or gap chain 2, and all the gaps in a gap chain are linked via forward references. The address of the first gap (the anchor) in a chain is located in the management information. The gaps have the same structure as described in section 10.7.3 for the standard area.

The free storage space located at the end of the area is designated the end portion and forms the end gap. It does not belong to a gap chain, and its start address is in the management information.

All the addresses in the management information and in the gaps are relative to the beginning of the area. In this way entire areas can be allocated to other areas and buffered in files as records.

If a variable is to be allocated storage space in an area (only via the statement ALLOCATE IN (area)), processing proceeds in the following sequence:

1. If gap chain 2 is not empty, a check is performed to ascertain whether the gap position corresponds to the required length (all gaps in this chain are the same length). If they are both the same length, then the storage space of the first gap is used and this is removed from gap chain 2.

   If the lengths are not equal, gap chain 2 is resolved and the gaps are inserted in gap chain 1 in ascending order of addresses, combining adjacent gaps into one.

2. A gap equal to or greater than the length required is sought in gap chain 1. If one is found, its storage space is allocated to the variable and the gap removed or shortened.

   For optimization reasons, gap chain 1 is searched cyclically, beginning at the position at which storage space was last allocated from the gap chain. For this reasons the address of the predecessor to this gap is retained in the field "cyclic chain 1", and cyclic searching begins with the successor to this gap.

3. If the free end portion is sufficiently large, storage space is allocated there.

4. If all attempts are unsuccessful, the AREA condition is raised.

If the storage space occupied by a variable in an area is to be freed, processing proceeds as follows:

1. If the storage space borders on the end portion, it is incorporated in this. Gap chain 2 is resolved, as described in 4.

2. If gap chain 2 is empty, the storage space is inserted as a new gap.

3. If gap chain 2 is not empty and the length of the gaps is equal to that of the storage space to be freed, the new gap is inserted at the beginning of the gap chain.

4. If gap chain 2 is not empty and the length of the gaps does not equal that of the storage space to be freed, gap chain 2 is resolved and its gaps are inserted in gap 1 in ascending order of addresses. Adjacent gaps are combined into one.

   The new gap is then inserted in the now empty gap chain 1.

10.7.5 Reference chain for CONTROLLED variable

For a variable with the CONTROLLED attribute, storage is allocated via the ALLOCATE statement. If the statement is entered a number of times for the same variable, new storage is allocated each time; storage already allocated remains so. The storage space is stacked, and if accessed, the last allocation is always valid.

The FREE statement, however, frees the last storage space allocated. If available, the previously allocated storage space becomes the current storage, which ensures that the same amount of storage space is freed as was allocated.

At the start of the program, a STATIC variable or for CONTROLLED (PLI1GLOBAL(n)), a pseudo-register entry at the length of 4 bytes is created for each CONTROLLED variable - the "anchor". If the CONTROLLED variable has been allocated storage space, the anchor contains the absolute address of the last link in a reference chain, via which the current allocation is reached. In all other cases the anchor contains a null pointer.

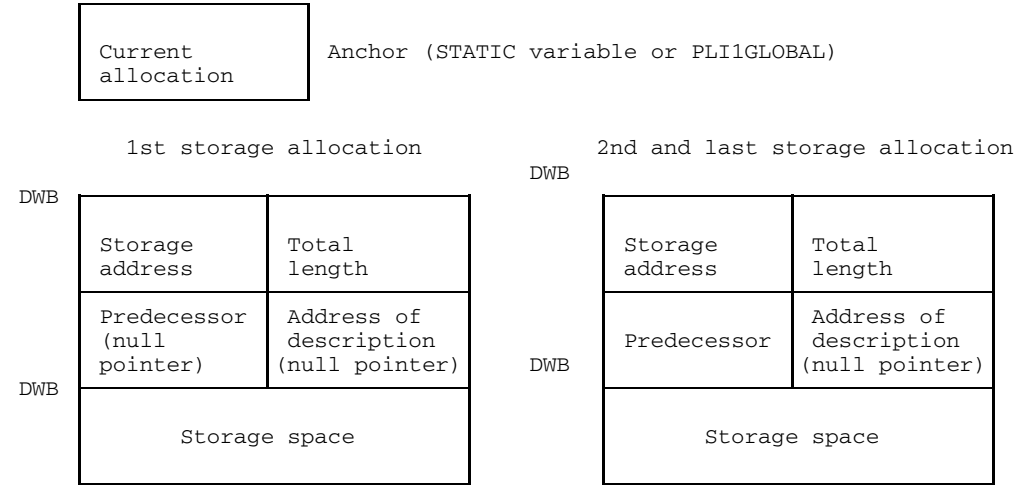
By means of the ALLOCATE statement a variable with the CONTROLLED attribute is allocated storage in the standard area, the size of which may be ascertained from the attributes. The size may vary for each allocation. If the attribute set of the CONTROLLED variable includes an option which represents a variable item (i.e. an expression), that expression will be calculated before storage is assigned. Additionally, in this case, the standard area receives a copy of the data description in which the variable items are replaced with the calculated values. They are required for every access to the CONTROLLED variables. The internal representation of the data description is explained in section 10.6. In addition a chain link with 2 doublewords is set up in the standard area, which becomes the last link of the reference chain for the controlled variable. The anchor is reset to point to the last link in the chain. A link contains the following management information:

- The first fullword contains an absolute pointer to the allocated storage space.
- The second fullword contains the total length of the allocated storage space, incl. the length of the chain link and if a data description is maintained, its length.
- An absolute pointer indicating the previous link (reverse reference) is located in the third fullword. In the first link this fullword contains a null pointer.
- The fourth fullword contains a null pointer or an absolute pointer to the data description.

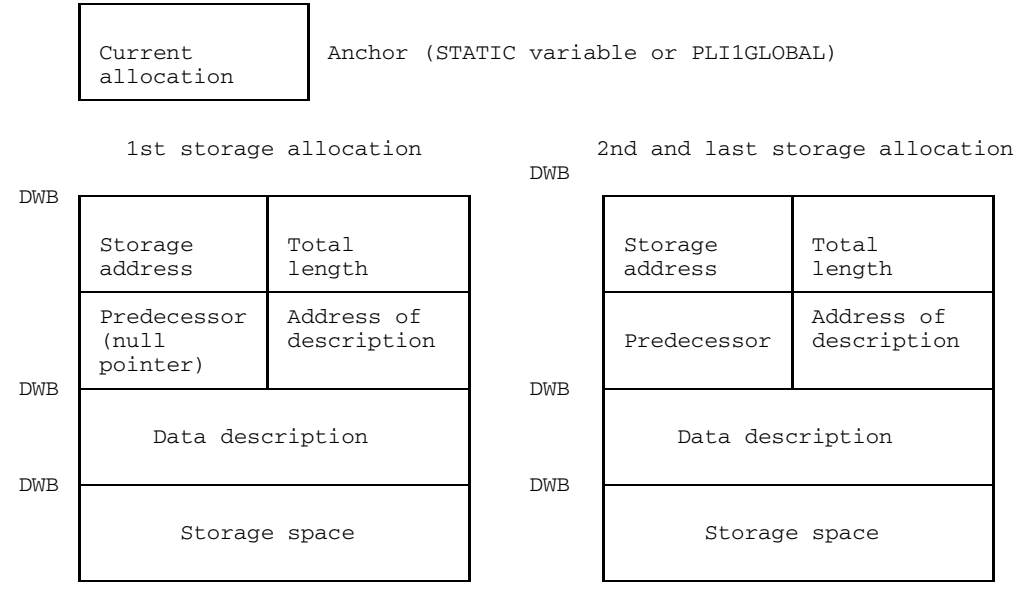
Chain link, data description, and storage space of the variable are arranged in a connected block.

Reference chain for CONTROLLED variable in the standard area (example)

Case 1: Without data description



Case 2: With data description



DWB: doubleword boundary

By means of a FREE statement the current storage space is deallocated together with the storage for the copy of the data description, if available. The length of storage space to be freed may be ascertained from the length specification retained in the link.

The anchor is set to the predecessor in the reference chain or contains a null pointer if no further predecessors are present. Storage space for the link is also released.

In the case of a reference to a CONTROLLED variable, the storage space and, if applicable, the data description are found via the anchor and the link of the reference chain.

11 Utilities

This section deals with facilities available to the user in the form of prefabricated procedures already compiled. These utilities extend beyond the scope of PL/I and cannot be expected to be equally implemented on a different system, when transferring programs.

In a PL/I program the procedures described here can be invoked by a subroutine reference (CALL) or by a function reference. The procedure is automatically incorporated into the program during linkage.

The procedures are described independently of one another and arranged in alphabetical order of their entry names. The relations between them are dealt with in separate sections.

All the attributes needed in the particular case are included in the data description. Attributes enclosed in brackets are completed by the PL/I default system; they can be omitted, so long as the default system is not overwritten by a DEFAULT statement. Possible attributes for a complete data description may be chosen according to requirements.

Note that in declarations for some of the procedures the option

```
OPTIONS (LIBRARY)
```

must be specified. It is described in chapter 7.

BS2SRTA (a, b, c) BS2SRTB (a, b, c, d) BS2SRTC (a, b, c, e) BS2SRTD (a, b, c, d, e) BS2SRT (h,..., c, d, e)	a: SORT/MERGE statement CHAR (*) b: RECORD/ALLOC statement CHAR (*) c: Acknowledgment FIXED BIN PREC (31,0) c = 0: error-free run c = 16: abnormal termination d: Input procedure ENTRY ( ) e: Output procedure ENTRY ( ) h: Control statement for SORT as from V7.0

RUNTIME	1) 2)	Runtime since start of program, in seconds
HEXDEC (a)	1) 2)	Conversion of bit string a to hexadecimal
ERROUT	1)	Error text of current ON unit to SYSOUT

TRACE (a) NOTRACE (a)	1)	Trace activate/deactivate: a: letter sequence P PROCEDURE trace C CALL trace R RETURN trace G GOTO trace L LABEL trace T also to terminal empty string ≙ 'PCRGL'
SNAP	1)	Call nesting to SYSOUT

RDUMP (a,b)	1)	b characters relative to pointer a	to SYSLST
ADUMP	1)	Standard area	Dump in hexadecimal and character representation
SDUMP	1)	Stack	

PLIRETC (a)	1)	Set monitor job variable to value a

CMD (a, b, c)	1)	Execute BS2000 command a: BS2000 command CHAR (*) b: SYSOUT listing CHAR (*) c: acknowledgment FIXED BIN (31)

1) with OPTIONS (LIBRARY)
2) Function

Fig. 11-1 Overview of procedures

ADUMP Dump from the standard area

```
ADUMP
```

Contents of standard area (CONTROLLED and BASED variables) are output to SYSLST in hexadecimal and character representation.

Input

```
DCL    ADUMP      ENTRY ( )
                  OPTIONS (LIBRARY) [EXTERNAL][CONSTANT];
```

Parameters

None

Effect

The current contents of the standard area are printed out. A detailed description of the structure of standard area appears in chapter 10.

Prior to the printout mentioned above, the pseudoregister vector (PRV) is output. Interpretation of this printout must be left to specialists.

```
DCL ADUMP ENTRY () OPTIONS (LIBRARY);

DCL FEST FIXED BIN(31,0) CONTROLLED INIT(255);

DCL ZEICHEN CHAR(20) VARYING BASED INIT((8)'4');
DCL P POINTER DIM(3);

DCL Z POINTER DIM(3);

DO I=1 TO 3;

     ALLOCATE FEST; FEST=FEST * (16**I);
     Z(1) = ADDR(FEST); PUT SKIP(2) LIST (UNSPEC(Z(I)));

     ALLOCATE ZEICHEN SET (P(I));
     PUT SKIP(2) LIST (UNSPEC(P(I))); PUT SKIP;

END;

CALL ADUMP;

FREE P(1)->ZEICHEN;
FREE FEST;
ALLOCATE FEST; FEST = FEST * (16**4);

CALL ADUMP;
```

Result:

```
****** DUMP OF THE STANDARD-AREA: ******

DUMP OF THE PRV:

REPL    VS ADR   MEMORY
        025000   000800D2  F1F7F3F3  D7D3C9C1  D3D3C740
        025020   C2C1C4E4  00000E00  00010001  00000000
        025040   00000002  00017BD4  01017484  02017754
        025060   070176B8  4F009FB2  01017484  00000002
        025080   00025CEC  00000000  00000090  7F000098
           .
           .
           .
           .
DUMP OF ALL LOGICAL STANDARD-AREA SEGMENTS:

REPL   VS ADR   MEMORY
       03E000   00000000  00000000  0003E000  0004E000
       03E020   00000001  00000000  0003E1B8  00000000
       03E040   0003E050  00000014  FFFEFFF8  FFFEFFF8
       03E060   00000000  00000000  00000000  00024008
       03E080   00000000  00000000  00000000  00000000
```

Fig. 11-2 Example of ADUMP call

BS2SRT Sort/merge

```
{BS2SRTA (a,b,c)   }
{BS2SRTB (a,b,c,d) }
{BS2SRTC (a,b,c,e) }
{BS2SRTD (a,b,c,d,e)}
```

```
BS2SRT (h1,..,hn,c,d,e)
```

Start of sort/merge program SORT (see SORT Manual [10]) without or with PL/I procedures for processing records before and/or after the sort phase.

Entries

```
DCL    BS2SRTA  ENTRY    (CHAR (*) [NONVARYING],
                          CHAR (*) [NONVARYING],
                         [REAL]FIXED BINARY PREC (31,0) [ALIGNED])
                [EXTERNAL][CONSTANT];

DCL    BS2SRTB  ENTRY    (CHAR (*) [NONVARYING],
                          CHAR (*) [NONVARYING],
                         [REAL]FIXED BINARY PREC (31,0) [ALIGNED],
                         ENTRY ( ))
                [EXTERNAL][CONSTANT];

DCL    BS2SRTC  ENTRY    (CHAR (*) [NONVARYING],
                          CHAR (*) [NONVARYING],
                         [REAL]FIXED BINARY PREC (31,0) [ALIGNED],
                         ENTRY ( ))
                [EXTERNAL][CONSTANT];

DCL    BS2SRTD  ENTRY    (CHAR (*) [NONVARYING],
                          CHAR (*) [NONVARYING],
                         [REAL]FIXED BINARY PREC (31,0) [ALIGNED],
                         ENTRY ( ),
                         ENTRY ( ))
                [EXTERNAL][CONSTANT];

DCL    BS2SRT ENTRY OPTIONS (VARIABLE);
```

Parameters

a: A character string representing a control statement SORT or MERGE for the SORT program.

b: A character string representing a control statement RECORD or ALLOC for the SORT program. If such a statement is unnecessary, a null string has to be specified.

c: Only one variable having the attributes mentioned above is permitted. The value passed on in the call is irrelevant. On return one of the following values will be passed in the variables:

c = 0: Execution of SORT program free from errors.
c = 16: SORT program aborted due to error.

d: Name of a PL/I function for processing each record prior to the sort operation; d = 0 if name is not present.

e: Name of a PL/I function for processing each record after the SORT operation; e = 0 if name is not present.

h: A character string representing a control statement for the SORT program (e.g. INCLUDE, SORT, SUM, RECORD etc.).

Files

For files to be processed by the SORT program the following link names must be declared where required:

```
Sort input file:          SORTIN or
                          SORTIN01, SORTIN02 etc. up to max. SORTIN99
Merge input files:        MERGE01, MERGE02 etc. up to max. MERGE99
Output file:              SORTOUT
```

When the LINK name SORTIN and/or SORTOUT is omitted, the records are passed to SORT or returned from it only via the appropriate PL/I function.

Where a program requires auxiliary SORT files, they can be provided by the following LINK names.

```
Workfile on disk:         SORTWK
Workfiles on tape:        SORTWK01 etc. up to max. SORTWK99
Checkpoint file:          SORTCKPT
```

If auxiliary files are needed for which LINK names have not been specified, files under the name SORTWK or SORTCKPT are automatically cataloged and created. These files will be erased at normal termination of the SORT program.

Detailed information on sorting and merging is found in the manual of the SORT program. Extracts of the most important elements in the control statements SORT, MERGE and RECORD are shown in Figures 11-3 and 11-4.

For examples see chapter 13.5.

```
SORT statement  ::= SORT FIELDS=({(Start,Length,Sequence,[,Format])},...),

MERGE statement ::= MERGE FIELDS=({Start,Length,Sequence,[,Format]},...),

Start           ::= Character position [.bit position]

Length          ::= Number of characters "max. 256 characters"

Character       ::= "Start of the sort field; 1 - 4096 characters"
position
Bit position    ::= "Bit position within characters; 0 - 7 bits;
                     for BI format only"

Number of       ::= "Length of the sort field; 1 - 256 characters"
characters
Number of bits  ::= "0 - 7; for format = BI only" "Remainder length of the
                     sort field; 0 - 7 bits; for BI format only"

                    [A "ascending"        ]
Sequence        ::= {D "descending"       }
                    [N "Remainder field"  ]

                    [BI "Binary"                                  ]
                    [CH "Character"                               ]
                    [SP "Special character"                       ]
                    [AA "ISO -> EBCDIC;sorting; -> ISO"           ]
                    [AE "ISO -> EBCDIC;sorting; -> EBCDIC"        ]
Format          ::= {EE "EBCDIC -> ISO;sorting; -> EBCDIC"      }
                    [EA "EBCDIC -> ISO;sorting; -> ISO"           ]
                    [FI "Fixed point"                             ]
                    [FL "Floating-point"                          ]
                    [PD "Packed decimal"                          ]
                    [ZD "Zoned decimal"                           ]
```

Fig. 11-3 Extract from syntax for control statements SORT and MERGE of program SORT

```
RECORD statement ::= RECORD LENGTH (length,...),TYPE=Type

Length           ::= "Integer"

                     ⎧F  "fixed record length"   ⎫
Type             ::= ⎨                           ⎬
                     ⎩V  "variable record length"⎭
```

	Presetting	
1st length: max. length of input record	-	
2nd length: max. length of sort/merge record	1st length	
3rd length: max. length of output record	2nd length	
4th length: min. length of sort/merge record	length of control field	only for TYPE = V
5th length: most frequently occurring length of sort/merge record	(2nd length + 4th length)/2	only for TYPE = V

Fig. 11-4 Syntax for the RECORD control statement of the SORT program (abbreviated version)

PL/I record processing functions

The following conditions apply to the functions for processing records:

Entry

Identifier: PROCEDURE (s,r)
RETURNS (CHAR (*) [NONVARYING]);
or appropriate internal function;

The sort program must be informed about the entry via parameters (’d’ and ’e’ of the BS2SRT call); the entry will be called once for each record, before or after the sort/merge process.

Parameters

```
DCL s CHAR (length) NONVARYING UNALIGNED PARAMETER;
```

The length must be a constant. Depending on parameter ’r’, parameter ’s’ contains a record or an undefined value.

```
DCL r REAL FIXED BINARY PREC (31,0) ALIGNED PARAMETER
```

On invocation one of the following values is passed via r.

r = 0: Record is passed to s. If no input file is specified, s is undefined. Interrogation possible through:

```
IF ADDR(s) = NULL( )
THEN no input file
ELSE input file
```

r = 4: Record is passed to s. Another record with the same sortfield is present (only possible after the sort operation).

r = 8: End of file; value of parameter s undefined.

On return one of the following values can be passed in r:

r = 0: Record is passed.

r = 4: Further processing of the record by program SORT not desired; passed value undefined. This value must be specified when the sorted records are output by the PL/1 function itself.

r = 8: End of processing.

r = 12: Additional record to be inserted prior to the passed record. The passed record, if any, will be supplied again by SORT. This value must be specified when the PL/1 function itself inputs the records to be sorted.

Result

The result returned by the function to SORT contains a record or an undefined value depending on the value returned via parameter r.

r = 0: Unmodified or modified record

r = 4: Value undefined

r = 8: Value undefined

r = 12: Additional record

CMD Execute BS2000 command

```
CMD (a, b, c)
```

Execution of a BS2000 command

Entry

```
DCL CMD ENTRY (CHAR(*) [NONVARYING] [UNALIGNED],
               CHAR(*) [NONVARYING] [UNALIGNED],
               [REAL] FIXED BINARY PREC (31,0) [ALIGNED)]
        OPTIONS(LIBRARY) [EXTERNAL] [CONSTANT];
```

Parameters

a: A character string which contains the BS2000 command to be executed. The maximum length of the string is 32763 characters.

b: A character string into which the SYSOUT listing of the BS2000 command is entered. The first 4 bytes of each record in the listing contains its record length field (bytes 0-1, bytes 2-3 are reserved). The records are entered sequentially into the character string. The maximum length is 32763. If the length = 0, the listing is output to SYSOUT.

c: A fixed-point number into which the acknowledgment of the execution of the command is entered. The field can have one of the following values:

c = 0: normal termination
c = 4: memory shortage; no request
c = 8: error in the operand list
c = 12: the list area is too small for the SYSOUT listing
c = 16: command error
c = 20: invalid command

Effect

The specified BS2000 command is executed by means of the system macro CMD. The listing is either entered in the list area or output to SYSOUT.

Commands which cannot be executed or which terminate the program are listed in the description of the system macro CMD in [16].

```
DCL CMD ENTRY(CHAR(*), CHAR(*), FIXED BIN(31))
        OPTIONS(LIB);
    LIST CHAR(1000),RET_CODE FIXED BIN(31);

/* THE LISTING IS ENTERED IN THE LIST AREA */
CALL CMD ('FSTATUS',LIST,RET_CODE);

/* THE LISTING IS OUTPUT TO SYSOUT */
CALL CMD ('FSTATUS','',RET_CODE);
```

Fig. 11-4a Example of using CMD

ERROUT Error text output

```
ERROUT
```

Output of current error text to SYSOUT

Entry

```
DCL ERROUT ENTRY ( ) OPTIONS (LIBRARY) [EXTERNAL][CONSTANT];
```

Parameters

None

Action

If, as a consequence of raising a condition, an ON unit was executed, the corresponding error text is output by CALL ERROUT.

The error text referred to is the same as that which would have been output by the system unit, if no ON unit had been present.

```
DCL ERROUT  ENTRY() OPTIONS(LIBRARY);

ON AREA  BEGIN;
         CALL ERROUT;
         IF ONCODE() = 360 THEN GOTO ALLOC_FEHLER;
         IF ONCODE() = 361 THEN GOTO ASSIGN_FEHLER;
         IF ONCODE() = 362 THEN GOTO SIGNAL_FEHLER;
         END;
```

Output from ERROUT

```
*****AREA-CONDITION, ONCODE=0362 IN LINE 15 IN STATEMENT 1
INSUFFICIENT SPACE IN A NAMED AREA SIGNALED
```

Fig. 11-5 Example of using ERROUT

HEXDEC (a) Hexadecimal characters

```
HEXDEC (a)
```

The bit-string a is converted to a character string of hexadecimal characters.

Entry

```
DCL HEXDEC ENTRY (BIT (*) [NONVARYING] [ALIGNED])
           RETURNS (CHAR (*) [NONVARYING] [UNALIGNED])
           OPTIONS (LIBRARY) [EXTERNAL] [CONSTANT];
```

Parameters

Scalar bit-string

Result

Character string containing the hexadecimal 0...9 and A...F only.

Action

Starting from the left, one hexadecimal character is generated for every four bits. The hexadecimal characters used are the numeric characters 0 to 9 and alphabetic characters A to F.

If the length of the bit-string is not a multiple of 4 bits, bits of value ’0’B will be filled in on the right. If filling in on the left is required, a suitable concatenation must be specified for the parameter (see example).

```
DCL HEXDEC  ENTRY    (BIT(*))
            RETURNS  (CHAR(*))     OPTIONS (LIBRARY);

DCL BIT  BIT(7)  INIT ('1111000'B);                          Result


PUT SKIP(1) LIST (HEXDEC(BIT));                              F0
PUT SKIP(2) LIST (HEXDEC('0'B || BIT));                      78
PUT SKIP(2) LIST (HEXDEC(COPY('0'B,4*CEIL(LENGTH(BIT)/4)
                 -LENGTH(BIT))||BIT));                       78
```

```
                    Bit string and          '0000'B  0
                    resulting               '0001'B  1
                    hexadecimal             '0010'B  2
                    character               '0011'B  3
                                            '0100'B  4
                                            '0101'B  5
                                            '0110'B  6
                                            '0111'B  7
                                            '1000'B  8
                                            '1001'B  9
                                            '1010'B  A
                                            '1011'B  B
                                            '1100'B  C
                                            '1101'B  D
                                            '1110'B  E
                                            '1111'B  F
```

Fig. 11-6 Example of HEXDEC call

NOTRACE Trace off

```
NOTRACE (a)
```

Deactivation of trace

Entry

```
DCL NOTRACE ENTRY (CHAR(*)) OPTIONS (LIBRARY) [EXTERNAL] [CONSTANT];
```

Parameters

Character string that may contain the following letters with the meanings indicated:

P Trace for procedure call (PROCEDURE)
C Trace for CALL statements
R Trace for return from procedure call (RETURN)
G Trace for branches (GOTO)
L Trace for labels (LABEL)
T Additional output for trace to terminal

Other letters are ignored. The empty string (") corresponds to the 'PCRGL' option.

Effect

An activated trace (see TRACE) can be deactivated by NOTRACE. (See also chapter 9.)

PLIRETC Set return code

Only for users with software product JV.

```
PLIRETC (a)
```

The program information of the return code is set to a.

Entry

```
DCL PLIRETC ENTRY ([REAL] FIXED BINARY PREC(31,0) [ALIGNED])
            OPTIONS (LIBRARY) [EXTERNAL][CONSTANT]
```

Parameters

a: Value between 0 and 999

Effect

The program information (digits 5 through 7) of the return code of a monitoring job variable receives the value a in the format PIC'999' at program termination, thus permitting the program to pass an item of information to the command level.

```
DCL PLIRETC ENTRY(FIXED BINARY(31,0))
            OPTIONS (LIBRARY);

CALL PLIRETC (17);
```

```
/DCLJV JV
/EXEC T.PLIRETC,MONJV=JV
   .
   .
   .
   .
   .
END OF PROGRAM....
/GETJV (JV,5,3)
%017
```

Fig. 11-6a Example of PLIRETC

RDUMP (a,b) Dump

```
RDUMP (a,b)
```

Output of b characters to SYSLST, starting at address a.

Input

```
DCL RDUMP ENTRY (POINTER [ALIGNED],
                 [REAL] FIXED BINARY PREC (15,0) [ALIGNED])
          OPTIONS (LIBRARY) [EXTERNAL] [CONSTANT];
```

Parameters

a: pointer to start of dumped memory area b: number of bytes to be dumped.

Effect

Starting at address a, at least b characters in hexadecimal notation are output to SYSLST.

```
DCL RDUMP  ENTRY (POINTER ALIGNED,
                  FIXED BINARY (15,0) ALIGNED)
                  OPTIONS (LIBRARY);

DCL GLEIT  FLOAT DECIMAL (33) INIT (12345E+16);

CALL RDUMP   (ADDR(GLEIT),16);
```

Output

```
****** DUMP DES BEREICHS 0181A0 - 0181AF: ******

REPL    VS ADR    SPEICHERINHALT
        0181A0    516B1368  0EF11F90  43000000  00000000
```

Fig. 11-8 Example of RDUMP

RUNTIME Computing time used

```
RUNTIME
```

Function returning computing time (in seconds) used since program was started.

Entry

```
DCL RUNTIME ENTRY ( )
            RETURNS (CHAR (8))
            OPTIONS (LIBRARY) [EXTERNAL] [CONSTANT];
```

Parameters

None

Result

Computing time in seconds in the form PIC 'ZZZZ9.V99'

Effect

When this procedure function is invoked the CPU time used since the start of the program is determined, and returned in the form of a character string:

5 characters integer part 1 character decimal point 2 characters fractional part

The time is measured in seconds. The value is rounded.

```
DCL  RUNTIME  ENTRY () RETURNS (CHAR(8)) OPTIONS(LIBRARY);

PUT  SKIP LIST (RUNTIME);


DCL  ZEIT   CHAR(8);

ZEIT = RUNTIME;

PUT  SKIP LIST (ZEIT);
```

Result

```
0.28
0.29
```

Fig. 11-9 Example of RUNTIME call

SDUMP Stack dump

```
SDUMP
```

The contents of the stack (block activations containing AUTOMATIC variables) is output to SYSLST in hexadecimal and character representation.

Entry

```
DCL SDUMP ENTRY ( )
          OPTIONS (LIBRARY) [EXTERNAL] [CONSTANT];
```

Parameters

None

Effect

Dumping of current contents of the stack segments. A detailed description of the stack structure is given in chapter 10. Prior to the dump the starting address of the current activation of the stack is output, i.e. the activation of the runtime system that outputs the dump.

Prior to the printout, referred to above, the pseudoregister vector (PRV) is output. Interpretation of this printout must be left to specialists.

```
DCL  SDUMP  ENTRY() OPTIONS(LIBRARY);

DCL  ZEICHEN  CHAR(8) AUTOMATIC INIT('ABCDEFGH');


CALL SDUMP;
```

Result:

```
****** DUMP OF THE STACK:******

DUMP OF THE PRV:

REPL    VS ADR   MEMORY
        014000   000800D2  F1F7F3F5  D7D3C9C1  D3D3C740
        014020   C2E3C4E4  00000E00  00010001  00000000
        014040   00000002  00010044  0100F8F4  0200FBC4
        014060   0700FB28  4F002422  0100F8F4  00000002



ACTUAL ACTIVATION (REGISTER 13 OF THE DUMPING PROCEDURE): 00019250

DUMP OF ALL LOGICAL STACK SEGMENTS:

REPL    VS ADR   MEMORY
        019000   00000000  00000000  7F019000  7F029000  00000000
        019020   00000010  00000002  00000000  00000014  00000014
        019040   00000000  00000000  00000000  00000000  00000000
```

Fig. 11-10 Example for SDUMP call

SNAP Call nesting

```
SNAP
```

Output of current call nesting to SYSOUT

Entry

```
DCL SNAP ENTRY ( ) OPTIONS (LIBRARY) [EXTERNAL] [CONSTANT];
```

Parameters

None

Effect

CALL SNAP produces a listing of the currently active procedures and their calling sequence. The oldest procedure in time (main procedure) is listed last. One line is shown for each procedure which has been called and is still active.

The structure of this listing is explained in detail in section 9.7.

```
VERSCHA:              /* EXAMPLE OF CALL NESTING */
          PROCEDURE OPTIONS(MAIN);

          DCL  SNAP ENTRY() OPTIONS(LIBRARY);

          PUT  SKIP LIST ('——— SNAP');

          ON ZERODIVIDE BEGIN;
                        PUT SKIP LIST ('———ZERO');
                        CALL FEHLER;
                        GOTO ENDE;
                        END;
          BEGIN;
          PUT SKIP LIST ('——— BEG');
          DCL (X,Y,Z) INIT(0),
          X = Y / Z;
          END;

          FEHLER: PROC;
                  PUT SKIP LIST ('——— ERROR');

                  CALL SNAP;

                  END;
```

SYSLST

```
... SNAP
... BEG
... ZERO
... FEHLER
```

SYSOUT

```
****************************** SNAP *************************


START OF PRINTING OF NESTED SUBROUTINES

CALLED FROM     TYPE      ON ADDRESS      SOURCE REFERENCE

  SNAP          SYSTEM      0020D2
  FEHLER        ENTRY       000406               25     1
  ZEROIDE       ON          000300               13     1
  ER$INTR       SYSTEM      0107EA
  ER$PUB        SYSTEM      010EA0
  SR$STXT       SYSTEM      00FD04
  ##00004       BEGIN       0003FA               19     1
  BSNAP         ENTRY       000244               11     1
```

Fig. 11-11 Example of call nesting (SNAP)

TRACE Trace on

```
TRACE (a)
```

Activation of trace

Entry

```
DCL TRACE ENTRY(CHAR(*)) OPTIONS (LIBRARY)
                         [EXTERNAL] [CONSTANT]
```

Parameters

Character string that may contain the following letters with the meanings indicated:

P Trace for procedure call (PROCEDURE)
C Trace for CALL statements
R Trace for return from procedure call (RETURN)
G Trace for branches (GOTO)
L Trace for labels (LABEL)
T Additional output for trace to terminal

Other letters are ignored. The empty string (") corresponds to the 'PCRGL' option.

Effect

A precondition for trace is that, when the procedure is being compiled, the correct trace has been incorporated by use of the compiler control command

```
*COMPOT DEBUG=option
```

The desired trace can be activated either by the object control command

```
*RUNOPT TRACE=option
```

(it then starts at the beginning of the procedure) or, dynamically, by the invocation

```
CALL TRACE (expression);
```

(it then starts immediately after the invocation). By use of the invocation NOTRACE the desired trace can be deactivated dynamically.

The presetting of the PL/I system does not initiate incorporation nor activation of a trace routine (see chapter 9).

12 Shareable programs

12.1 Prerequisites

For programs using shareable modules the following conditions apply:

- Shareable modules must be loaded dynamically.
- Shareable modules may be accessed in read mode only.
- Address referencing from shareable modules to nonshareable ones is not permitted.
- Variable data must be located in dynamically allocated memory areas. This condition is necessary if all modules are to belong to storage class 4. This must always be the case in the current implementation for shareable PLI1 programs.

These conditions will be fulfilled by PL/I programs if they do not contain STATIC variables, CONTROLLED variables, or input/output statements.

Modules ITP#$RTS# and ITP#IOS# of the "shareable" PL/I runtime system comply with the conditions for shareable modules.

When using assembler modules, they, too, will be subject to the above conditions.

12.2 PL/I programs

This section gives a description of methods for obtaining shareable programs from PL/I programs which do not comply with the conditions listed above. Extended language is available in PLI1 for this purpose.

12.2.1 STATIC variables

By language extensions, STATIC variables can be removed from external procedures if the user does not lose sight of any of his STATIC variables. STATIC variables are changed to BASED variables via the PLI1 system's predefined (built-in) absolute pointer array, PLI1GLOBAL (0:127).

Example

```
DCL name STATIC EXTERNAL INIT (value);
```

is replaced by

```
DCL name BASED (PLI1GLOBAL(n)) INIT (value);
```

where n is an integer between 0 and 127. A separate number is required for each name.

The statement

```
ALLOCATE name;
```

must be inserted at the start of the program.

If several STATIC variables have to be removed, it is advisable to collect them in a structure.
INTERNAL STATIC variables with attribute INITIAL which are only referenced by read access operations, should not be handled as described above. Instead

```
STATIC (CONSTANT)
```

should be written in the declaration, if they are not recognized as constants by the compiler anyway (see chapter 8).

12.2.2 Input/output statements

In the declaration of a file the compiler creates a STATIC variable for the file constant.

Creation of this STATIC variable may be suppressed by an extension of PL/I language. This is achieved by adding the specification

```
ENVIRONMENT (PLI1GLOBAL (n))
```

to the file constant. When this is done, care must be taken that in the presence of more than one external procedure the same index n is only used for identical file constants and nowhere else.

In statements for the input/output stream without file specification the file constants SYSIN and SYSPRINT are implicitly declared for input and output respectively. These file constants must also be explicitly declared with the environment specification mentioned above. Analogous rules apply to the SYSOUT file.

The only input/output statement which can be used without special precautions is DISPLAY with/without REPLY.

12.2.3 CONTROLLED variables

A STATIC variable is implicitly generated for each CONTROLLED variable by the compiler. Creation of this variable can be suppressed by an extension of PL/I language. For this purpose it is necessary to specify

```
CONTROLLED (PLI1GLOBAL (n))
```

in the declaration. Note that ’n’ must be uniquely assigned to the name of the CTL variable.

12.3 Entry into class 4 memory

When all the modules of a program have been made shareable and compiled, the module linkage editor is used to generate a prelinked module, linking ITP#AOD#, the linkage module, to the dynamic runtime system of PLI1. The prelinked module is declared "READ ONLY" by means of the LMR [3] and is then entered in the share table of the system by the SHARE command (see BS2000 System Controller's Guide).

The ITP#RTS# and ITP#IOS# modules of the runtime system should also have been entered in the share table.

Then the dynamic linkage loader is used to start the program:

```
/EXEC (module-name [,lib-name]) [,further information]
```

Caution:

The SORT routine is not shareable with the BS2SRT program from chapter 11.

13 PLI1 ASSEMBLER macro interface

13.1 General

When ASSEMBLER programs are connected to PL/I programs, and vice versa, the PLI1 conventions must be observed in the ASSEMBLER program (see chapter 7). Since this would involve a considerable programming effort, a set of definition and action macros is provided which normally restricts this effort to the insertion of a few macros. Especially the conversion of existing ASSEMBLER subroutines to permit them to be invoked by PL/I programs will only require the removal of a few ASSEMBLER instructions and the insertion of two macro calls.

These macros can only be used in PL/1 modules if "*COMOPT OPTIONS=NOXS" was used in compilation.

13.1.1 Table of macros

(A = action macro, D = definition macro)

```
P$CALL      -A-    Invokes a PL/I subroutine
P$ENTRY     -A-    Generates an entry to be invoked by PL/I
P$ENVIRM    -A-    Initializes the PLI1 environment
P$ERROR     -A-    Sets the ERROR condition
P$LINK      -A-    Loads a PL/I subroutine
P$PRV       -D-    PLI1 dummy register vector (DSECT or presetting)
P$REGEQU    -D-    EQU statements for register notation
P$RETURN    -A-    Return to the calling PL/I program
P$STACK     -D-    DSECT for register and AUTOMATIC storages
P$STOP      -A-    Terminates the entire program run
```

13.1.2 User considerations

- All macros are reentrant, i.e. ASSEMBLER programs that are reentrant remain this even after PLI1 macros have been inserted.
- The macros have been designed in such a way that normally no control parameters need be specified, i.e. the most frequent application is assumed.
- For coordination purposes the macros use the global variables &ENVIRM#, &STACK#, &MAXPAR#, &PRV#, &REG#.
- All ASSEMBLER programs that are run in conjunction with PL/I programs must observe the PL/I register conventions, i.e. registers R12 and R13 must not be changed outside of the particular macros. If this condition cannot be ensured, the PL/I specific contents of R12 and R13 must be restored each time before the action macros are invoked (exception: P$ENVIRM and P$ENTRY macros).
- In the event of individual STXIT processing in the ASSEMBLER program care must be taken that the PLI1 STXIT processing is reactivated when the PLI1 subroutine is called as well as when control is returned to the calling PL/I program (option STXIT=YES in the P$CALL and P$RETURN macros).
- The P$ENTRY or P$ENVIRM macros enable the ASSEMBLER program to request dynamic storage from the PLI1 storage management. Programs that request storage individually (REQM macro) must be converted in order to avoid fragmentation of virtual address space.
- For examples see chapter 14.6.

13.2 Macros

P$CALL

Invokes a PL/I procedure from an ASSEMBLER program, with optional parameter list transfer to the PL/I program

```
name P$CALL PL/I procedure,param | (param1, param2,...),
PARNUM=0|blank|number,STXIT=N[O] | Y[ES]
```

Rules:

- The macro can only be called after the P$ENVIRM or P$ENTRY macro, a check being performed via the global variable &ENVIRM#.
- A base register is not required; R10 is used internally.
- The registers R0, R1, R2, R3, R4, and R14 are changed and not restored; R10 and R15 are used and restored.
- R12 and R13 must contain the PLI1 specific values.

Description of parameters:

name	Is not used
PL/I procedure	Name of the PL/I program to be called The V-type constant of the PL/I procedure to be invoked must be passed through the R1 parameter register if it is not specified, i.e. R1 contains the address of the V-type constant. Special case: See next parameter
	Param | (param1, param2,...)
	Name(s) of the parameter(s) to be passed to the PL/I program.
	If not specified, either no parameter is passed or a parameter list of ASSEMBLER type is passed.

An ASSEMBLER parameter list has the following structure:

```
R1            Parameter register
A (param1)    Address of the first parameter
A (param2)    Address of the second parameter
                          .
                          .
                          .
X'80' A (param n) Address of the last parameter
                  identified
           by X'80' ('80'B4).
```

Special case:

If the parameter "PL/I procedure" is not specified, the address of the V-type constant of the PL/I procedure to be invoked must be specified as the first entry of the assembler parameter list.

```
PARNUM = 0 | blank | number
```

– Default value 0 means that no parameter must be passed to the PL/I program. Exception: When an explicit parameter list ("param") is specified, 0 is treated as a blank entry.

– The blank entry means that no check on the current parameter must be performed.

– If "number" is specified, a check will always be performed to ensure that the specified number corresponds with the current parameter number, i.e. if "param" is specified, the number of entries in the sublist will be compared with "number" and if no match is found the macro generation will be abnormally terminated with MNOTE. If the transfer is effected through parameter register R1, a check can only be performed at runtime based on the option X'80' ('80'B4) in the last parameter. If no match is found the error condition ONCODE=5010 is set. The parameter "PL/I procedure", if specified, is not taken into account.

```
STXIT = Y[ES] |N[O]
```

– Default NO means no action

– YES means that, prior to calling the PL/I program the PLI1 specific STXIT routines are activated. This is advisable or necessary if a STXIT macro was used in the calling ASSEMBLER program.

Notes

- The maximum number of parameters to be passed to the PL/I program can be defined in the P$ENTRY and P$ENVIRM macros. If errors are detected at compilation time (number of parameters greater than global variable &MAXPAR) the macro generation is terminated with MNOTE. If errors are detected at runtime the error condition P$CALL with ONCODE=5010 is set.
- The invoked PL/I procedure may be a function. If so, the PLI1 return conventions for function value transfer, described in section 7.1.4, must be observed. For the return of some data types, register R1 is used.
- The parameters passed to the PL/I program should not need any descriptors. This means that parameters with * options are not permitted. Alternatively, the use of VARYING character sets or BASED variables with variable dimensioning may be considered.

Example

```
/* ALTERNATES FOR CHAR(*)PARM*/
DCL A CHAR(100) VARYING PARAMETER;
DCL A CHAR(N)   BASED(P),
    P POINTER   PARAMETER,
    N BIN FIXED PARAMETER;
```

P$ENTRY

Generates a procedure entry and generates the procedure heading which is expected when an entry point is called by PL/I.

```
name  P$ENTRY  LENGTH = bytes   , MAXPAR=  number
```

Rules:

– The entry generated with the aid of the P$ENTRY macro can only be called if the PLI1 environment has been activated.

– The macro may be inserted several times in one program, an external entry point being generated each time.

Description of parameters:

name — If the macro appears immediately after the START statement, the option "name" must be omitted; if "name" is specified an ENTRY statement is generated in addition to the procedure heading.

LENGTH = bytes — Dynamic user storage is allocated to the ASSEMBLER subroutine by the PLI1 storage management.

If "bytes" = 0 (default) no user storage is reserved. See also parameter MAXPAR.

MAXPAR = blank | number

The default '_' means that the presetting of the P$STACK macro is used.
If "number" > 4 an area of (number-4) fullwords is reserved after the register save area for the parameter list used by the P$CALL macro. For details see P$STACK macro.
If "number" ≤4 the parameter transfer by macro P$CALL is effected through registers R1,...,R4; additional storage for the parameter list is not necessary.

Notes

- The global variable &ENVIRM# is used.
- The entry specified by the macro can be defined on the PL/I side by means of the following OPTIONS options:

  ASSEMBLER, VARIABLE, PLI1 (default). Do not use: LIBRARY, COBOL, FORTRAN.
- The RETURNS attribute is not permitted if OPTIONS (ASSEMBLER) is not specified. The return of the function value should be effected in accordance with PLI1 conventions (see section 7.1.4). If the return is made through register R1, parameter R1RETRN=YES must be specified in the P$RETURN macro.

P$ENVIRM

Establishes the PLI1 environment, activates PLI1 STXIT processing and storage management

```
name P$ENVIRM  MAXPAR =  number  ,  LENGTH=bytes, BASEREG = RXX
```

Rules:

- The macro can be called only once for any program; it must not be called after P$ENTRY otherwise MNOTE.
- For use of registers see parameter BASEREG=

Description of parameters:

name — Not used

MAXPAR= — blank | number
"blank" denotes that the presetting of macro P$STACK is used.
"number" indicates the maximum number of parameters to be specified in the P$CALL macro. This specification affects the size of the dynamic storage area; see next parameter.

LENGTH = bytes — Default value 0 denotes that no dynamic storage is requested from the register save area and, if applicable, from the parameter area.
"bytes" specifies how much dynamic storage is to be allocated to the ASSEMBLER program by the PLI1 storage management (mandatory for shareable programming).

BASEREG = RXX | blank
"blank" denotes that no base register is specified when P$ENVIRM is called. R10 is assumed and remains valid until after the end of macro generation.
"RXX" (XX=1,2,...,(11),14,15) may be specified to inform the macro that at the time the macro is called a specific base register is valid. This register continues to be a base register with the previously valid base address, after the macro call.
Within the macro, register R10 is always used as the base register.

R12 and R13 are not allowed since these registers must retain the PLI1-specific value after the macro has been executed. R11 is not recommended since this register is the only one that is not changed by the P$ENVIRM macro. It may be used to store any other register (necessary for shareable programming).

Notes

- The macro uses all registers other than R11. Because of its reenter feature, they cannot be restored. See also parameter BASEREG.
- If the macro is called more than once inadvertently, no warning can be issued. Depending on the setting of the runtime option STORAGE, each call causes a specific amount of virtual storage to be inhibited, or the STORAGE condition may arise.

P$ERROR

Sets the PL/I ERROR condition analogous to SIGNAL ERROR, however with indication of a specific ONCODE value and, if applicable, a message number

```
name      P$ERROR  ONCODE = code   ,MSG=subcode
```

Rules:

- The macro can only be called after the macros P$ENTRY or P$ENVIRM, a check being performed through global variable &ENVIRM#.
- A base register is not necessary. R10 is used internally. No registers are restored.
- R12 and R13 must contain the PLI1-specific values.

Description of parameters:

name	Not used
ONCODE =	code,MSG=subcode Preset ONCODE=5000, MSG=00 "code" indicates the PL/I ONCODE to be used by the PLI1 error recovery in order to issue a special error text. The option MSG=subcode is only mandatory if there are several error texts for one specific ONCODE. The texts must be contained in the files $TSOS.PLI1.TEXT.E or D or be transferred to them by the user.
MSG = subcode	See parameter ONCODE If special error messages and ONCODE values are to be defined, the text files must be extended accordingly. See parameters ONCODE= and MSG=. The ISAM key to be used in the text files for the insertion of texts is eight positions long 015CCCMM, where 5CCC is the ONCODE with leading zeros and MM, the additional message number.

P$LINK

Loads a PL/I subroutine (DLL)

```
name  P$LINK  PL/I name  ,addr.const   ,LIBNAM=lib-name
```

Rules:

- The macro should only be used when the PLI1 environment (R12 and R13) has been established because, in the event of an error, the PLI1 error recovery is invoked through macro P$ERROR with ONCODE=5015.
- A base register R1 is required.
- Registers R0 and R1 are changed but not restored; R15 is used and restored.

Description of parameters:

name	Not used
PL/I name	Name of the PL/I subroutine that is to be loaded. If not specified the address of a LINK parameter block is expected in R1 as it is generated, for example, by the macro call LINK MF=L,... . In this parameter block at least the name field must be filled.
Addr. const	Name of an address constant to be assigned to the PL/I subroutine to be loaded. If not specified, the load address will be in R1.
LIBNAM= lib-name	Specification of a special load library. Default NONE indicates that only the $TASKLIB library is used if a library has not been specified in the parameter block addressed through register R1.

Notes

– The loaded program is not started (INHIBIT=YES); this must be done with the aid of the P$CALL macro.

– Any LMR or LMS library can be assigned to the TASKLIB ($TASKLIB) library with the aid of the /SYSFILE TASKLIB=library command.

– The macro should only be used in conjunction with the loadable (shareable) runtime system. If the runtime system is not loadable (static runtime system, control module P$ANFOS#) all required runtime system modules will be made addressable in the dummy register vector when the program is started. If the loaded PL/I program should require additional modules, the associated addresses are not provided in the dummy register vector which may lead to addressing errors (ONCODE=8095).

  Remedy: If the use of the static runtime system cannot be dispensed with, all associated modules must be linked explicitly into the calling program.

P$PRV

Generates a DSECT or an initialization constant for the PLI1 dummy register vector (Pseudo Register Vector, PRV).

```
&name    P$PRV  TYPE=  D[SECT] |C [ONST]
```

Rules:

- The macro is expanded only once for each program system. Control is effected through global variable &PRV.
- The macro-generated DSECT is used in all P$ action macros. The macro is therefore called by these macros.
- The base register R12, which is associated with the PRV-DSECT, is only set in the P$ENVIRM macro or in a PL/I main program.

Description of parameters:

name — Name of the generated DSECT or 4 K constant
Default when TYPE=DSECT, if not specified.

TYPE = D[SECT]|C[ONST]
Default: DSECT
The option CONST is normally not meaningful for the user.

Notes

- Normally, the dummy register vector is not referenced by the ASSEMBLER user program outside of the PLI1 macro.

  Exception: The PRV entry PLI1GLOBAL (128 pointers) can be used with shareable programming in order to address the predefined pointer array PLI1GLOBAL(0:127) of the PL/I programs involved, in order to replace the EXTERNAL or COMMON variables which are not permitted in this case.
- The listing of the macro expansion is suppressed. The previously issued PRINT statement remains valid.

P$REGEQU

Generates EQU statements for PLI1 register notation (RXX)

```
P$REGEQU
```

Rules:

– The macro is generated only once for each program. Control is effected through the global variable &REG#.

– The macro is used by all PLI1 macros.

Description of parameters:

none

Notes

The generation of this macro in existing ASSEMBLER programs may lead to M flags. If the removal of identical names gives rise to problems the generation of the macro may be inhibited by setting $REG# SETA 1.

P$RETURN

Return from an ASSEMBLER subroutine to the calling PL/I program.

```
name  P$RETURN      R1RETURN = N[O]|Y[ES] ,
                    STXIT    = N[O]|Y[ES]
```

Rules:

- This macro should only be used in conjunction with the P$ENTRY macro. This means that return from an ASSEMBLER program can only be effected with P$RETURN if P$ENTRY was called at its entry point.
- No base register required.
- Register R13 must have the same setting as it had in the P$ENTRY macro. If this rule is violated the program to which a return is made will fall back on illegal generations of dynamic storage.
- The register settings valid at the time of invocation are restored.
  Exception: Register 1 when parameter R1RETURN = YES

Description of parameters:

R1RETURN = N[O]|Y[ES]

Default NO indicates that all registers are reset to the values they contained at the time of invocation (macro P$ENTRY).
YES must be specified when the ASSEMBLER routine returns returns a function value by way of register R1. This is possible only with ASSEMBLER programs written in accordance with PLI1 standards - see the relevant description in section 7.1.4.

STXIT = N[O]|Y[ES]

Default NO means no action.
YES must be specified when an STXIT macro is used in the ASSEMBLER routine. The PLI1 STXIT will then be restored before control is returned to the PL/I program.

Notes

- The PLI1 STXIT option affects all ERROR conditions with ONCODE = 8089 as well as the applicability of the /INTR command (ATTENTION condition).
- The use of register R12 should be avoided in the ASSEMBLER routine, if possible. However, this macro restores the value anticipated from PLI1. A precondition is that the value of R13 is correct.
- The STXIT-SVC, which is performed if STXIT=YES, requires a few thousand instructions in the operating system (P2, P3 time). It should therefore not be used frequently as it may affect the system throughput rate adversely. Hence the default value STXIT=NO.

P$STACK

Defines a DSECT in order to address the individual fields of a dynamic storage area.

```
name    P$STACK    MAXPAR = number
```

Rules:

- The macro can be generated only once for each program. Coordination is achieved through the global variable &STACK#
- The resulting DSECT is addressed using base register R13
- The parameter MAXPAR is passed on to the macros P$ENTRY and P$ENVIRM through the global variable &MAXPAR#

Description of parameters:

name
: Name of the generated DSECT
If not specified, P$STACK will be assumed.

MAXPAR = number
: The maximum number of parameters to be passed on to a PL/I subroutine that is to be invoked through P$CALL
Default value: 64. A higher value is not permitted.
A field of (number-4) fullwords is reserved in dynamic storage for the PLI1 parameter list. The generated names are PLIPAR4, PLIPAR5, ..., PLIPAR64.

Notes

- Generally this macro is not called by the user but by the macros P$ENTRY and P$ENVIRM.
- The user should call it only if he wants to change the defaults of "name" and "number". In this case the macro must be called before P$ENTRY, P$ENVIRM or other action macros. An entry "number" ≤4 is only effective if an appropriate check is performed.

P$STOP

Terminates the program analogous to the PL/I statement STOP

```
name  P$STOP
```

Rules:

– The macro may only be called after P$ENTRY or P$ENVIRM. A check is performed through global variable &ENVIRM#.

Description of parameters:

name	Not used.

14 Appendix

14.1 List of compiler warnings and error messages

A complete list of the messages which are generated by the compiler as a result of syntactic or semantic errors in statements or due to constraints on implementation, is given in the text files

```
PLI1.TEXT.D  (German)
PLI1.TEXT.E  (English).
```

The texts are largely self-explanatory.

Since all diagnostic messages are completed by a unique reference to the source line, debugging of the source program is generally simple.

For some errors an additional remark, indicated how the problem can be solved, is given.

In the majority of cases, detection of an illegality and output of message is followed by normal continuation of the compiling operation. Any assumptions made by the compiler to enable it to continue will also be recorded in the message.

The device to which the message is output can be determined by the control statements DIAGNOST and MESSAGE. For further details see section 3.5. The default value for all messages is SYSLST.

The records in the text files have the following structure:

No. Weight Text

where:

No.: The first 8 positions give the number (consecutive) of the error.

Weight: The meaning of the following 3 positions is as follows:

C01	Warning
C02	Error
C03	Severe error; object module not generated.
C04	Unrecoverable error; immediate abortion of compiling operation.

Text: If the error text contains the character @, it will be replaced by the appropriate current value.

If the generation of object modules is to depend on the number of messages of a definite weight which have occurred, the control statement OBJECT can be used. For further information see section 3.6.1.

Output of messages below a certain weight can be suppressed by using the control statement DIAGNOST (see section 3.5.2).

14.2 List of error messages from object programs (ONCODE values)

If errors are encountered during the execution of a PL/I program, and there is no ON unit associated with the error class, one of the following error texts is issued by default.

Each error text has an ONCODE value (error number) assigned to it, which can be recalled in an ON unit with the builtin function ONCODE. Further details can be found in chapter 11 of the language reference manual [1].

For errors detected in the Data Management System (DMS), the appropriate error key is output as part of an explanatory text. The meaning of these error keys is found in publication [6].

The following applies to the subsequent list of ONCODE-Values and associated messages:

- The list is arranged according to ascending ONCODE values.
- A heading specifies the condition under which the subsequent ONCODES may occur. A condition may appear as heading more than once.
- If the same ONCODE value appears more than once with different messages, then each message represents an alternative. The respective alternative can be shown by CALL ERROUT.
- As the listing is printed, the character '@' in the text is replaced with the appropriate current value.
- The error texts which follow are stored in ISAM file PLI1.TEXT.E with the key '01ccccmm', where 'c' is the ONCODE with leading zeros and 'm' is an additional number for an otherwise identical ONCODE.

ERROR condition (see also ONCODE = 9 and 1000)

```
 3    NO WHEN CLAUSE SATISFIED AND NO OTHERWISE CLAUSE SPECIFIED

 3    DATA TYPE OF THE RETURN-EXPRESSION CONFLICTS WITH THE
      RETURNS ATTRIBUTE

 3    RETURN FROM FUNCTION WITHOUT RETURN VALUE
```

FINISH condition

```
 4    END OF PROGRAM REACHED

 4    END OF PROGRAM SIGNALED
```

ERROR condition

```
9       ERROR CONDITION SIGNALED
```

NAME condition

```
10    GET FILE DATA; SYNTAX ERROR IN IDENTIFIER
      (ONFILE='@',ONFIELD='@')

10    GET FILE DATA: MORE THAN 256 CHARACTERS IN THE NAME
      (ONFILE='@',ONFIELD='@')

10    GET FILE DATA: AN IDENTIFIER HAS NO COUNTERPART IN THE DATA
      LIST (ONFILE='@',ONFIELD='@')

10    GET FILE DATA: AN ARRAY SUBSCRIPT IS MISSING OR INDICATES
      TOO MANY DIMENSIONS (ONFILE='@',ONFIELD='@')

10    GET FILE DATA: A SUBSCRIPT IS BEYOND THE DECLARED RANGE
      (ONFILE='@',ONFIELD='@')

10    GET FILE DATA: ILLEGAL DATA TYPE (ONFILE='@',ONFIELD='@')

10    GET FILE DATA: MORE THAN 64 QUALIFIERS IN THE NAME
      (ONFILE='@',ONFIELD='@')

10    GET FILE DATA: IDENTIFIER NOT KNOWN IN THE BLOCK
      (ONFILE='@',ONFIELD='@')

10    GET FILE DATA; PARTIALLY QUALIFIED NAME IS AMBIGUOUS IN THE
      BLOCK (ONFILE='@,ONFIELD='@')

10    UNRECOGNIZABLE IDENTIFIER IN GET DATA ON FILE @ SIGNALED
```

RECORD condition (see also ONCODE = 3 and 1000)

```
20    RECORD LENGTH ERROR ON FILE @ SIGNALED

21    LENGTH OF RECORD VARIABLE LESS THAN RECORD LENGTH.
      (ONFILE='@')

21    RECORD LENGTH GREATER THAN ACTUAL LINESIZE FOR SYSDTA
      (ONFILE='@')

22    LENGTH OF RECORD VARIABLE GREATER THAN RECORD LENGTH
      (ONFILE='@')

23    RECORD VARIABLE TOO SHORT TO CONTAIN EMBEDDED KEY
      (ONFILE='@')

23    RECORD VARIABLE HAS ZERO LENGTH (ONFILE='@')
```

TRANSMIT condition

```
40    I/O TRANSMISSION ERROR ON FILE @ SIGNALED

41    UNCORRECTABLE TRANSMISSION ERROR ON OUTPUT (ONFILE='@')

41    UNCORRECTABLE ERROR IN OUTPUT (ONFILE='@',DMS-ERROR:@)

41    RECSIZE GREATER THAN (BLKSIZE - PAD) OR WRONG SEQUENCE OF
      ACCESS TO SHARED UPDATE FILES (ONFILE='@',DMS-ERROR:@)

41    NOT ENOUGH SPACE AVAILABLE FOR SECONDARY ALLOCATION
      (ONFILE='@',DMS-ERROR:@)

42    UNCORRECTABLE TRANSMISSION ERROR ON INPUT (ONFILE='@')

42    UNCORRECTABLE ERROR IN INPUT (ONFILE='@',DMS-ERROR:@)
```

KEY condition

```
50    ERROR IN RECORD KEY ON FILE @ SIGNALED

51    KEY SPECIFIED CANNOT BE FOUND
      (ONFILE='@',ONKEY='@',DMS-ERROR:@)

52    KEY SPECIFIED ALREADY IN USE ON DATA SET
      (ONFILE='@',ONKEY='@',DMS-ERROR:@)

53    KEY VALUE IS NOT GREATER THAN VALUE OF PREVIOUS KEY,
      (ONFILE='@',ONKEY='@')

53    KEY VALUE IS LESS THAN VALUE OF PREVIOUS KEY,
      (ONFILE='@',ONKEY='@')

53    KEY VALUE IS NOT GREATER THAN VALUE OF PREVIOUS KEY,
      (ONFILE='@',ONKEY='@',DMS-ERROR:@)

54    KEY SPECIFIED CANNOT BE CONVERTED TO VALID DATA
      (ONFILE='@',ONKEY='@')

55    KEY SPECIFIED IS INVALID (ONFILE='@',ONKEY='@')

56    KEY SPECIFIES POSITION OUTSIDE REGIONAL DATA SET
      (ONFILE='@',ONKEY='@')

57    NO SPACE AVAILABLE TO ADD KEYED RECORD
      (ONFILE='@',ONKEY='@',DMS-ERROR:@)

57    NO SPACE AVAILABLE TO ADD KEYED RECORD
      (ONFILE='@',ONKEY='@')
```

ENDFILE condition

```
70    ENDFILE CONDITION RAISED. (ONFILE='@')

70    END OF FILE PREVIOUSLY ENCOUNTERED ON STREAM INPUT
      (ONFILE='@')

70    ENDFILE CONDITION RAISED. (ONFILE='@',DMS-ERROR:@)

70    END OF FILE @ SIGNALED

71    ENDFILE CONDITION FOR TAPE RAISED : DOUBLE TAPEMARK DETECTED
      (ONFILE='@')
```

UNDEFINEDFILE condition

```
80    ERROR IN ATTRIBUTES OF FILE @ SIGNALED

81    CONFLICTING DECLARE AND OPEN ATTRIBUTES  (ONFILE='@')

81    CONFLICTING DECLARE AND OPEN ATTRIBUTES: INPUT AND OUTPUT
      (ONFILE='@')

81    CONFLICTING DECLARE AND OPEN ATTRIBUTES: INPUT AND UPDATE
      (ONFILE='@')

81    CONFLICTING DECLARE AND OPEN ATTRIBUTES: OUTPUT AND UPDATE
      (ONFILE='@')

81    CONFLICTING DECLARE AND OPEN ATTRIBUTES: RECORD AND STREAM
      (ONFILE='@')

81    CONFLICTING DECLARE AND OPEN ATTRIBUTES: SEQUENTIAL AND
      DIRECT (ONFILE='@')

81    CONFLICTING DECLARE AND OPEN ATTRIBUTES: BACKWARDS AND
      STREAM (ONFILE='@')

81    INCORRECT ATTRIBUTE IN DECLARE OR OPEN FOR TRANSIENT-FILE.
      (ONFILE='@')

82    CONFLICTING ATTRIBUTES AND FILE ORGANIZATION: DIRECT, KEYED
      AND CONSECUTIVE (ONFILE='@')

82    INDEXED OR REGIONAL CONFLICTS WITH PRINT, STREAM, TRANSIENT
      OR BACKWARDS. (ONFILE='@')

82    CONFLICTING ATTRIBUTES AND FILE ORGANIZATION: OUTPUT WITHOUT
      KEYED AND INDEXED OR REGIONAL (ONFILE='@')

82    CONFLICTING ATTRIBUTES AND FILE ORGANIZATION: DIRECT OUTPUT
      AND INDEXED (ONFILE='@')

82    INCORRECT ENVIRONMENT-OPTION FOR SYSTEM-FILE. (ONFILE='@')

82    REGIONAL(1/3) ORGANIZED FILES MUST NOT HAVE THE FILENAME
      *DUMMY (ONFILE='@')

82    CONFLICT BETWEEN FILE ATTRIBUTES AND PHYSICAL ORGANIZATION :
      AN UPDATE FILE MUST RESIDE ON RANDOM ACCESS DEVICE
      (ONFILE='@')

82    OUTPUT FOR BTAM NOT ALLOWED. (ONFILE='@')

82    CONFLICT BETWEEN FILE ATTRIBUTES AND PHYSICAL ORGANIZATION :
      INDEXED FILES MUST RESIDE ON RANDOM ACCESS DEVICE
      (ONFILE='@')

82    CONFLICT BETWEEN FILE ATTRIBUTES AND DATA SET ORGANIZATION :
      DEVICE TYPE OR VSN IN ERROR OR IN CONFLICT WITH FILE
      ORGANIZATION (ONFILE='@',DMS-ERROR: @)
```

```
83    DATA SET SPECIFICATION INCOMPLETE : RECFORM NOT GIVEN (ONFILE='@')

83    DATA SET SPECIFICATION INCOMPLETE : NO RECSIZE FOR
      REGIONAL(1) FILE GIVEN (ONFILE='@')

83    DATA SET SPECIFICATION INCOMPLETE : NO RECSIZE FOR
      REGIONAL(3) FILE GIVEN (ONFILE='@')

83    DATA SET SPECIFICATION INCOMPLETE : NO KEYLEN FOR
      REGIONAL(3) FILE GIVEN (ONFILE='@')

83    DATA SET SPECIFICATION INCOMPLETE : NO RECSIZE OR BLKSIZE
      FOR TAPE-FILE GIVEN (ONFILE='@')

83    DATA SET SPECIFICATION INCOMPLETE : NO RECSIZE FOR FILE WITH
      RECFORM=F GIVEN (ONFILE='@')

84    NO CORRECT FILE COMMAND|MACRO EXECUTED
      (ONFILE='@',DMS-ERROR: @)

85    REGIONAL DATA SET CANNOT BE FORMATTED (ONFILE='@')

86    CONFLICT BETWEEN ENVIRONMENT OPTION AND DATA SET
      SPECIFICATION: VALUE OF LINESIZE CONTRADICTS VALUE OF
      BLKSIZE|RECSIZE|KEYLEN AND RECFORM (ONFILE='@')

86    LINESIZE ≤ 0. (ONFILE='@')

86    BLOCKMODE: CONFLICT BETWEEN LINESIZE/PAGESIZE AND DEVICE
      SPECIFICATIONS (ONFILE='@')

86    THIS TERMINAL DOES NOT ALLOW TO USE BLOCKMODE

87    CONFLICT BETWEEN DATA SET SPECIFICATIONS
      (ONFILE='@',DMS-ERROR: @)

87    REGIONAL(1) FILES MUST BE ASSOCIATED WITH PAM-ACCESSED DATA
      SETS (ONFILE='@')

87    FILE IS NOT REGIONAL(1) ORGANIZED (ONFILE='@')

87    CONFLICT BETWEEN FILE ATTRIBUTES AND DATA SET SPECIFICATION
      : REGIONSIZE > SPACE FOR REGIONAL(1) FILE (ONFILE='@')

87    REGIONAL(3) FILES MUST BE ASSOCIATED WITH PAM ACCESSED DATA
      SETS (ONFILE='@')

87    FILE IS NOT REGIONAL(3) ORGANIZED (ONFILE='@')

87    CONFLICT BETWEEN FILE ATTRIBUTES AND DATA SET SPECIFICATION
      : REGIONSIZE > SPACE FOR REGIONAL(3) FILE (ONFILE='@')

87    CONFLICT BETWEEN FILE ATTRIBUTES AND DATA SET SPECIFICATION
      : RECSIZE > REGIONSIZE FOR REGIONAL(3) FILE (ONFILE='@')
```

```
87    INCORRECT NOTATION OF THE REGIONSIZE (=BLKSIZE PARAMETER)
      FOR A REGIONAL(3) FILE : REGIONSIZE MUST BE OF THE FORM
      STD|(STD,N) WITH 1<=N<=16 (ONFILE='@')

87    CONFLICT WITHIN DATA SET SPECIFICATIONS : THE VALUES OF THE
      RECSIZE OR THE RECFORM OR THE BLKSIZE PARAMETER ARE
      INCORRECT, CONFLICTING OR NOT YET GIVEN
      (ONFILE='@',DMS-ERROR: @)

87    CONFLICT WITHIN DATA SET SPECIFICATION : KEY SPECIFICATION
      IN ERROR OR CONTRADICTING TO REFORM|RECSIZE VALUE
      (ONFILE='@',DMS-ERROR: @)

87    CONFLICT BETWEEN FILE ATTRIBUTES AND DATA SET SPECIFICATION:
      OUTPUT FILES MUST HAVE BEEN EXPIRED|MUST NOT BE SPECIFIED
      BY STATE=FOREIGN (ONFILE='@',DMS-ERROR: @)

87    FAILURE: RECSIZE 0 FOR F-RECFORM. (ONFILE='@')

87    VALUE OF RECSIZE GREATER THAN ALLOWED FOR THIS VALUE OF
      BLKSIZE (ONFILE='@')

87    CONFLICT WITHIN DATA SET SPECIFICATION : DATA SETS RESIDING
      ON RANDOM ACCESS DEVICES MUST HAVE A BLKSIZE PARAMETER OF
      THE FORM : BLKSIZE=STD|(STD,N) WITH 1<=N<=16 (ONFILE='@')

87    SCALARVARYING AND (INCOMPATIBLE KEYPOS OR PRINT-FILE OR
      PRINTER-CONTROL SPECIFIED IN FILE COMMAND). (ONFILE='@')

87    CONFLICT BETWEEN ENVIRONMENT OPTION AND DATA SET
      SPECIFICATION : INDEXED ORGANIZED FILES MUST BE ASSOCIATED
      WITH ISAM DATA SETS (ONFILE='@')

87    CONFLICT WITHIN DATA SET SPECIFICATION : ISAM DATA SETS MUST
      HAVE RECFORM=F|V (ONFILE='@')

87    CONFLICT BETWEEN ENVIRONMENT-OPTION AND DATA SET
      SPECIFICATION: WRONG KEY-SPECIFICATION  FOR
      CONSECUTIVE-ORGANIZED FILES (ONFILE=@)

87    CONFLICT BETWEEN KEYLEN OR KEYPOS IN FILE
      COMMAND/ENVIRONMENT OPTIONS AND IN THE CATALOG ENTRY
      (ONFILE='@')

87    CONFLICT BETWEEN ENVIRONMENT OPTIONS : FOR INDEXED FILES
      RECFORM=U IS NOT ALLOWED (ONFILE='@')

87    CONFLICT BETWEEN ENVIRONMENT OPTIONS : IF KEYLOC(0) IS
      SPECIFIED, RECFORM MUST BE ALSO GIVEN AS AN ENVIRONMENT
      OPTION (ONFILE='@')

87    CONFLICT BETWEEN FILE ATTRIBUTES AND DATA SET SPECIFICATION:
      STREAM FILES MUST NOT BE ASSOCIATED WITH PAM DATA SETS
      (ONFILE='@')
```

```
87    CONFLICT BETWEEN FILE ATTRIBUTES AND DATA SET SPECIFICATION:
      PRINT FILES MUST NOT BE ASSOCIATED WITH PAM|ISAM DATA SETS
      (ONFILE='@')

87    CONFLICT WITHIN DATA SET SPECIFICATION : FOR PRINT-FILES THE
      VALUES OF THE RECFORM AND RECSIZE|BLKSIZE PARAMETERS MUST
      ALLOW AT LEAST ONE DATA-ITEM IN EACH RECORD (ONFILE='@')

87    CONFLICT BETWEEN FILE ATTRIBUTES AND DATA SET SPECIFICATION
      : BACKWARDS NEEDS FCBTYPE=SAM   (ONFILE='@')

88    CONFLICT BETWEEN FILE ORGANIZATION AND DATA SET
      SPECIFICATION : REGIONAL(1) FILES MUST HAVE RECFORM=F IN THE
      FILE COMMAND (ONFILE='@')

88    CONFLICT BETWEEN FILE ORGANIZATION AND DATA SET
      SPECIFICATION: REGIONAL(3) FILES MUST SPECIFY RECFORM=F|V IF
      ASSOCIATED WITH A PAM DATA SET (ONFILE='@')

88    CONFLICT BETWEEN CATALOGUE AND ENVIRONMENT: PRINTER-CONTROL.
      (ONFILE='@')

88    CONFLICT BETWEEN CATALOGUE AND ENVIRONMENT: RECSIZE.
      (ONFILE='@')

88    CONFLICT BETWEEN CATALOGUE AND ENVIRONMENT: RECFORM.
      (ONFILE='@')

88    CONFLICT BETWEEN CATALOGUE AND ENVIRONMENT: KEYLEN.
      (ONFILE='@')

88    CONFLICT BETWEEN CATALOGUE AND ENVIRONMENT: KEYLOC.
      (ONFILE='@')

89    PASSWORD INVALID OR NOT SPECIFIED (ONFILE='@')
```

ENDPAGE condition

```
90    ATTEMPT TO START NEW LINE, WHEN LINE NUMBER IS EQUAL TO
      CURRENT PAGESIZE (ONFILE='@')

90    PAGESIZE OVERFLOW ON FILE @ SIGNALED
```

UNDEFINEDFILE condition

```
93    WHILE OPENING A DATA SET DMS SIGNALED AN ERROR UNSPECIFIC
      FOR I/O-SYSTEM (E.G. LOOK AT YOUR BLKSIZE PARAMETER)
      (ONFILE='@',DMS-ERROR: @)

93    UNIDENTIFIED I/O-ERROR DETECTED (BLKSIZE MAY BE INCORRECT)
      (ONFILE='@',DMS-ERROR: @)

93    NOT ENOUGH CLASS-5-STORAGE AVAILABLE
      (ONFILE='@',DMS-ERROR:@)

93    FCBTYPE IN FILE-COMMAND IS INCONSISTENT WITH CATALOG-ENTRY
      (ONFILE='@')

94    INPUT FILES MUST EXIST AND MUST NOT BE EMPTY
      (ONFILE='@',DMS-ERROR: @)

94    INPUT ATTEMPTED BUT FILE NOT YET CATALOGUED. (ONFILE='@')

95    ATTEMPT DURING OPEN TO ASSOCIATE TO A FILE , CLOSED WITH
      'CLOSE LEAVE' , ANOTHER DATA SET  (ONFILE='@')

96    BACKWARDS/OUTPUT AFTER CLOSE LEAVE FOR NSTD-LABELED TAPES
      NOT ALLOWED   (ONFILE='@')

98    ONLY ONE TRANSIENT-FILE PER JOB POSSIBLE. (ONFILE='@')

99    ATTEMPT TO OPEN LOCKED OR NON SHAREABLE FILE
      (ONFILE='@',DMS-ERROR: @)

110   NO SPACE FOR REGIONAL-FILE. (ONFILE='@')

110   VALUE OF SPACE PARAMETER SPECIFIED IN FILE COMMAND TOO SMALL
      OR IN CONFLICT WITH THE VALUE OF BLKSIZE (ONFILE='@')

110   NO SPACE FOR ISAM-FILE. (ONFILE='@')
```

STRINGSIZE condition

```
150   STRING TRUNCATED BY ASSIGNMENT

150   @: CHARACTERS HAVE BEEN LOST BY ASSIGNING @ CHARACTERS TO A
      @ BYTE TARGET.

150   @: BITS HAVE BEEN LOST BY ASSIGNING @ BITS TO A @ BIT
      TARGET.

150   STRING TRUNCATED BY DISPLAY-STATEMENT

150   STRING TRUNCATION SIGNALED
```

OVERFLOW condition

```
300   FLOATING-POINT OVERFLOW RAISED

300   @ SHORT FLOATING POINT:OVERFLOW

300   @ SHORT FLOATING POINT EXPONENTIATION: OVERFLOW

300   @ SHORT FLOATING POINT: SINGULARITY AT (2 * N + 1)*PI/2

300   @ LONG FLOATING POINT: OVERFLOW

300   @ LONG FLOATING POINT EXPONENTIATION: OVERFLOW

300   @ LONG FLOATING POINT: SINGULARITY AT (2 * N + 1)*PI/2

300   @ EXTENDED FLOATING POINT: OVERFLOW

300   @ EXTENDED FLOATING POINT EXPONENTIATION: OVERFLOW

300   @ EXTENDED FLOATING POINT: SINGULARITY AT (2 * N + 1)*PI/2

300   @ COMPLEX SHORT FLOATING POINT: ABS(REAL(X)) > 174.673

300   @ COMPLEX SHORT FLOATING POINT: ABS(IMAG(X)) > 174.673

300   @ COMPLEX SHORT FLOATING POINT: ABS(2 * IMAG(X)) > 174.673

300   @ COMPLEX SHORT FLOATING POINT: ABS(2 * REAL(X)) > 174.673

300   @ COMPLEX SHORT FLOATING POINT EXPONENTIATION: IMMEDIATE
      OVERFLOW ABS(REAL(D2 * LOG(C2))) > 174.673

300   @ COMPLEX SHORT FLOATING POINT: SINGULARITY AT (2 * N + 1)
      *PI/2 +0I

300   @ COMPLEX SHORT FLOATING POINT: SINGULARITY AT 0
      +((2*N+1)*PI/2)I

300   @ COMPLEX LONG FLOATING POINT: ABS(REAL(X)) > 174.673

300   @ COMPLEX LONG FLOATING POINT: ABS(IMAG(X)) > 174.673

300   @ COMPLEX LONG FLOATING POINT: ABS(2*IMAG(X)) > 174.673

300   @ COMPLEX LONG FLOATING POINT: ABS(2*REAL(X)) > 174.673

300   @ COMPLEX LONG FLOATING POINT EXPONENTIATION: IMMEDIATE
      OVERFLOW ABS(REAL(D4 * LOG(C4))) > 174.673

300   @ COMPLEX LONG FLOATING POINT: SINGULARITY AT (2*N+1)*PI/2 + 0I

300   @ COMPLEX LONG FLOATING POINT: SINGULARITY AT 0 +
      ((2*N+1)*PI/2)I
```

```
300   @ COMPLEX EXTENDED FLOATING POINT: ABS(REAL(X)) > 174.673

300   @ COMPLEX EXTENDED FLOATING POINT: ABS(IMAG(X)) > 174.673

300   @ COMPLEX EXTENDED FLOATING POINT: ABS(2 * IMAG(X)) >
      174.673

300   @ COMPLEX EXTENDED FLOATING POINT: ABS(2 * REAL(X)) >
      174.673

300   @ COMPLEX EXTENDED FLOATING POINT EXPONENTIATION: IMMEDIATE
      OVERFLOW ABS(REAL(D8 * LOG(C8))) > 174.673

300   @ COMPLEX EXTENDED FLOATING POINT: SINGULARITY AT
      (2*N+1)*PI/2 + 0I

300   @ COMPLEX EXTENDED FLOATING POINT: SINGULARITY AT 0
      +((2*N+1)*PI/2)I

300   @ COMPLEX EXTENDED FLOATING POINT BASE AND INTEGER EXPONENT:
      OVERFLOW

300   @ COMPLEX EXTENDED FLOATING POINT: CHARACTERISTIC OVERFLOW

300   ROUND(X,K): CHARACTERISTIC OVERFLOW

300   FLOATING-POINT OVERFLOW SIGNALED
```

FIXEDOVERFLOW condition

```
310   FIXED-POINT OVERFLOW RAISED BY BINARY OR DECIMAL ARITHMETIC

310   @ COMPLEX DECIMAL OVERFLOW

310   @ COMPLEX BIN FIXED: LENGTH OF RESULT EXCEEDS 15 BIT

310   FIXED-POINT OVERFLOW SIGNALED
```

ZERODIVIDE condition

```
320   ATTEMPT TO DIVIDE BY ZERO

320   @ EXTENDED FLOATING POINT: ZERODIVIDE

320   @ COMPLEX BIN FIXED: ZERODIVIDE

320   @ EXTENDED FLOATING POINT: SECOND ARGUMENT IS ZERO

320   @ COMPLEX EXTENDED FLOATING POINT: ZERODIVIDE

320   @ COMPLEX SHORT FLOATING POINT: ZERODIVIDE

320   @ COMPLEX LONG FLOATING POINT: ZERODIVIDE

320   DIVISION BY ZERO SIGNALED
```

UNDERFLOW condition

```
330   FLOATING-POINT UNDERFLOW RAISED

330   @ COMPLEX EXTENDED FLOATING POINT INTEGER EXPONENTIATION:
      CHARACTERISTIC UNDERFLOW

330   @ COMPLEX EXTENDED FLOATING POINT: CHARACTERISTIC UNDERFLOW

330   FLOATING-POINT UNDERFLOW SIGNALED
```

SIZE condition

```
340   LOSS OF HIGH-ORDER SIGNIFICANT DIGITS IN ASSIGNMENT

340   LOSS OF HIGH-ORDER SIGNIFICANT DIGITS IN CONVERSION

340   @ COMPLEX DEC FIXED: HIGH ORDER NON ZERO DIGITS HAVE BEEN
      LOST. TARGET PRECISION IS TOO SMALL.

340   @ COMPLEX BIN FIXED: TARGET PRECISION IS TOO SMALL, BUT NO
      DIGITS HAVE BEEN LOST.

340   LOSS OF HIGH-ORDER SIGNIFICANT DIGITS SIGNALED

341   LOSS OF HIGH-ORDER SIGNIFICANT DIGITS IN CONVERSION FOR AN
      I/O OPERATION
```

STRINGRANGE condition

```
350   SUBSTRING EXCEEDS STRING BOUNDARY

350   SUBSTRING ERROR SIGNALED
```

AREA condition

```
360   INSUFFICIENT SPACE IN A NAMED AREA FOR ALLOCATE

361   TARGET AREA TO SMALL FOR 'ASSIGN' STATEMENT

362   INSUFFICIENT SPACE IN A NAMED AREA SIGNALED
```

ATTENTION condition

```
400   INTERUPT BY '/INTR @' COMMAND FROM TERMINAL

400   INTERRUPT BY 'SIGNAL ATTENTION' STATEMENT
```

CONDITION condition

```
500   @-CONDITION WAS SIGNALED
```

SUBSCRIPTRANGE condition

```
520   SUBSCRIPT EXCEEDS ITS SPECIFIED BOUNDS

520   SUBSCRIPT ERROR SIGNALED

521   ISUB-SUBSCRIPT EXCEEDS BOUNDS OF BASE ARRAY
```

STORAGE condition

```
530   SPECIFIED MAXIMUM SIZE OF STANDARD AREA EXCEEDED

531   INSUFFICIENT MAIN STORAGE FOR STANDARD AREA

540   INSUFFICIENT MAIN STORAGE SIGNALED

550   MAXIMUM NUMBER OF STORAGE SEGMENTS REACHED IN PROCEDURE
      STACK

551   INSUFFICIENT MAIN STORAGE FOR PROCEDURE STACK

553   ATTEMPT TO ALLOCATE TOO MUCH STORAGE IN STACK
```

CONVERSION condition

```
600   ERROR DURING CONVERSION TO CHARACTER ON INPUT FOR A GET
      STRING STATEMENT (ONSOURCE='@',ONCHARPOS=@)

600   CONVERSION ERROR SIGNALED

601   ERROR DURING CONVERSION TO CHARACTER ON INPUT FOR A GET FILE
      STATEMENT (ONFILE='@',ONSOURCE='@',ONCHARPOS=@)

601   ERROR DURING CONVERSION TO CHARACTER ON INPUT FOR A GET
      STRING STATEMENT (ONSOURCE='@',ONCHARPOS=@)

602   ERROR DURING CONVERSION TO CHARACTER ON INPUT FOR A GET FILE
      STATEMENT AFTER 'TRANSMIT' DETECTED
      (ONFILE='@',ONSOURCE='@',ONCHARPOS=@)

603   ERROR DURING CONVERSION FROM F-FORMAT ON INPUT FOR GET
      STRING STATEMENT (ONSOURCE='@',ONCHARPOS=@)

604   ERROR DURING CONVERSION FROM F-FORMAT ON INPUT FOR GET FILE
      STATEMENT (ONFILE='@',ONSOURCE='@',ONCHARPOS=@)

605   ERROR DURING CONVERSION FROM F-FORMAT ON INPUT FOR GET FILE
      STATEMENT AFTER 'TRANSMIT' DETECTED
      (ONFILE='@',ONSOURCE='@',ONCHARPOS=@)
```

606 ERROR DURING CONVERSION FROM E-FORMAT ON INPUT FOR GET STRING STATEMENT (ONSOURCE='@',ONCHARPOS=@)

607 ERROR DURING CONVERSION FROM E-FORMAT ON INPUT FOR GET FILE STATEMENT (ONFILE='@',ONSOURCE='@',ONCHARPOS=@)

608 ERROR DURING CONVERSION FROM E-FORMAT ON INPUT FOR GET FILE STATEMENT AFTER 'TRANSMIT' DETECTED (ONFILE='@',ONSOURCE='@',ONCHARPOS=@)

609 ERROR DURING CONVERSION FROM B-FORMAT ON INPUT FOR GET STRING STATEMENT (ONSOURCE='@',ONCHARPOS=@)

610 ERROR DURING CONVERSION FROM B-FORMAT ON INPUT FOR GET FILE STATEMENT (ONFILE='@',ONSOURCE='@',ONCHARPOS=@)

611 ERROR DURING CONVERSION FROM B-FORMAT ON INPUT FOR GET FILE STATEMENT AFTER 'TRANSMIT' DETECTED (ONFILE='@',ONSOURCE='@',ONCHARPOS=@)

612 ERROR DURING CHARACTER STRING TO ARITHMETIC CONVERSION (ONSOURCE='@',ONCHARPOS=@)

612 ERROR DURING CONVERSION FROM CHARACTER TO ARITHMETIC ON INPUT OR OUTPUT FOR GET OR PUT STRING STATEMENT (ONSOURCE='@',ONCHARPOS=@)

613 ERROR DURING CONVERSION FROM CHARACTER TO ARITHMETIC ON INPUT OR OUTPUT FOR GET OR PUT FILE STATEMENT (ONFILE='@',ONSOURCE='@',ONCHARPOS=@)

614 ERROR DURING CONVERSION FROM CHARACTER TO ARITHMETIC ON INPUT FOR GET FILE STATEMENT AFTER 'TRANSMIT' DETECTED (ONFILE='@',ONSOURCE='@',ONCHARPOS=@)

615 ERROR DURING CONVERSION FROM CHARACTER STRING TO BIT STRING (ONSOURCE='@',ONCHARPOS='@')

615 ERROR DURING CONVERSION FROM CHARACTER TO BIT ON INPUT OR OUTPUT FOR GET OR PUT STRING STATEMENT (ONSOURCE='@',ONCHARPOS=@)

616 ERROR DURING CONVERSION FROM CHARACTER TO BIT ON INPUT OR OUTPUT FOR GET OR PUT FILE STATEMENT (ONFILE='@',ONSOURCE='@',ONCHARPOS=@)

617 ERROR DURING CONVERSION FROM CHARACTER TO BIT ON INPUT FOR GET FILE STATEMENT AFTER 'TRANSMIT' DETECTED (ONFILE='@',ONSOURCE='@',ONCHARPOS=@)

618 ERROR DURING CHARACTER STRING TO PICTURE CONVERSION (ONSOURCE='@',ONCHARPOS=@)

```
618   ERROR DURING CONVERSION FROM CHARACTER TO PICTURE CHARACTER
      STRING ON INPUT OR OUTPUT FOR GET OR PUT STRING STATEMENT
      (ONSOURCE='@',ONCHARPOS=@)

619   ERROR DURING CONVERSION FROM CHARACTER TO PICTURE CHARACTER
      STRING ON INPUT OR OUTPUT FOR GET OR PUT FILE STATEMENT
      (ONFILE='@',ONSOURCE='@',ONCHARPOS=@)

620   ERROR DURING CONVERSION FROM CHARACTER TO PICTURE CHARACTER
      STRING ON INPUT FOR GET FILE STATEMENT AFTER 'TRANSMIT'
      DETECTED (ONFILE='@',ONSOURCE='@',ONCHARPOS=@)

621   ERROR DURING CONVERSION FROM P-FORMAT (ARITH.) ON INPUT FOR
      GET STRING STATEMENT (ONSOURCE='@',ONCHARPOS=@)

622   ERROR DURING CONVERSION FROM P-FORMAT (ARITH.) ON INPUT FOR
      GET FILE STATEMENT (ONFILE='@',ONSOURCE='@',ONCHARPOS=@)

623   ERROR DURING CONVERSION FROM P-FORMAT (ARITH.) ON INPUT FOR
      GET FILE STATEMENT AFTER 'TRANSMIT' DETECTED
      (ONFILE='@',ONSOURCE='@',ONCHARPOS=@)

624   ERROR DURING CONVERSION FROM P-FORMAT (CHAR.) ON INPUT FOR
      GET STRING STATEMENT (ONSOURCE='@',ONCHARPOS=@)

625   ERROR DURING CONVERSION FROM P-FORMAT (CHAR.) ON INPUT FOR
      GET FILE STATEMENT (ONFILE='@',ONSOURCE='@',ONCHARPOS=@)

626   ERROR DURING CONVERSION FROM P-FORMAT (CHAR.) ON INPUT FOR
      GET FILE STATEMENT AFTER 'TRANSMIT' DETECTED
      (ONFILE='@',ONSOURCE='@',ONCHARPOS=@)
```

ERROR condition (see also ONCODE = 3 and 9)

```
1002  GET/PUT STRING EXCEEDS STRINGSIZE

1003  CLOSE ERROR AFTER KEY|TRANSMIT-CONDITION WAS RAISED
      (ONFILE='@')

1004  ATTEMPT TO USE PAGE/LINE FOR A NON PRINT FILE (ONFILE='@')

1004  ATTEMPT TO USE PAGE/LINE IN A PUT STRING STATEMENT

1004  ATTEMPT TO USE SKIP/COLUMN IN A GET STRING STATEMENT

1007  NO PRECEDING READ SET OR READ INTO FOR REWRITE OR DELETE
      (ONFILE='@')

1008  INVALID ELEMENT VARIABLE IN STRING FOR GET STRING DATA

1009  INVALID FILE OPERATION (ONFILE='@')

1011  I/O - ERROR

1016  IMPLICIT OPEN UNSUCCESSFUL (ONFILE='@')
```

1017 END OF FILE FOR A NSTD-LABELED TAPE PREVIOUSLY ENCOUNTERED (ONFILE='@')

1018 UNEXPECTED END OF FILE DETECTED IN STREAM INPUT (ONFILE='@')

1101 ERROR IN I/O-SYSTEM (ONFILE='@')

1102 ERROR DURING PROCESSING OF REGIONAL DATA SET (ONFILE='@')

1400 ATTEMPT TO FREE NON-ALLOCATED STORAGE IN STANDARD AREA

1401 ATTEMPT TO FREE STORAGE IN STANDARD AREA NOT ON DOUBLE-WORD BOUNDARY

1410 NORMAL RETURN FROM AN AREA-ONUNIT WITHOUT 'FREE' STATEMENT

1411 ATTEMPT TO FREE NON-ALLOCATED STORAGE IN A NAMED AREA

1412 ATTEMPT TO FREE PART OF AN ELEMENT IN THE FREE-CHAIN OF A NAMED AREA

1413 CONTROL DATA OF A NAMED AREA DESTROYED

1414 ATTEMPT TO FREE STORAGE IN A NAMED AREA NOT ON DOUBLE-WORD BOUNDARY

1500 @ SHORT FLOATING POINT: ARGUMENT NEGATIVE

1501 @ LONG FLOATING POINT: ARGUMENT NEGATIVE

1502 @ EXTENDED FLOATING POINT: ARGUMENT NEGATIVE

1503 @ EXTENDED FLOATING POINT: ARGUMENT NOT POSITIVE

1503 @ EXTENDED FLOATING POINT EXPONENTIATION: BASE NEGATIVE

1504 @ SHORT FLOATING POINT: ARGUMENT NOT POSITIVE

1504 @ SHORT FLOATING POINT EXPONENTIATION: BASE NEGATIVE

1505 @ LONG FLOATING POINT: ARGUMENT NOT POSITIVE

1505 @ LONG FLOATING POINT EXPONENTIATION: BASE NEGATIVE

1506 @ SHORT FLOATING POINT: ABSOLUTE VALUE OF ARGUMENT EXCEEDS (2**18) * PI

1506 @ SHORT FLOATING POINT: ABSOLUTE VALUE OF ARGUMENT EXCEEDS (2**18) * 180

1506 @ COMPLEX SHORT FLOATING POINT: ABS(IMAG(X)) > 2**18 * PI

1506 @ COMPLEX SHORT FLOATING POINT: ABS(REAL(X)) > 2**18 * PI

1506 @ COMPLEX SHORT FLOATING POINT: ABS(2 * REAL(X)) > 2**18 * PI

1506 @ COMPLEX SHORT FLOATING POINT: ABS(2 * IMAG(X)) > 2**18 * PI

```
1506  @ COMPLEX SHORT FLOATING POINT EXPONENTIATION: IMMEDIATE
      OVERFLOW  ABS(IMAG(D2 * LOG(C2))) > 2**18 * PI

1507  @ LONG FLOATING POINT: ABSOLUTE VALUE OF ARGUMENT EXCEEDS
      (2**50)*PI

1507  @ LONG FLOATING POINT: ABSOLUTE VALUE OF ARGUMENT EXCEEDS
      (2**50)*180

1507  @ COMPLEX LONG FLOATING POINT: ABS(IMAG(X)) > 2**50 * PI

1507  @ COMPLEX LONG FLOATING POINT: ABS(REAL(X)) > 2**50 * PI

1507  @ COMPLEX LONG FLOATING POINT: ABS(2 * REAL(X)) > 2**50 * PI

1507  @ COMPLEX LONG FLOATING POINT: ABS(2 * IMAG(X)) > 2**50 * PI

1507  @ COMPLEX LONG FLOATING POINT EXPONENTIATION: IMMEDIATE
      OVERFLOW ABS(IMAG(D4 * LOG(C4))) > 2**50 * PI

1508  @ SHORT FLOATING POINT: ABSOLUTE VALUE OF ARGUMENT EXCEEDS
      (2**18) * PI

1508  @ SHORT FLOATING POINT: ABSOLUTE VALUE OF ARGUMENT EXCEEDS
      (2**18) * 180

1509  @ LONG FLOATING POINT: ABSOLUTE VALUE OF ARGUMENT EXCEEDS
      (2**50)*PI

1509  @ LONG FLOATING POINT: ABSOLUTE VALUE OF ARGUMENT EXCEEDS
      (2**50) * 180

1510  @ SHORT FLOATING POINT: ARGUMENTS BOTH ZERO

1510  @ COMPLEX SHORT FLOATING POINT: ARGUMENT = 0 + 0I

1511  @ LONG FLOATING POINT: ARGUMENTS BOTH ZERO

1511  @ COMPLEX LONG FLOATING POINT: ARGUMENT = 0 + 0I

1514  @ SHORT FLOATING POINT: ABSOLUTE VALUE OF ARGUMENT > = 1

1515  @ LONG FLOATING POINT:  ABSOLUTE VALUE OF ARGUMENT > = 1

1516  @ EXTENDED FLOATING POINT: ABSOLUTE VALUE OF ARGUMENT > = 1

1517  @ EXTENDED FLOATING POINT: ABSOLUTE VALUE OF ARGUMENT
      EXCEEDS (2**106) * PI

1517  @ EXTENDED FLOATING POINT: ABSOLUTE VALUE OF ARGUMENT
      EXCEEDS (2**106) * 180

1517  @ COMPLEX EXTENDED FLOATING POINT: ABS(IMAG(X)) > 2**100

1517  @ COMPLEX EXTENDED FLOATING POINT: ABS(REAL(X)) > 2**100

1517  @ COMPLEX EXTENDED FLOATING POINT: ABS(2 * REAL(X)) > 2**100
```

```
1517  @ COMPLEX EXTENDED FLOATING POINT: ABS(2 * IMAG(X)) > 2**100

1517  @ COMPLEX EXTENDED FLOATING EXPONENTIATION: IMMEDIATE
      OVERFLOW ABS(IMAG(D8 * LOG(C8))) > 2**100

1518  @ SHORT FLOATING POINT: ABSOLUTE VALUE OF ARGUMENT EXCEEDS 1

1519  @ LONG FLOATING POINT: ABSOLUTE VALUE OF ARGUMENT EXCEEDS 1

1520  @ EXTENDED FLOATING POINT: ABSOLUTE VALUE OF ARGUMENT
      EXCEEDS 1

1521  @ EXTENDED FLOATING POINT: ARGUMENTS BOTH ZERO

1521  @ COMPLEX EXTENDED FLOATING POINT: ARGUMENT = 0 + 0I

1522  @ EXTENDED FLOATING POINT: ABSOLUTE VALUE OF ARGUMENT
      EXCEEDS (2**106) * PI

1522  @ EXTENDED FLOATING POINT: ABSOLUTE VALUE OF ARGUMENT
      EXCEEDS (2**106) * 180

1549  @ INTEGER EXPONENTIATION: BASE ZERO, EXPONENT NOT POSITIVE

1550  @ SHORT FLOATING POINT: BASE ZERO, INTEGER EXPONENT NOT
      POSITIVE

1551  @ LONG FLOATING POINT: BASE ZERO, INTEGER EXPONENT NOT
      POSITIVE

1552  @ SHORT FLOATING POINT: BASE ZERO, EXPONENT NOT POSITIVE

1553  @ LONG FLOATING POINT: BASE ZERO, EXPONENT NOT POSITIVE

1554  @ COMPLEX SHORT FLOATING POINT: BASE IS 0 + 0I, INTEGER
      EXPONENT NOT POSITIVE

1555  @ COMPLEX LONG FLOATING POINT: BASE IS 0 + 0I, INTEGER
      EXPONENT NOT POSITIVE

1556  @ COMPLEX SHORT FLOATING POINT: BASE IS 0 + 0I, EXPONENT NOT
      POSITIVE REAL

1557  @ COMPLEX LONG FLOATING POINT: BASE IS 0 + 0I, EXPONENT NOT
      POSITIVE REAL

1558  @ COMPLEX SHORT FLOATING POINT: ARGUMENT = 1I OR -1I

1558  @ COMPLEX SHORT FLOATING POINT: ARGUMENT = 1 OR -1

1559  @ COMPLEX LONG FLOATING POINT: ARGUMENT = 1I OR -1I

1559  @ COMPLEX LONG FLOATING POINT: ARGUMENT = 1 OR -1

1560  @ EXTENDED FLOATING POINT: BASE ZERO, INTEGER EXPONENT NOT
      POSITIVE
```

1561 @ EXTENDED FLOATING POINT: BASE ZERO, EXPONENT NOT POSITIVE

1562 @ COMPLEX EXTENDED FLOATING POINT: BASE IS 0 + 0I, INTEGER EXPONENT NOT POSITIVE

1563 @ COMPLEX EXTENDED FLOATING POINT: BASE IS 0 + 0I, EXPONENT NOT POSITIVE REAL

1564 @ COMPLEX EXTENDED FLOATING POINT: ARGUMENT = 1I OR -1I

1564 @ COMPLEX EXTENDED FLOATING POINT: ARGUMENT = 1 OR -1

1570 ROUND(X,K): K IS NOT POSITIVE

1599 @: ERROR IN MATHEMATICAL LIBRARY ROUTINE

1600 DSCEXT: ERROR-STOP WITH AN UDS-FUNCTION. PARAMETER 'USERINFORMATION' MISSING OR DESTROYED.

1610 FORPL: ERROR IN PLI1 RUN-TIME-SYSTEM - REGISTRATION OF PLI1 TERMINATION ROUTINE IN FOR1 IMPOSSIBLE

1611 PLFOR: ERROR IN PLI1 RUN-TIME-SYSTEM - REGISTRATION OF FOR1 TERMINATION ROUTINE IN PLI1 IMPOSSIBLE

1612 FUNCTION CALL FOR1/ASS FROM PLI1: ILLEGAL TYPE OF 'RETURNS' ATTRIBUTE IN DECLARATION

1613 THE PLI1 RUNTIME SYSTEM DOES NOT ALLOW THE REGISTRATION OF A TERMINATION ROUTINE

1614 THE FOR1 RUNTIME SYSTEM DOES NOT ALLOW THE REGISTRATION OF A TERMINATION ROUTINE

3000 FORMAT SPECIFICATION E(W,D,S) CAUSES LOSS OF DIGITS

3000 VALUE OF W - FIELD TOO SMALL IN F-FORMAT SPECIFICATION

3001 MORE THAN 10 FORMAT LISTS NESTED

3002 TOO MANY OR TOO FEW PARAMETERS IN C-FORMAT ITEM

3006 INVALID ASSIGNMENT TO PICTURED CHARACTER STRING

3010 ILLEGAL FORMAT ITEM ON INPUT OR OUTPUT

3011 TARGET TYPE ILLEGAL ON INPUT

3012 SOURCE TYPE ILLEGAL ON OUTPUT

3013 BITSTRING IS SPECIFIED WITH INVALID RADIXFACTOR

3790 NORMAL RETURN FROM A SUBSCRIPTRANGE-ONUNIT

3791 NORMAL RETURN FROM A STORAGE-ONUNIT

3792 NORMAL RETURN FROM A STRINGRANGE-ONUNIT

3799 NORMAL RETURN FROM A CONVERSION-ONUNIT WITHOUT HAVING CORRECTED THE INVALID STRING

3802 @(A,N): ARRAY BOUND OUT OF RANGE, N NOT POSITIVE

3802 @(A,N): ARRAY BOUND OUT OF RANGE, N GREATER THAN DIMENSION OF A

3804 NAME HAS LENGTH GREATER THAN PERMITTED MAXIMUM (NAME='@')

4001 ATTEMPT TO ASSIGN TO UNALLOCATED CONTROLLED VARIABLE IN GET FILE DATA (ONFILE='@')

4001 ATTEMPT TO ASSIGN TO UNALLOCATED CONTROLLED VARIABLE IN GET STRING DATA

5000 P$ERROR CALLED WITHOUT ONCODE

5010 P$CALL: X'80' IN LAST PARAMETER MISSING OR TOO MANY PARAMETERS

5015 P$LINK: MODULE NOT LINKED, SEE DLL-MESSAGE

7000 UNABLE TO CLOSE THE FILE @ PROPERLY.

8081 CPU TIMER HAS ELAPSED (SEE MACRO SETIC)

8082 REAL-TIME TIMER HAS ELAPSED (SEE MACRO SETIC)

8091 NON-EXISTENT OPERATION CODE

8092 PRIVILEGED OPERATION EXCEPTION

8094 STORAGE PROTECTION EXCEPTION

8095 ADDRESSING EXCEPTION

8097 INVALID FIXED DECIMAL DATA

8098 ALIGNMENT ERROR

8099 TIME RUNOUT

14.3 Constraints on implementation

The constraints imposed by PLI1 on the use of arithmetic values, strings and picture specifications, length of names, input/output operations and on listings are tabulated below. When these bounds are being exceeded, the compiler and in some cases the runtime system will issue an appropriate error message.

1. Bounds of Arithmetic Values

Language elements	Limitation
Max. number of digits (incl. the leading zeros of a fixed-point constant)	56 for BINARY 16 for DECIMAL (although more digits may be specified, these digits will only affect the scale factor)
Max. number of digits in exponent of a floating-point constant	3 for BINARY 2 for DECIMAL
Max. number of digits in mantissa of a floating-point constant	112 for BINARY 33 for DECIMAL
Range of values of a fixed-point constant x	$-2^{31} + 1 \leq x \leq 2^{31}-1$ for BINARY $-(10^{15} - 1) \leq x \leq 10^{15} - 1$ for DECIMAL
Range of values of a floating-point constant x	$2^{-260} < \|x\| < 2^{252}$ approx. and x = 0 for BINARY $10^{-78} < \|x\| < 10^{75}$ and x = 0 for DECIMAL
Precision range p of an arithmetic fixed-point item	$1 \leq p \leq 31$ for BINARY $1 \leq p \leq 15$ for DECIMAL
Precision range p of an arithmetic floating-point item	$1 \leq p \leq 109$ for BINARY $1 \leq p \leq 33$ for DECIMAL
Range of scale k for an arithmetic fixed-point item	$-128 \leq k \leq 127$

2. Bounds of strings and picture specifications

Language elements	Limitation
Max. fixed length of a bit string or character string	32767 bits or characters. 509 bits or 510 characters, if conversion is necessary. (Both limitations apply after application of replicators.)
Limits of length l of declared character string	1 ≤ l ≤ 32767 characters
Limits of length l of declared bit string	1 ≤ l ≤ 32767 bits
Range of value of replicator r for strings	0 ≤ r ≤ 32767
Limits of length l of alphanumeric picture	1 ≤ l ≤ 511 characters
Limits of length l of numeric picture	1 ≤ l ≤ 255 characters
Max. numbers of numeric picture character in numeric picture specifications	p ≤ 15 fixed-point p ≤ 16 floating-point, 2 for exponent
Range of scale k in picture specifications	- 128 ≤ k ≤ 127

3. Bounds of names

Language elements	Limitation
Max. number of characters for identifier	255 (although more characters may be specified, the excess characters will be ignored)
Max. number of characters for identifier with attribute EXTERNAL	7 (longer names are condensed into the first 4 and last 3 characters. Character _ is replaced by $).
Max. number of characters for identifier with attribute FILE CONSTANT	31 for builtin function ONFILE, rest is truncated. 8 for LINK-name, rest is truncated. Character _ is replaced by $.
Max. number of characters for identifier in a data stream (with PUT/GET DATA)	255 incl. up to max. 64 qualifiers
Max. number of characters for names of %INCLUDE texts or libraries	8 (longer names are condensed into the first 5 and last 3 characters)

4. Bounds of arrays and areas (AREA)

Language elements	Limitation
Max. number of dimensions of an array	255
Max. length of an area (AREA)	16 777 191 characters

5. Bounds of lists and nestings

Language elements	Limitation
Max. number of %INCLUDE texts	255
Depth of nesting of %INCLUDE texts	5
Max. depth of logical nesting for all types of parentheses (blocks, DO statements, expressions, factorization in DECLARE, etc.)	only limited by stack, see control statement STORAGE in chapter 8
Max. number of bases in the qualifier chain of a BASED variable	128
Max. number of items in an INITIAL list	1024 for AUTOMATIC
Max. number of references to the left of an assignment symbol	128
Max. number of parameters	64

6. Bounds of input/output operations

Language elements	Limitation
Max. number of data elements in I/O lists	128
Depth of nesting in format lists	10
Max. value of n for SKIP(n), LINE(n), COLUMN(n)	2^{31}
Max. length of expression for GET STRING	32767 characters
Max. number of characters of value for GET	32767 (incl. subsequent blanks with GET LIST)
Max. number of characters of value for PUT	65536 (after conversion to external representation)
Length of names for PUT DATA	256 (without equal symbol)
Length of names for GET DATA	256 (with equal symbol)
Length of character string for DISPLAY	2044 characters
Range of record length r for record-formatted transfer in characters	$1 \leq r \leq 32767$ (min. of 12 for tape files, max. of 2048 for CONSECUTIVE on PAM and 32763 for ISAM) To disk files without PAM key (BLKCTRL=DATA) applies a max. of 32752 on SAM for CONSECUTIVE, a max. of 32492 on ISAM for CONSECUTIVE or INDEXED
Range of key length k in characters	$1 \leq k \leq 255$
Range of key location c	$1 \leq c \leq r - k$ and 0 in connection with KEYLOC
Additional rules for record length r for:	
RECFORM = F (except ISAM)	$r \leq$ BLKSIZE
RECFORM = F (for ISAM)	$r \leq$ BLKSIZE - 4
RECFORM = V (except REGIONAL(i))	$r \leq$ BLKSIZE - 4
RECFORM = V (for REGIONAL(3))	$r \leq$ BLKSIZE - 8
RECFORM = F/V, FCBTYPE=SAM/ISAM,BLKCTRL=DATA	$r \leq$ BLKSIZE - 16
ENVIRONMENT (INDEXED)	$r \geq c + k$ (c,k: see above)

14.4 Distinctions from PL/I-D

The space of the PLI1 compiler language is largely upward compatible with PL/I-D.

There are however some restrictions to be observed when converting programs.

1. Differences in Scope of Language

- GET STRING and COPY cannot be specified at the same time in PLI1.
- If an internal procedure is additionally declared by DECLARE, PLI1 will ignore the DCL statement, while PL/I-D will perform parameter adjustments under certain conditions.
- There are differences in the rules for the aggregate assignment. For instance, in the case of DCL A(10,20); A = A/A (1,1); PLI1 saves the value of A(1,1) in a dummy variable, prior to the evaluation of the expression.
- In PLI1 the number of picture characters for alphanumeric pictures (PICTURE) must not exceed 511.
- The keyword RETURNS in the PROCEDURE statement or ENTRY statement is mandatory (PL/I-D admits, for instance, the form A: PROC PTR;).
- Function procedures without arguments (also and in particular the BUILTIN functions DATE, TIME, etc.) must be called with empty parentheses unless they were declared explicitly; e.g.

```
A = DATE();     or    DCL DATE BUILTIN;
                      A = DATE;
```

- The %INCLUDE statement allows only the "%INCLUDE library (member);" format; "%INCLUDE library, member;" is not supported.

2. Differences in operational features

- Control of compiler and object program are different.
- The constraints on implementation are not the same.
- The tab settings for PUT LIST differ.
- The DISPLAY statement optionally works on the interactive terminal, on the operator console or on the system files SYSOUT or SYSDTA.
- A BUFFERS specification in ENVIRONMENT has no effect.
- The default record format for STREAM input/output is ’F’ for PL/I(D) and ’U’ for PLI1. Users converting from fixed to variable record length for better file handling in BS2000 should be aware of the special features of STREAM input/output with variable record length, esp. for edit-directed input.
- Variables are not preset by PLI1.

3. Differences at the procedure interface

- Procedures compiled by PL/I-D cannot be linked together with programs compiled by PLI1.
- Internal procedures cannot be declared explicitly; that is, DCL statements to that effect are ignored. Differences may arise if conversions (based on the declaration of parameters in the internal subroutine) are generated or not by PLI1.
- Assembler procedures written for PL/I-D can be executed in simple cases. Differences which may lead to errors exist in the following points:
  - The supply of parameters is different for more than 4 parameters, so is the return of function values for special data types.
  - The backtrace information (SNAP) that can be obtained via R13 and R15 has a different structure; condition handling etc. cannot be initiated.
  - The pseudo-register vector (PRV) has a different structure.
  - The rule of formation for abbreviated identifier with EXTERNAL attribute is different.
  - The management information for areas (AREA) has a different structure and is longer for PLI1.
  - The representation of the null pointer is different.
  - It is generally advisable to convert larger assembler subroutines to the standard assembler conventions (OPTIONS (ASSEMBLER)). See chapter 7.

14.5 Examples of sorting

Example 1

Processing an F file using separate user exits for input and output record processing

```
/*  EXAMPLE OF A PL1 SORT APPLICATION          */
/*  TASK:                                      */
/*  RECORDS OF AN F FILE ARE TO BE UPDATED     */
/*  ACCORDING TO THE FOLLOWING CRITERIA AND TO */
/*  BE SORTED ALPHABETICALLY:                  */
/*  1.THE CONTENTS OF A FIELD IS TO BE CHANGED */
/*    ACCORDING TO INSTRUCTIONS                */
/*  2.OF RECORDS THAT ARE PRESENT TWICE ONLY   */
/*    ONE RECORD IS ENTERED IN THE SORT FILE   */
/*  3.A SPECIFIC RECORD IS TO BE DELETED       */
/*  4.A RECORD IS TO BE ENTERED AND INSERTED   */
/*    AT THE PROPER PLACE                      */
PL1SRTA:PROC OPTIONS(MAIN);
DCL RETCODE       FIXED BIN(31) INIT(0);
DCL M             FIXED BIN(15) EXTERNAL;
DCL N             FIXED BIN(15) EXTERNAL;
DCL BS2SRTD       ENTRY (CHAR(*),CHAR(*),FIXED BIN(31),ENTRY,ENTRY);
DCL PL1E21        ENTRY EXTERNAL;
DCL PL1E23        ENTRY EXTERNAL;
DCL SYSPRINT      FILE CONSTANT;
/*
  M = 0;
  N = 0;
/*
D:    CALL BS2SRTD(
' SORT FIELDS=(1,15,A,CH,23,4,N,CH),FORMAT=CH,OPT=REC,SIZE=15',
' RECORD LENGTH=(80,80,80),TYPE=F',
RETCODE,PL1E21,PL1E23);
/*
EXIT:      PUT SKIP(2) LIST('RETCODEHP = ',RETCODE);
/*
END;

PL1E21: PROC (SATZEIN,RETCODE) RETURNS(CHAR(*));
  DCL RETCODE     FIXED BIN(31) PARM;
  DCL SATZEIN     CHAR(80) PARM;
  DCL M           FIXED BIN(15) EXTERNAL;
  DCL N           FIXED BIN(15) EXTERNAL;
  DCL EINFUEGESATZ CHAR(80) INIT('INSERT RECORD FOR CHECK ON PL1E21');
  DCL SYSPRINT    FILE CONSTANT;
/*                                                              */
    IF RETCODE = 8 THEN GOTO EXIT;
/*                                                              */
    M = M + 1;
    PUT SKIP LIST('RECORD NO.    = ',M);
/*                                                              */
  IF SUBSTR(SATZEIN,17,2) = '99'
        THEN DO;
               SUBSTR(SATZEIN,17,2) = '77';
               RETURN(SATZEIN);
             END;
```

```
/*                                                                    */
   IF SUBSTR(SATZEIN,18,12) = 'SORTIERDATEI'
         THEN DO;
                RETCODE = 4;
                RETURN(SATZEIN);
              END;
/*                                                                    */
  RETURN(SATZEIN);
/*                                                                    */
  EXIT: IF N = 0
         THEN DO;
                RETCODE = 12; N = 1;
                RETURN(EINFUEGESATZ);
               END;
  PUT SKIP(3) LIST('******************** DATEIENDE ************');
  N=0;M=0;
  RETURN(' ');
 END;

PL1E23: PROC (SATZEIN,RETCODE) RETURNS(CHAR(*));
  DCL RETCODE        FIXED BIN(31) PARM;
  DCL SATZEIN        CHAR(80) PARM;
  DCL N              FIXED BIN(15) EXTERNAL;
  DCL SYSPRINT       FILE CONSTANT;
/*                                                                     */
    IF RETCODE = 8 THEN GOTO EXIT;
/*                                                                     */
    N = N + 1;
    PUT SKIP LIST('RECORD NO.   = ',N);
/*                                                                     */
    IF RETCODE = 4
       THEN DO;
              PUT SKIP LIST('DELETED INPUT RECORD = ',SATZEIN);
              RETURN(SATZEIN);
             END;
/*                                                                     */
RETURN(SATZEIN);
/*                                                                     */
EXIT: PUT SKIP(3) LIST('************** DATEIENDE ************');
      RETURN(' ');
END;
```

Example 2

Processing a V-type file using separate user exits for input and output record

```
/*  EXAMPLE IF A PL1 SORT APPLICATION            */
/*  TASK:                                        */
/*  RECORDS OF A V FILE ARE TO BE UPDATED        */
/*  ACCORDING TO THE FOLLOWING CRITERIA AND TO   */
/*  BE SORTED ALPHABETICALLY:                    */
/*  1.IF A SPECIFIC FIELD HAS A SPECIFIED        */
/*    CONTENTS, IT IS TO BE CHANGED ACCORDING    */
/*    TO INSTRUCTIONS.                           */
/*  2.OF RECORDS THAT ARE PRESENT TWICE ONLY     */
/*    ONE IS ENTERED IN THE SORT FILE.           */
/*  3.AN IDENTIFIER RECORD IS TO BE ENTERED AS   */
/*    THE LAST RECORD OF THE SORT FILE.          */
PL1SORT:PROC OPTIONS(MAIN);
DCL RETCODE       FIXED BIN(31) INIT(0);
DCL M             FIXED BIN(15) EXTERNAL;
DCL N             FIXED BIN(15) EXTERNAL;
DCL BS2SRTD       ENTRY (CHAR(*),CHAR(*),FIXED BIN(31),ENTRY,ENTRY);
DCL PL1E21V       ENTRY EXTERNAL;
DCL PL1E23V       ENTRY EXTERNAL;
DCL SYSPRINT      FILE CONSTANT;
/*
  M = 0;
  N = 0;
/*
D:    CALL BS2SRTD(
' SORT FIELDS=(13,15,A,CH,23,4,N,CH),FORMAT=CH,OPT=REC,SIZE=15',
' RECORD LENGTH=(80,80,80),TYPE=F',
RETCODE,PL1E21V,PL1E23V);
/*
EXIT:      PUT SKIP(2) LIST('RETCODEHP = ',RETCODE);
/*
END;

PL1E21V: PROC (SATZEIN,RETCODE) RETURNS(CHAR(*));
  DCL RETCODE             FIXED BIN(31) PARM;
  DCL SATZEIN             CHAR(76) PARM;
    IF RETCODE = 8 THEN GOTO EXIT;
    BEGIN;
  DCL 1  VSATZ            BASED(P),
       2 SATZLNG1         FIXED BIN(15),
       2 FILLER1          CHAR(2),
       2 STRINGEIN        CHAR(1 REFER(SATZLNG1));
  DCL    P PTR;
  DCL 1  ARBEITSSATZ,
       2 SATZLNG2         FIXED BIN(15),
       2 FILLER2          CHAR(2),
       2 ARBEITSSTRING    CHAR(SATZLNG1 - 4);
  DCL  UEBERGABESATZ      CHAR(SATZLNG2) BASED(ADDR(ARBEITSSATZ));
  DCL  N                  FIXED BIN(15) EXTERNAL;
  DCL SYSPRINT            FILE CONSTANT;
/*                                                    */
    P = ADDR(SATZEIN);
    IF P = NULL() THEN GOTO EXIT;
```

```
/*                                                                    */
SATZLNG2 = SATZLNG1;FILLER2 = '  ';ARBEITSSTRING = STRINGEIN;N = N+1;
PUT SKIP LIST('RECORDNO-E21 = ',N);
/*                                                                    */
    IF SUBSTR(ARBEITSSTRING,27,12) = 'MAINT.-PROG.'
       THEN DO;
             SUBSTR(ARBEITSSTRING,27,12) = 'DIENSTPROG.';
             RETURN(UEBERGABESATZ);
            END;
    END;
/*                                                                    */
RETURN(UEBERGABESATZ);
/*                                                                    */
EXIT: PUT SKIP(4) LIST('************  DATEIENDE  ***************');
      N = 0;
      RETURN('  ');
/*                                                                    */
END;

PL1E23V: PROC (SATZEIN,RETCODE) RETURNS(CHAR(*));
  DCL RETCODE                 FIXED BIN(15) PARM;
  DCL SATZEIN                 CHAR(76) PARM;
    IF RETCODE = 8 THEN GOTO EXIT;
    BEGIN;
  DCL 1  VSATZ                BASED(ADDR(SATZEIN)),
       2 SATZLNG1             FIXED BIN(15),
       2 FILLER1              CHAR(2),
       2 STRINGEIN            CHAR(1 REFER(SATZLNG1));
  DCL 1  VSATZH               BASED(ADDR(SATZEIN)),
       2 SATZLNGH             FIXED BIN(15),
       2 FILLERH              CHAR(2),
       2 AUSSTRING            CHAR(SATZLNG1-4);
  DCL 1  ARBEITSSATZ,
       2 SATZLNG2             FIXED BIN(15),
       2 FILLER2              CHAR(2),
       2 ARBEITSTRING         CHAR(SATZLNG1-4);
  DCL UEBERGABESATZ           CHAR(SATZLNG2) BASED(ADDR(ARBEITSSATZ));
  DCL M                       FIXED BIN(15) EXTERNAL;
  DCL N                       FIXED BIN(15) EXTERNAL;
  DCL SYSPRINT                FILE CONSTANT;
/*                                                                    */
SATZLNG2 = SATZLNG1;FILLER2 = '  ';ARBEITSSTRING = STRINGEIN;N = N+1;
PUT SKIP LIST('RECORDNO-E23 = ',N);
/*                                                                    */
  IF RETCODE = 4
     THEN DO;
            PUT SKIP LIST('DOUBLE INPUT RECORD = ',VSATZH);
            RETURN(UEBERGABESATZ);
  END;
/*                                                                    */
  RETURN(UEBERGABESATZ);
/*                                                                    */
      IF M = 0
         THEN DO;
                M = 1;
                RETCODE = 12;
                SATZLNG2 = 76;
        ARBEITSSTRING = '99999999' || '9999-DATEIENDE'||
```

```
        '*****************************************';
                PUT SKIP LIST(EINGABESATZ = ',
                 '99999999'   '9999-DATEIENDE'
                 '*****************************************');
                RETURN(UEBERGABESATZ);
              END;
    END;
         PUT SKIP(4) LIST('*************** DATEIENDE ********');
         RETURN('  ');
END;
```

Example 3

Processing an F file using separate user exits for input and output record processing.

```
PL1SRTA:PROC OPTIONS(MAIN);
DCL RETCODE    FIXED BIN(31) INIT(0);
DCL M          FIXED BIN(15) EXTERNAL;
DCL N          FIXED BIN(15) EXTERNAL;
DCL BS2SRT     ENTRY OPTIONS(VARIABLE);
DCL PL1E21V    ENTRY EXTERNAL;
DCL PL1E23V    ENTRY EXTERNAL;
DCL SYSPRINT   FILE CONSTANT;
     DCL SPRUNG BIN FIXED(31);
/*                                                                  */
  M = 0;
  N = 0;
/*                                                                  */
/*           ****************************************               */
/*     SORT CALL WITH TYPE=V                                        */
/*     INPUT FILE IS ISAM FILE                                      */
/*     LENGHT IN THE FIRST 4 BYTES, ISAM KEY 8 BYTES                */
/*     SORT STARTING WITH 13TH BYTE                                 */
/*           ****************************************               */
/*     USER ENTRY PL1E21V:                                          */
/*     'MAINT. PROG.' IS CHANGED INTO 'UTILITY'                     */
/*     USER ENTRY PL1E23V:                                          */
/*     ONE RECORD IS INSERTED                                       */
/*                                                                  */
/*                                                                  */
/*     SORT 13 THROUGH 22 BY CHARACTER IN ASCENDING ORDER           */
/*     WITH PL1E21V AND PL1E23V                                     */
/*     OUTPUT FILE IS ISAM FILE                                     */
/*           ****************************************               */
DV:  CALL BS2SRT(
' SORT FIELDS=(13,10,A,B)',
' RECORD LENGTH=(92,92,92),TYPE=V'
RETCODE,PL1E21V,PL1E23V);
/*                                                                   */
/*           ****************************************               */
        PUT SKIP(2) LIST('RETCODEHP = ',RETCODE);
/*                                                                   */
END;
```

Example 4

Preparing a statistics deletions file arranged according to customer numbers.

```
/* EXAMPLE OF A PL1 SORT                                              */
/* TASK:                                                              */
/* 1. SORT OUTPUT RECORDS ACCORDING TO                                */
/*    CUSTOMER NUMBERS.                                               */
/* 2. SELECT OUTGOING RECORDS FROM                                    */
/*    TRANSACTION FILE.                                               */
/* 3. CREATE A SUMMARY FILE IN WHICH EVERY                            */
/*    CUSTOMER IS LISTED TO WHOM ANY ATICLE                           */
/*    HAS BEEN SOLD.                                                  */
  SORTP:PROC OPTIONS(MAIN);
  DCL BS2SRT ENTRY OPTIONS(VARIABLE);
  DCL RET BIN FIXED(31) INIT(0);
  CALL BS2SRT(' SORT FIELDS=(13,7,A,CH)',
  ' INCLUDE COND=(1,1,CH,EQ,C''I'')',
  ' SUM FIELDS=((40,3,ZD),(43,8,ZD,2))',
   RET,0,0);
   END;
```

14.6 Additional information on information messages

For the information messages described in section 3.9, the texts which are listed below provide further information on:

1. Information message 500
2. Information message 503
3. Information message 504

1. On information message 500

Information message 500 indicates that an out-line sequence with number 'n' was generated. The leftmost column of the following listing shows the number 'n' in ascending order, followed by an explanatory text describing the purpose of the outline sequence generated, and then the name of the runtime module in which the particular out-line sequence is implemented. The name of the module can be found e.g. in the linkage editor printout (see chapter 4).

No.	Function	Module
19	Normal program end, STOP	ITPRAHM#
102	ALLOCATE in standard area	ITPSTVW#
103	FREE in standard area	ITPSTVW#
104	ALLOCATE in named area	ITPSTVW#
105	FREE in named area	ITPSTVW#
108	Stack extension in block prolog	ITPSTVW#
110	Stack ext. for variable length AUTO and temporary variable	ITPSTVW#
114	Stack extension for RETURNS with * option	ITPSTVW#
200	Trace of a RETURN statement	ITPTHTR#
201	Trace of a GOTO statement	ITPTHTR#
202	Trace: GOTO with indexed label size	ITPTHTR#
203	Trace of a CALL statement	ITPTHTR#
204	Trace: program label trace	ITPTHTR#
205	Debugging aid checkpoint/halt point	ITPTHBK#
226	Test/Message on various conditions	ITPCDHD#
228	Test/Message on the CONVERSION condition	ITPCDHD#
229	SIGNAL statement	ITPCDHD#
248	ON statement	ITPCDHD#
249	REVERT statement	ITPCDHD#
256	Assignment of BIT strings, target NONVARYING	ITPBIT##
257	Assignment of BIT strings, target VARYING	ITPBIT##
258	NOT operation on BIT string, target NONVARYING	ITPBIT##
259	NOT operation on BIT string, target VARYING	ITPBIT##
260	Assignment of BIT/CHAR strings, with (possible) overlap	ITPBIT##
261	AND operation with BIT strings, target NONVARYING	ITPBIT##
262	AND operation with BIT strings, target VARYING	ITPBIT##
263	OR operation with BIT strings, target NONVARYING	ITPBIT##
264	OR operation with BIT strings, target VARYING	ITPBIT##
265	XOR operation with BIT strings, target NONVARYING	ITPBIT##

No.	Function	Module
266	XOR operation with BIT strings, target VARYING	ITPBIT##
267	Comparision of BIT strings	ITPBIT##
268	First call for chaining of BIT strings, target NONVARYING	ITPBIT##
269	First call for chaining of BIT strings, target VARYING	ITPBIT##
270	Follow-up call for chaining of BIT strings	ITPBIT##
271	Final call for chaining of BIT strings, special case	ITPBIT##
272	Final call for chaining of BIT strings, normal case	ITPBIT##
273	Check if all bits in BIT string equal '0'B, '1'B, otherwise	ITPBIT##
275	data conversion	ITPKONV#
276	Builtin function BOOL, target NONVARYING	ITPSBOB#
277	Builtin function BOOL, target VARYING	ITPSBOB#
278	Builtin function INDEX for BIT strings	ITPSIXB#
279	Builtin function INDEX for CHAR strings	ITPSIXC#
280	Builtin function SEARCH	ITPSSVC#
283	Builtin function VERIFY	ITPSSVC#
285	Trace of an ENTRY/PROC statement	ITPTHPT#
286	Builtin function SQRT, argument FLOAT single	ITPRRE##
287	Builtin function SQRT, argument FLOAT double	ITPRRD##
288	Builtin function SQRT, argument FLOAT extended	ITPRRW##
289	Builtin function SQRT, argument CPLX FLOAT single	ITPRCE##
290	Builtin function SQRT, argument CPLX FLOAT double	ITPRCD##
291	Builtin function SQRT, argument CPLX FLOAT extended	ITPRCW##
292	Builtin function SIN, argument FLOAT single	ITPRRE##
293	Builtin function SIN, argument FLOAT double	ITPRRD##
294	Builtin function SIN, argument FLOAT extended	ITPRRW##
295	Builtin function SIN, argument CPLX FLOAT single	ITPRCE##
296	Builtin function SIN, argument CPLX FLOAT double	ITPRCD##
297	Builtin function SIN, argument CPLX FLOAT extended	ITPRCW##
298	Builtin function SIND, argument FLAT single	ITPRRE##
299	Builtin function SIND, argument FLOAT double	ITPRRD##
300	Builtin function SIND, argument FLOAT extended	ITPRRW##
304	Builtin function COS, argument FLOAT single	ITPRRE##
305	Builtin function COS, argument FLOAT double	ITPRRD##
306	Builtin function COS, argument FLOAT extended	ITPRRW##
307	Builtin function COS, argument CPLX FLOAT single	ITPRCE##
308	Builtin function COS, argument CPLX FLOAT double	ITPRCD##
309	Builtin function COS, argument CPLX FLOAT extended	ITPRCW##
310	Builtin function COSD, argument FLOAT single	ITPRRE##
311	Builtin function COSD, argument FLOAT double	ITPRRD##
312	Builtin function COSD, argument FLOAT extended	ITPRRW##
316	Builtin function TAN, argument FLOAT single	ITPRRE##
317	Builtin function TAN, argument FLOAT double	ITPRRD##
318	Builtin function TAN, argument FLOAT extended	ITPRRW##
319	Builtin function TAN, argument CPLX FLOAT single	ITPRCE##
320	Builtin function TAN, argument CPLX FLOAT double	ITPRCD##
321	Builtin function TAN, argument CPLX FLOAT extended	ITPRCW##
322	Builtin function TAND, argument FLOAT single	ITPRRE##
323	Builtin function TAND, argument FLOAT double	ITPRRD##
324	Builtin function TAND, argument FLOAT extended	ITPRRW##
328	Builtin function ASIN, argument FLOAT single	ITPRRE##
329	Builtin function ASIN, argument FLOAT double	ITPRRD##
330	Builtin function ASIN, argument FLOAT extended	ITPRRW##
334	Builtin function ASIND, argument FLAOT single	ITPRRE##

No.	Function	Module
335	Builtin function ASIND, argument FLOAT double	ITPRRD##
336	Builtin function ASIND, argument FLOAT extended	ITPRRW##
340	Builtin function ACOS, argument FLOAT single	ITPRRE##
341	Builtin function ACOS, argument FLOAT double	ITPRRD##
342	Builtin function ACOS, argument FLOAT extended	ITPRRW##
346	Builtin function ACOSD, argument FLOAT single	ITPRRE##
347	Builtin function ACOSD, argument FLOAT double	ITPRRD##
348	Builtin function ACOSD, argument FLOAT extended	ITPRRW##
352	Builtin function ATAN, argument FLOAT single	ITPRRE##
353	Builtin function ATAN, argument FLOAT double	ITPRRD##
354	Builtin function ATAN, argument FLOAT extended	ITPRRW##
355	Builtin function ATAN, argument CPLX FLOAT single	ITPRCE##
356	Builtin function ATAN, argument CPLX FLOAT double	ITPRCD##
357	Builtin function ATAN, argument CPLX FLOAT extended	ITPRCW##
358	Builtin function ATAND, argument FLOAT single	ITPRRE##
359	Builtin function ATAND, argument FLOAT double	ITPRRD##
360	Builtin function ATAND, argument FLOAT extended	ITPRRW##
364	Builtin function LOG2, argument FLOAT single	ITPRRE##
365	Builtin function LOG2, argument FLOAT double	ITPRRD##
366	Builtin function LOG2, argument FLOAT extended	ITPRRW##
370	Builtin function LOG, argument FLOAT single	ITPRRE##
371	Builtin function LOG, argument FLOAT double	ITPRRD##
372	Builtin function LOG, argument FLOAT extended	ITPRRW##
373	Builtin function LOG, argument CPLX FLOAT single	ITPRCE##
374	Builtin function LOG, argument CPLX FLOAT double	ITPRCD##
375	Builtin function LOG, argument CPLC FLOAT extended	ITPRCW##
376	Builtin function LOG10, argument FLOAT single	ITPRRE##
377	Builtin function LOG10, argument FLOAT double	ITPRRD##
378	Builtin function LOG10, argument FLOAT extended	ITPRRW##
382	Builtin function EXP, argument FLOAT single	ITPRRE##
383	Builtin function EXP, argument FLOAT double	ITPRRD##
384	Builtin function EXP, argument FLOAT extended	ITPRRW##
385	Builtin function EXP, argument CPLX FLOAT single	ITPRCE##
386	Builtin function EXP, argument CPLX FLOAT double	ITPRCD##
387	Builtin function EXP, argument CPLX FLOAT extended	ITPRCW##
388	Builtin function ATANH, argument FLOAT single	ITPRRE##
389	Builtin function ATANH, argument FLOAT double	ITPRRD##
390	Builtin function ATANH, argument FLOAT extended	ITPRRW##
391	Builtin function ATANH, argument CPLX FLOAT single	ITPRCE##
392	Builtin function ATANH, argument CPLX FLOAT double	ITPRCD##
393	Builtin function ATANH, argument CPLX FLOAT extended	ITPRCW##
394	Builtin function COSH, argument FLOAT single	ITPRRE##
395	Builtin function COSH, argument FLOAT double	ITPRRD##
396	Builtin function COSH, argument FLOAT extended	ITPRRW##
397	Builtin function COSH, argument CPLX FLOAT single	ITPRCE##
398	Builtin function COSH, argument CPLX FLOAT double	ITPRCD##
399	Builtin function COSH, argument CPLX FLOAT extended	ITPRCW##
400	Builtin function ERF, argument FLOAT single	ITPRRE##
401	Builtin function ERF, argument FLOAT double	ITPRRD##
402	Builtin function ERF, argument FLOAT extended	ITPRRW##
406	Builtin function ERFC, argument FLOAT single	ITPRRE##
407	Builtin function ERFC, argument FLOAT double	ITPRRD##
408	Builtin function ERFC, argument FLOAT extended	ITPRRW##
412	Builtin function SINH, argument FLOAT single	ITPRRE##

No.	Function	Module
413	Builtin function SINH, argument FLOAT double	ITPRRD##
414	Builtin function SINH, argument FLOAT extended	ITPRRW##
415	Builtin function SINH, argument CPLX FLOAT single	ITPRCE##
416	Builtin function SINH, argument CPLX FLOAT double	ITPRCD##
417	Builtin function SINH, argument CPLX FLOAT extended	ITPRCW##
418	Builtin function TANH, argument FLOAT single	ITPRRE##
419	Builtin function TANH, argument FLOAT double	ITPRRD##
420	Builtin function TANH, argument FLOAT extended	ITPRRW##
421	Builtin function TANH, argument CPLX FLOAT single	ITPRCE##
423	Builtin function TANH, argument CPLX FLOAT extended	ITPRCW##
424	Builtin function ATAN2, argument FLOAT single	ITPRRE##
425	Builtin function ATAN2, argument FLOAT double	ITPRRD##
426	Builtin function ATAN2, argument FLOAT extended	ITPRRW##
430	Builtin function ATAND2, argument FLOAT single	ITPPRE##
431	Builtin function ATAND2, argument FLOAST double	ITPRRD##
432	Builtin function ATAND2, argument FLOAT extended	ITPRRW##
436	Exponentiation integer short ** integer short	ITPRND##
437	Exponentiation integer short ** integer long	ITPRND##
438	Exponentiation integer long ** integer short	ITPRND##
439	Exponentiation integer long ** integer long	ITPRND##
440	Exponentiation FLOAT single ** integer short	ITPRRE##
441	Exponentiation FLOAT double ** integer short	ITPRRD##
442	Exponentiation FLOAT extended ** integer short	ITPRRW##
443	Exponentiation CPLX FLOAT single ** integer short	ITPRCE##
444	Exponentiation CPLX FLOAT double ** integer short	ITPRCD##
445	Exponentiation CPLX FLOAT extended ** integer short	ITPRCW##
446	Exponentiation FLOAT single ** integer long	ITPRRE##
447	Exponentiation FLOAT double ** integer long	ITPRRD##
448	Exponentiation FLOAT extended ** integer long	ITPRRW##
449	Exponentiation CPLX FLOAT single ** integer long	ITPRCE##
450	Exponentiation CPLX FLOAT double ** integer long	ITPRCD##
451	Exponentiation CPLX FLOAT extended ** integer long	ITPRCW##
452	Exponentiation FLOAT single ** FLOAT single	ITPRRE##
453	Exponentiation FLOAT double ** FLOAT double	ITPRRD##
454	Exponentiation FLOAT extended ** FLOAT extended	ITPRRW##
455	Exponentiation CPLX FLOAT single ** CPLX FLOAT single	ITPRCE##
456	Exponentiation CPLX FLOAT double ** CPLX FLOAT double	ITPRCD##
457	Exponentiation CPLX FLOAT extended ** CPLX FLOAT extended	ITPRCW##
458	Division of type FLOAT operands extended	ITPRRW##
459	Builtin function MOD, argument FLOAT extended	ITPRRW##
460	Builtin function COPY for BIT strings, target NONVARYING	ITPSCRB#
461	Builtin function COPY for BIT strings, target VARYING	ITPSCRB#
462	Builtin function REVERSE for BIT strings, target NONVARYING	ITPSCRB#
463	Builtin function REVERSE for BIT strings, target VARYING	ITPSCRB#
464	Builtin function COPY for CHAR strings, target NONVARYING	ITBSCRC#
465	Builtin function COPY for CHAR strings, target VARYING	ITBSCRC#
466	Builtin function REVERSE for CHAR strings, target NONVARYING	ITBSCRC#
467	Builtin function REVERSE for CHAR strings, target VARYING	ITBSCRC#
468	Builtin function TRANSLATE, 3 arguments, target NONVARYING	ITPSTRC#
469	Builtin function TRANSLATE, 3 arguments, target VARYING	ITPSTRC#
470	Builtin function TRANSLATE, 2 arguments, target NONVARYING	ITPSTRC#
471	Builtin function TRANSLATE, 2 arguments, target VARYING	ITPSCRC#
485	Test/message on the SUBSCRIPTRANGE condition	ITPCOND#
486	Test/Message on the STRINGRANGE condition	ITPCOND#

No.	Function	Module
487	Builtin function ROUND (ISO case), argument DEC FLOAT	ITPRND##
488	Builtin function ROUND (ISO case), argument BIN FLOAT	ITPRND##
489	Builtin function ROUND (ISO case), argument CPLX BIN FLOAT	ITPRND##
490	Builtin function ROUND (ISO case), argument CPLX DEC FLOAT	ITPRND##
491	Builtin function ROUND (NOISO case), argument CPLX BIN FLOAT	ITPRND##
492	Builtin function ROUND (NOISO case), argument CPLX DEC FLOAT	ITPRND##
493	Builtin function ABS, argument CPLX BIN FLOAT single	ITPRCE##
494	Builtin function ABS, argument CPLX BIN FLOAT double	ITPRCD##
485	Builtin function ABS, argument CPLX BIN FLOAT extended	ITPRCW##
496	Builtin function ABS, argument CPLX BIN FIXED short	ITPACB##
497	Builtin function ABS, argument CPLX BIN FIXED long	ITPACB##
498	Comparison of operands of type CPLX BIN FIXED	ITPACB##
499	Addition of operands of type CPLX BIN FIXED	ITPACB##
500	Subtraction of operands of type CPLX BIN FIXED	ITPACB##
501	Multiplication of operands of type CPLX BIN FIXED	ITPACB##
502	Division of operands of type CPLX BIN FIXED	ITPACB##
503	Builtin function ROUND, argument CPLX BIN FIXED	ITPACB##
504	Builtin function ABS, argument CPLX DEC FIXED	ITPACD##
505	Comparison of operands of type CPLX DEC FIXED	ITPACD##
506	Addition of operands of type CPLX DEC FIXED	ITPACD##
507	Subtraction of operands of type CPLX DEC FIXED	ITPACD##
508	Multiplication of operands of type CPLX DEC FIXED	ITPACD##
509	Division of operands of type CPLX DEC FIXED	ITPACD##
510	Builtin function ROUND, argument CPLX DEC FIXED	ITPACD##
511	Comparison of operands of type CPLX FLOAT	ITPACF##
512	Addition of operands of type CPLX FLOAT	ITPACF##
513	Subtraction of operands of type CPLX FLOAT	ITPACF##
514	Multiplication of operands of type CPLX FLOAT	ITPACF##
515	Division of operands of type CPLX FLOAT	ITPACF##
516	Builtin function REPEAT for BIT strings, target NONVARYING	ITPSCRB#
517	Builtin function REPEAT for BIT strings, target VARYING	ITPSCRB#
518	Builtin function REPEAT for CHAR strings, target NONVARYING	ITPSCRC#
519	Builtin function for CHAR, target VARYING	ITPSCRC#
520	Calling an entry of type OPTIONS(FORTRAN)	ITPLXFV#
521	Calling an entry of type OPTIONS(FORTRAN INTER)	ITPLXFV#
522	Calling an entry of type OPTIONS(COBOL)	ITPLXFV#
770	End call for GET statement	ITPGET##
771	End call for PUT statement	ITPPUT##
772	PUT DATA statement with list, follow-up call for 1 data elem.	ITPPVD##
773	GET LIST statement, follow-up call for 1 data element	ITPGVL##
774	GET EDIT statement, follow-up call for 1 data element	ITPGVE##
775	PUT LIST statement, follow-up call for 1 data element	ITPPVL##
776	PUT EDIT statement, follow-up call for 1 data element	ITPPVE##
777	Record oriented I/O; transport statement	ITPIORC#
778	OPEN statement	ITPOPEN#
779	CLOSE statement	ITPOPEN#
780	GET DATA statement or initial call for GET statement	ITPGET##
781	PUT DATA without list or initial call for PUT statement	ITPPUT##
831	Trace for ENTRY/PROC in optimized procedure	ITPTHPT#

2. On information message 503

Information message 503 indicates that an out-line sequence was generated whose name is 'e'. The leftmost column of the following listing shows the names 'n' in ascending order, followed by an explanatory text describing the purpose of the out-line sequence generated, and then the name of the runtime module in which the particular out-line sequence is implemented. The name of the module can be found e.g. in the linkage editor printout (see chapter 4).

Entry	Function	Module
AFTERB##	Builtin function AFTER, argument BIT string	ITPSAFB#
AFTERC##	Builtin function AFTER, argument CHAR string	ITPSAFC#
ALL#####	Builtin function ALL	ITPBALL#
ANY#####	Builtin function ANY	ITPBANY#
BEFOREB#	Builtin function BEFORE, argument BIT string	ITPSAFB#
BEFOREC#	Builtin function BEFORE, argument CHAR string	ITPSAFC#
BOUND###	Builtin function DIM, HBOUND, LBOUND	ITPBBND#
COUNT###	Builtin function COUNT	ITPOPEN#
DATAFLD#	Builtin function DATAFIELD	ITPCDHD#
DATE####	Builtin function DATE	ITPBDAT#
DECATB##	Builtin function DECAT, argument BIT string	ITPSAFB#
DECATC##	Builtin function DECAT, argument CHAR string	ITPSAFC#
DISPPLY#	DISPLAY statement without REPLY	ITPIODI#
DISPRLY#	DISPLAY statement with REPLY	ITPIODI#
EVERY###	Builtin function EVERY	ITPBEVR#
GETLENO#	Builtin function LINENO	ITPOPEN#
GETPANO#	Builtin function PAGENO	ITPOPEN#
ON$CHAR#	Builtin function ONCHAR	ITPCDHD#
ON$CNT##	Builtin function ONCOUNT	ITPCDHD#
ON$CODE#	Builtin function ONCODE	ITPCDHD#
ON$FILE#	Builtin function ONFILE	ITPCDHD#
ON$FLD##	Builtin function ONFIELD	ITPCDHD#
ON$INTR#	Builtin function ONINTR	ITPCDHD#
ON$KEY##	Builtin function ONKEY	ITPCDHD#
ON$LOC##	Builtin function ONLOC	ITPCDHD#
ON$SRCE#	Builtin function ONSOURCE	ITPCDHD#
POLY####	Builtin function POLY	ITPBPLY#
PV$CHAR#	Pseudo variable ONCHAR	ITPCDHD#
PV$SRCE#	Pseudo variable ONSOURCE	ITPCDHD#
SAMEKEY#	Builtin function SAMEKEY	ITPBSKY#
SETPENO#	Pseudo variable PAGENO	ITPOPEN#
SOME####	Builtin function SOME	ITPBSOM#
ST$NMAS#	Assignment of named AREAs	ITPSTVW#
TIME####	Builtin function TIME	ITPBDAT#
VALID###	Builtin function VALID	ITPKONV#

3. On information message 504

Information message 504 indicates that an out-line sequence for a type 't' conversion was generated. The leftmost column of the following listing shows the types 't' in ascending order, followed by an explanatory text describing the purpose of the out-line sequence generated.

This listing uses the following abbreviations:

–→	Direction of conversion
BIT	BIT strings ALIGNED or UNALIGNED
CHARACTER	CHARACTER strings ALIGNED or UNALIGNED
FLOAT	Unless otherwise specified: DEC FLOAT or BIN FLOAT, single or double precision
PIC(...)	PICTURE of alphanumeric or numeric type or with numeric type specified
numeric	Arithmetic data types DECIMAL or BINARY, FLOAT, or FIXED

Conversion from/to COMPLEX, unless otherwise stated, is generally carried out separately for real and imaginary parts; 2 calls are therefore issued.

Conversions from/to FLOAT/PIC (float) with extended precision (4-fold) on the one hand and with single and double precision on the other hand is carried out via separate keys.

Type	Conversion function
1	BIT -> BIT
2	BIT -> CHARACTER
3	BIT -> BINARY FIXED
4	BIT -> DECIMAL FIXED
5	BIT -> FLOAT
6	BIT -> PICTURE(decimal fixed)
7	BIT -> PICTURE(decimal float)
8	BIT -> PICTURE(alphanumeric)
9	CHARACTER or PICTURE(alphanumeric) -> CHARACTER or PICTURE (alphanumeric)
10	CHARACTER or PICTURE(alphanumeric) -> BIT
11	CHARACTER or PICTURE(alphanumeric) -> numeric or PICTURE(numeric), real or complex
12	CHARACTER or PICTURE(alphanumeric) -> PICTURE(alphanumeric)
13	numeric or PICTURE(numeric), real or complex -> CHARACTER or PICTURE(alphanumeric)
17	BINARY FIXED -> DECIMAL FIXED
18	BINARY FIXED -> FLOAT
19	BINARY FIXED - > BIT
20	BINARY FIXED -> PICTURE(decimal fixed)
21	BINARY FIXED -> PICTURE(decimal float)
24	FLOAT -> BINARY FIXED
25	FLOAT -> DECIMAL FIXED
26	FLOAT -> PICTURE(decimal fixed)
27	FLOAT -> PICTURE(decimal float)
28	FLOAT -> BIT
29	FLOAT -> CHARACTER according to E format
31	DECIMAL FIXED -> DECIMAL FIXED
32	DECIMAL FIXED -> BINARY FIXED
33	DECIMAL FIXED -> FLOAT
34	DECIMAL FIXED -> BIT
35	DECIMAL FIXED -> PICTURE(decimal fixed)
36	DECIMAL FIXED -> PICTURE (decimal float)
48	PICTURE(numeric) -> PICTURE(decimal fixed)
49	PICTURE(numeric) -> BINARY FIXED
50	PICTURE(numeric) -> DECIMAL FIXED
51	PICTURE(numeric) -> FLOAT
52	PICTURE(numeric) -> PICTURE(decimal float)
53	PICTURE(numeric) -> BIT
67	numeric, BIT, CHARACTER or PICTURE -> FLOAT extended precision
69	FLOAT extended precision -> CHARACTER according to E format
70	FLOAT extended precision -> numeric, BIT, CHARACTER or PICTURE
73	FLOAT -> FLOAT
74	BINARY FIXED -> BINARY FIXED
75	BINARY FIXED -> BIT
76	FLOAT -> BIT
77	FLOAT -> CHARACTER according to E format

14.7 Runtime modules

The following is a listing of the names of the modules of the static runtime system, followed by an explanation of the features provided by each module. The modules are incorporated whenever their particular services are required in the user program.

On those lines which begin with "***", you find those modules of the runtime system which refer to the preceding module so that their incorporation into a program makes it mandatory to incorporate that module also.

```
Module    Function / invoked by module (***)

ITP#AOS#  Initial handling for PLI1 objects when static runtime system
          is used
ITPACB##  Out-line strings for addition, subtraction, multiplication,
          division, comparison, absolute value and rounding of CPLX BIN
          FIXED operands.
ITPACD##  Out-line strings for addition, subtraction, mulitplication,
          division, comparison, absolute value and rounding of CPLX DEC
          FIXED operands.
ITPACF##  Out-line strings for addition, subtraction, multiplication,
          division and comparison of CPLX FLOAT
ITPBALL#  Builtin function ALL
ITPBANY#  Builtin function ANY
ITPBBND#  Builtin functions DIM, RBOUND,LBOUND
ITPBDAT#  Builtin functions DATE, TIME
ITPBEVR#  Builtin function EVERY
ITPBIT##  Various operations with BIT strings (AND, OR, XOR, NOT, assign,
          chain, compare, test all bits  for '0'B or '1'B); assign CHAR
          strings with possible overlapping
     ***  ITPBALL#, ITPBANY#, ITPBEVR#, ITPBSOM#, ITP#AOS#
ITPBITN#  Pad storage area with binary zeros
     ***  ITPSBOB#, ITPSCRB#
ITPBPLY#  Builtin function POLY
ITPBSKY#  Builtin function SAMEKEY
ITPBSOM#  Builtin function SOME
ITPCDHD#  Condition handling, message from various conditions,
          ON, REVERT and SIGNAL statement, builtin functions/
          pseudo-variables DATAFIELD, ONCHAR, ONCODE, ONCOUNT,
          ONFIELD, ONFILE, ONINTR, ONKEY, ONLOC, ONSOURCE,
          ERROUT utility
     ***  ITP#AOS#, ITPGVD##, ITPGVE##, ITPOPEN##, ITPPVD##, ITPPVE##
ITPCOND#  Message from STRINGRANGE and SUBSCRIPTRANGE condition
     ***  ITP#AOS#tility
ITPDASI#  Nucleus for builtin function ASIN/ACOS, double precision
     ***  ITPRRD##
ITPDATH#  Nucleus for builtin function ATANH, double precision
     ***  ITPYATA#, ITPRRD##
ITPDAT2#  Nucleus for builtin function ATAN/ATAN2, double precision
     ***  ITPYATA#, ITPYLOG#, ITPRRD##
ITPDERF#  Nucleus for builtin function  ERF/ERFC, double precision
     ***  ITPRRD##
```

Module	Function / invoked by module (***)
ITPDEXP#	Nucleus for builtin function EXP, double precision
***	ITPDERF#, ITPDPWR#, ITPDSIH#, ITPDTAH#, ITPYEXP#,
***	ITPYSIN#, ITPRRD##
ITPDLOG#	Nucleus for builtin function LOG/LOG10/LOG2, double precision
***	ITPDATH#, ITPDPWR#, ITPYLOG#, ITPYPWC#, ITPRRD##
ITPDPWI#	Nucleus for exponentiation FLOAT double precision ** integer
***	ITPRRD##
ITPDPWR#	Nucleus for exponentiation FLOAT double ** FLOAT double precision
***	ITPRRD##
ITPDSIH#	Nucleus for builtin function SINH/COSH, double precision
***	ITPYTAN#, ITPRRD##
ITPDSIN#	Nucleus for builtin function SIN/COS, double precision
***	ITPYEXP#, ITPYSIN#, ITPYTAN#, ITPRRD#
ITPDSQR#	Nucleus for builtin function SQRT, double precision
***	ITPDASI#, ITPYABS, ITPYSQR#, ITPRRD##
ITPDTAH#	Nucleus for builtin function TANH, double precision
***	ITPRRD##
ITPDTAN#	Nucleus for builtin function TAN/COTAN, double precision
***	ITPRRD##
ITPEASI#	Nucleus for builtin function ASIN/ACOS, single precision
***	ITPRRE##
ITPEATH#	Nucleus for builtin function ATANH, single precision
***	ITPXATA#, ITPRRE#
ITPEAT2#	Nucleus for builtin function ATAN/ATAN2, single precision
***	ITPXATA#, ITPXLOG#, ITPRRE##
ITPEERF#	Nucleus for builtin function ERF/ERFC, single precision
***	ITPRRE##
ITPEEXP#	Nucleus for builtin function EXP, single precision
***	ITPEERF#, ITPEPWR#, ITPESIH#, ITPETAH#, ITPXEXP#, ITPXSIN#,
***	ITPRRE##
ITPELOG#	Nucleus for builtin function LOG/LOG10/LOG2, single precision
***	ITPEATH#, ITPEPWR#, ITPXLOG#, ITPXPWC#, ITPRRE##
ITPEPWI#	Nucleus for exponentiation FLOAT single precision ** integer
***	ITPRRE##
ITPEPWR#	Nucleus for exponentiation FLOAT single ** FLOAT single precision
***	ITPRRE##
ITPESIH#	Nucleus for builtin function SINH/COSH, single precision
***	ITPXTAN#, ITPRRE##
ITPESIN#	Nucleus for builtin function SINH/COSH, single precision
***	ITPXEXP#, ITPXSIN#, ITPXTAN#, ITPRRE#
ITPESQR#	Nucleus for builtin function SQRT, single precision
***	ITPEASI#, ITPXABS#, ITPXSQR#, ITPRRE##
ITPETAH#	Nucleus for builtin function TANH, single precision
***	ITPRRE##
ITPETAN#	Nucleus for builtin function TAN/COTAN, single precision
***	ITPRRE##

Module	Function / invoked by module (***)
ITPFL###	Supply format description for a variable during I/O in EDIT mode
***	ITPGVE##, ITPPVE##
ITPGDT##	Supply associated variable address and descriptor for the GET DATA statement for a variable
***	ITPGVD##
ITPGDTX#	Supply associated variable address and descriptor for the GET DATA statement for a variable in the XS case
***	ITPGVDX##
ITPGET##	Initial and final conditions for all GET statements, (follow-up) calls for reading an element (entry date/name) for all GET statements, control read operations (set line or column, skip character), Perform GET DATA statement with/without listing
***	ITPFL###, ITPGVD##, ITPGVE##, ITPGVL##, ITPOPCL#
ITPGVD##	Nucleus routine for performing GET DATA statement with/without listing
***	ITPGET##
ITPGVDX#	Nucleus routine for performing GET DATA statement with/without listing (in XS case)
***	ITPGET##
ITPGVE##	Input data element from GET DATA statement (follow-up call)
ITPGVL##	Input data element from GET LIST statement (follow-up call); also with an element when GET DATA is used.
***	ITPGVD##
ITPHXDC#	HEXDEC utility
***	ITPTHRD#
ITPIODI#	DISPLAY statement with/without REPLY
ITPIORC#	Transport statement for record-oriented I/O; also perform transport and positioning jobs for stream-oriented I/O; format REGIONAL files
***	ITPGET##, ITPIOSY#, ITPOPCL#, ITPOPEN#, ITPPUT##
ITPIORX#	Transport statement for record-oriented I/O etc. according to ITPIORC# in XS case
***	ITPGET##, ITPIOSY#, ITPOPCX#, ITPOPEN#, ITPPUT##
ITPIOSY#	Perform I/O transfer jobs from/to BS2000 system files (SYSDTA, SYSCMD, SYSOUT, SYSLST, terminal, operator console); analyze SYSFILE runtime option and set tabulator stops
***	ITPIODI#, ITPGET#, ITPOPEN#, ITPPUT#, ITPIORC#
ITPIPWI#	Nucleus for exponentiation of integer operands
***	ITPRND##
ITPKONV#	Conversion package for all on-line conversions for object routines, I/O and mathematical library; builtin function VALID
***	ITPGVE##, ITPGVL##, ITPPVE##, ITPPVL##, ITPRND##
ITPLXFC#	Common area for transition routine ITPLXFN
ITPLXFN#	Transition routine for converting the program environment when calling PLI1 programs from COBOL or FORTRAN main program
***	ITPLXFC#

Module	Function / invoked by module (***)
ITPLXFV#	Transition routine for converting the program environment when calling COBOL and FORTRAN entry points with/without INTER option
ITPOPCL#	Nucleus routine for opening and closing files
***	ITPOPEN#
ITPOPCX#	Nucleus routine for opening and closing files in XS case
***	ITPOPEN#
ITPOPEN#	OPEN and CLOSE statement; implicit opening and closing of files; builtin functions/pseudo-variables COUNT, LINENO, PAGENO
***	ITPGET##, ITPPUT##, ITPIORC#
ITPOPRD#	Routine for reading and analyzing object control options (RUNOPT)
***	ITP#AOS#
ITPOPWR#	Routine for logging object control options (RUNOPT)
***	ITP#AOS#
ITPPDBA#	Control output for PUT DATA without listing
***	ITPPUT##
ITPPDBX#	Control output for PUT DATA without listing in XS case
***	ITPPUT##
ITPPDSD#	Subroutine for I/O in DATA mode
***	ITPGDT##, ITPPVD##
ITPPDSX#	Subroutine for I/O in DATA mode (XS case)
***	ITPGDTX#, ITPPVDX#
ITPPDVA#	Nucleus routine for output using PUT DATA without listing
***	ITPPDBA#
ITPPDVX#	Nucleus routine for output using PUT DATA without listing (XS case) ITPPDBX#
ITPPUT##	Initial and final conditions for all PUT statements, (follow-up) calls for outputting an element (output date/name) for all PUT statements, control output operations (set line, page or column), perform COPY option in GET statement, perform PUT DATA without listing
***	ITPFL###, ITPGET##, ITPIOSY#, ITPOPCL#, ITPOPEN#, ITPPVD##, ITPPVE##, ITPPVL##
ITPPVD##	Follow-up call for PUT DATA with listing; nucleus routine for outputting a variable in DATA mode
***	ITPPDVA#
ITPPVDX#	Follow-up call for PUT DATA with listing; nucleus routine for outputting a variable in DATA mode (XS case)
***	ITPPDVX
ITPPVE##	Output data element from PUT EDIT statement (follow-up call)
ITPPVL##	Output data element from GET LIST statement (follow-up call); also for an element when PUT DATA used
***	ITPPVD##
ITPRAHM#	Standard framework for PLI1 objects; start, end and interrupt handling ; error handling for utilities; program termination/ STOP statement; RUNTIME utility
***	ITP#AOS#, ITPTHBK#, ITPOPEN#, ITPTHRD#, ITPTHTR#

Module	Function / invoked by module (***)
ITPRCD##	Builtin functions ABS, ATAN, ATANH, COS, COSH, EXP, LOG, SIN, SINH, SQRT, TAN, TANH for CPLX FLOAT-type arguments, double precision, exponentiation with CPLX FLOAT base, double precision
ITPRCE##	Builtin functions ABS, ATAN, ATANH, COS, COSH, EXP, LOG, SIN, SINH, SQRT, TAN, TANH for CPLX FLOAT-type arguments, single precision; exponentiation with CPLX FLOAT base, single precision
ITPRCW##	Builtin functions ABS, ATAN, ATANH, COS, COSH, EXP, LOG, SIN, SINH, SQRT, TAN, TANH for CPLX FLOAT-type arguments, extended precision; exponentiation with CPLX FLOAT base, extended precision
ITPRND##	Exponentiation integer ** integer, builtin function ROUND or rounding for real and complex FLOAT operands
ITPRRD##	Builtin functions ACOS, ACOSD, ASIN, ASIND, ATAN, ATAN2, ATAND, ATAND2, ATANH, COS, COSD, COSH, ERF, ERFC, EXP, LOG, LOG10, LOG2, SIN, SIND, SINH, SQRT, TAN, TAND, TANH for FLOAT-type arguments, double precision; exponentiation with FLOAT base, double precision
***	ITPBPLY#
ITPRRE##	Builtin functions ACOS, ACOSD, ASIN, ASIND, ATAN, ATAN2, ATAND, ATAND2, ATANH, COS, COSD, COSH, ERF, ERFC, EXP, LOG, LOG10, LOG2, SIN, SIND, SINH, SQRT, TAN, TAND, TANH for FLOAT-type arguments, single precision; exponentiation with FLOAT basis, single precision
***	ITPBPLY#
ITPRRW##	Builtin functions ACOS, ACOSD, ASIN, ASIND, ATAN, ATAN2, ATAND, ATAND2, ATANH, COS, COSD, COSH, ERF, ERFC, EXP, LOG, LOG10, LOG2, MOD, SIN, SIND, SINH, SQRT, TAN, TAND, TANH for FLOAT-type arguments, extended precision; exponentiation with FLOAT base, extended precision; division of FLOAT-type operands, extended precision
***	ITPKONV#, ITPBPLY#
ITPRTAD#	Subroutine for I/O in DATA mode
***	ITPGDT##, ITPPDVA#, ITPRTOF#, ITPRTPT#
ITPRTAX#	Subroutine for I/O in DATA mode (XS case)
***	ITPGDTX#, ITPPDVX#, ITPRTOX#, ITPRTPX#
ITPRTOF#	Subroutine for I/O in DATA mode
***	ITPRTPT#
ITPRTOX#	Subroutine for I/O in DATA mode (XS case)
***	ITPRTPX#
ITPRTPT#	Subroutine for I/O in DATA mode
***	ITPRTAD#
ITPRTPX#	Subroutine for I/O in DATA mode (XS case)
***	ITPRTAX#
ITPRTSX#	Subroutine for I/O in DATA mode (XS case)
***	ITPGDTX#
ITPRTSY#	Subroutine for I/O in DATA mode
***	ITPGDT##
ITPRTVA#	Subroutine for I/O in DATA mode
***	ITPGDT##, ITPPDVA#, ITPRTAD#, ITPPDSD#
ITPRTVX#	Subroutine for I/O in DATA mode (XS case)
***	ITPGDTX#, ITPPDVX#, ITPRTAX#, ITPRTSX

Module	Function / invoked by module (***)
ITPSAFB#	Builtin functions AFTER, BEFORE, DECAT for BIT strings
ITPSAFC#	Builtin functions AFTER, BEFORE, DECAT for CHAR strings
ITPSBOB#	Builtin function BOOL
ITPSCRB#	Builtin functions COPY, REPEAT, REVERSE for BIT strings
ITPSCRC#	Builtin functions COPY, REPEAT, REVERSE for CHAR strings
ITPSIXB#	Builtin function INDEX for BIT strings
***	ITPSAFB#
ITPSIXC#	Builtin functions INDEX for CHAR strings
ITPSSVC#	Builtin functions SEARCH, VERIFY
ITPSTRC#	Builtin function TRANSLATE
ITPSTVW#	Storage management for stack (for AUTOMATIC and temporary variables), standard area (system storage; for BASED variables not stored in a named AREA, CONTROLLED variables, buffers for I/O, etc.) and named AREAs (for BASED variables); ALLOCATE and FREE statements, assignment of named AREAs;
***	ITP#AOS#
ITPTHAI#	Administration for the debugging aid AID
***	ITP#AOS#
ITPTHBK#	Handle the checkpoint/breakpoint debugging aid; analyze control options entered during breakpoint
ITPTHBO#	Read in control options entered at checkpoint/breakpoint
***	ITPTHBK#
ITPTHLF#	ADUMP, RDUMP, SDUMP utilities (debugging aids)
***	ITPTHBK#
ITPTHPT#	Trace for ENTRY/PROC statements (PROCTRACE); PTON, PTOFF utilities
ITPTHRD#	Debugging aids: SNAP, binary dumps; determining a source reference (numbers of INCLUDE file, line and statement); SNAP utility (debugging aid)
***	ITP#AOS#, ITPTHBK#, ITPTHLF#
ITPTHTR#	Trace for CALL, GOTO and RETURN statements (CALLTRACE, GOTOTRACE, RETURNTRACE) and program label trace (LABELTRACE)
ITPTLIN#	Editing of line references
***	ITPTHRD#, ITPTHTR#, ITPTXST#
ITPTXBS#	Basic text output; text output to SYSOUT and/or SYSLST, basic services for diagnostics listings
***	ITP#AOS#, ITPTHBK#, ITPTHRD#, ITPTHTR#
ITPTXST#	Standard text output; text output from message files to SYSOUT and/or SYSLST; if necessary, enlarge texts or supply with header; analyze options regarding language and output device from the RUNOPT controller
***	ITP#AOS#, ITPOPEN#, ITPTHRD#
ITPWASI#	Nucleus for builtin function ASIN/ACOS, extended precision
***	ITPRRW##
ITPWATH#	Nucleus for builtin function ATANH, extended precision
***	ITPZATA#, ITPRRW##
ITPWAT2#	Nucleus for builtin function ATAN/ATAN2, extended precision
***	ITPZATA#, ITPZLOG#, ITPRRW#

Module	Function / invoked by module (***)
ITPWDIV#	Nucleus for division of FLOAT-type operands, extended precision
***	ITPWASI#, ITPWATH#, ITPWAT2#, ITPWERF#, ITPWEXP#, ITPWMOD#,
***	ITPWPWI#, ITPWSIH#, ITPWTAN#, ITPZATA#, KZLMT###, ITPZSQRW#,
***	ITPZTAN#, ITPRRW##
ITPWERF#	Nucleus for builtin function ERF/ERFC, extended precision
	ITPRRW##
ITPWEXP#	Nucleus for builtin function EXP/LOG/LOG10/LOG2 and exponentiation FLOAT extended precision ** FLOAT extended precision
***	ITPWATH#, ITPWERF#, ITPWSIH#, ITPWTAH#, ITPZEXP#, ITPZLOG#,
***	ITPZPWC#, ITPZSIN#, ITPRRW##
ITPWMOD#	Nucleus for builtin function MOD, extended precision
***	ITPRRW##
ITPWPWI#	Nucleus for exponentiation FLOAT extended precision ** integer
***	ITPRRW##
ITPWSIH#	Nucleus for builtin function SINH/COSH, extended precision
***	ITPZTAN#, ITPRRW##
ITPWSIN#	Nucleus for builtin function SIN/COS, extended precision
***	ITPZEXP#, ITPZSIN#, ITPZTAN#, ITPRRW##
ITPWSQR#	Nucleus for builtin function SQRT, extended precision
***	ITPWASI#, ITPZABS#, ITPZSQR#, ITPRRW##
ITPWTAH#	Nucleus for builtin function TANH, extended precision
***	ITPRRW##
ITPWTAN#	Nucleus for builtin function TAN/COTAN, extended precision
***	ITPRRW##
ITPXABS#	Nucleus for builtin function ABS, cplx single precision
***	ITPRCE##
ITPXATA#	Nucleus for builtin function ATAN/ATANH, cplx single precision
***	ITPRCE##
ITPXDIV#	Nucleus for devision of FLOAT-type operands, single precision
***	ITPACF##
ITPXEXP#	Nucleus for builtin function EXP, cplx single precision
***	ITPXPWC#, ITPRCE##
ITPXLOG#	Nucleus for builtin function LOG, cplx single precision
***	ITPXPWC#, ITPRCE##
ITPXMLT#	Nucleus for multiplication of CPLX FLOAT-type operands, single precision
***	ITPXPWI#, ITPACF##
ITPXPWC#	Nucleus for exponentiation CPLX FLOAT single precision ** CPLX FLOAT single precision
***	ITPRCE##
ITPXPWI#	Nucleus for exponentiation CPLX FLOAT single precision ** integer
***	ITPRCE##
ITPXSIN#	Nucleus for builtin function SIN/COS/SINH/COSH, cplx single precision
***	ITPRCE##
ITPXSQR#	Nucleus for builtin function SQRT, cplx single precision
***	ITPRCE##
ITPXTAN#	Nucleus for builtin function TAN/TANH, cplx single precision
***	ITPRCE##

Module	Function / invoked by module (***)
ITPYABS#	Nucleus for builtin function ABS, cplx double precision
***	ITPRCD##
ITPYATA#	Nucleus for builtin function ATAN/ATANH, cplx double precision
***	ITPRCD##
ITPYDIV#	Nucleus for division of CPLX FLOAT-type operands, double precision
***	ITPACF##
ITPYEXP#	Nucleus for builtin function EXP, cplx double precision
***	ITPYPWC#, ITPRCD##
ITPYLOG#	Nucleus for builtin function LOG, cplx double precision
***	ITPYPWC#, ITPRCD##
ITPYMLT#	Nucleus for multiplication of CPLX FLOAT-type operands, double precision
***	ITPYPWI#, ITPACF##
ITPYPWC#	Nucleus for exponentiation CPLX FLOAT double precision ** CPLX FLOAT double precision
***	ITPRCD##
ITPYPWI#	Nucleus for exponentiation CPLX FLOAT double precision ** integer
***	ITPRCD##
ITPYSIN#	Nucleus for builtin function SIN/COS/SINH/COSH, cplx double precision
***	ITPRCD##
ITPYSQR#	Nucleus for builtin function SQRT, cplx double precision
***	ITPRCD##
ITPYTAN#	Nucleus for built function TAN/TANH, cplx double precision
***	ITPRCD##
ITPZABS#	Nucleus for builtin function ABS, cplx extended precision
***	ITPRCW##
ITPZATA#	Nucleus for builtin function ATAN/ATANH, cplx extended precision
***	ITPRCW##
ITPZEXP#	Nucleus for builtin function EXP, cplx extended precision
***	ITPZPWC#, ITPRCW##
ITPZLOG#	Nucleus for builtin function LOG, cplx extended precision
***	ITPZPWC#, ITPRCW##
ITPZMLT#	Nucleus for multiplication and division of CPLX FLOAT-type operands, extended precision
***	ITPZPWC#, ITPZPWI#, ITPACF##
ITPZPWC#	Nucleus for exponentiation CPLX FLOAT extended precision ** CPLX FLOAT extended precision
***	ITPRCW##
ITPZPWI#	Nucleus for exponentiation CPLX FLOAT extended precision ** integer
***	ITPRCW##
ITPZSIN#	Nucleus for builtin function SIN/COS/SINH/COSH, cplx extended precision
***	ITPRCW##

Module	Function / invoked by module (***)
ITPZSQR#	Nucleus for builtin function SQRT, cplx extended precision
***	ITPRCW##
ITPZTAN#	Nucleus for builtin function TAN/TANH, cplx extended precision
***	ITPRCW##
ITP2SRT#	Intermediate routine for calling BS2000 SORT from PLI1 programs and (optionally) interfacing user routines for record I/O during sort run; utilities BS2SRTA, BS2SRTB, BS2SRTC, BS2SRTD

14.8 Messages of the PLI1 runtime system

These messages appear for inability to call the PL/I error handling routine, e.g. because the necessary initializations have not been carried out (e.g. during the prolog of the program) or because the routines needed for condition handling are affected by error conditions themselves (e.g. text output routines, program interrupt handling). Depending on the type of error, the program is terminated or an interactive prompt appears to that effect (***CONTINUE (Y/N)).

Format:
***Ennn ERROR IN mmmmmmm:message

where:

- 'nnn' is the number of the message,
- 'mmmmmmm' is the name of the module which caused the message;
- 'message' is the text of the message.

The following table provides a listing of message numbers and texts together with additional information.

Number	Message	Description/Cause	Reaction
001	INTERRUPT OR WROUT ERROR IN RUNTIME SYSTEM	Error in PLI1 frame routine	Inform System Service
002, 003	PROGRAM CANNOT BE INITIATED	Unable to sign on program interrupt handling; e.g. load module too large	Remove storage bottleneck; reduce object module
004	STACK POINTER (R13) DESTROYED-NO COND' HANDLING	Error handling impos.; R13 destroyed; possibly user error	If necessary, correct the program
007	IRREGULAR INTERRUPT	Undefined interrupt or error in tracer	Inform System Service
011, 012, 013	STORAGE SHORTAGE	Storage shortage for stack initialization or after STORAGE condition	Increase entry for *RUNOPT STORAGE = SPACE
014, 015, 016, 017	ILLEGAL VALUE IN STORAGE OPTION IGNORED	Invalid STORAGE option parameter	Correct the STORAGE option

Number	Message	Description/Cause	Reaction
021	MULTIPLE INITIALISATION	Attempted multiple initialization of the PLI1 environment	
022	ILLEGAL PARAMETER	Inconsistent STORAGE parameter	Correct the STORAGE option
023	STORAGE SHORTAGE	Storage shortage for stack initialization or after STORAGE condition	Increase entry for *RUNOPT STORAGE = STACK
024 025 026 027	ILLEGAL VALUE IN STORAGE OPTION IGNORED	Invalid STORAGE option parameter	Correct the STORAGE option
029	FREE CHAIN DESTROYED IN STANDARD AREA	Free chain of the standard area de-stroyed; possibly user error	Program may have to be corrected
030	LINESIZE TOO LARGE, ADJUSTED TO BUFFERSIZE	Line size too large for interactive device	Correct the LINESIZE parameter
031	CONTROL DATA DESTROYED	Control information destroyed or parameter illegal	Inform System Service
032	IRREGULAR INTERRUPT	Undefined interrupt or error in the backtracer	Inform System Service
033	CONTROL DATA DESTROYED	Control information destroyed or parameter illegal	Inform System Service
034	ERROR ON SYSOUT/SYSLST	Error in text output	Inform System Service
035	STORAGE SHORTAGE	Storage shortage for text output routines	Increase option for *RUNOPT STORAGE = AREA
036	PLI1 TEXTFILE NOT AVAILABLE (MISSING, LOCKED, ETC.)	Text file improperly assigned	Make text file available, e.g. by /FILE..., LINK = TEXTLINK
037	ERROR ON SYSOUT/SYSLST	Error in text output	Inform System Service

Number	Message	Description/Cause	Reaction
038	FILE NOT USABLE FOR SAVLST	Incompatible SAVLST file parameters	Use default SAVLST file parameters
039	I/O ERROR on SAVLST	I/O error for SAVLST file	Inform System Service
050 051 052 053	INTERNAL ST$STVW# ERROR	PLI1 storage management error	Inform System Service
055	REQM ERROR (MEMORY SATURATION)	REQM macro error	Inform System Service
056	RELM ERROR	RELM macro error	Inform System Service
070	UNRECOVERABLE SYNTAX ERROR IN OPTIONS	Error in OPTIONS entry	Correct the *RUNOPT entry
071	INPUT TOO LARGE FOR OPTION routine	OPTIONS entry too long	Reduce the line with *RUNOPT
080	ILLEGAL OPERATION FOR OVERLAPPING BIT STRINGS	Module BIT#OP## of the PLI1 runtime system has been called while using an invalid entry	Inform System Service
101	I/O UNABLE TO INITIALIZE TERMINATION	Termination procedure cannot be signed on for OPEN	Inform System Service
102 103 104 105 106 107 108 109	I/O-System ERROR	DMS error for OPEN/CLOSE	Inform System Service

Number	Message	Description/Cause	Reaction
110	ATTEMPT TO START A FOR1-PROGRAM OUTSIDE THE MAIN-PROCEDURE	A program containing FOR1 procedures is started in the PLI1 runtime system	Link in the correct sequence or with a START specification
111	INCONSISTENT RUNTIME/ I/O-SYSTEM	Prelinked sections of the PLI1 runtime system (P$RTS###/P$IOS###) do not agree with the connection module (P$ANFOD#)	Obtain suitable runtime system; relink if necessary
112	ATTEMPT TO START A PLI1-PROGRAM OUTSIDE THE MAIN-PROCEDURE	A program invoked by foreign procedures is started in the PLI1 runtime system	Link in the correct sequence

14.9 Examples of PLI1 ASSEMBLER macros

1. PL/I Main Program

```
PLIMAC :PROC  OPTIONS(MAIN);

        PUT SKIP LIST
            ('TEST OF PLI1.MACLIB');
        PUT SKIP LIST
            ('CALL OF ASSEMBLER-SUBROUTINES, THAT ARE CALLING');
        PUT SKIP LIST
            ('PL/I-SUBROUTINES ETC.');
        PUT SKIP;

        DCL  (UP1,UP2,UP3,UP4,UP5)
             ENTRY OPTIONS(ASM);
        DCL  (UP6,UP7) RETURNS (PTR);
        DCL  (UP8,UP9) OPTIONS(PLI1);
        DCL   A CHAR(40) VAR INIT
                ('1234567890  3 CHAR.S ARE NOT PRINTED'),
              B CHAR(40) BASED (P),
              C CHAR(40) VAR INIT('CCCCCCCCCCCC'),
              D BIT (256) INIT ('C0C1C2C3C4C5C6C7C8C9CACBCCCDCECF'B4 ||
                                '808182838485868788898A8B8C8D8E8F'B4),
              P   PTR INIT(ADDR(X)),
              PT6 PTR,
              PT7 PTR,
              X CHAR(100) INIT((100)'X'),
              ERROUT OPTIONS(LIB);

        ON ERROR BEGIN;
              CALL ERROUT;
              GOTO ERR;
        END;

        CALL UP1 (A,B,C,D);
        CALL UP2 (A);
        CALL UP3;
        CALL UP4;
 ERR:   CALL UP5 (PLIUP3,P1,P2,P3,P4,P5,P6);
        PT6 = UP6 ();
        PT7 = UP7 ();
        PUT SKIP EDIT
            ('R12=',UNSPEC(PT6)) (A,B4);
        PUT SKIP EDIT
            ('====',UNSPEC(ADDR(PLI1GLOBAL(1))) | 'FFFFF000'B4)
            (A,B4);

        PUT SKIP EDIT
            ('R13=',UNSPEC(PT7)) (A,B4);
        CALL UP8;
 FIN:
        CALL UP9;
       PUT SKIP LIST('NORMAL END');

       STOP;
```

```
PLIUP1:ENTRY(A1,A2,A3,A4);
       DCL  (A1,
             A2,
             A3 ) CHAR(40) VAR,
             A4   CHAR(32);

       PUT SKIP DATA (A1);
       PUT SKIP DATA (A2);
       PUT SKIP DATA (A3,A4);
       PUT SKIP;
       RETURN;

PLIUP2:ENTRY(B1,B2,B3,B4,B5,B6);
       DCL  (B1,B3,B5(P1,P3,P5)STATIC) CHAR(15) VAR,
            (B2,P2 STATIC) BIN FIXED(31),
            (B4,P4 STATIC) DEC FIXED(5),
            (B6,P6 STATIC) PTR,
             B7 CHAR(15) VAR BASED (B6);
       P1=B1;P2=B2;P3=B3;P4=B4;P5=B5;P6=B6;

       PUT SKIP DATA (B1,B2);
       PUT SKIP DATA (B3,B4);
       PUT SKIP DATA (B5,B7);

       RETURN;

PLIUP3:ENTRY(C1,C2,C3,C4,C5,C6);
       DCL (C1,C3,C5) CHAR(15) VAR,
            C2 BIN FIXED(31),
            C4 DEC FIXED(5),
            C6 PTR,
            C7 CHAR(15) VAR BASED (C6);

       PUT SKIP DATA (C1,C2);
       PUT SKIP DATA (C3,C4);
       PUT SKIP DATA (C5,C7);
       PUT SKIP;

       RETURN;

PLIUP4:ENTRY;

       PUT SKIP LIST ('PLIUP4 ENTERED');
       PUT SKIP;

       RETURN;

PLIUP5:ENTRY;

       PUT SKIP LIST ('PLIUP5 ENTERED');
       PUT SKIP;

       RETURN;

     END PLIMAC;

UP1         START
            PRINT   NOGEN
```

```
SSS         P$STACK
PRV         P$PRV
            P$ENTRY LENGTH=200
*         PARAMS FOR THE FOLLOWING CALL VIA R1 FROM PL/1
            P$CALL  PLIUP1,PARNUM=4
            P$RETURN
UP2         P$ENTRY
            LA      R1,0(,R1)
            WRLST   (1)
            P$RETURN
UP3         P$ENTRY LENGTH=0
            P$CALL  PLIUP2,(P1,P2,P3,P4,P5,P6)
            P$RETURN
P1          DC      H'15',CL40'FIRST PARAMETER'
P2          DC      F'99999'
P3          DC      H'12',CL15'2ND PARAMETER'
P4          DC      P'99999'
P5          DC      H'00',CL15'3RD PARAMETER'
P6          DC      A(P11)
P11         DC      H'17',CL15'WRONG LENGTH STRING'
P12         DC      C'!!!!!'
UP4         P$ENTRY
            P$ERROR   ONCODE =0003,MSG=01
            P$RETURN
UP5         P$ENTRY
            P$CALL    PARNUM=7
            P$RETURN  STXIT=YES
UP6         P$ENTRY
            LR        R1,R12
            P$RETURN  R1RETN=YES,STXIT=YES
UP7         P$ENTRY
            LR        R1,R13
            P$RETURN  R1RETN=YES,STXIT=NO
UP8         P$ENTRY
            P$CALL    PLIUP4
            P$RETURN  STXIT=YES
UP9         P$ENTRY
            BALR      10,0
            USING     *,R10
            LA        R1,=V(PLIUP5)
            P$CALL
            P$RETURN
            END
```

The PL/I program PLIMAC calls the entries of the ASSEMBLER program UP1, which call the entries of the main program via the P$CALL macro. As a result the following is output:

```
TEST  OF PLI1.MACLIB
CALL OF ASSEMBLER-SUBROUTINES, THAT ARE CALLING
PL/1-SUBROUTINES ETC.

A1='1234567890  3 CHAR.S ARE NOT PRINTED';
A2='XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX';
A3='CCCCCCCCCCCC'   A4=' ABCDEFGHI       abcdefghi ÄÖÜ';
4567890  3 CHAR.S ARE NOT PRINT

B1='FIRST PARAMETER'        B2=           99999;
B3='2ND PARAMETER'  B4=  99999;
B5=''     B7='WRONG LENGTH ST';
*****ERROR-CONDITION, ONCODE=0003 AT OFFSET '027312' IN PROCEDURE WITH ENTRY
UP4
NO WHEN CLAUSE SATISFIED AND NO OTHERWISE CLAUSE SPECIFIED

C1='FIRST PARAMETER'          C2=          99999;
C3='2ND PARAMETER'   C4=   99999;
C5=''     C7='WRONG LENGTH ST';

R12=00028000
====FFFFFDEC
R13=7F02E580
PLIUP4 ENTERED

PLIUP5 ENTERED

NORMAL END

END OF PROGRAM PLIMAC  , RTS 3.2A-600, TIME USED:    0.09 SEC
```

2. ASSEMBLER Main Program

```
HP2       START
          PRINT     GEN
          BALR      R11,0
          USING     *,R11
          P$ENVIRM  BASEREG=R11
          P$LINK    PLIUP4,LIBNAM=BIS.OBJ
          ST        R1,PAREND
          LA        R1,PAREND
          P$CALL
          P$LINK    PLIUP2,LIBNAM=BIS.OBJ
          ST        R1,PAREND
          LA        R1,PAREND
          P$CALL    ,(P1,P2,P3,P4,P5,P6)
          P$STOP
P1        DC        H'15',CL40'FIRST PARAMETER'
P2        DC        F'99999'
P3        DC        H'12',CL15'2ND PARAMETER'
P4        DC        P'99999'
P5        DC        H'00',CL15'3RD PARAMETER'
P6        DC        A(P11)
P11       DC        H'17',CL15'WRONG LENGTH STRING'
P12       DC        C'!!!!!'
          END
```

```
 PLIUP2 : PROC (B1,B2,B3,B4,B5,B6) ;
          DCL  (B1,B3,B5) CHAR(15) VAR,
                B2 BIN FIXED(31),
                B4 DEC FIXED(5),
                B6 PTR,
                B7 CHAR(15) VAR BASED (B6);

          PUT SKIP DATA (B1,B2,B3);
          PUT SKIP DATA (B4,B5,B7);
          PUT SKIP;
 END;
```

```
 PLIUP4 : PROC ;
          PUT SKIP LIST
              ('PLIUP4 ENTERED');
          PUT SKIP;

          RETURN;
 END;
```

The ASSEMBLER main program HP2 invokes PL/I programs PLIUP2 and PLIUP4 which are dynamically linked by means of the P$LINK macro. As a result the following is output:

```
 PLIUP4 ENTERED

 B1='FIRST PARAMETER'         B2=         99999   B3='2ND PARAMETE';
 B4=   99999         B5=''    B7='WRONG LENGTH ST';
```

15 References

[1] **PLI1** (BS2000)
PL/I Compiler
Language Reference Manual

Target group
PL/I users in BS2000.
Contents
Language elements and program modules, data types and attributes, storage allocation, statements, program structure, I/O, expressions, data conversion, formats and picture characters, built-in functions, pseudo-variables, preprocessor and uncommon program results.
The reference manual is also suitable as a textbook for PL/I novices.

[2] BS2000
Control System Command Language
Reference Manual

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.

[3] BS2000
Utility Routines
Reference Manual

Target group
BS2000 users (non-privileged).
Contents
Utility routines for non-privileged BS2000 users.
Applications
BS2000 timesharing mode.

[4] **EDOR** (BS2000)
Reference Manual

Target group
Data entry operators, programmers.
Contents
Description of the statements to the EDOR File Editing System.
Applications
BS2000 interactive mode.

[5] **EDT** (BS2000)
Reference Manual

Target group
Data entry operators, programmers.
Contents
Description of the statements for the EDT file editor; EDT procedures; EDT subroutine interface.
Applications
BS2000 interactive/batch mode.

[6] BS2000
System Messages
Reference Manual

Target group
BS2000 users.
Contents
Standard format messages of the BS2000 control system, including SPOOL, RSO, SDF; standard format messages of the software products DCAM, TIAM, RBAM.

[7] BS2000
DMS Disk Processing
Reference Manual

Target group
BS2000 users, assembly language programmers (both non-privileged).
Contents
Functions of the Data Management System in BS2000; DMS commands and macros, service and action macros; access methods UPAM, SAM, ISAM and EAM for disk files.
Applications
BS2000 interactive/batch mode, programming.

[8] **COB1**(BS2000)
COBOL Compiler
User Guide

Target group
COBOL users in BS2000.
Contents
Operation of the COB1 compiler and the software required for the development, linking, execution and debugging of COBOL programs; structure of the COB1 system and the generated object modules; programming notes; compiler messages and the COB1 interface with UDS.

[10] **SORT** (BS2000)
Reference Manual

Target group
BS2000 users.
Contents
Functions and statements for sorting and merging files.

[11] **PLI1** (BS2000)
PL/I Compiler
Reference Guide

Target group
PL/I users in BS2000.
Contents
Tables of language elements, debugging aids and compiler and object control functions.

[12] BS2000
Linkage Editor and Loaders
Reference Manual

Target group
BS2000 users.
Contents
Description of the statements for linking and loading programs with TSOSLNK, ELDE and DLL.
Applications
BS2000 interactive/batch mode.

[13] **LMS** (BS2000)
Reference Manual

Target group
BS2000 users.
Contents
Description of the statements for creating and managing PLAM libraries with LMS and storing library members using the delta method.
Applications
BS2000 interactive/batch mode.

[14] BS2000
Interactive Debugging Aid (IDA)
Reference Manual

Target group
Programmers.
Contents
Description of the commands and macros for the Interactive Debugging Aid (IDA).
Applications
BS2000 interactive mode.

[15] BS2000
Job Variables
Reference Manual

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.

[16] BS2000
Executive Macros
Reference Manual

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.

[17] BS2000
DMS Tape Processing
Reference Manual

Target group
BS2000 users, assembly language programmers (both non-privileged).
Contents
Functions of the Data Management System in BS2000; DMS commands and macros, service and action macros; access methods UPAM, SAM and BTAM for tape files.
Applications
BS2000 interactive/batch mode, programming.

[18] **AID** (BS2000)
Advanced Interactive Debugger
Debugging of PL/1 Programs
User's Guide

Target group
PL/1 programmers.
Contents
Preparations for the symbolic debugging of PL/1 programs; description of all the AID commands available for symbolic debugging; examples of AID sessions; messages.
Applications
Debugging of PL/1 programs in interactive and batch modes.

[19] BS2000
User Commands (SDF Format)
Reference Manual

Target group BS2000 users *Contents* All BS2000 user commands in SDF format in alphabetical order with detailed explanations and examples. *Applications* BS2000 interactive mode, procedures, batch mode

Ordering manuals

The manuals listed above and the corresponding order numbers are to be you how to order manuals. New publications are listed in the *Druckschriften-Neuerscheinungen Datentechnik (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

$TSOS 150
$TSOSLNK 95
%INCLUDE 47, 66, 70, 75, 129
*COMOPT 51
*COMOPT DIAGNOST 122
*COMOPT LIST 99
*COMOPT MODULE = destination 93
*COMOPT MODULE=target control statement 91
*DUMMY 78, 207
*EAM file 93
  object module 95
*END 51, 67
*RUNOPT 172

A

abbreviation for control statement 52
ABORT 85
absolute pointer 406
access authorization 218
access method 209
access method EAM 209
access method ISAM 209
access method PAM 209
access method SAM 209
access protection 218
acess method BTAM 209
activation of check points 383
activation record 437
ACTIVE control statement 187
administration information 211
ADUMP 455
AGGREGATE 80, 112
aggregates 369

AID 90
  debugger 386
aliased variable 419
ALIGN 187
ALIGNED 389
alignment 434
alphanumeric picture 435
AREA 65, 181, 183
area 408
AREA attribute 408, 446
argument 441
ARGUMENT control statement 180
arithmetic expressions 362
arithmetic variable 389
array element 414
ASACNTRL 76
ascertainment of storage requirements 416
ASM 298
assembler convention 309
assembler procedure 307
assembly listing 119
ASSIGN-SYSDTA, SDF operand 195
ASSIGN-SYSLST, SDF operand 195
ASSIGN-SYSOUT, SDF operand 195
assigning, file 236
assignment 360
  file name 236
assignment of organization methods 239
ASSM 81, 119
attribute 357
attribute listing 80
AUTOMATIC 437
auxiliary variable 441

B

BACKWARDS 283
BASED attribute 426, 442
BASED variable 478
batch mode 46, 52, 204, 254
batch processing 13
BIT 428
bit string 370
BITPTR 87
BLKSIZE for ENVIRONMENT 225

BLKSIZE parameter 212
BLOCK 212
block condition 440
block example 213
block length field 212
block size 212
block type 439
Boolean expressions 362
branch 304
BREAK command 204
BREAK function 188
BREAKPOINT 86
breakpoint 48, 380
BS2SRT 457
BTAM, access method 209
buffer 212
buffer length 212

C

call nesting 473
calling the linkage editor 151
CALLTRACE 86, 186, 380
carriage control 76, 228
carriage control character 211
CATALOG command 218
CHARACTER 403
character string 370, 400
character string variable 403
class 4 memory 480
COBOL 314
COBOL procedure 311, 316
CODE 85
code generation 85
code module 92, 93, 161
code output 85
COMLIB 129
COMLIB control statement 66, 75
command procedure 5
common expression 340
common expression elimination 350
COMOPT 119
COMP 235
compilation 5
compiler, controlling 50

compiler control 380
  example 46
compiler listing 20
COMPILER-ACTION, SDF operand 136
COMPILER-TERMINATION, SDF operand 144
computing time used 470
COND 181
condition 371
condition ENDPAGE 251
condition UNDEFINEDFILE 227
CONSECUTIVE 224
  organization 258
CONSECUTIVE file organization 37
constraints on implementation 521
control character 251
CONTROL control statement 187
control optimization 353
control statement
  abbreviation 52
  object program 165
  rule 51, 172
  specification 51
  validity 52
control statement preprocessor 129
control statements for compiler 41
CONTROLLED attribute 442, 449
controlled editing DATA 248
controlled editing LIST 248
controlled output DATA 250
controlled output LIST 250
CONTROLLED variable 479
controlling, listing output 173
controlling object module generation 85
controlling source input 66
controlling the compiler 50ff
controlling the linkage editor 150
controlling the listing output 78
conversion 362
CPU-LIMIT, SDF operand 194
cross-reference listing 109
CTLASA 228, 251
CTLMACH 228, 251

D
DATA controlled editing 248
DATA controlled output 250
data conversion 334
DATA input 253
data management system 221
data module 161
data set 207, 211, 222
DEBUG control statement 86
debugger 32
  AID 386
debugging aid 86, 90, 379
declaration 357
default 50
description of the data type 428
DEUTSCH 122
DEVICE 217
DEVICE parameter 217
DIAGNOST control statement 82
diagnostic message 122
DIMENSION attribute 414
dimension BIT 428
direct access media 217
DIRECT KEYED 239
DISPLAY 184
DISPLAY statement 237
distinction from PL/I-D 525
DMS macro 221, 238
DO command 204
DO groups 366
drifting character 434
dummy record 271, 273
DUMP 384
dump 384, 469
DUMP control statement 181
duplicate key for REGIONAL(3) 278

E
EAM 42
  access method 209
EAM file, erase 6
EAM object modules file 151
EDIT input 253
EDOR 80, 102

EDT 80, 102
element 70, 91
elementary runtime system 158, 159
enable control statement 41
enabled control statement 81
ENABLING 88
END statement 154
ENDFILE condition 260, 280
ENDPAGE condition 251
ENGLISCH 122
ENGLISH 84
ENTER command 46, 204
ENTRY 149, 412
ENTRY attribute 161
entry variable 412
ENVIRONMENT 224
ENVIRONMENT attribute 222, 240
EOF 188
EOF command 204
ERASE 6
ERASE command 206
ERROR 82, 122
error 41, 122
error handling during control statement evaluation 53
error monitoring job variable 44, 168
error text 45, 464
error weight 44
ERROUT 464
ERROUT procedure 173
ESCAPE function 188
ESD 80, 106
example for REGIONAL(3) 276
example of compiler control 46
example of PLI1 ASSEMBLER macro 556
example of the linkage editor 155
examples of sorting 528
EXEC command 204
EXECUTE $TSOSLNK 151
EXECUTE command 43, 167
executive 221, 238
EXPAND 79, 102
expression simplification 347
EXTEND 78, 246, 259
extend file 246

EXTERNAL 161
EXTERNAL attribute 236
EXTERNAL items 149
external name 106
external reference 150, 154
EXTERNAL variable 419

F

F format 211
FCBTYPE parameter 209
FILE 413
file, characteristics 242
file access 203
FILE attribute 222
file attribute 206
file attribute RECORD 222
file attribute STREAM 222
FILE command 151, 207, 240
FILE command for CONSECUTIVE file 261
FILE command for INDEXED file 267
FILE command for REGIONAL(3) files 281
file link name 70, 208
file name 207
file organization 37, 209
file organization CONSECUTIVE 37
file organization INDEXED 37
file organization REGIONAL (1) 37
file SAVLST 125
file size 219
file type 241
file variable 413
FIXED BINARY 390
FIXED BINARY PRECISION 390
fixed binary variable 390
FIXED DECIMAL PRECISION 393
fixed decimal variable 393
flag 439
FLOAT PRECISION 396
floating point variable 396
FORMAT 411
FORMAT control statement 83, 181
format variable 411
FORTRAN 312
FORTRAN procedure 311, 316

frame character 105
FROM-FILE, SDF operand 193
FULLXREF 80, 109
function, PLI1 41

G
GAM 70
GAM file 41, 47
GAMKEY 70, 72, 77
gap chain 444
generation 207
GENKEY 264
GERMAN 84
GET statement 249
global optimization 340
GOTOTRACE 86, 380
group file 66
group key 72
group name 72

H
HEAP-ADMINISTRATION, SDF operand 199
hexadecimal character 465
HEXDEC 465

I
I/O buffer for CONSECUTIVE file 259
I/O statement for INDEXED organization 263
I/O statements for REGIONAL(1) and REGIONAL(3) organization 270
IDA statement 156
ILCASE 235
ILCS
  interrupt handling 322
  mapping of files 322
  OPTION 64
  parameter types 322
  termination processing 322
ILCS procedures 320
in-line operation 334
include 103
INCLUDE reference 41, 75
INCLUDE reference listing 80, 126
INCLUDE statement 150, 153
INCLUDE text 41, 79
include text 108

INCLUDE-LIBRARY, SDF operand 133
INDEXED 224
INDEXED file organization 37
INDEXED organization 263
Industry Standard 316
INFORMATION 82, 122
information 122
information message 535
INITIAL attribute 478
initialization 360
initialization of aggregate 348
INOUT 246, 259
input buffer 260
input DATA 253
input EDIT 253
input file 204
input LIST 253
input/output 372
input/output statement 479
INSOURCE 80, 101
INTER 298
interactive mode 46, 52, 204
interactive task 18
interlanguage facility 311
internal procedure calls 349
internal representation 387
INTERRUPT 87
interrupt handling 317, 341
INTR command 188
invocation from COBOL program 319
invocation from FORTRAN program 319
invocation interface 288
IREF 80, 108
ISAM, access method 209
ISO 87
ISO code 213

J

job variable, monitoring 168

K
key 241
KEY condition 266, 273, 274, 276, 280
key for CONSECUTIVE file 259
key options 227
key specification 264
key specification for REGIONAL(1) and REGIONAL(3) organization 271
key specification for REGIONAL(3) 276
KEYLEN parameter 227
KEYLENGTH 227
KEYLOC 227
KEYPOS parameter 227
keyword 51

L
LABEL 410
label variable 410
LABTRACE 86, 186, 380
LANGUAGE, SDF operand 146, 194
leader 104
LEAVE 285
length specification 428
LIBRARY 305
library procedure 305
LIMCT 278
limit value 521
limits for arithmetic values 524
limits for input/output procedures 524
limits for matrices and areas 524
limits for names 524
limits for strings and pictures 524
LINE 235
line length 83
line mode 254
line reference 102
LINECNT 80, 102
LINESIZE 184, 254
LINESIZE attribute 249
LINID 76, 102, 130
LINK 20
LINK name 75
LINK parameter 208
LINK=SAVLINK 125
LINK=TEXTLINK 45

linkage editor call 151
linkage editor example 155
linking 5, 95, 149
LIST control statement 79, 182
LIST controlled editing 248
LIST controlled output 250
LIST input 253
LIST=INSOURCE 126
LIST=IREF 126
LIST=MAP 385
LIST=SUMMARY 444
LISTING, SDF operand 138, 198
listing 20, 81
  compiler 20
listing output, controlling 78, 173
listing preprocessor 101
LMS 70
LMS element 91
LMS library 70, 91, 92
  object module 97
load module 149, 156
loading 5, 149, 156
LOCATE 426
LOCATE SET 234
LOCATE statement 256
logical block 212

M

machine code 81
MACRO 85, 87
macro 481
macro library 41
macro-organized library 66
magnetic tape 283
MAIN 87
main and substructure 420
MAIN OPTIONS 63
main procedure 87
manual optimization 326
MAP 80, 115
mapping of SDF to RUNOPT operands 202
mapping SDF, COMOPT operands 147
MARGINS 70, 102
MARGINS control statement 76

MARGINS option 67
MARGINS=SAVMAC 128
member 70
MERGE 457
MERGE control statement 458
MESSAGE control statement 82, 182, 458
messages of the PLI1 runtime system 552
MLU 70
MLU file 47
modular programming 333
MODULE 91
module library 28
MODULE-LIBRARY, SDF operand 137
monitoring job variable 44
MONJV 44
  SDF operand 145, 194

N

name convention, object module 161
named area 446
NEST 80, 102
nesting level 101
nestings 80
NOLINID 76
nonstandard block 212
NOTRACE 467
number of errors 85
number of lines 83, 181
numeric picture 435

O

OBJECT 85
object code listing 41, 156
object listing 81
object module 5, 41, 149
  LMS library 97
  name convention 161
object module *EAM file 95
object module generation controlling 85
object module library 153
object module maintenance 93
object modules 150
object modules file EAM 151
object program, control statement 165
OBJECT=MACRO 126

OFFSET 80, 119, 406
OFFSET list 41
OFFSET listing 81
offset listing 119
ON chain 440
ON unit 371
ONCODE=value 501
ONINTR 189
opening a REGIONAL(3) file 279
OPTIMIZATION, SDF operand 143
optimization 88, 323
optimization control 353
optimization global 340
optimization manual 326
optimization of Boolean expressions 346
optimization register 352
optimization time 353
OPTIMIZE control statement 88
OPTIMIZE=TIME 385
option PAGESIZE 251
option RUNOPT 237
option SPACE 219
OPTIONS 81, 87, 182, 296, 453
OPTIONS attribute 63
OPTIONS control statement 87
OPTIONS entry ASSEMBLER 64
OPTIONS entry COBOL 64
OPTIONS entry FORTRAN 64
OPTIONS entry ILCS 64
OPTIONS entry INTER 64
OPTIONS entry LIBRARY 64
OPTIONS entry VARIABLE 64
OPTIONS entry WXTRN 64
OPTIONS option BITPTR 63
OPTIONS option ENABLING 63
OPTIONS option ISO 63
OPTIONS option MAIN 63
OPTIONS option OVERLAP 63
OPTIONS option REENTRANT 63
OPTIONS option REORDER 63
OPTIONS option XS 63
OPTIONS=MACRO 126
ORDER option 343
organizing, various ways of 224

OUT 85
OUTTEXT 79, 102
OVERLAP 88, 354
overlapping 354

P
P$CALL 483
P$ENTRY 486
P$ENVIRM 488
P$ERROR 490
P$LINK 491
P$PRV 493
P$REGEQU 494
P$RETURN 495
P$STACK 497
P$STOP 498
PAD 76, 102
PAD parameter 209
PAD specification 213
page mode 254
PAGESIZE 184, 255
PAGESIZE option 251
PAM 210
PAM block 212
PARAM command 41
parameter 441
passing of parameters 292
password 218
PASSWORD command 218
physical block 212
picture 434
PICTURE attribute 405, 428
picture description 433
picture length 434
picture type 434
picture variable 405
PL/I interface 288
PL/I standard 87
PL/I-D 525
PLI1 ASSEMBLER macro interface 481
PLI1 function 41
PLI1.SAVMAC 128
PLI1GLOBAL(n) 478
PLIRETC 468

POINTER 406
pointer 406
pointer for CONSECUTIVE file 259
pointer for regional(1) 273
PRECISION 389
precision 428
prelinked runtime system 158, 159
PREPROCESSING, SDF operand 135
preprocessor 126
preprocessor listing 101
preprocessor output 128
preprocessor statement 128
presetting for access methods 239
presetting for organization methods 239
PRIMARY 68, 78
PRINT 80, 102
PRINT command 258
PRINT file 251
PRINTER 83
private volume 217
PROCEDURE 6
procedure 46
  COBOL 311
  FORTRAN 311
procedure ERROUT 173
procedure file 204
procedure nesting 181
PROCEDURE OPTIONS 356
PROCEDURE-TRACE 186
PROCEDURETRACE 86
PROCTRACE 86, 380
PROcTRACE 186
program
  execution 167
  started 165
program control 356, 381
program control variable 406
program execution 5
program interrupt 188, 384
PROGRAM statement 152
pseudo variable 370
public volume 217
PUT statement 249

Q
QUICK 385

R
RANGE 181
RDPASS parameter 218
RDUMP 469
READ INTO 230, 426
READ ONLY 480
READ SET 232, 426
READ statement 256
record 426
RECORD condition 259, 266
RECORD control statement 458
RECORD file attribute 222
record format 211
record key 227
record length 211
record length field 211
record structure 211, 225
record-oriented input and output 256
record-oriented transfer 222
record-oriented transmission 256
RECSIZE parameter 211
RECSIZE(r) for ENVIRONMENT 225
reducible function 346
reduction of linear expressions 352
reduction of linear expressions in DO loops 343
REENTRANT 87
REFER 424
reference chain 449
reference listing 41, 109
region-specific record key 276
REGIONAL (1) file organization 37
REGIONAL (1) organization 269
REGIONAL(1) 224
REGIONAL(3) 224
register and address optimization 352
register in DO statement 349
register in DO statements 352
register saving 439
relative pointer 406
REORDER 88, 353
REORDER option 343

RESOLVE statement 150, 154
result of the preprocessor 128
RESUME 156, 170, 188
RESUME command 52
RETPD parameter 218
return 303
return address 439
return of the result 299
return when * is specified 301
RETURNS option 301
RETURNTRACE 86, 186, 380
REWRITE statement 256
rule of reduction 161
rules for control statement 51
rules governing REGIONAL(3) organization 275
rules of the industrial standard 87
RUNOPT 381
RUNOPT option 237
RUNTIME 470
runtime library 149
runtime module 543
runtime system 149, 157, 165, 325
  elementary 158, 159
  prelinked 158, 159
  storage 160

S

SAM, access method 209
SAVLST 81, 82, 122
SAVMAC 77
SCALARVARYING 211, 229
SCALARVARYING specification 258
scaling 428, 434
SDF mapping, RUNOPT operands 202
SDUMP 471
self-defining structure 424
SEMANTIC 85
semantics run 85
SEQUENTIAL KEYED 239
set return code 468
SETSW command 169
SEVERE 82, 122
SHARE command 480
shareability 265

shareable programs 477
shared update 265
SHARUPD 218
SHRTXREF 80, 109
simultaneous access 218
SNAP 181, 385, 473
SORT 528
SORT control statement 458
sort/merge program SORT 457
SORTCKPT 458
SORTIN 458
SORTOUT 458
SORTWK 458
SOURCE 79, 102
  SDF operand 132
SOURCE control statement 70
source key 276
source line 104
source listing 79
source program 41
source protocol 102
source text 127
SOURCE-PROPERTIES, SDF operand 134
SPACE 251
SPACE option 219
specification for control statement 51
specifying OPTIONS 292
spool file 204
spool-in file 68
spoolin file 204
STACK 65, 181, 183
stack 81, 183
stack dump 471
stack request 81
STACK-ADMINISTRATION, SDF operand 200
standard area 183, 442
Standard assembler convention 298
standard block 212
START PLI1 COMPILER 43
START-PARAMETERS, SDF operand 194
START-PLI1-COMPILER, operand overview 131
START-PLI1-PROGRAM 167
  operand overview 192
started, program 165

statement condition 440
STATIC 478
STATIC attribute 436
static module 92, 93
static parent block 439
STATIC variable 478
static variable 436
statistics 81
statistics listing 121
STMT 86, 380
storage 151
  runtime system 160
STORAGE control statement 65, 183
storage management 436
storage map 41
storage occupancy 115
storage occupancy statistics 41
storage statistics 183
STORAGE=AREA 443
STREAM file attribute 222
STREAM input/output for interactive devices 253
stream-oriented input and output (STREAM) 248
stream-oriented transfer 222
string variables 400
STRUCTURE 416
structure 416
structure length 112
structure length table 80
SUMMARY 81, 121, 182
summary of control statements 54
SYMTEST 90
SYNTAX 85
syntax check 136
syntax run 85
SYSCMD 68, 204
SYSDATA 41, 66
SYSDTA 68, 184, 204
  system file 236
SYSFILE 6, 170
SYSFILE command 18, 52, 66, 68, 78, 238
SYSFILE control statement 184
SYSIN 13
SYSIPT 205
SYSLST 78, 82, 184, 204

system file 236
SYSLYST 41
SYSOPT 205
SYSOUT 42, 78, 184, 204
system file 236
SYSPRINT 13
system defaults for file characteristics 244
system file 204
system file SYSDTA 236
system file SYSLST 236
system file SYSOUT 236

T

TABULATOR control statement 186
TABULATOR-POSITION, SDF operand 201
task switch 0 53
TASKLIB 150
TCHNG 188
temporary write protection 218
TERMINAL 81, 82, 83, 122, 182, 186, 235
terminating a procedure 302
termination code 44
TEST-SUPPORT, SDF operand 141, 196
TEXT 76, 102, 130
TIME 88
TITLE 222, 236
TITLE entry 208
TRACE 32, 383, 475
TRACE control statement 186
trace off 467
trace on 475
trace output 383
trailer 104
transfer of invariant expressions 352
TRANSIENT file 235
TRANSMIT condition 218, 260, 266
TSOSLNK 149
tuning a program for virtual storage 331
type BIT 428

U
U format 211
undefined records 211
UNDEFINEDFILE condition 227, 238, 259, 268
UNLOAD 285
user identification 207
user library 28
utilities 453

V
V format 211
valid key 276
validity, control statement 52
VARIABLE 296, 309
variable 478
variable length entry 377
variable-length records 211
VARYING attribute 229
version 70, 91, 207
VOLUME 217
volumes 217

W
WARNING 82, 85, 122
warning 41, 122
warnings and error messages 499
WRITE FROM 232, 426
WRITE statement 256
WRPASS parameter 218
WXTRN 150, 306

Information on this document

On April 1, 2009, Fujitsu became the sole owner of Fujitsu Siemens Computers. This new subsidiary of Fujitsu has been renamed Fujitsu Technology Solutions.

This document from the document archive refers to a product version which was released a considerable time ago or which is no longer marketed.

Please note that all company references and copyrights in this document have been legally transferred to Fujitsu Technology Solutions.

Contact and support addresses will now be offered by Fujitsu Technology Solutions and have the format *...@ts.fujitsu.com*.

The Internet pages of Fujitsu Technology Solutions are available at *http://ts.fujitsu.com/*...
and the user documentation at *http://manuals.ts.fujitsu.com*.

Hinweise zum vorliegenden Dokument

Zum 1. April 2009 ist Fujitsu Siemens Computers in den alleinigen Besitz von Fujitsu übergegangen. Diese neue Tochtergesellschaft von Fujitsu trägt seitdem den Namen Fujitsu Technology Solutions.

Das vorliegende Dokument aus dem Dokumentenarchiv bezieht sich auf eine bereits vor längerer Zeit freigegebene oder nicht mehr im Vertrieb befindliche Produktversion.

Bitte beachten Sie, dass alle Firmenbezüge und Copyrights im vorliegenden Dokument rechtlich auf Fujitsu Technology Solutions übergegangen sind.

Kontakt- und Supportadressen werden nun von Fujitsu Technology Solutions angeboten und haben die Form *...@ts.fujitsu.com*.

Die Internetseiten von Fujitsu Technology Solutions finden Sie unter *http://de.ts.fujitsu.com/*..., und unter *http://manuals.ts.fujitsu.com* finden Sie die Benutzerdokumentation.