English

FUJITSU

COBOL2000 V1.6

COBOL Compiler

User Guide

June 2018

Table of Contents

COBOL Compiler. User Guide. 7
1 Preface 8
1.1 Objectives and target groups of this manual 9
1.2 Summary of contents 10
1.3 Expansion levels of the COBOL2000 system 11
1.4 Changes compared to the predecessor version 12
1.5 Notational conventions 13
1.6 Definitions of terms used in this manual 14
2 From compilation unit to executable program 16
2.1 Preparing the compilation unit 18
2.1.1 Input from cataloged files 19
2.1.2 Input from PLAM libraries 20
2.2 Source data input 22
2.2.1 Assigning the compilation unit with the ASSIGN-SYSDTA command 23
2.2.2 Input of program segments 24
2.2.3 Assignment to compiler variables to control source text manipulation 28
2.3 I/O for repositories 30
2.3.1 Principle of a repository 31
2.3.2 Assigning a repository 32
2.4 Output from the compiler 33
2.4.1 Output of modules 34
2.4.2 Output of listings and messages 36
2.5 Compiler control options 37
2.6 Terminating the compiler run 38
2.7 Compiling a compilation group 39
2.8 Parametrized classes and interfaces 40
3 Controlling the compiler via SDF 44
3.1 Calling the compiler and entering options 45
3.1.1 SDF expert mode 46
3.1.2 SDF menu mode 47
3.2 SDF syntax description 49
3.3 SDF options for controlling the compiler run 53
3.3.1 SOURCE option 54
3.3.2 SOURCE-PROPERTIES option 55
3.3.3 ACTIVATE-FLAGGING option 57
3.3.4 COMPILER-ACTION option 58
3.3.5 MODULE-OUTPUT option 61

3.3.6 LISTING option 63
3.3.7 TEST-SUPPORT option 70
3.3.8 OPTIMIZATION option 72
3.3.9 RUNTIME-CHECKS option 73
3.3.10 COMPILER-TERMINATION option 75
3.3.11 MONJV option 76
3.3.12 RUNTIME-OPTIONS option 77
3.3.13 VERSION option 79
4 Controlling the compiler with COMOPT statements 80
4.1 Source data input under COMOPT control 82
4.1.1 Assigning the compilation unit with the END statement 83
4.1.2 Assigning the compilation unit with the ADD-FILE-LINK command and COMOPT SOURCE-ELEMENT 84
4.2 Table of COMOPT operands 85
5 Controlling the compiler with compiler directives 95
5.1 IMP COMPILER-ACTION 96
5.2 IMP LISTING-OPTIONS 97
5.3 IMP PRINT-DIRECTIVES 98
5.4 IMP RUNTIME-ERRORS 100
6 Linking, loading, starting 101
6.1 Functions of the linkage editor 103
6.2 Static linkage using TSOSLNK 105
6.3 Linking using BINDER 108
6.4 Dynamic linking and loading using DBL 110
6.5 Loading and starting executable programs 112
6.6 Program termination 113
6.7 Shareable COBOL programs 116
7 Debugging aids for program execution 118
7.1 Advanced Interactive Debugger (AID) 119
7.1.1 Conditions for symbolic debugging 120
7.1.2 Symbolic debugging with AID 122
7.1.3 Predefined information 124
7.1.4 Notes on symbolic debugging of nested programs 125
7.1.5 Notes on debugging object-oriented COBOL programs 126
7.1.6 Information on testing programs with user-defined types 128
7.2 Debugging lines 130
8 Interface between COBOL programs and BS2000/OSD 131
8.1 Input/output via system files 132
8.1.1 COBOL language elements 133
8.1.2 System files: primary assignments, reassignments, record formats 135
8.2 Job switches and user switches 137

8.3 Job variables 141
8.4 Accessing an environment variable 144
8.5 Compiler and operating system information 145
9 Processing of cataloged files 149
9.1 Basic information on the structure and processing of cataloged files 150
9.1.1 Basic concepts relating to the structure of files 151
9.1.2 Assignment of cataloged files 153
9.1.3 Definition of file attributes 156
9.1.4 Disk and file formats 159
9.2 Sequential file organization 162
9.2.1 Characteristics of sequential file organization 163
9.2.2 COBOL language tools for the processing of sequential files 164
9.2.3 Permissible record formats and access modes 169
9.2.4 Open modes and types of processing (sequential processing) 170
9.2.5 Line-sequential files 172
9.2.6 Creating print files 174
9.2.7 Processing files in ASCII or in ISO 7-bit code 178
9.2.8 Processing magnetic tape files 179
9.2.9 I-O status 181
9.3 Relative file organization 185
9.3.1 Characteristics of relative file organization 186
9.3.2 COBOL language tools for processing relative files 187
9.3.3 Permissible record formats and access modes 192
9.3.4 Open modes and types of processing (relative files) 193
9.3.5 Random creation of a relative file 197
9.3.6 I-O status 200
9.4 Indexed file organization 204
9.4.1 Characteristics of indexed file organization 205
9.4.2 COBOL language tools for the processing of indexed files 207
9.4.3 Permissible record formats and access modes 212
9.4.4 Open modes and types of processing (indexed files) 213
9.4.5 Positioning with START 217
9.4.6 I-O status 219
9.5 Shared updating of files (SHARED-UPDATE) 223
9.5.1 ISAM files 224
9.5.2 PAM files 229
10 Processing XML documents 231
10.1 Making XML documents available 232
10.2 Using XML language elements in programs 233
10.3 Linking, loading, starting programs with XML language elements 234

10.4 Encoding identification . . . 236
10.5 Obtaining the parser . . . 238
10.6 Extended I-O status for XML statements (CBX code) . . . 239
11 Sorting and merging . . . 243
11.1 COBOL language elements for sorting and merging files . . . 244
11.2 Files for the sort program . . . 245
11.3 Checkpointing and restart for sort programs . . . 247
11.4 Sorting tables . . . 248
11.5 Sorting with extended character sets . . . 249
12 Checkpointing and restart . . . 250
12.1 Checkpointing . . . 251
12.2 Restart . . . 252
13 Program linkage . . . 253
13.1 Linking and loading subprograms . . . 254
13.2 COBOL special register RETURN-CODE . . . 259
13.3 Passing parameters to programs in other languages . . . 260
13.4 Unloading COBOL subroutines . . . 261
14 COBOL2000 and POSIX . . . 262
14.1 Overview . . . 263
14.1.1 Compiling . . . 264
14.1.2 Linking . . . 265
14.1.3 Debugging . . . 267
14.2 Reading in the compilation unit . . . 268
14.3 Controlling the compiler . . . 269
14.3.1 General options . . . 270
14.3.2 Option for compiler statements . . . 271
14.3.3 Option for compiler listing output . . . 273
14.3.4 Options for the linkage run . . . 274
14.3.5 Debugger option . . . 275
14.3.6 Input files . . . 276
14.3.7 Output files . . . 277
14.4 Introductory examples . . . 278
14.5 Comparison with COBOL2000 in BS2000 . . . 279
14.5.1 Restrictions on the functionality of the language . . . 280
14.5.2 Extensions to the functionality of the language . . . 281
14.5.3 Differences in the program/operating system interfaces . . . 282
14.6 Processing POSIX files . . . 284
14.6.1 Program execution in the BS2000 environment . . . 285
14.6.2 Program execution in the POSIX shell . . . 287
14.6.3 I-O status . . . 288

15 Useful software for COBOL users 293
15.1 Advanced Interactive Debugger (AID) 294
15.2 Library Maintenance System (LMS) 296
15.3 Job variables 297
15.4 Database interface ESQL-COBOL 298
15.5 Universal Transaction Monitor openUTM 299
15.6 Net Express® development environment with the BS2000/OSD option . . . 300
16 Messages of the COBOL2000 system 303
17 Appendix 306
17.1 Structure of the COBOL2000 system 307
17.1.1 Structure of the COBOL2000 compiler 308
17.1.2 The COBOL2000 runtime system 310
17.2 Database operation (UDS/SQL) 316
17.3 Description of listings 319
17.3.1 Header line 320
17.3.2 Control statement listing 321
17.3.3 Source listing for a compilation unit 322
17.3.4 Format control statements TITLE, EJECT, SKIP 326
17.3.5 Diagnostic (error message) listing 328
17.3.6 Locator map listing 329
18 Related publications 331

COBOL Compiler. User Guide.

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 9001:2015

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 9001:2015.

Copyright and Trademarks

Copyright © 2018 Fujitsu Technology Solutions GmbH.

All rights reserved.
Delivery subject to availability; right of technical modifications reserved.

All hardware and software names used are trademarks of their respective manufacturers.

1 Preface

COBOL2000 is the COBOL compiler for object oriented programming in BS2000/OSD.

1.1 Objectives and target groups of this manual

This User Guide describes how COBOL programs are processed in a BS2000 system environment.

It is intended for users who are familiar with the programming language COBOL as well as the BS2000 operating system.

1.2 Summary of contents

This User Guide describes how COBOL programs produced in a BS2000 environment can be

- prepared for compilation,
- compiled with the COBOL2000 compiler,
- linked into executable programs and loaded into main memory, and
- tested for logical errors in debugging sessions.

In addition, it includes details of how COBOL programs

- utilize BS2000 facilities for information exchange,
- process cataloged files,
- sort and merge files,
- take checkpoints and use them for subsequent restarts, and
- communicate with other programs (program linkage).

In chapter "Processing XML documents" it also explains how XML documents are processed.

In chapter “COBOL2000 and POSIX” the manual also describes how the COBOL2000 compiler and the programs it generates can be used within the POSIX subsystem of BS2000/OSD and on how the POSIX file system is accessed.

Familiarity with the COBOL programming language and with simple applications of BS2000 is prerequisite to understanding this manual.
Language elements of the COBOL2000 compiler are discussed in detail in the “COBOL2000 Reference Manual” [1].

Other publications are referred to in the text by their abbreviated titles or by numbers enclosed in square brackets. The full titles are listed with their corresponding numbers under “Related publications”.

Readme file

The functional changes to the current product version and revisions to this manual are described in the product-specific Readme file.

Readme files are available to you online in addition to the product manuals under the various products at http://manuals.ts.fujitsu.com. You will also find the Readme files on the Softbook DVD.

Information under BS2000/OSD

When a Readme file exists for a product version, you will find the following file on the BS2000 system:

`SYSRME.<product>.<version>.<lang>`

This file contains brief information on the Readme file in English or German (<lang>=E/D). You can view this information on screen using the `/SHOW-FILE` command or an editor. The `/SHOW-INSTALLATION-PATH INSTALLATION-UNIT=<product>` command shows the userID under which the product’s files are stored.

Additional product information

Current information, version and hardware dependencies, and instructions for installing and using a product version are contained in the associated Release Notice. These Release Notices are available online at http://manuals.ts.fujitsu.com.

1.3 Expansion levels of the COBOL2000 system

The COBOL2000 system V1.6 is supplied in two configurations:

- COBOL2000 (maximum configuration)
- COBOL2000-BC (basic configuration)

The BC version of COBOL2000 does not include the following control and language facilities:

- Symbolic debugging with AID
- Output of a list of all error messages
- COBOL-DML language elements for database links
- Report-Writer language module
- Compiler and program execution in the POSIX subsystem
- Starter phase

This User Guide refers to the full-featured configuration. Descriptions of the functions that are not supported by the COBOL2000-BC basic configuration are indicated by an appropriate note.

The COBOL compiler supplied with COBOL2000 Version 1.6 does not include the COBOL runtime system.
The COBOL runtime system is a component of CRTE, the common runtime environment for COBOL, C and C++ programs.
All programs compiled with COBOL85 compilers as of Version 1.0A as well as the COBOL2000 Compiler V1.0A and higher are supported by the COBOL runtime system included in CRTE.

1.4 Changes compared to the predecessor version

Processing of XML documents

- Provision of the XML generator

1.5 Notational conventions

The following metalinguistic conventions are followed in this user guide:

COMOPT	Uppercase letters denote keywords that must be entered exactly as shown.
name	Lowercase letters denote variables which must be replaced by current values when being entered.
YES NO	Underlining indicates a default value that is automatically used when no value has been specified by the user.
{YES | NO}	Braces enclose alternatives, i.e. one of the specified values must be selected. The alternatives are depicted one under the other. If one of the listed values is a default value, no entry need be specified when the default value is desired.
{ | a | b |}	Vertical bars within braces enclose optional entries. Here at least one (a or b) but also more than one entry (a and b) can be selected. Each alternative should, however, be used no more than once.
{YES/NO}	A slash separating two adjacent entries also indicates that the entries represent alternatives from which one must be selected. No entry is required if the given default value is desired.
[ ]	Brackets enclose optional entries that may be omitted by the user.
[| a | b |]	Vertical bars within brackets enclose optional entries. Here the specification can be omitted, or more than one of the specified values can be selected. Each alternative should, however, be used no more than once.
( )	Parentheses must be entered.
'BLANK'	This symbol denotes that at least one blank is required for syntactical reasons.
Special characters	Must be entered as given.

Note

The usual COBOL conventions apply with regard to the COBOL formats shown in this manual (see “COBOL2000 Reference Manual” [1]).

1.6 Definitions of terms used in this manual

A number of different terms are frequently used for the same object when describing the process of generating a program. The result obtained from a compiler run, for instance, is called an object module, whereas the same object module in LLM format is referred to as a link-and-load module.

The synonymous use of such terms serves a practical purpose within the context of specific components, but may occasionally be confusing to the user. To prevent any misunderstanding, the most important terms that are used synonymously are defined below for reference.

Object module, prelinked module

The term “object module” refers to object modules as well as “prelinked” modules.

Object modules and prelinked modules have the same structure and are stored in the same format (object module format). Such modules are stored in PLAM libraries as elements of type R.

Object modules are generated by the compiler. These modules are created whenever one or more source compilation units are compiled.

Prelinked modules are generated by the linkage editor TSOSLNK. A prelinked module is a single module that contains a combination of one or more object modules and/or other prelinked modules.

Object modules can be processed further by the static linkage editor TSOSLNK, the dynamic binder loader DBL, and the linkage editor BINDER.

Module, object module, link-and-load module (LLM)

The term “module” is a generic term for the result obtained by compiling a compilation unit with the COBOL2000 compiler. An “object module” is a module in OM format; a “link-and-load module” is a module in LLM format.

Executable program, program, load module, object program

Executable programs are programs that are generated by linkage editors and stored in PLAM libraries as elements of type C. They are often referred to simply as “programs” in this manual. In contrast to object modules, executable programs cannot be processed further by the linkage editor TSOSLNK; they are loaded into memory by a (static) loader.

In some documentation, the term “load module” is also used synonymously for an executable program. Technically speaking, however, a load module is a loadable unit **within** a program. A segmented program, for example, may consist of multiple load modules.

The synonymous use of the term “object program” for a load module could lead to confusion in COBOL terminology. The COBOL Standard uses the term object program for the object generated by the COBOL compiler, without taking any implementation-specific need for a linkage run into account.

Job, task, process

A job is a sequence of commands, statements, etc., that are specified between the SET-LOGON-PARAMETERS and EXIT-JOB (or LOGOFF) commands. A distinction is made between batch jobs (ENTER jobs) and interactive jobs (executed in a dialog).

A job is considered a task if system resources are allocated to it (CPU, memory, devices, etc.). In interactive mode, a job becomes a task as soon as the SET-LOGON-PARAMETERS command is accepted.

The activities which run within a task, e.g. program runs, are referred to as processes.

In the past the terms “task” and “job” have often been used as synonyms for “process”. In this User Guide these terms are used as described above. The phrase “at process termination” thus means: when a program run has terminated. “End of task” refers to the time after the EXIT-JOB or LOGOFF command. The term “task switch” is used instead of “process switch”.

Compilation group

A group of compilation units that are compiled together.

Compilation unit

A source unit that cannot be nested in other source units (e.g. a program prototype, program definition, class definition or interface definition). Compilation units can be grouped together as elements of a compilation group, but may also be compiled separately.

Source unit

A sequence of statements that begins with an Identification Division and ends with an associated END entry (can be nested).

2 From compilation unit to executable program

Three steps are needed to convert a COBOL compilation unit into an executable program:

1. Reading in the compilation unit (see section “Preparing the compilation unit”)
2. Compilation: The compilation unit must be converted into machine language. The compiler generates an object module or a link-and-load module (LLM) and logs the sequence and results of the compilation in listings.
3. Linkage: One or more modules are linked with so-called runtime modules to create an executable program (see chapter “Linking, loading, starting”).

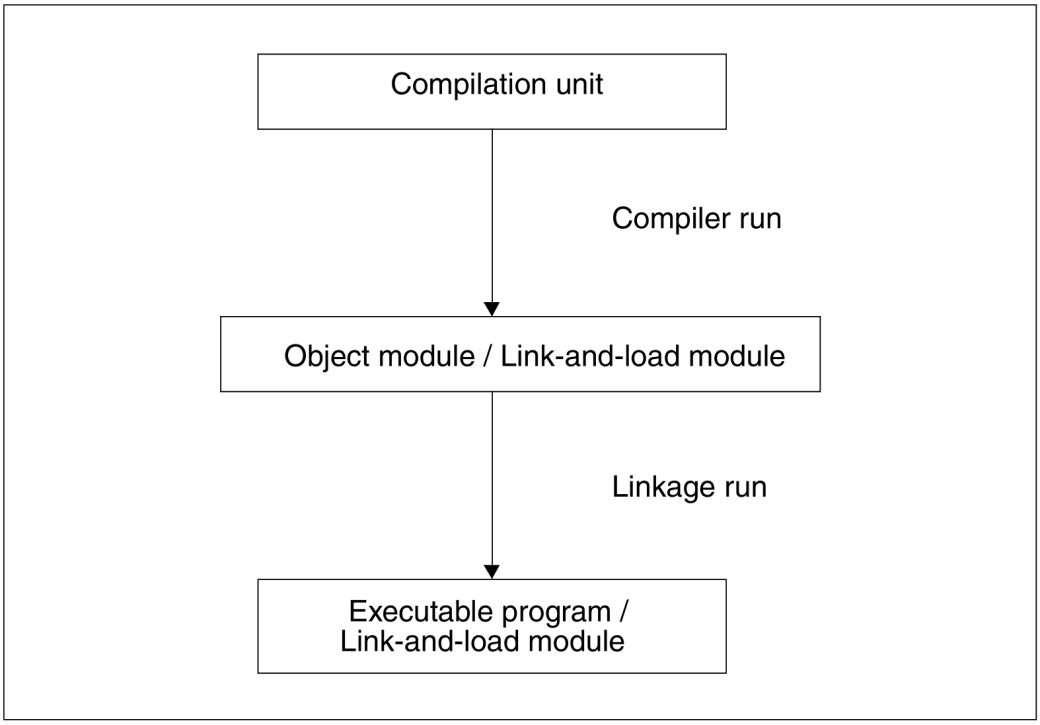


Figure 1: Producing an executable program

Three functions are performed by the compiler during the compilation run:

- Checking of the compilation unit for syntax and semantic errors,
- Conversion of COBOL code into machine language,
- Output of messages, listings, and modules.

The user can make use of control statements to

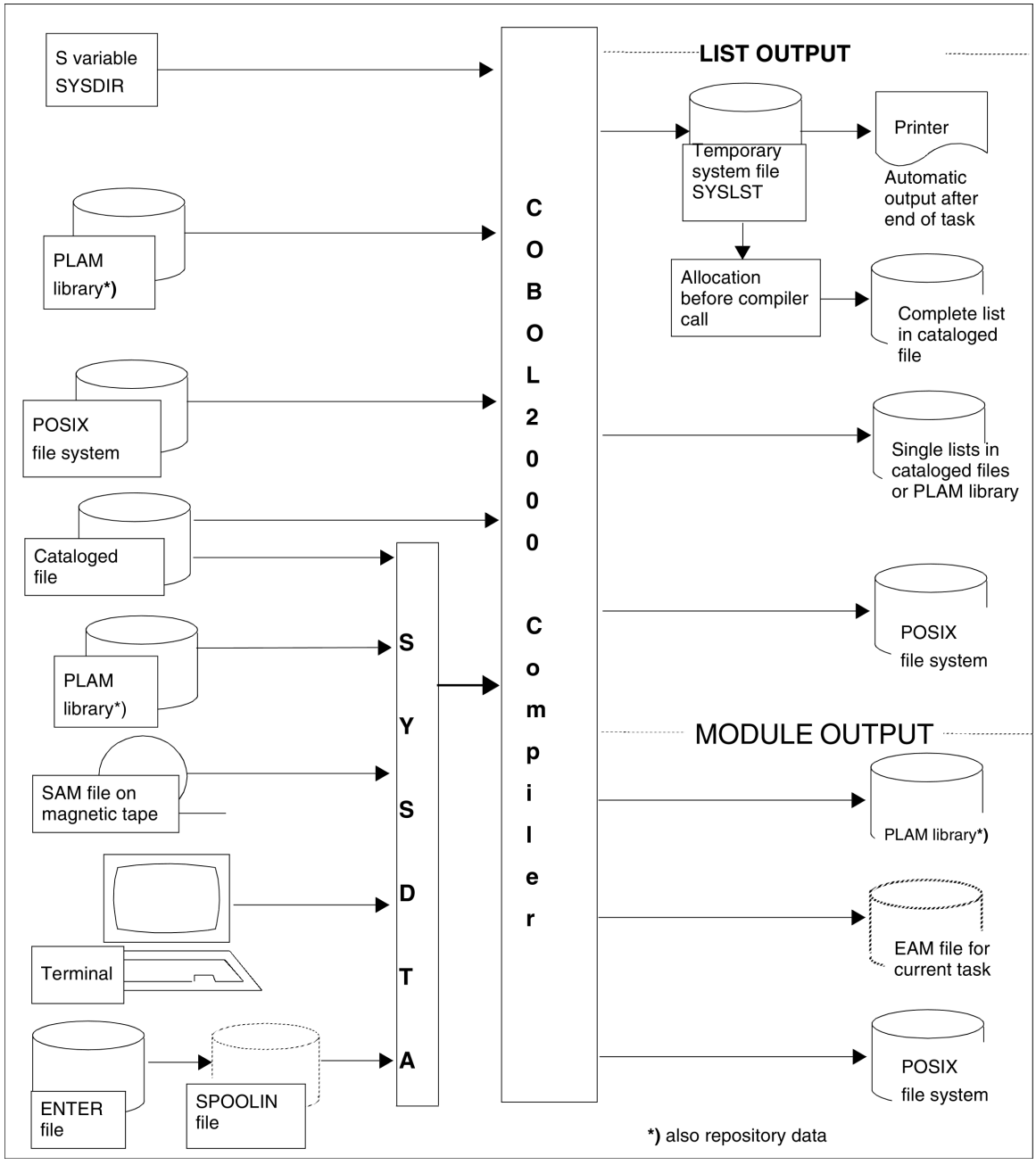
- select COBOL2000 functions,
- allocate resources for input and output,
- define characteristics of the modules,
- specify the type and scope of listing output.

The control options provided by COBOL2000 and by the operating system are described in detail in the chapter “Controlling the compiler via SDF” and the chapter “Controlling the compiler with COMOPT statements”.

A compilation unit is a COBOL source program that can be compiled in **one** compilation run. However, it is also possible to compile a series of compilation units, a so-called compilation group, in a single compilation run.

> The information relating to compilation units presented in the following sections also applies to compilation groups unless stated otherwise.

Possible input sources and output locations for the compiler:



2.1 Preparing the compilation unit

After a COBOL compilation unit has been coded, it must be made available to the compiler for compilation. This can be achieved in several ways, the most common methods being:

- input from a file and
- input from a PLAM library.

The operating system supports the loading of compilation units into files or PLAM libraries through various commands and utility routines.

2.1.1 Input from cataloged files

COBOL2000 can process compilation units from SAM or ISAM files. If ISAM files are used, they must be cataloged with KEYPOS=5 and KEYLEN=8. The method used to enter a compilation unit into such a file depends on the form in which it is available:

- If the compilation unit is already stored on an external volume (e.g. magnetic tape), it can be moved to a cataloged file by using suitable
  - BS2000 commands (see “Commands” manual [3]); e.g. the COPY-FILE command (for compilation units on magnetic tape) or
  - utility routines, e.g. ARCHIVE for magnetic tapes.
- If a new compilation unit is to be created, the file editor EDT (see the “EDT” manual [19]) can be used. This editor can process both SAM and ISAM files and includes special functions to support the formatted input and subsequent editing of COBOL compilation units. Some of these functions are listed below:
  - The option of setting tabs enables the programmer to quickly and reliably move to the starting column of the program text area and thus facilitates compliance with the reference format for COBOL programs (see “COBOL2000 Reference Manual” [1]).
  - Functions to insert, delete, copy, transfer and edit single source lines and ranges of lines or columns.
  - Statements to insert, delete and replace character strings in the file.

2.1.2 Input from PLAM libraries

Apart from SAM or ISAM files, PLAM libraries are another important input source for the COBOL2000 compiler.

Characteristics of PLAM libraries

PLAM libraries are PAM files that are processed by using the PLAM (**P**rimary **L**ibrary **A**ccess **M**ethod) access method. These libraries can be created and maintained with the help of the LMS utility routine (see “LMS” manual [11]).

The elements of a PLAM library typically include not only compilation units and program segments (COPY elements), but also, for example, modules and executable programs. The individual element types are characterized by different type designations.

Among others, elements of the following types may be stored in a PLAM library:

Type designation	Content of the library elements
S	compilation units, COPY elements
R	Object modules or prelinked modules
C	Executable programs
J	Procedures
L	Link-and-load modules (LLMs)
P	Print-edited data (lists)
X	REPOSITORY data

Table 1: PLAM element types

A PLAM library may also contain elements of the same name, provided they can be differentiated by version or type designation.

The advantages of maintaining data in PLAM libraries are listed below:

- Up to 30% storage space can be saved by combining different types of elements and by using additional compression techniques.
- Access times are shorter for the various types of elements in the same program library as opposed to access times for conventional data maintenance.
- The burden on EAM storage space is reduced when link-and-load modules are directly stored as PLAM library elements.

Input into PLAM libraries

PLAM libraries can accept compilation units

- from files
- from other libraries
- via SYSDTA or SYSIPT, i.e. from a terminal or a temporary spoolin file.

The method used to enter a compilation unit into the PLAM library depends on the form in which it exists:

- If the compilation unit exists as a cataloged file or as an element of a library, it can be copied to a PLAM library by using the LMS utility routine (see Example 2-1).
  When transferring a compilation unit from an ISAM file with LMS, it should be noted that the ISAM key is not copied with PAR KEY=YES and SOURCE-ATTRIBUTES=KEEP. The COBOL2000 compiler cannot process any compilation unit with an ISAM key from a library.
- If the compilation unit is being entered for the first time, it can also be directly written into a PLAM library (as an element) by using the EDT file editor.

Example 2-1

Transferring a compilation unit from a cataloged file to a PLAM library

```
/START-LMS————————————————————————————————————————————————————————(1)
%  LMS0310 LMS VERSION  '03.3A30' STARTED
//OPEN-LIBRARY LIB=PLAM.LIB,MODE=UPDATE(STATE=NEW)————————————————————(2)
//ADD-ELEM FROM-FILE=SOURCE.EINXEINS,TO-E=LIB-ELEM(ELEM=EINXEINS,TYPE=S)(3)
//END——————————————————————————————————————————————————————————————(4)
%  LMS0311 LMS V03.3A30   TERMINATED NORMALLY
```

(1) The LMS utility routine is invoked.

(2) PLAM.LIB is defined as the new (STATE=NEW) output library (USAGE=OUT). By default, it is created as a PLAM library by LMS.

(3) The compilation unit is transferred from the cataloged file SOURCE.MULTABLE and is included under the name MULTABLE as an S-type element in the PLAM library.

(4) The LMS run is terminated; all open files are closed.

2.2 Source data input

Input to the compiler may consist of the following source data:

- Compilation units (individual compilation units or a compilation group)
- Program segments (COPY elements)
- Compiler control statements (COMOPT statements or SDF options)
- Repository data (interface definitions)

The compiler can process compilation units from cataloged SAM or ISAM files, elements of PLAM libraries and POSIX files. Input from compilation units is described in the chapter “From compilation unit to executable program” and the chapter “COBOL2000 and POSIX”.

The control statements for the input are detailed in the chapter “Controlling the compiler via SDF” and the chapter “Controlling the compiler with COMOPT statements”. The required assignment of the system file SYSDTA, which is common to both control modes, is presented below.

2.2.1 Assigning the compilation unit with the ASSIGN-SYSDTA command

By default, the compiler expects source data from the system file SYSDTA. SYSDTA can be assigned to a cataloged file or library element before the compiler is called. The command for this is:

```
/ASSIGN-SYSDTA [TO =] {filename | *LIB-ELEM(LIB=library,ELEM=element)}
```

Detailed information on the ASSIGN-SYSDTA command can be found in the “BS2000/OSD-BC Commands” manual [3].

Example 2-2

Reading a compilation unit from a cataloged file

```
/ASSIGN-SYSDTA SOURCE.MULTABLE ————————————————————————————(1)
Call to compiler ——————————————————————————————————————————(2)
/ASSIGN-SYSDTA *PRIMARY  ——————————————————————————————————(3)
```

(1) The cataloged file SOURCE.MULTABLE, which contains the compilation unit to be compiled, is assigned to the SYSDTA system file.

(2) The compiler is loaded and started. It processes the data that is received from SYSDTA. This applies only when the compiler is not called via the SDF interface and when source = ... is not specified here.

(3) The SYSDTA system file is reset to its primary assignment for subsequent tasks.

Example 2-3

Reading a compilation unit from a library

```
/ASSIGN-SYSDTA *LIBRARY-ELEMENT(LIB=PLAM.LIB,ELEM=EXAMP3) ————————(1)
Call to compiler ——————————————————————————————————————————(2)
/ASSIGN-SYSDTA *PRIMARY  ——————————————————————————————————(3)
```

(1) The system file SYSDTA is assigned to the element EXAMP3 in the PLAM library PLAM.LIB.

(2) The compiler is invoked. It accesses the assigned library element via SYSDTA. See the example above.

(3) SYSDTA is reset to its primary assignment.

Other methods for the input of source data are related to controlling the compiler with COMPOPT statements and are described in the chapter “Controlling the compiler with COMOPT statements”.

2.2.2 Input of program segments

Program segments (COPY elements) can be stored in libraries as distinct entities independent of the compilation unit in which they are used. This is especially recommended when identical program segments are used in different compilation units.
In the compilation unit itself, such program segments are represented by a COPY statement. COPY statements may be located at any position in the compilation unit (except for comment lines and non-numeric literals).

When the compiler encounters a COPY statement in the compilation unit being compiled, it inserts the element specified in the COPY statement from the appropriate library. The COPY element is then compiled as if it were a part of the compilation unit itself.The COPY statement format is shown and explained in chapter “Controlling the compiler” in the “COBOL2000 Reference Manual” [1].

Input of COPY elements from PLAM libraries

Before invoking the compiler, the libraries that contain the COPY elements must be assigned to the compiler using the ADD-FILE-LINK command and linked to the file link names specified below.
If a library name is specified in the COPY statement, the link name is formed from the first 8 characters of the library name.
If no library name has been declared in the COPY statement, up to ten libraries can be linked using the standard link names COBLIB, and COBLIB1 through COBLIB9. The compiler then searches the assigned libraries in hierarchical order until the required COPY element is found.

Depending on how the COPY statement is formulated in the compilation unit, the following assignments are required in the SET-FILE-LINK command:

COPY statement	**ADD-FILE-LINK command**
`COPY textname` textname element name (max. 31 characters long)	`ADD-FILE-LINK [LINK-NAME=]standard-linkname, [FILE-NAME=]libname` standard-linkname COBLIB COBLIB1..COBLIB9 libname Name of the cataloged library in which the COPY element is stored
`COPY textname OF library` library library name (max. 31 characters long)	`ADD-FILE-LINK [LINK-NAME=] linkname, [FILE-NAME=] libname` linkname The first eight characters of the name of the library specified in the COPY statement libname Name of the cataloged library in which the COPY element is stored

Input of COPY elements from the POSIX file system

If the POSIX subsystem is available, you can also pass COPY texts from the POSIX file system to the compiler. To do this you use an S variable with the default name of SYSIOL-COBLIB or SYSIOL-libraryname. The formulation of the COPY statement in the compilation unit influences the S variable as follows (see also “Example 2-6”); this does not apply to BC (basic configuration):

COPY statement	S variable
`COPY textname` textname name of POSIX file (max. 31 characters long) containing the COPY text. textname must not contain lowercase letters.	`DECL-VAR SYSIOL-COBLIB,INIT='*POSIX (pfad)', SCOPE=*TASK` pathname Absolute pathname (beginning with /) of the directory in which a search is made for the file textname
`COPY textname OF library` library Library name (max. 31 characters long) for forming the S variable with the name SYSIOL-library. library must not contain lowercase letters.	`DECL-VAR SYSIOL-libname,INIT='*POSIX (pfad)', SCOPE=*TASK` libname The first 8 characters of library pathname Absolute pathname (beginning with /) of the directory in which a search is made for the file textname

Example 2-4

Input of two COPY elements

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG.
...
    COPY XYZ.————————————————————————————————————————(1)
    COPY ABC OF LIBRARY.————————————————————————————(2)
...
```

Assignment and linkage:

```
/ASSIGN-SYSDTA EXAMPLE1 ————————————————————————————(3)
/ADD-FILE-LINK COBLIB,LIB1 —————————————————————————(4)
/ADD-FILE-LINK LIBRARY,LIB2 ————————————————————————(5)
```

Call to compiler

The compilation unit in the file EXAMPLE1 includes the following COPY statements:

(1) XYZ is the name of the element under which the COPY element is stored in the PLAM library LIB1.

(2) ABC is the name of the element under which the COPY element is stored in the PLAM library LIB2 with the link name LIBRARY.

(3) SYSDTA is assigned to the file EXAMPLE1. From this file, the compiler receives a compilation unit in which two COPY statements are written.

(4) The first ADD-FILE-LINK command assigns the PLAM library LIB1 and links it to the standard link name COBLIB.

(5) The second ADD-FILE-LINK command assigns the PLAM library LIB2 and links it to the library name LIBRARY specified in the COPY statement.

Example 2-5

Input of several COPY elements from different libraries

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG1.
...
      COPY A1.                                                  (1)
      COPY B1.                                                   |
      COPY D1.                                                  (1)
...
```

Assignment and linkage:

```
/ASSIGN-SYSDTA EXAMPLE2 ————————————————————————————————— (2)

/ADD-FILE-LINK COBLIB,A                                         (3)
/ADD-FILE-LINK COBLIB1,B                                         |
/ADD-FILE-LINK COBLIB3,D                                        (3)
Call to compiler ———————————————————————————————————————— (4)
```

The compilation unit EXAMPLE2 includes the following COPY statements:

(1) A1, B1 and D1 are the names under which the COPY elements have been stored in the cataloged libraries A, B and D.

(2) SYSDTA is assigned to the cataloged file EXAMPLE.2. From this file, the compiler receives a compilation unit in which three COPY statements are written.

(3) Libraries A, B and D are assigned and linked to standard link names. Whereas the standard link name COBLIB must always be assigned, the number and sequence of links to COBLIB1 through COBLIB9 are freely selectable.

(4) After invocation, the compiler searches COBLIB, COBLIB1, and COBLIB3 in the given order for the elements specified in the COPY statements.

Example 2-6

Input of a COPY element from the POSIX file system

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG1.
...
COPY ATEXT. ————————————————————————————————————————(1)
...
```

Assigning the POSIX file system by declaring and setting an S variable:

```
/DECL-VAR SYSIOL-COBLIB,INIT=‘*POSIX(/usr/dir1),*POSIX(/usr/dir2)‘, –
/                            SCOPE=*TASK ————————————————(2)
/START-COBOL2000-COMPILER? ————————————————————————————(3)
```

(1) The COPY element ATEXT is a file in the POSIX file system.

(2) The SDF-P command DECL-VARIABLE sets the variable to the paths of the POSIX directories dir1 and dir2 which are to be searched for ATEXT.

(3) Access to the POSIX file system is only possible when the compiler is invoked under SDF control. By means of the “?” appended to the call command, the user is placed in SDF menu mode (see section “SDF menu mode”) in which further entries on controlling the compiler run can be made.

(4) The compiler accepts COPY elements from the POSIX file system only if their file names consist only of uppercase letters.

2.2.3 Assignment to compiler variables to control source text manipulation

Compiler directives allow the COBOL programmer to control the manipulation of the source text.

The following compiler directives are available:

- DEFINE directive
- EVALUATE directive
- IF directive

The compiler directives are described in detail in the “COBOL2000 Reference Manual” [1].

The DEFINE directive allows the programmer to define compiler variables in the source program. It is also possible to use S variables to assign values to these compiler variables prior to compilation. To do this, the programmer must define the variables in the program with the suffix ASPARAMETER. Compiler variables are assigned to S variables via the variable name which must be formed as follows:

DEFINE directive	S variable
`>>DEFINE variable AS PARAMETER`	`DECL-VAR SYSDIR-variable ...,SCOPE=*TASK`

The S variables must be declared with SCOPE=*TASK.

If values are to be supplied externally to the compiler variables, two different types of S variable are available and these must be declared with the required TYPE:

- numeric variables with TYPE=*INTEGER
- alphanumeric variables with TYPE=*STRING

The two examples below demonstrate how compiler variables are used in BS2000/OSD. The use of compiler variables when the compiler is called under POSIX is described in subsection "Using compiler variables under POSIX" of section "Compiling".

Example 2-7

Passing a numeric value

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG1
...
    >>DEFINE VLADIMIR AS PARAMETER. ———————————————————— (1)
...
```

Assignment and linkage:

```
/DECLARE-VARIABLE SYSDIR-VLADIMIR(TYPE=*INTEGER),SCOPE=*TASK  ——— (2)
/SET-VARIABLE SYSDIR-VLADIMIR=1234 ———————————————————————— (3)
```

Call to compiler

(1) The DEFINE directive specifies a compiler variable with a content which the COBOL compiler expects to find in an S variable.

(2) The SDF-P command DECLARE-VARIABLE declares an S variable: VLADIMIR is the name of the numeric compiler variable in the source program. The associated S variable is declared as SYSDIR-VLADIMIR with TYPE=*INTEGER.

(3) The SDF-P command SET-VARIABLE assigns the numeric value 1234 to the S variable SYSDIR-VLADIMIR.

Example 2-8

Passing an alphanumeric literal

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG2
...
    >>DEFINE JERRY AS PARAMETER.————————————————————————————— (1)
...
```

Assignment and linkage:

```
/DECLARE-VARIABLE SYSDIR-JERRY (TYPE=*STRING),SCOPE=*TASK ————————— (2)
/SET-VARIABLE SYSDIR-JERRY='This is a string'————————————————————— (3)
```

Call to compiler

(1) The DEFINE directive specifies a compiler variable with a content which the COBOL compiler expects to find in an S variable.

(2) The SDF-P command DECLARE-VARIABLE declares an S variable: JERRY is the name of the alphanumeric compiler variable in the source program. The associated S variable is declared as SYSDIR-JERRY with TYPE=*STRING.

(3) The SDF-P command SET-VARIABLE assigns the alphanumeric value “This is a string” to the S variable SYSDIR-JERRY. The surrounding quotes do **not** form part of the literal.

2.3 I/O for repositories

- Principle of a repository
- Assigning a repository

2.3.1 Principle of a repository

In order to compile object-oriented COBOL programs, you will need an external library (which is logically one library) called a “repository”, which contains the required definitions of the program interfaces, classes and other interfaces. Even for programs that are not object-oriented, a repository is needed whenever interfaces are to be checked in a CALL (see Format 3 of the CALL statement in the “COBOL2000 Reference Manual” [1]). These definitions are read by the COBOL compiler so that additional checks can be run on the compilation unit itself with the goal of preventing runtime errors.
A repository need not be a single physical library; it could also consist of many libraries in a hierarchy, as in the case of COPY libraries.
Repository data includes both input and output data.

2.3.2 Assigning a repository

Two repositories can be used during compilation:

- For input: this repository, which could also consist of a hierarchy of libraries, is searched for the interfaces being used.
- For output: this repository is used to store the interface definition of the source text being compiled. Only a single library is possible in this case.

Repository data is stored in PLAM libraries as elements of type X (see section “Input from PLAM libraries”).

The link names of the the files from which entries of the repository are to be imported can be specified using ADD-FILE-LINK commands.
These link names are REPLIB, REPLIB1,...,REPLIB9 and must be assigned by the user before the compiler is called. The files are searched in the specified order until a suitable interface definition is found.
If no repository entry is found in these libraries or no repository is specified, the library SYS.PROG.LIB is searched.

The link name for the library in which the output of an interface is to be placed is REPOUT. This library, which is also specified via a ADD-FILE-LINK command, could also be one of the input libraries.
If no link name is specified, the library SYS.PROG.LIB is used for the output by default. This output occurs only if UPDATE-REPOSITORY=YES has been specified.

2.4 Output from the compiler

- Output of modules
- Output of listings and messages

2.4.1 Output of modules

The compiler translates the source data input into machine language and generates one or more object modules (OM format) or link-and-load modules (LLM format) in the process. Each such module can be assigned a **L**ist for **S**ymbolic **D**ebugging (LSD) containing the symbolic addresses of the compilation unit.

By default, the compiler places the object modules in the temporary EAM file of the current task. The object modules are simply added to the library, i.e. stored without defining any relationship between them.
The EAM file belongs to the task under which the compilation is performed. It is created for this task during the first compilation run and is automatically deleted at task end (LOGOFF). If the results of the compilation are to be used later, it is up to the user to store the contents of the EAM file in a backup file for further processing. The LMS utility routine (see “LMS” manual [11]) is available to the user for backup of object modules from the EAM file in PLAM libraries.

If the compiled object modules are no longer needed in the EAM file, e.g. because the compilation unit still contains errors that need to be corrected, it is advisable to delete the EAM file - at the latest, before the next compiler run - by using the command:

```
/DELETE-SYSTEM-FILE [SYSTEM-FILE=] OMF
```

Link-and-load modules (LLMs) are always written to a PLAM library by the compiler as elements of type L.

If the POSIX subsystem is available, modules can be written to the POSIX file system. This option is described in section “MODULE-OUTPUT option”.

Formation of element names when modules are output to libraries

Compilation unit	Module format OM	Module format LLM
	Standard name derived from	
Code that is not shareable [1] not segmented segmented	ID-name[2] ..8[3] PROGRAM-ID-name 1..6 + segment number (for each segment)	ID-name[2] 1..30[4] PROGRAM-ID-name 1..30 [4] (segmentation ignored)
Shareable code [5]	ID-name [2] 1..7@ (code module) ID-name [2] 1..7 (data module)	ID-name [2] 1..30 [3]

Table 2: Formation of element names at module generation and output

1) Module generated with COMPILER-ACTION=MODULE-GENERATION(SHAREABLE-CODE=NO) or COMOPT GENERATE-SHARED-CODE=NO

2) ID-name is the PROGRAM-ID-name, CLASS-ID-name or INTERFACE-ID-name.

3) The name should be unique in the first 7 characters.

4) A separate element name may be selected instead of the standard name by specifying MODULE-OUTPUT=*LIBRARY-ELEMENT(LIBRARY=<filename>, ELEMENT=<composed-name>) or COMOPT MODULE-ELEMENT=element-name.
It should be noted, however, that this option has no effect on the name of the entry point, i.e. the name that is specified in the CALL statement.
(not allowed for program sequences).

5) Module generated with
COMPILER-ACTION=MODULE-GENERATION(SHAREABLE-CODE=YES) or
COMOPT GENERATE-SHARED-CODE=YES

2.4.2 Output of listings and messages

Output of listings

The compiler can generate the following listings during the compilation run:

Control statement listing	OPTION LISTING
Compilation unit listing	SOURCE LISTING
Library listing	LIBRARY LISTING
Object listing	OBJECT PROGRAM LISTING
Locator map Cross-reference listing	LOCATOR MAP LISTING
Error message listing	DIAGNOSTIC LISTING

The compiler writes each requested listing to a separate cataloged file by default. The listings stored in cataloged files can then be printed at any time by using the PRINT-FILE command (see “Commands” manual [3]).

Instead of being written to cataloged files, the requested listings can also be written as elements in a PLAM library.

If desired, the user can have the requested listings output to the system file SYSLST by means of an appropriate control statement. The system automatically sends the temporary file created for this purpose to the printer.

The generation and output of listings can be controlled by the user via

- the SDF option LISTING (see the chapter “Controlling the compiler via SDF”) or
- the COMOPT statements LISTFILES, LIBFILES or SYSLIST (see the chapter “Controlling the compiler with COMOPT statements”).

If the POSIX subsystem is available, listings (with the exception of the object listing) can be output to the POSIX file system. This option is described in section “LISTING option”.

Output of messages

All compiler messages related to the execution of the compilation run (COB90xx) are output to the terminal via the system file SYSOUT by default.
The texts of all the COB90xx messages that can be issued by the compiler are listed, together with comments, in chapter “Messages of the COBOL2000 system”.

2.5 Compiler control options

The method used for the input of source data, the attributes of the generated module, the output of messages and listings, as well as the output of the object module itself can be controlled by means of statements issued to the COBOL2000 compiler.
The COBOL2000 compiler can be controlled in different ways:

- by means of options in the SDF syntax format
- by means of COMOPT statements
- by means compiler directives

The user chooses one of the two control options through the type of command used to call the compiler:

Compiler invocation command	**Control mode**
/START-COBOL2000-COMPILER options	SDF control, expert mode
/?	SDF control, menu mode
/START-COBOL2000-COMPILER?	SDF control, menu mode
/START-PROGRAM *name compiler-phase* or *name starter-phase* [1]	COMOPT control
/START-COBOL2000-COMPILER	None; input of compilation unit from SYSDTA

Table 3: Compiler invocation commands and control modes
[1] does not apply to COBOL2000-BC

SDF control options are described in the chapter “Controlling the compiler via SDF”, COMOPT control options in the chapter “Controlling the compiler with COMOPT statements”.

Compiler control in the POSIX subsystem is described in the chapter “COBOL2000 and POSIX”.

Compiler control via compiler directives is described in chapter “Controlling the compiler with compiler directives” and in the “COBOL2000 Reference Manual” [1].

2.6 Terminating the compiler run

The termination behavior of the COBOL2000 compiler depends on

- the class of any errors detected in the compilation unit and
- whether the compiler itself executes without error.
  This behavior is particularly significant when the COBOL2000 compiler is called from within a procedure or is monitored by monitoring job variables.
  The following table provides an overview of the possible events, their impact on the further course of the procedure, and the contents of the return code indicator of the monitoring job variable:

Error	**Termination**	**Dump**	**Return code indicator in monitoring job variable**	**Trigger spin-off in procedures** [1)]
No error	Normal	No	0000	No
Error class F	Normal	No	0001	
Error class I	Normal	No	0001	
Error class 0	Normal	No	1002	
Error class 1	Normal	No	1003	
Error class 2	Normal	No	2004	No
Error class 3	Norma	No	2005	Yes
Compiler error	Abnormal	Yes	3006	

Table 4: Termination behavior of the compiler

1) When a spin-off is triggered, all subsequent commands are ignored with the exception of the SET-JOB-STEP, EXIT-JOB, LOGOFF, CANCEL-PROCEDURE, END-PROCEDURE and EXIT-PROCEDURE commands. The SET-JOB-STEP command terminates the spin-off, and processing is continued with the next command.

2.7 Compiling a compilation group

The special aspects to be noted when compiling a group of compilation units are discussed below.

Control statements:
Control statements that are specified before the compiler is called apply to all compilation units in the group.
No control statements must come between the compilation units in a group.

Output of listings via SYSLST:
Requested listings are output sequentially in a single SPOOL file in program-specific order.

Output of listings to cataloged files:
If standard names are used, the same number of files is created for each compilation unit in the sequence as the number of requested listings.
If standard link names are used, the creation of files is based on the type of listing: the file linked with OPTLINK contains a single options listing for all compilation units; the file linked with SRCLINK contains all compilation units; the file linked with ERRLINK contains all diagnostic listings, and the file linked with LOCLINK contains all locator map and cross-reference listings.

Output of listings to a PLAM library:
For each compilation group, the number of elements created is equal to the number of listings requested (the options listing is only created once).

Values indicated in monitoring job variables:
The return code of the compilation unit containing the error with the highest error weight is always indicated in the monitoring job variable.

Compiler termination:
If a compilation unit contains an error that aborts compilation of the program, the entire compiler run is terminated, i. e. none of the following compilation units are compiled.

Module output:
A separate module is generated for each compilation unit in the sequence. These modules are entered into the EAM file sequentially, or stored as individual elements in a PLAM library.

Repository output:
A repository entry is created for each compilation unit (if requested).

> ¡ When working with a repository (especially if a hierarchy is involved) and newly generated repository entries which are to be used in preceding or following programs, special care must be taken to ensure that the desired contents are actually accessed.

2.8 Parametrized classes and interfaces

3 steps must be distinguished when working with parameterized classes/interfaces:

1. **Precompilation** of a parameterized class/interface
2. **Usage** of a parameterized class/interface
3. **Expansion** of a parameterized class/interface

Precompilation of a parameterized class/interface takes place without any knowledge of the current parameters. One aim here is to detect syntax errors. The other is to store the interface and the source text of the parameterized class/interface in the repository (see the section “I/O for repositories”). The status of the source text, the status of the COPY elements and the status of the compiler directives (see the section “Source data input”) which the compilation unit of the parameterized class/interface addresses are recorded at the time precompilation takes place and are used for the later usages and expansions (i.e. subsequent modifications to these statuses have no affect on the usage and expansion).

The **usage** of parameterized classes/interfaces takes place in the compilation unit using EXPANDS clauses in the REPOSITORY paragraph. As a result new, concrete classes/interfaces are created. These consist of the current parameters specified in the compilation unit and of the repository data recorded during precompilation. The concrete classes/interfaces behave like non-parameterized classes/interfaces. Their characteristics are included in the user’s compilation of the parameterized classes/interfaces.

The **expansions** are initiated automatically following the user of the parameterized classes/interfaces or, in the case of compilation groups, after the last compilation unit. Here all concrete classes/interfaces which result from the usage of parameterized classes/interfaces are compiled. Only the data of the parameterized class/interface and the current parameters (including their repository data) are used for this purpose. No further source texts, library elements, etc. are required for the expansions. All the compiler options (see the chapter “Controlling the compiler via SDF”) which are valid for the compilation unit of the user of parameterized classes/interfaces are, as with compilation groups, also effective for the subsequent expansions. However, no source listing is created in the case of expansions. The effect of >>IMP directives is not suppressed. In contrast, compiler directives which apply for users of parameterized classes/interfaces have no influence on subsequent expansions.

Further details on parameterized classes/interfaces are provided in the “COBOL2000 Reference Manual” [1].

Example 2-9

Precompilation of a parameterized class

Source code

```
CLASS-ID. pkl USING fp.————————————————————————————————— (1)
...
REPOSITORY.
    CLASS fp.
...
01 obj-fp USAGE OBJECT REFERENCE fp.
01 obj-pkl USAGE OBJECT REFERENCE pkl.
...
```

Assignment and call to compiler

```
/ADD-FILE-LINK REPOUT,REPOSITORY———————————————————————— (2)
/START-COBOL2000-COMPILER ————————————————————————————— (3)
/ ... UPDATE-REPOSITORY=*YES ...———————————————————————— (4)
```

(1) The name of the parameterized class is `pkl`, the name of a formal parameter is `fp`.

(2) The library REPOSITORY is assigned to incorporate the repository data.

(3) Precompilation takes place using the COBOL2000 compiler; the compiler automatically recognizes whether precompilation is to be performed - no additional control is required for this.

(4) The repository data is stored as an X element with the name PKL$PCL (see the section “COMPILER-ACTION option”).

Example 2-10

Usage of a parameterized class

Source code

```
PROGRAM-ID. n.
...
REPOSITORY.
    CLASS pkl
    CLASS exp EXPANDS pkl USING ap ————————————————————— (1)
    CLASS ap.
...
01 obj-exp USAGE OBJECT REFERENCE exp.
...
```

Assignment and call to compiler

```
/ADD-FILE-LINK REPLIB,REPOSITORY ——————————————————————— (2)
/START-COBOL2000-COMPILER ... ————————————————————————— (3)
```

(1) The name of the expansion of the parameterized class is `exp`, the name of the current parameter is `ap`.

(2) The repository data of the precompiled parameterized class `pkl` and of the (non-parameterized) class `ap` is expected in the library REPOSITORY.

(3) Compilation of `n` is followed automatically by the compilation of the concrete expansion `exp` of the parameterized class `pkl`.

> In the event of dependencies between various expansions (example: the expansion of a parameterized class is used as the current parameter for another expansion), the input repository must also be assigned as output library and the UPDATE-REPOSITORY=*YES option must be set when the user is compiled.

Example 2-11

Expansion of a parameterized class

```
Source code temporary created  ———————————————————————————— (1)
CLASS-ID. exp USING ap.  ——————————————————————————————— (2)(3)
...
REPOSITORY.
    CLASS ap.  ————————————————————————————————————————— (3)
...
01 obj-fp USAGE OBJECT REFERENCE ap.  ———————————————————— (3)
01 obj-pkl USAGE OBJECT REFERENCE exp.  ——————————————————— (2)
...
```

(1) Generation and compilation take place automatically. The user does not need to enter any additional commands or statements for this purpose.

(2) The name of the parameterized class is replaced in all places by the name `exp` of the concrete expansion.

(3) The name of the formal parameter is replaced in all places by the name `ap` of the current parameter.

> The compiler does not have to perform the subsequent expansions in the order in which they have been written in the program, but in such a way that the data required for the current parameters is available before the expansion takes place.

3 Controlling the compiler via SDF

The COBOL2000 compiler can be controlled via SDF (**S**ystem **D**ialog **F**acility).

The principal ways of working with SDF are described in the following sections. For a detailed description of the SDF dialog interface, refer to the manuals “Introductory Guide to the SDF Dialog Interface” [5] and “Commands” [3].

3.1 Calling the compiler and entering options

In interactive mode SDF offers the following ways of entering options:

- input from the display terminal without user guidance, referred to below as “expert mode”.
- input from the display terminal with three different levels of user guidance, referred to below as “menu mode”.

3.1.1 SDF expert mode

SDF expert mode is preset as the default mode following the LOGON command. In this mode the user starts the compilation run as follows:

```
/START-COBOL2000-COMPILER options

guaranteed abbreviation: START-COBOL2-COMP options
```

The compilation is started immediately after input of the command.

If no options are specified, the compiler will read the compilation unit from SYSDTA, so the file or library element containing the compilation unit (see section “Assigning the compilation unit with the ASSIGN-SYSDTA command”) must be assigned to SYSDTA before the compiler is called.

The following general rules apply when entering options in expert mode:

- All options, parameters and operand values must be separated from one another by commas.
- If there is not enough room to enter all the options in one line
  - continuation lines can be generated by entering a hyphen (“-”) after the last character in a line,
  - or all the options can be written continuously (i.e. without regard for the end of the line).

Options may be specified as keyword operands or as positional operands.

- Keyword operands

  The keywords must be specified in the correct format; they can, however, be abbreviated as desired provided they remain unique within their respective SDF environment. Illegal abbreviations and typing errors are reported as syntax errors and can be corrected immediately.

  > ¡ It is generally advisable to avoid using abbreviations (especially in procedures), since the possible abbreviations could change in future SDF versions.

- Positional operands

  The operand keywords (i.e. the keywords to the left of the equal sign in the format) and the equal sign itself can be omitted provided the predetermined order of the operands, and their values, are strictly adhered to. All operands that are not specified because their default value is to be used must be indicated by the comma separator “,”. If there are further possible options after the last option to be specified explicitly, their position does not have to be indicated by separators.

  Options should not be specified as positional operands in procedures.

3.1.2 SDF menu mode

There are two ways of using the SDF menu mode to control the compiler:

Permanent menu mode

The user switches to the SDF main menu by entering the SDF command:
/MODIFY-SDF-OPTIONS GUIDANCE = MAXIMUM / MEDIUM / MINIMUM
The available commands for calling the compiler are given there under the entry “PROGRAMMING-SUPPORT”. Specifying the associated number in the input line calls up the PROGRAMMING-SUPPORT menu. The compiler can then be called from this menu by entering the command number.

The values of the MODIFY-SDF-OPTIONS commands have the following meanings:

MAXIMUM	Maximum help level, i.e. all operand values with options, help texts for commands and operands.
MEDIUM	All operand values without options; help texts for commands only.
MINIMUM	Minimum help level, i.e. only default values for the operands; no options, no help texts.

The permanent menu mode remains active until the user explicitly switches back to expert mode by entering the command:
MODIFY-SDF-OPTION GUIDANCE=EXPERT

Temporary menu mode

There are two ways of controlling the compiler in the temporary menu mode:

1. By moving from the SDF menu to the operand form in steps

   When the user enters a question mark at the system level, the SDF main menu is displayed.

   ```
   /?
        The SDF main menu appears
     User specifies the number of the PROGRAMMING-SUPPORT menu
         The PROGRAMMING-SUPPORT menu appears
     User specifies the number of the command to call the compiler
         The operand form appears
   ```

2. By directly switching to the operand form

   A question mark is appended immediately after START-COBOL2000-COMPILER.

   ```
   /START-COBOL2000-COMPILER? [options]
           Switch to operand form
   ```

   Entering START-COBOL2000-COMPILER? causes control to switch to menu mode, and the first page of the operand form is opened.
   The form may contain the operand values of options that were specified immediately after START-COBOL2000-COMPILER?.

   The user can immediately return to expert mode from any menu by specifying *CANCEL in the NEXT line or by pressing the K1 key.

   After the compilation has terminated, the user is back in expert mode (indicated by /).

Notes on processing the operand form

The operand form is largely self-explanatory. During processing the main thing to note is that only the entry in the input line (“NEXT:...”) determines which operation will be executed. The permitted inputs are listed below this line.

The most important control characters for processing the operand form are summarized below.
A detailed description of the best way to use SDF is given in the manual “Introductory Guide to the SDF Dialog Interface” [5].

Control characters for processing the operand form

?	as an operand value provides a help text and indicates the value range for this operand. If SDF produced the message “CORRECT INCORRECT OPERANDS” after a previous invalid input, the question mark supplies additional detailed error messages. The remainder of the line does not have to be deleted.
!	as an operand value reinserts the default value for this operand if the displayed default value was previously overwritten. The remainder of the line does not have to be deleted.
<operand>(	An open parenthesis after a structure-initiating operand produces the sub-form for the associated structure. Operands specified after the open parenthesis are displayed in the sub-form.
–	as the last character in an input line causes a continuation line to be output (up to 9 continuation lines are possible per operand).
Line (LZF) key	deletes all characters in the input line from the cursor position.

3.2 SDF syntax description

The metasyntax used in the option formats is explained in the following tables.

Table 5: Metacharacters

The option formats make use of certain symbols and notational conventions whose meaning is explained in the following table.

Symbol	Meaning	Example
UPPERCASE LETTERS	Uppercase letters indicate keywords. Some keywords are prefixed with *	LISTING = STD SOURCE = *SYSDTA
=	The equal sign links an operand name with its associated operand values.	LINE-SIZE = 132
< >	Angle brackets indicate variables whose range of values is described by data types and suffixes (see table 6 and table 7).	... = <integer 1..100>
underscoring	Underscoring is used to indicate the default value of an operand.	MODULE-LIBRARY = *OMF
/	A slash separates alternative operand values.	SHAREABLE-CODE = NO / YES
(...)	Parentheses indicate operand values which introduce a structure.	TEST-SUPPORT = AID (...)
indentation | |	Indentation indicates dependence on a higher-ranking operand. The vertical bar indicates related operands belonging to the same structure. It extends from the start to the end of the structure. A structure may contain additional structures within itself. The number of vertical bars preceding an operand corresponds to the structure depth.	LISTING = PARAMETERS(...) PARAMETERS(...) | SOURCE = YES (...) | YES(...) | | COPY-EXP... | | . | | .
,	A comma precedes further operands on the same structure level.	,SHARABLE-CODE = ,ENABLE-INITIAL-STATE=

Table 5: Metacharacters

Table 6: Data types

Variable operand values are represented in SDF by data types. Each data type represents a specific set of values. The number of data types is limited to those described in table 6.

The description of the data types is valid for all options. Therefore only deviations from table 6 are described in the relevant operand descriptions.

Data type	Character set	Special rules
alphanum-name	A...Z 0...9 $,#,@	
composed-name	A...Z 0...9 $,#,@ hyphen period	Alphanumeric string that may be delimited by periods or commas into several substrings
c-string	EBCDIC characters	A string of EBCDIC characters in single quotes, optionally with the letter C prefixed.

filename	A...Z
	0...9
	$,#,@
	hyphen
	period
		Input format:
		`cat:$user. {  file`
		`  | file(no)`
		`  | group`
		`  | group{(*abs) | (+rel) | (-rel)}`
		`}`
		:cat:
		optional entry of the catalog identifier; character set limited to A....Z and 0....9; maximum of 4 characters; must be enclosed in colons; default value is the catalog identifier assigned to the user ID, as specified in the JOIN entry.
		$user.
		optional entry of the user ID; character set restricted to A...Z and 0...9; maximum of 8 characters; $ and period are mandatory; default value is the user' s own ID.
		$. (special case)
		system default ID
		file
		file or job variable name; last character must not be a hyphen or period; a maximum of 41 characters; must contain at least A...Z.
		#file (special case)
		@file (special case)
		# or @ used as the first character identifies temporary files or job variables, depending on system generation.
		file(no)
		tape file name
		no: version number;
		character set is A...Z, 0...9, $, #, @.
		Parentheses must be specified.

		group name of a file generation group (character set: as for “file”) `group{(*abs) | (+rel) | (-rel)}` (*abs) relative generation number (0-99); positive or negative signs and parentheses must be specified. (+rel) (-rel) relative generation number (0-99); positive or negative signs and parentheses must be specified.
integer	0...9,+, -	+ or -, if specified, must be the first character.

Table 6: Data types

Table 7: Suffixes for data types

Data-type suffixes define additional rules for data-type input. They can be used to limit or extend the set of values. This manual makes use of the following short codes to represent data-type suffixes:

generation	gen
cat-id	cat
user-id	user
version	vers

The description of the data-type suffixes is valid for all options and operands. Therefore only deviations from table 7 are described in the relevant operand descriptions.

Suffix	Meaning
x..y	Length specification x Minimum length for the operand value; x is an integer. y Maximum length for the operand value; y is an integer. x=y The length of the operand value must be x exactly.
with-low	Lowercase letters accepted
without	Restricts the specification options for a data type.
-gen	A file generation or file generation group may not be specified.
-vers	The version (see file(no)) may not be specified for tap files.
-cat	A catalog ID may not be specified.
-user	A user ID may not be specified.

Table 7: Suffixes for data types

3.3 SDF options for controlling the compiler run

Name of the option	Purpose
SOURCE	Defines the input source of the compilation group
SOURCE-PROPERTIES	Defines certain properties of the compilation group
ACTIVATE-FLAGGING	Flags specific language elements in the error listing with a message of class F
COMPILER-ACTION	Partial execution of the compiler run; determines some attributes of the generated code and the module format (object module, LLM)
MODULE-OUTPUT	Specifies the name and output destination for object modules or LLMs
LISTING	Specifies the type of listings to be output, the layout of these listings, and where they are to be written
TEST-SUPPORT*	Determines if information for debugging with AID is generated
OPTIMIZATION	Activates/deactivates optimization for the compiler
RUNTIME-CHECKS	Activates check routines of the runtime system
COMPILER-TERMINATION	Defines the number of errors at which the compiler run is to be terminated
MONJV	Initializes a job variable to monitor the compiler run
RUNTIME-OPTIONS	Defines some of the runtime characteristics of the program
VERSION	Selects the compiler via its version number

Table 8: Overview: Options to control the compiler

* This option is not available in COBOL2000-BC

3.3.1 SOURCE option

The parameters for this option determine whether the compilation unit will be read from SYSDTA, from a cataloged BS2000 file, from a PLAM library or from a POSIX file.

Format

```
SOURCE = *SYSDTA / <filename 1..54> / <c-string 1..1024 with-low> / *LIBRARY-ELEMENT(...)
  *LIBRARY-ELEMENT(...)
    |  LIBRARY = <filename 1..54>
    |  ,ELEMENT = <composed-name 1..40>(...)
    |     <composed-name>(...)
    |        |  VERSION = *HIGHEST-EXISTING / *UPPER-LIMIT / <composed-name 1..24>
```

SOURCE = *SYSDTA
The compilation group is read from the SYSDTA system file. In interactive mode, this is assigned to the terminal by default. If SYSDTA was assigned to the compilation unit file with the ASSIGN-SYSDTA command before starting the compilation run, the SOURCE option may be omitted.

SOURCE = <filename 1..54>
The <filename> parameter is used to assign a cataloged file. After compilation there is a TFT entry for the link name SRCFILE, which is linked with the file name <filename>. The file must be “SYSDTA-compatible”, i.e. an ASSIGN-SYSDTA command for this file must run error-free.

SOURCE = <c-string 1..1024 with-low>
If the POSIX subsystem is available, this parameter can be used to request a source file from the POSIX file system. <c-string> defines the name of the POSIX file. If <c-string> does not include a directory name, the compiler will look for the source file under the specified file name in the home directory of the current BS2000 user ID. If the file is in any other directory, <c-string> must include the absolute path name.

This operand is not available in COBOL-BC.

SOURCE = *LIBRARY-ELEMENT(...)
This parameter specifies a PLAM library and an element (member) held in that library.

LIBRARY = <filename 1..54>
Name of the PLAM library in which the compilation group is stored as an element. After compilation there is a TFT entry for the link name SRCLIB, which is linked with the file name <filename> of the PLAM library.

ELEMENT = <composed-name 1..40>(...)
Name of the library element in which the compilation group is stored.

VERSION = *HIGHEST-EXISTING / *UPPER-LIMIT / <composed-name 1..24>
Version designation of the library element. If no version or *HIGHEST-EXISTING is specified, the compiler reads the version of the element with the highest version designation present in the library. If *UPPER-LIMIT is specified, the compiler reads the version of the element with the highest possible version number (indicated by LMS with “@”).

3.3.2 SOURCE-PROPERTIES option

This option defines certain properties of the compilation group.

Format

```
SOURCE-PROPERTIES = *STD / *PARAMETERS(...)
    *PARAMETERS(...)
        |   RETURN-CODE = *FROM-COBOL-SUBPROGRAMS / *FROM-ALL-SUBPROGRAMS
        |   ,ENABLE-KEYWORDS=*COBOL85 / *STD(...)
        |       *STD(...)
        |           |   XML-SUPPORT = *YES / *NO
        |   ,STANDARD-DEVIATION=*YES / *NO
        |   ,LINE-SEQUENTIAL=*STD / *SEQUENTIAL
        |   ,XML-NAMES=*KEEP / *UPPER
```

SOURCE-PROPERTIES = *STD
The default value of the following PARAMETERS structure is accepted.

SOURCE-PROPERTIES = *PARAMETERS(...)

RETURN-CODE = *FROM-COBOL-SUBPROGRAMS
The special register RETURN-CODE is used for information exchange between the COBOL programs in a compilation unit.

RETURN-CODE = *FROM-ALL-SUBPROGRAMS
The special register RETURN-CODE is also for accepting the function value from a subprogram (register 1).

ENABLE-KEYWORDS = *COBOL85 / *STD
If COBOL85 is specified, the additional keywords reserved by the COBOL2000 compiler (as opposed to those in COBOL85) are not recognized as such and can therefore be used as file names.

XML-SUPPORT = *YES / *NO
If YES is specified, the keywords of the new language elements are recognized for XML processing; these language elements are compiled and the special registers are available.
If NO is specified, the keywords of the new language elements are not reserved for XML processing, and these language elements are not recognized.

STANDARD-DEVIATION = *YES / *NO
If YES is specified, the compiler accepts certain deviations from the rules set out in the COBOL standard:

- The address of another section or the contents of a pointer can be assigned to the data descriptions in the Linkage Section (level number 01 or 77) by means of a SET statement even **without** any BASED specification. No check is then performed to determine whether each used parameter has also been specified in the USING clause of the Procedure Division.
- Data structures containing pointer data items or universal object references are also permitted as receiving items.

- The following may also be redefined
  - Data structures containing pointer data items or universal object references, or
  - Pointer data items or universal object references with level number 01 or 77.
- Data structures containing pointer data items or universal object references may also be subject to reference modification.

> If YES is specified then the user is entirely responsible for ensuring that data structures are correctly aligned (at word or double word boundaries).

LINE-SEQUENTIAL = *STD / *SEQUENTIAL
If STD is specified, the compiler handles sequential files in compliance with their declarations.
If SEQUENTIAL is specified, the compiler ignores the keyword LINE in SELECT clauses and handles LINE SEQUENTIAL files as if they are with SEQUENTIAL organization.

XML-NAMES = *KEEP / *UPPER
If KEEP is specified, the statement XML GENERATE transfers data item names from the source code to the XML element names without any changes.
If UPPER is specified, the XML element names are additionally converted in upper case.

3.3.3 ACTIVATE-FLAGGING option

This option causes the compiler to flag certain language elements according to ANS85 or according to the “Federal Information Processing Standard” (FIPS) with a class F message in the diagnostic listing.

Format

```
ACTIVATE-FLAGGING = *NO / *ANS85
```

ACTIVATE-FLAGGING = *NO
No language elements are flagged in the diagnostic listing.

ACTIVATE-FLAGGING = *ANS85
When ANS85 is specified, obsolete language elements and also any non-standard language extensions are flagged with a class F message (severity code F) in the diagnostic listing.

The following flags are used in the message texts:

“obsolete”	for obsolete language elements
“nonconforming nonstandard”	for all language extensions (additions to ANS85)

3.3.4 COMPILER-ACTION option

This option specifies a point in the compilation after which the compiler run is to be terminated. If a module is to be generated, this option can also be used to define its format and attributes.

Format

```
COMPILER-ACTION = *PRINT-MESSAGE-LIST / *SYNTAX-CHECK / *SEMANTIC-CHECK / *MODULE-GENERATION(...)
    *MODULE-GENERATION(...)
        |     ,SHAREABLE-CODE = *NO / *YES
        |     ,ENABLE-INITIAL-STATE = *NO / *YES
        |     ,MODULE-FORMAT = *OM / *LLM (...)
        |          LLM (...)
        |               |       ALIGNMENT = *PAGE / *DOUBLE-WORD
        |     ,SUPPRESS-GENERATION = *NO / *AT-SEVERE-ERROR
        |     ,SEGMENTATION = *ELABORATE / *IGNORE
        |     ,UPDATE-REPOSITORY = *NO / *YES
        |     ,CALL-CONVENTION = *COBOL / *COMPATIBLE
        |     ,OPTION-DIRECTIVES = *KEEP / *IGNORE
```

COMPILER-ACTION = *PRINT-MESSAGE-LIST
The compiler prints a list of all possible error messages. No compilation takes place. This operand is not available in COBOL-BC.

COMPILER-ACTION = *SYNTAX-CHECK
The compiler only checks the compilation units for syntax errors.

COMPILER-ACTION = *SEMANTIC-CHECK
The compiler runs a syntax check on the compilation units and also verifies that they comply with the semantic rules. Since no module is to be generated, only a source listing and diagnostic listing can be requested.

COMPILER-ACTION = *MODULE-GENERATION(...)
A complete compilation run is to be performed and - unless explicitly suppressed - object modules are to be generated.

SHAREABLE-CODE = *NO / YES
If YES is specified, the compiler writes the code of the PROCEDURE DIVISION (without DECLARATIVES) into a shareable code module (see section “Shareable COBOL programs”).
For the name convention see table 2 in "Output of modules".
Any segmentation of the PROCEDURE DIVISION is ignored.

ENABLE-INITIAL-STATE = *NO / *YES
If YES is specified, the compiler sets up areas for initialization. If NO is specified, programs to which a CANCEL statement refers or that contain the INITIAL clause or INITIALIZE statements with VALUE specification do not run as standard.

MODULE-FORMAT = *OM / *LLM (...)
The following specifications are ignored if the module is written to the POSIX file system (see MODULE-OUTPUT = <c-string...>).
OM: To enable further processing with BINDER, TSOSLNK, or DBL, the module is to be generated in OM format (object module format).
Maximum length for external names: 8 characters.
LLM: To enable further processing with BINDER or DBL, the module is to be generated in LLM format (link-and-load module format).
Maximum length for external names: 30 characters.

> When classes and interfaces are compiled, the *LLM format should always be selected. Classes or interfaces that inherit from each other **must** all be available in the same module format.

ALIGNMENT = *PAGE / *DOUBLE-WORD
If PAGE is specified then the CSECTS have the PAGE attribute in the generated module and are therefore aligned at the boundary.
If DOUBLE-WORD is specified then the CSECTS are only aligned at double word boundaries.

SUPPRESS-GENERATION = *NO / *AT-SEVERE-ERROR
AT-SEVERE-ERROR can be specified to suppress the generation of the module and the expansion of the parameterized classes/interfaces used if an error with a severity code >= 2 occurs during compilation.
SUPPRESS-GENERATION = *AT-SEVERE-ERROR also results in the operand SUPPRESS-GENERATION = *AT-SEVERE-ERROR in the LISTING option. This also prevents the object, address and cross-reference lists from being output.

SEGMENTATION=*ELABORATE / *IGNORE
ELABORATE: permits segmentation. If the program contains nested programs and non-fixed segments (segment number greater than or equal to segment limit), the compilation is aborted and a message is output. Otherwise, only segmentation-related language elements are rejected with appropriate warnings.
If `SEGMENTATION = ELABORATE` is specified together with `SHAREABLE-CODE = YES` or `MODULE-FORMAT = LLM`, it is rejected with an error message.
IGNORE: ignores segmentation-related language elements (`SEGMENT-LIMIT` clause, segment numbers in section header). When they occur, they are indicated with appropriate warnings.

UPDATE-REPOSITORY = *NO / *YES
If YES is specified, the compiler places the external interface of the compilation units in the external repository assigned with the link name REPOUT. If a corresponding interface already exists in the repository, **no** check is performed to determine whether any changes in the interface have occurred, i.e., the existing definition is blindly overwritten with the new one. If no link with the name REPOUT exists, the library SYS.PROG.LIB is used.
This output always occurs and cannot be suppressed with SUPRESS-GENERATION. Repository data is stored as an element of type X. To enable a differentiation, classes are assigned the suffix $CLS, interfaces the suffix $IFC, parameterized classes the suffix $PCL, parameterized interfaces the suffix $PIF and programs or program prototypes the suffix $PRO.

CALL-CONVENTION = *COBOL / *COMPATIBLE

When COBOL is specified, the value COBOL is set for the >>CALL-CONVENTION directive.
When COMPATIBLE is specified, COMPATIBLE is assumed as the default value for the >>CALL-CONVENTION directive.

OPTION-DIRECTIVES = *KEEP / *IGNORE

When IGNORE is specified, all >>IMP directives in the source text which relate to compiler options (LISTING-OPTIONS, COMPILER-ACTION and RUNTIME-ERRORS) are ignored. The result of this is that the options set externally are effective regardless of the directives specified in the source text.
In the case of expansion of parameterized classes/interfaces, OPTION-DIRECTIVES=*KEEP is always assumed.

3.3.5 MODULE-OUTPUT option

This option enables the user to control the library the object module is to be stored in and the name it is to be stored under.

Format

```
MODULE-OUTPUT = *STD / *OMF / <c-string 1..1024 with-low> / *LIBRARY-ELEMENT(...)
  *LIBRARY-ELEMENT(...)
    | LIBRARY=<filename 1..54>
    | ,ELEMENT = *STD (...) / <composed-name 1..32>(...)
    |    *STD (...)
    |       | VERSION = *UPPER-LIMIT / *INCREMENT / *HIGHEST-EXISTING / <composed-name 1..24>
    |    <composed-name>(...)
    |       | VERSION = *UPPER-LIMIT / *INCREMENT / *HIGHEST-EXISTING / <composed-name 1..24>
```

MODULE-OUTPUT = *STD
An object module is placed in the temporary EAM file of the current task.A link-and-load module (LLM) is placed in a PLAM library with the standard name PLIB.COBOL.<prog-id-name>, using the program name as the element name, and *UPPER-LIMIT (i.e. the highest possible version number) as the version designation.

MODULE-OUTPUT = *OMF
An object module is written to the temporary EAM file. If *OMF is specified for a link-andload module (LLM), the compiler issues a class I (information) message, and the module is placed in the PLAM library PLIB.COBOL.<prog-id-name>.

MODULE-OUTPUT = <c-string 1..1024 with-low>
If the POSIX subsystem is available, you can use this parameter to output a module (LLMs only) to the POSIX file system as an object file.
If <c-string> does not include a directory name, the object file will be stored under the specified file name in the home directory of the current BS2000 user ID. If the object file is to be written to any other directory, <c-string> must include the absolute path name.When selecting a file name, note that object files cannot be further processed, i.e. linked, in the POSIX subsystem unless they have a name ending with the extension “.o”. The compiler does not do any name checking.

This operand is not available in COBOL-BC.

MODULE-OUTPUT = *LIBRARY-ELEMENT(...)
This parameter specifies the PLAM library (LIBRARY=) the module is to be stored in and the element name (ELEMENT=) it is to be stored under.

LIBRARY = <filename 1..54>(...)
Name of the PLAM library in which the module is to be placed. If the PLAM library does not exist, it is created automatically.

ELEMENT = *STD
The element name of the module is derived from the PROGRAM-ID name. The formation of standard element names is described in section “Output of modules”, in table 2).

VERSION =
Specifies the version designation

VERSION = *UPPER-LIMIT
If no version designation or *UPPER-LIMIT is specified, the element receives the highest possible version number (indicated by LMS with “@”).

VERSION = *INCREMENT
The element receives the version number of the highest existing version incremented by 1, provided that the highest existing version designation ends with a digit that can be incremented. Otherwise, the version designation cannot be incremented. In this case, *UPPER-LIMIT is assumed and an appropriate error message is output.

Example 3-1

Highest existing version	Version generated by *INCREMENT
ABC1	ABC2
ABC	@ and error message
ABC9	@ and error message
ABC09	ABC10
003	004
none	001

VERSION = *HIGHEST-EXISTING
The highest existing version in the library is overwritten.

VERSION = <composed-name 1..24>
The element receives the specified version designation. If the version designation is to be incrementable, at least the last character must be an incrementable digit (see Example 3-1).

ELEMENT = <composed-name 1..32>
The user may optionally specify a freely-selected name for link-and-load modules (LLMs).
If a compilation group is being compiled, this operand is ignored, and the element names of LLMs are derived from the respective PROGRAM-ID name (see section “Output of modules”, table 2)) instead.

VERSION = *UPPER-LIMIT / *INCREMENT / *HIGHEST-EXISTING / <composed-name 1..24>
Version designation (see the above description of the VERSION operand for object modules).
When a compilation group is compiled, each element is assigned the same version designation.

3.3.6 LISTING option

The parameters of this option control which listings the compiler is to generate, their layout, and where they are to be output. Only one options listing is generated per compilation group. The other listings are created individually for each compilation unit.

Format

```
LISTING = *NONE / *STD / *PARAMETERS(...)
    *PARAMETERS(...)
      | OPTIONS = *NO / *YES
      | ,SOURCE = *NO / *YES(...)
      |     *YES(...)
      |       | COPY-EXPANSION = *NO / *VISIBLE-COPIES / *ALL-COPIES
      |       | ,SUBSCHEMA-EXPANSION = *NO / *YES
      |       | ,INSERT-ERROR-MSG = *NO / *YES
      |       | ,CROSS-REFERENCE = *NO / *YES
      |       |     *YES(...)
      |       |       | STMT-ADDRESS = *NO / *FIRST
      | ,DIAGNOSTICS = *NO / *YES(...)
      |     *YES(...)
      |       | MINIMAL-WEIGHT = *NOTE / *WARNING / *ERROR / *SEVERE-ERROR / *FATAL-ERROR
      |       | ,IMPLICIT-SCOPE-END = *STD / *REPORTED
      |       | ,MARK-NEW-KEYWORDS = *NO / *YES
      |       | ,REPORT-2-DIGIT-YEAR = *ACCEPT-STMT / *NO
      | ,NAME-INFORMATION = *NO / *YES(...)
      |     *YES(...)
      |       | SORTING-ORDER = *ALPHABETIC / *BY-DEFINITION
      |       | ,CROSS-REFERENCE = *NONE / *REFERENCED / *ALL
      |       | ,SUPPRESS-GENERATION = *NO / *AT-SEVERE-ERROR
      | ,LAYOUT = *STD / *PARAMETERS(...)
      |     PARAMETERS(...)
      |       | LINES-PER-PAGE = 64 / <integer 20..128>
      |       | ,LINE-SIZE = 132 / <integer 119..172>
      | ,OUTPUT = *SYSLST / *STD-FILES / *LIBRARY-ELEMENT(...)
      |     *LIBRARY-ELEMENT(...)
      |       | LIBRARY = <filename 1..54>
```

LISTING = *NONE
The compiler is to generate no listings.

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

LISTING = *PARAMETERS(...)
The following parameters determine which listings are to be generated, their layout, and the output destination to which they are to be directed.

OPTIONS = *NO / *YES
By default, the compiler generates a listing specifying the control statements that apply during compilation, the environment of the compilation process and some information for maintenance and diagnostic purposes.

SOURCE = *YES(...)
The compiler generates a source listing and a library listing.

COPY-EXPANSION = *NO
The COPY elements copied into the compilation unit will not be printed in the source listing. This setting is recommended for frequently occurring COPY elements, in order to save paper.

COPY-EXPANSION = *VISIBLE-COPIES
Only COPY elements containing no SUPPRESS entry will be printed in the source listing for the compilation unit. Each line of a COPY element is identified by a “C” in column 1.

COPY-EXPANSION = *ALL-COPIES
All COPY elements will be printed in the source listing for the compilation unit, including those that contain a SUPPRESS entry. Each line of a COPY element is identified by a “C” in column 1.

SUBSCHEMA-EXPANSION = *NO / *YES
If YES is specified, the SUB-SCHEMA SECTION will be listed and each line will be identified by a “D” in column 1.
This operand is not available in COBOL-BC.

INSERT-ERROR-MSG = *NO / *YES
If YES is specified, any (error) messages that occur during compilation are “merged” with the source listing for the compilation unit. The message line always appears immediately after the source line in which the construct responsible for triggering the message begins. Messages that cannot be allocated to a particular source line by the compiler are output after the last source line.
The operand also works if no error listing has been requested.
For merging to function correctly, the source listing for the compilation unit should not contain more than 65535 source lines (see for the for the compilation unit source listing in the section “Source listing for a compilation unit”).

CROSS-REFERENCE = *YES(...)
If YES is specified, the address and length of the definitions contained in the line follow in the source listing on the right next to the source lines. In the case of definitions, there are cross-references to the users, including the usage type, and in the case of the users, references back to the definition.

The operand has no effect if the compilation unit comprises more than 65535 lines.

When using this operand, it is advisable to increase the line length (see the LAYOUT operand) and then to use a corresponding character set or wider paper to print out the listing (see 3-5).

In the case of lines that are not listed in the source listing, the additional specifications generated by the operand (see COPY-EXPANSION, SUBSCHEMA-EXPANSION and LISTING directive) are not included. References from listed lines to suppressed lines are retained.

STMT-ADDRESS = *NO / *FIRST
If FIRST is specified, for the first statement in a line the source listing contains, on the right next to the source lines from the Procedure Division, the address of the first machine instruction generated for this.

DIAGNOSTICS = *YES(...)
The compiler is to generate a diagnostic listing.

MINIMAL-WEIGHT = *NOTE / *WARNING / *ERROR / *SEVERE-ERROR / *FATAL-ERROR
The diagnostic listing will contain no messages with a weighting less than the specified value. The default value NOTE causes all (error) messages that occurred during the compilation to be listed.

IMPLICIT-SCOPE-END = *STD / *REPORTED
If REPORTED is specified, a remark message is added to the diagnostic listing each time a structured statement is ended by a period.

MARK-NEW-KEYWORDS = *NO / *YES
If YES is specified, keywords from the future standard will be marked in the diagnostic listing with a message with severity code I. A value of YES is only meaningful if *COBOL85 has been specified for ENABLE-KEYWORDS.

REPORT-2-DIGIT-YEAR = *ACCEPT-STMT / *NO
If *ACCEPT-STMT is specified, the compiler indicates that the year numbers are processed without century digits for every ACCEPT statement and for every variable accessed in the statement. MINIMAL-WEIGHT should be set to NOTE. If *NO is specified, these indications are suppressed.

NAME-INFORMATION = *NO / *YES(...)
If YES is specified, the compiler will generate a locator map or a locator map and cross-reference listing. The listing contains data, section and paragraph names.

SORTING-ORDER = *ALPHABETIC
The symbolic names are to be listed in ascending alphabetical order.

SORTING-ORDER = *BY-DEFINITION
The symbolic names are to be listed in the order in which they are defined in the compilation unit.

CROSS-REFERENCE = *NONE
No cross-reference listing will be generated.

CROSS-REFERENCE = *REFERENCED
Only the data and procedure names that are actually addressed in the program will be listed in the cross-reference listing.

CROSS-REFERENCE = *ALL
A cross-reference listing containing all data and procedure names will be generated.

SUPPRESS-GENERATION = *NO / *AT-SEVERE-ERROR
AT-SEVERE-ERROR can be specified to suppress the generation of the module if an error with a severity code >= 2 occurs during compilation.

LAYOUT = *STD
The layout of the generated listings is to correspond to the default settings of the PARAMETERS structure.

LAYOUT = *PARAMETERS(...)
The following parameters can be used to modify the layout of the generated listings.

LINES-PER-PAGE = 64 / <integer 20..128>
This parameter can be used to define the maximum number of lines to be printed per page. A page throw will be performed when this line number is reached.

LINE-SIZE = 132 / <integer 119..172>
This parameter defines the maximum number of characters to be printed per line.

OUTPUT = *SYSLST
This causes the generated listings to be written into the temporary system file SYSLST, from where they will automatically be output on the printer at end of task (i.e. after LOGOFF). The first requested listing is preceded by a title page (COMOPT listing) with details concerning the system environment and a list of all the COMOPT operands in effect at compilation.

OUTPUT = *STD-FILES
This setting causes each requested listing to be placed in a separate cataloged file. The cataloged files created in this way have the default names given in the right-hand column of the following table. *program-name* is derived from the PROGRAM-ID name and may, if necessary, be abbreviated to 16 characters.

Listing	File name
control statement listing	OPTLST.COBOL.*program-name*
source listing or library listing	SRCLST.COBOL.*program-name*
diagnostic listing	ERRFIL.COBOL.*program-name*
locator map listing / cross-reference listing	LOCLST.COBOL.*program-name*

File names and file characteristics for these cataloged files are preset by default. However, the user can divert the output to other cataloged files. In order to do this, the desired characteristics must be defined in an ADD-FILE-LINK command before the compiler is called so that they can be linked with the respective file link name used by the compiler:

Listing	Link name
control statement listing	OPTLINK
source listing/library listing	SRCLINK
address listing/cross-reference listing	LOCLINK
diagnostic listing	ERRLINK

To store the generated listings in the POSIX file system, you must assign them to the POSIX file system using S variables. The default names of these variables are:

Listing	Name of S variable
control statement listing	SYSIOL-OPTLINK
source listing/library listing	SYSIOL-SRCLINK
locator map/cross-reference listing	SYSIOL-LOCLINK
diagnostic listing	SYSIOL-ERRLINK

OUTPUT = LIBRARY-ELEMENT(LIBRARY = <filename 1..54>)
The requested listings are output to the PLAM library specified by <filename>. Each listing occupies its own type R library element, which has the highest possible version number. The following standard names are assigned to these elements:

Listing	Element name
control statement list	OPTLST.COBOL.*program-name*
source listing/library listing	SRCLST.COBOL.*program-name*
address listing/cross-reference listing	LOCLST.COBOL.*program-name*
diagnostic listing	ERRLST.COBOL.*program-name*

program-name is derived from the PROGRAM-ID name and truncated to 16 characters is required. If the truncation causes the program name to end in a '-' character, the '-' is replaced by a '#' character. After the compilation, there is a TFT entry for the link name LIBLINK, which is linked with the <filename> of the PLAM library.

Example 3-2

Writing listings to cataloged files

The compiler is to generate only a diagnostic (error) listing and save this in the cataloged file ERRORS.

```
/ADD-FILE-LINK ERRLINK,ERRORS ———————————————————————— (1)
/START-COBOL2000-COMPILER? ———————————————————————————— (2)

Entry in operand form:
LISTING=PAR(OPTIONS=NO,SOURCE=NO) ————————————————————— (3)
```

(1) The ADD-FILE-LINK command assigns the cataloged file ERRORS to the default link name ERRLINK.

(2) The compiler is called in menu mode.

(3) The default setting (generation of options, source and diagnostic listings) is changed; the compiler is to generate only a diagnostic listing and output it by default to the cataloged file ERRORS.

Example 3-3

Writing listings to a PLAM library

The compiler is to generate all listings and save them as elements in the PLAM library LISTLIB.

```
/START-COBOL2000-COMPILER? ————————————————————————————————————(1)

Entry in operand form:
LISTING=PAR(NAME-INFORMATION=YES(CROSS-REFERENCE=ALL),-
OUTPUT=*LIBRARY-ELEMENT(LIBRARY=LISTLIB)) ————————————————————————(2)
```

(1) The compiler is called in menu mode.

(2) The default setting (generation of options, source and diagnostic listings) is changed; the compiler is also to generate a locator map and cross-reference listing and save all listings in a PLAM library named LISTLIB.

Example 3-4

Writing listings to the POSIX file system

The compiler is to generate a source listing and a diagnostic listing and store them in the POSIX file system.

```
/DECL-VAR SYSIOL-SRCLINK,INIT='*P(xpl.srclst)',SCOPE=*TASK————————— (1)
/DECL-VAR SYSIOL-ERRLINK,INIT='*P(xpl.errlst)',SCOPE=*TASK————————— (1)
/START-COBOL2000-COMPILER? ————————————————————————————————————— (2)
```

(1) The DECL-VARIABLE command assigns the desired name to the variable. Since the file name does not include a path specification, the file will be stored in the home directory.

(2) The compiler is called in SDF menu mode.

Example 3-5

Writing a compressed listing for utilizing print pages

A compressed listing is to be generated to utilize the print pages as far as possible.

```
LISTING=*PAR(SOURCE=*YES(CROSS-REFERENCE=YES),-  ———————————————— (1)
LAYOUT=*PAR(LINES-PER-PAGE=60,LINE-SIZE=172))              *) ——— (1)
/PRINT-FILE srclst.cobol.programmname,LOOP=98,CHAR-SET=R01 ————— (2)
```

*) These specifications have been optimized for a page width of 32 cm and a page length of 22 cm.

(1) Options used

(2) Command for printing out the listing file

3.3.7 TEST-SUPPORT option

This option controls whether a program execution is to be monitored with the AID debugger. It can also determine certain characteristics of the AID debugger.

This option is not available in COBOL-BC.

Format

```
TEST-SUPPORT = *NONE / *AID(...)

    *AID(...)

        | STMT-REFERENCE = *LINE-NUMBER / *COLUMN-1-TO-6

        | ,PREPARE-FOR-JUMPS = *NO / *YES

        | ,VIRTUAL-NAMES = *NO / *YES
```

TEST-SUPPORT = *NONE
No debugging aid is requested. The compiler generates only ESD debugger information of the type compilation unit. This means that the module (or every module in the case of segmented programs) is assigned a symbolic name consisting of the first 8 characters of the name in the ID paragraph of the compilation unit. When debugging with AID, this name can be used to qualify the compilation unit.

TEST-SUPPORT = *AID(...)
This parameter is required if the program is to be monitored symbolically using AID. It causes the compiler to generate both LSD information and ESD debugger information, which means that symbolic names from the compilation unit can be used for debugging with AID (see description in “AID” manual [8]).
In segmented programs it is possible to generate LSD information - and thus create conditions for symbolic debugging with AID - only if the object module is written to a PLAM library.

STMT-REFERENCE = *LINE-NUMBER
The AID source references are formed using the line numbers generated by the compiler.

STMT-REFERENCE = *COLUMN-1-TO-6
The AID source references are created by means of the user-assigned sequence numbers in the compilation unit (columns 1 to 6).
Debugging with AID is useful only if the assigned sequence numbers are sorted in ascending numeric order.

PREPARE-FOR-JUMPS = *NO / *YES
YES must be specified if, during the AID debugging session,

- the AID command %JUMP is to be used (see “AID” manual [8] and section “Advanced Interactive Debugger (AID)”) or
- test points are to be set selectively on paragraphs or chapters, e.g. when debugging nested GO TO loops (as generated by the COLUMBUS preprocessor COLCOB) in which several paragraph headings follow one another in immediate succession or follow after a section heading.
- the AID command %TRACE is to be used to individually trace each COBOL statement (see “AID” manual [8]).

Use of this function increases the size of the object and lengthens the program run time.

VIRTUAL-NAMES = *NO / *YES

YES must be specified if, during the AID debugging session:

- FILLER-elements of structure are to be referenced.
- entries of SPECIAL-NAMES paragraph are to be displayed as elements of the virtual structure SPECIAL-NAMES.

3.3.8 OPTIMIZATION option

This option can be used to activate and deactivate the optimization actions of the compiler.

Format

```
OPTIMIZATION = *STD / *PARAMETERS(...)

    *PARAMETERS(...)

        |  CALL-IDENTIFIER = *STD / *OPTIMIZE
```

OPTIMIZATION = *STD
The default of the PARAMETERS structure applies.

OPTIMIZATION = *PARAMETERS(...)

CALL-IDENTIFIER = *STD / *OPTIMIZE
If *OPTIMIZE is specified, the optimization is activated. Multiple calls of the same subprogram by means of CALL identifier are processed without calling by system interfaces (possible for the first 100 subprograms called).

3.3.9 RUNTIME-CHECKS option

This option can be used to activate the check routines of the runtime system.

Format

```
RUNTIME-CHECKS = *NONE / *ALL / *PARAMETERS(...)

    *PARAMETERS(...)

        | TABLE-SUBSCRIPTS = *NO / *YES

        | ,FUNCTION-ARGUMENTS = *NO / *YES

        | ,PROC-ARGUMENT-NR = *NO / *YES

        | ,RECURSIVE-CALLS = *NO / *YES

        | ,REF-MODIFICATION = *NO / *YES
```

RUNTIME-CHECKS = *NONE
No check routines of the runtime system are requested.

RUNTIME-CHECKS = *ALL
All check routines of the runtime system that are named in the PARAMETERS structure are to be activated.

RUNTIME-CHECKS = *PARAMETERS(...)

TABLE-SUBSCRIPTS = *NO / *YES
If YES is specified, the runtime system checks that table bounds are kept to (both for subscripting and for indexing).
Checks are made to determine whether

- index values are greater than zero,
- index values are not greater than the number of elements in the corresponding dimensions,
- index values are not greater than associated values in DEPENDING ON items,
- values in DEPENDING ON items are within the bounds defined in corresponding OCCURS clauses.

The runtime system responds with message COB9144 or COB9145 and aborts the program in the event of an error if ERROR-REACTION = TERMINATION was specified in the RUNTIME-OPTIONS option.

FUNCTION-ARGUMENTS = *NO / *YES
If YES is specified, the value range, number, and length of function arguments are checked at runtime. If invalid values are detected, one of the messages COB9123, COB9125, COB9126 or COB9127 is issued; the program will abort if ERROR-REACTION = TERMINATION was specified in the RUNTIME-OPTIONS option.

PROC-ARGUMENT-NR = *NO / *YES
If YES is specified, a check is made when a COBOL subprogram is called to determine whether the number of parameters passed matches the number expected. If there is a discrepancy, message COB9132 is issued, and the program will abort if ERROR-REACTION = TERMINATION was specified in the RUNTIME-OPTIONS option.The check is only effective if the called program was compiled with this option and if the calling program was compiled with a compiler version >= 2.0.

RECURSIVE-CALLS = *NO / *YES

If YES is specified, a check will be made on the call hierarchy of a program run unit; that is, the runtime system uses a table to check whether a subprogram is being called recursively, i.e. is still active. If there is a recursive call and the CALL statement contains no ON EXCEPTION phrase, the program run is aborted with the error message COB9157.

Every program that contains a CALL identifier and/or CANCEL should be compiled using RECURSIVE-CALLS=YES.

This option is ignored for compilation units that are not programs and is rejected for programs with a RECURSIVE specification in the PROGRAM-ID (when YES is set).

REF-MODIFICATION = *NO / *YES

Specifying YES causes the runtime system to verify compliance with data-item limits for identifiers subject to reference modification. If data-item limits are not complied with, error message COB9140 is issued and the program is continued or aborted depending on the ERROR-REACTION parameter of the RUNTIME-OPTIONS option.

3.3.10 COMPILER-TERMINATION option

This option can be used to initiate a termination of the compilation run dependent on the number of errors that have occurred.

Format

```
COMPILER-TERMINATION = *STD / *PARAMETERS(...)

    *PARAMETERS(...)

        |  MAX-ERROR-NUMBER = *NONE / <integer 1..100>
```

COMPILER-TERMINATION = *STD
The default settings of the PARAMETERS structure are to apply.

COMPILER-TERMINATION = *PARAMETERS(...)

MAX-ERROR-NUMBER = *NONE / <integer 1..100>
An integer can be used to specify the number of errors allowed before the compilation run is terminated. The count begins from the error severity class specified in the MINIMAL-WEIGHT parameter of the LISTING option (default value: NOTE, see section “LISTING option”).

The specified error number can be exceeded because the compilation is terminated only after execution of a compiler segment has been completed (see “Appendix”).

3.3.11 MONJV option

This option can be used to create a job variable which will monitor the compiler run.

Format

```
MONJV = *NONE / <filename 1..54>
```

MONJV = *NONE / <filename 1..54>
The user uses <filename> to define a monitoring job variable. During the compilation run, the compiler will then store a code in the return code indicator of the job variable, giving information about any errors that occurred during the execution of the compiler.

3.3.12 RUNTIME-OPTIONS option

The parameters of this option control certain characteristics of the executable COBOL program.

Format

```
RUNTIME-OPTIONS = *STD / *PARAMETERS(...)

    *PARAMETERS(...)

        | ACCEPT-STMT-INPUT = *UNMODIFIED / *UPPERCASE-CONVERTED

        | ,FUNCTION-ERR-RETURN = *UNDEFINED / *STD-VALUE

        | ,SORTING-ORDER = *STD / *BY-DIN

        | ,ACCEPT-DISPLAY-ASSGN = *SYSIPT-AND-SYSLST / *TERMINAL

        | ,ERR-MSG-WITH-LINE-NR = *NO / *YES

        | ,ERROR-REACTION = *CONTINUATION / *TERMINATION

        | ,ENABLE-UFS-ACCESS = *NO / *YES

        | ,EXTRA-ALTERNATE-KEYS = *IGNORE / *STD

        | ,XML-LINE-FEED = *INSERTED / *IGNORED
```

RUNTIME-OPTIONS = *STD
The preset default values of the PARAMETERS structure will be used.

RUNTIME-OPTIONS = *PARAMETERS(...)

ACCEPT-STMT-INPUT = *UNMODIFIED / *UPPERCASE-CONVERTED
If UPPERCASE-CONVERTED is specified, letters entered in lowercase in an ACCEPT statement will be converted to uppercase if the input is typed in from the terminal.

FUNCTION-ERR-RETURN = *UNDEFINED / *STD-VALUE
If STD-VALUE is specified, the value range, number, and length of function arguments are checked at runtime. If invalid argument values are detected, the appropriate return code for the error is assigned to the function in which the error occurs.

SORTING-ORDER = *STD / *BY-DIN
Specifying BY-DIN causes the SORT utility routine to perform the sort according to the DIN standard for EBCDIC; that is,

- lowercase letters are equated to the corresponding uppercase letters
- the character
  “ä” or “Ä” is identified with “AE”,
  “ö” or “Ö” is identified with “OE”,
  “ü” or “Ü” is identified with “UE” and
  “ß” is identified with “SS”.
- digits are sorted before letters.

ERR-MSG-WITH-LINE-NR = *NO / *YES
If YES is specified, the message COB9102 is output instead of the COB9101 message and is supplemented by the compilation unit line number assigned by the compiler to the statement that was being executed when the message was output.

ACCEPT-DISPLAY-ASSGN = *SYSIPT-AND-SYSLST / *TERMINAL
Specifying *TERMINAL causes the system files SYSDTA and SYSOUT to be assigned instead of system files SYSIPT and SYSLST (defaults) for ACCEPT and DISPLAY statements without FROM and UPON phrases.

ERROR-REACTION = *CONTINUATION / *TERMINATION
By default (CONTINUATION), the program run will continue after the following messages are output:
COB9120 to COB9127, COB9131, COB9132, COB9134, COB9140, COB9144, COB9145 and COB9197.
If TERMINATION is specified, the aforementioned error conditions lead to abnormal program termination (see also section “Program termination”).

ENABLE-UFS-ACCESS = *NO / *YES
If YES is specified, the compiler generates an object

- that is capable of accessing the POSIX file system as a program
- that can be further processed (linked) in the POSIX subsystem.

The chapter “COBOL2000 and POSIX” describes how to access a file from the POSIX file system and the conditions to which file processing is subject.

This operand is not available in COBOL-BC.

EXTRA-ALTERNATE-KEYS = *IGNORE / *STD
A prerequisite for processing an indexed file with secondary keys is, by default (STD), an identical description of the secondary keys in the program and in the file’s catalog entry. Specifying IGNORE means that an indexed file with secondary keys can be processed in read mode (OPEN INPUT) even if more keys are described in the file’s catalog entry than in the program.

XML-LINE-FEED = *INSERTED / *IGNORED
If INSERTED is specified, the record structure of a **file** containing an XML document remains visible while the document is processed: record changes are forwarded to the XML parser in the form of an end-of-line character.
If IGNORED is specified, the end of a file record remains invisible for the parser: the document appears like a single record in the file.
This parameter has no effect for XML documents which are provided in the working memory.

> i This parameter refers only to the record structure of a file, but **not** to end-of-line characters which are contained **in** a file record.

3.3.13 VERSION option

This option enables you to use the version number to select the compiler with which the compilation group is to be compiled.

Format

VERSION = *STD / <product-version>

VERSION = *STD
The last COBOL2000 compiler installed in the system by means of IMON is called.

VERSION = <product-version>
Specify the version number to select the compiler required for compilation if multiple COBOL2000 compilers with different versions are installed in the system simultaneously using IMON.
product-version must be specified in the following format: mm.n[a[kk]]

mm	Main version 1..99
n	Revision version 0..9
a	Modification status of the interface (user interface) A..Z
kk	Correction status (source/object correction) 00..99

Detailed information about installing a compiler using IMON is provided in the IMON manual [34] and in the SOLIS delivery letter.

When the compiler is controlled via SDF, the SDF syntax file activated with the last COBOL2000 compiler installed determines the options that are available. To ensure that the latest options are available, you should install the COBOL2000 compiler with the highest version number last.

4 Controlling the compiler with COMOPT statements

The COBOL2000 compiler can also be controlled as usual via COMOPT statements. In this case it is invoked with the command

```
/START-PROGRAM [FROM-FILE =] $.COBOL2000
```

The input of the compilation unit, the output of listings and of the module, and the internal execution of the compilation run are controlled by means of options that the user specifies in one or more COMOPT statements. The options are read via SYSDTA after COBOL2000 is invoked.
There are three ways in which the user can enter compiler options:

- The COMOPT statement(s) can be entered directly by calling the compiler without first reassigning the system file SYSDTA with the ASSIGN-SYSDTA command. In this case, the compiler explicitly requests the entry of the options by entering an asterisk (*) in column 1.
- The user can write the COMOPT statement(s) into a file and issue them via the file. This file could be a compilation unit file (the options are entered before the compilation unit) or a separate file.
  The file used is assigned to the system file SYSDTA with an ASSIGN-SYSDTA command before the compiler is called.
- COMOPT statement(s) can be entered directly, and the END statement can be used to reassign SYSDTA to a file that contains further COMOPT statements before the compilation unit.

When no further control statements are encountered, the compiler immediately begins to read the program text. The compiler determines the location of the compilation unit via the END statement and continues reading at that point.

If invalid COMOPT or END statements are entered in a batch process, the compilation is aborted (with error message CBL9005).

Format of the COMOPT statement

```
{COMOPT | COBRUN} operand= {YES | NO | option | (option[,option]...)}
```

- Input lines for COMOPT statements can be up to 128 characters in length. For ISAM files, this includes the length of the record key. The standardized reference format for writing COBOL compilation units has no significance for the input of COMOPT statements.
- An operand consists of a keyword, followed by a sign of equality and one or more parameters. If several parameters can be specified in a single operand, these must be enclosed in parentheses.

If errors are detected during the processing of a COMOPT statement, all previously evaluated options from the same line remain in effect. As indicated in the error message, the rest of the operand line or the remaining part of the operand is then ignored. Error messages for operands are only output to SYSOUT. The COMOPT statements only apply to the compiler run for which they were specified.
If the same COMOPT statement is entered more than once, the last specified value applies.If conflicting COMOPT statements are entered, the statement specified last is applicable.

Format of the END statement

```
END {filename | libname(elementname)}
```

END filename or libname can be used to reassign SYSDTA to a file or a library element.

END (without any further qualification) indicates to the compiler that the input of COMOPT statements has ended and that the compilation can therefore be started.

4.1 Source data input under COMOPT control

Input to the compiler may consist of the following source data:

- Individual compilation units
- Program segments (COPY elements)
- COMOPT statements
- Repository data
- Current values for DEFINE directives

The compiler expects the source data from the system input file SYSDTA.

By default, SYSDTA is assigned to the terminal in interactive mode and to the SPOOLIN file or the ENTER file in batch mode.
If source data is to be entered directly, no input control operations are required. The compiler is simply called, and the control statements and compilation units are entered directly.
However, if the source data is to come from a cataloged file or a library, the input file must be explicitly assigned to SYSDTA. Separate control statements are available to control the input of COPY elements. The assignments to be made with the ASSIGN-SYSDTA command and the input of COPY elements are described in section “Source data input”. The procedure for supplying values for compiler directives is described in section “Assignment to compiler variables to control source text manipulation”.

4.1.1 Assigning the compilation unit with the END statement

The input of compilation units and control statements can also be achieved without using the ASSIGN-SYSDTA command. After invocation, the compiler expects input from the terminal via SYSDTA. When the asterisk appears in the first column, the user can enter source code or compiler options. All entered characters that do not represent a valid COMOPT control statement are interpreted as source code by the compiler.
The END statement can be used to assign a cataloged file or a library element. If a file or library element is specified with it, the END statement can also be the first statement to be issued after the compiler is called. Further COMOPT statements may be included at the start of the assigned file.

> ¡ If the END statement is used to assign a library element, the name of the compilation unit cannot be correctly mapped in the compiler listings and at the AID-FE interface.
>
> If the END statement is used to assign a file, this file must be “SYSDTA-compatible” i.e. an ASSIGN-SYSDTA command must run without errors for this file.

Example 4-1

Assigning a cataloged file after input of COMOPT statements

```
/START-PROGRAM $COBOL2000————————————————————————(1)
COMOPT...————————————————————————————————————————(2)
END SOURCE.MULTABLE——————————————————————————————(3)
```

(1) The compiler is invoked. In interactive mode SYSDTA is assigned to the terminal.

(2) The keyword COMOPT informs the compiler that the following entries are control statements.

(3) The END statement assigns SYSDTA to the cataloged file SOURCE.MULTABLE, which contains the compilation unit to be compiled or a sequence of control statements.
At the end of compilation SYSDTA and SYSCMD are linked together.

Example 4-2

Assigning a library without the use of COMOPT statements

```
/START-PROGRAM $COBOL2000————————————————————————(1)
END PLAM.LIB(EXAMP3)—————————————————————————————(2)
```

(1) Invocation of the compiler; in interactive mode SYSDTA is assigned to the terminal.

(2) The system file SYSDTA is assigned to the element EXAMP3 in the PLAM library PLAM.LIB. At the end of the compilation SYSDTA and SYSCMD are linked together.

4.1.2 Assigning the compilation unit with the ADD-FILE-LINK command and COMOPT SOURCE-ELEMENT

Input from libraries can also be initiated directly - bypassing SYSDTA - with the ADD-FILE-LINK command. The standard link name SRCLIB must be used in this case. The general format of the ADD-FILE-LINK command for the input of compilation units from libraries is shown below:

```
/ADD-FILE-LINK [LINK-NAME=]SRCLIB,[FILE-NAME=]libname
```

Example 4-3

Input from a PLAM library

```
/ADD-FILE-LINK SRCLIB,PLAM.LIB ————————————————————————(1)
/START-PROGRAM $COBOL2000 ————————————————————————————(2)
COMOPT SOURCE-ELEMENT=EXAMP3 ————————————————————————(3)
COMOPT SOURCE-VERSION=V001 ——————————————————————————(4)
END ——————————————————————————————————————————————(5)
```

(1) The SDF command (in positional operand form) assigns the PLAM library PLAM.LIB and links it with the standard link name SRCLIB.

(2) Invocation of the compiler.

(3) The compilation unit to be compiled is stored under the element name EXAMP3 in the PLAM library assigned with the ADD-FILE-LINK command.

(4) The PLAM.LIB library contains several versions of the element named EXAMP3. In this case, the version designated as V001 is referenced.

(5) The input of options is terminated, and the compiler begins the compilation run.

4.2 Table of COMOPT operands

Almost all the options have a default value. This automatically applies if the user does not explicitly specify an alternative. If all the default values of the system are to be used as options, COMOPT entries are superfluous.

The following table summarizes all COMOPT operands that can be used to control the compiler.
The following points refer to the representation of the statement formats:

- If an operand can be abbreviated, the short form is indicated below its full designation (e.g. ACC-L-T-U for ACCEPT-LOW-TO-UP). The equal sign must be specified between the operand and the value in every case.
- The default operand values are either shown underlined in the format or mentioned explicitly in the summarized description of the function.

In the “Function” column, under the keyword “SDF option”, you will find the short form of the SDF operand equivalent of the respective COMOPT operand. If there is no equivalent SDF operand, this is indicated by a dash.

Operand format	Function
ACCEPT-LOW-TO-UP={YES/NO} ACC-L-T-U	specifies whether letters entered in lowercase are to be converted to uppercase when an ACCEPT statement is executed. The conversion is performed only if the input is typed in at the terminal. *SDF option:* RUNTIME-OPTIONS = PARAMETERS(...) ACCEPT-STMT-INPUT =
ACTIVATE-WARNING-MECHANISM={YES/NO} ACT-W-MECH	specifies whether the existence of • obsolete language elements and • non-standard language extensions that are detected in the program during compilation should be identified in the diagnostic listing by means of a message of severity code F. *Note* The COMOPT operands listed below are ineffective in the case of compilation runs for which ACTIVATE-WARNING-MECHANISM=YES has been specified. They would otherwise produce a deviation from the ANS85 Standard during compilation. RESET-PERFORM-EXITS = NO USE-APOSTROPHE = YES REPLACE-PSEUDOTEXT = NO In addition, the operand MINIMAL-SEVERITY=I is set in this case in order that messages of severity code F can also be listed. *SDF option:* ACTIVATE-FLAGGING = ANS85
ACTIVATE-XPG4-RETURNCODE= {YES/NO}	specifies that, after a subprogram is called, its function value (register 1) is available in the COBOL special register RETURN-CODE. *SDF option:* SOURCE-PROPERTIES = PARAMETERS(...) RETURN-CODE =.
ALIGN-LLM-PAGE={YES/NO} A-L-P	specifies whether CSECTs in the generated module should be aligned on the page (YES) or double word boundary (NO). *Note* This option only applies to LLMs, not to OMs. *SDF option:* COMPILER-ACTION MODULE-FORMAT=LLM(...) ALIGNMENT=PAGE
CHECK-CALLING-HIERARCHY={YES/NO}	

CHECK-C-H	specifies whether the calling hierarchy should be checked. A program in which the statements CALL identifier and/or CANCEL are used, must be compiled using CHECK-CALLING-HIERARCHY=YES. *SDF option:* RUNTIME-CHECKS = PARAMETERS(...) RECURSIVE-CALLS =
CHECK-DATE={YES/NO} CHECK-D	specifies whether or not the compiler outputs a note concerning two-digit year values in the case of ACCEPT FROM DATE/DAY *SDF option:* LISTING=PARAMETERS(...) DIAGNOSTICS=YES REPORT-2-DIGIT-YEAR=
CHECK-FUNCTION-ARGUMENTS={YES/ NO } CHECK-FUNC	causes function arguments to be checked for validity and a message to be issued by the runtime system when errors occur. *SDF option:* RUNTIME-CHECKS = PARAMETERS(...) FUNCTION-ARGUMENTS =
CHECK-PARAMETER-COUNT={YES/ NO } CHECK-PAR-C	specifies whether the number of parameters passed should be compared with the number of parameters expected when a COBOL subprogram is called. This does not work for subprograms called via an ENTRY. *SDF option:* RUNTIME-CHECKS = PARAMETERS(...) PROC-ARGUMENT-NR =
CHECK-REFERENCE-MODIFICATION = {YES/ NO} CHECK-REF	determines whether the runtime system should verify compliance with data-item limits for identifiers subject to reference modification. *SDF option:* RUNTIME-CHECKS = PARAMETERS(...) REF-MODIFICATION
CHECK-SCOPE-TERMINATORS={YES/ NO } CHECK-S-T	checks the syntax of the statements in the PROCEDURE DIVISION for correct scope termination. *SDF option:* LISTING = PARAMETERS(...) DIAGNOSTICS = YES(...) IMPLICIT-SCOPE-END =
CHECK-SOURCE-SEQUENCE={YES/ NO } CHECK-S-SEQ	determines whether record pairs that are found not to be in ascending order should be identified in the diagnostic listing by an error message of severity code 0. CHECK-SOURCE-SEQUENCE does not apply for free format. *SDF option: --*
CHECK-TABLE-ACCESS={YES/NO} CHECK-TAB	determines whether the runtime system should verify compliance with table limits (both for subscripts and for indexing). *SDF option:* RUNTIME-CHECKS = PARAMETERS(...) TABLE-SUBSCRIPTS =
CONCATENATE-XML-LINES={YES/NO} C-X-L	determines whether the line feed caused by the record structure of a file(which contains an XML document) is to be supplied to the parser as an end-of-line character. *SDF option:* RUNTIME-OPTIONS=PARAMETERS(...) XML-LINE-FEED=
CONTINUE-AFTER-MESSAGE={YES/NO} CON-A-MESS	determines whether the runtime system should be continued or terminated following specific COB91 messages. *SDF option:* RUNTIME-OPTIONS = PARAMETERS(...) ERROR-REACTION =

DEFAULT-CALL-CONVENTION={COBOL/ COMPATIBLE} DEF-C-C	COMPATIBLE is assumed as the default value for the CALL-CONVENTION directive. If COBOL is specified, the value COBOL is assumed for the CALL-CONVENTION directive. *SDF option:* COMPILER-ACTION=MODULE-GENERATION(...) CALL-CONVENTION =*COBOL
ELABORATE-SEGMENTATION={YES/NO}	If NO is specified, segmentation-related language elements are ignored (SEGMENT-LIMITclause, segment numbers in section header). Warnings are output. If YES is specified, compiler directives are ignored which apply during the compilation phase and are specified within a compilation unit. YES supports segmentation. However, the compilation is aborted with a message if the program contains nested compilation units and non-fixed segments. If this combination does not exist, only segmentation-related language elements are rejected with warnings. If ELABORATE-SEGMENTATION=YES is specified together with GENERATE-SHARED-CODE=YES or GENERATE-LLM=YES, it is also rejected. *SDF option:* COMPILER-ACTION=MODULE-GENERATION(...) SEGMENTATION=
ENABLE-COBOL85-KEYWORDS-ONLY={YES / NO}	If YES is specified, only the keywords defined for COBOL85 are reserved. The additional keywords reserved by COBOL2000 can then be used as file names. *SDF option:* SOURCE-PROPERTIES = PARAMETERS(...) ENABLE-KEYWORDS=
ENABLE-UFS-ACCESS={YES/NO} Not available in COBOL2000-BC!	specifies whether the compiler is to generate an object which is also capable of processing files from the POSIX file system. *SDF option:* RUNTIME-OPTIONS = PARAMETERS(...) ENABLE-UFS-ACCESS =
ENABLE-XML-PROCESSING={YES/NO}	If NO is specified, the new statements for processing XML documents are not available. The keywords reserved for this purpose can then be used as data names. *SDF option:* SOURCE-PROPERTIES=PARAMETERS(...) ENABLE-KEYWORDS=STD(...) XML-SUPPORT=
EXPAND-COPY={YES/NO} EXP-COPY	controls whether COPY elements inserted in the compilation unit are printed in the source listing. *SDF option:* LISTING = PARAMETERS(...) SOURCE = YES(...) COPY-EXPANSION =
EXPAND-SUBSCHEMA={YES/NO} EXP-SUB Not available in COBOL2000-BC!	controls whether the Sub-schema Section of the compilation unit is logged on the source listing. *SDF option:* LISTING = PARAMETERS(...) SOURCE = YES(...) SUBSCHEMA-EXPANSION =
FLAG-NONSTANDARD={YES/NO}	In the diagnostic listing, all non-standard language extensions are flagged with F. *Note* The COMOPT operands listed below are ineffective in the case of compilation runs for which FLAG-NONSTANDARD=YES has been specified. They would otherwise produce a deviation from the ANS85 Standard during compilation. RESET-PERFORM-EXITS = NO USE-APOSTROPHE = YES REPLACE-PSEUDOTEXT = NO *SDF option:* ACTIVATE-FLAGGING = FIPS(...) NONSTANDARD-LANGUAGE =

FLAG-OBSOLETE={YES/NO}	In the diagnostic listing, all obsolete language extensions are flagged with F. *Note* The COMOPT operands listed below are ineffective in the case of compilation runs for which FLAG-OBSOLETE=YES has been specified. They would otherwise produce a deviation from the ANS85 Standard during compilation. RESET-PERFORM-EXITS = NO USE-APOSTROPHE = YES REPLACE-PSEUDOTEXT = NO *SDF option:* ACTIVATE-FLAGGING = FIPS(...) OBSOLETE-FEATURES =
GENERATE-INITIAL-STATE={YES/NO} GEN-INIT-STA	specifies whether the compiler should make arrangements to return the program to its initial state. All programs which are affected by a CANCEL statement or which contain an INITIAL clause or an INITIALIZE statement with TO VALUE specification must be compiled with GENERATE-INITIAL-STATE=YES in order to conform to the standard. *SDF option:* COMPILER-ACTION = MODULE-GENERATION(...) ENABLE-INITIAL-STATE =
GENERATE-LINE-NUMBER={YES/NO} GEN-L-NUM	determines whether the COB9101 message is output instead of the COB9102 message. The COB9102 message is supplemented by the source line number (generated by COBOL2000) of the statement being executed when the message was issued. *SDF option:* RUNTIME-OPTIONS = PARAMETERS(...) ERR-MSG-WITH-LINE-NR =
GENERATE-LLM={YES/NO} GEN-LLM	defines the module format for the module to be generated. If YES is specified, a link-and-load module (LLM) is generated; if NO is specified, an object module (OM) is generated. These specifications are ignored if MODUL-OUTPUT=<c-string...> has been specified. *SDF option:* COMPILER-ACTION = MODULE-GENERATION(...) MODULE-FORMAT = OM / LLM
GENERATE-SHARED-CODE={YES/NO} GEN-SHARE	specifies whether the Procedure Division code (without DECLARATIVES) is written to a separate code module. The name for this module is program name, shortened to 7 characters if necessary, with appended “@”. *SDF option:* COMPILER-ACTION = MODULE-GENERATION(...) SHAREABLE-CODE =
IGNORE-COPY-SUPPRESS={YES/NO} IGN-C-SUP	determines whether or not existing COPY elements with the SUPPRESS option in the compilation unit should be listed in the source listing . IGNORE-COPY-SUPPRESS=YES has the additional effect of the EXPAND-COPY=YES operand. *SDF option:* LISTING = PARAMETERS(...) SOURCE = YES(...) COPY-EXPANSION =
IGNORE-EXTRA-ALTERNATE-KEYS={YES/ NO } IGN-EXT-ALTKEY	determines whether or not secondary keys which are defined only in the catalog entry but not in the program are to be ignored for indexed files and no error message output in the case of OPEN INPUT. *SDF option:* RUNTIME-OPTIONS = PARAMETERS(...) EXTRA-ALTERNATE-KEYS =
IGNORE-LINE-SEQUENTIAL={YES/NO}	specifies whether the compiler is to ignore LINE in SELECT clauses and process LINE SEQUENTIAL files as if they are with SEQUENTIAL organization. If default NO is specified, the compiler processes sequential files in accordance with their declarations. *SDF option:* SOURCE-PROPERTIES = PARAMETERS(...) LINE-SEQUENTIAL =

IGNORE-OPTION-DIRECTIVES={YES/NO} IGN-O-DIR	determines whether or not >>IMP directives which apply for the compiler options (LISTING-OPTIONS, COMPILER-ACTION and RUNTIME-ERRORS) are ignored. *SDF option:* COMPILER-ACTION = MODULE-GENERATION(...) OPTION-DIRECTIVES =
INHIBIT-BAD-SIGN-PROPAGATION={YES/NO}	NO enables faster code to be generated when one data item is transferred to another, and both items are numerical and described with USAGE DISPLAY. No code is generated which would prevent encoded operational signs which do not match the PICTURE clause from being transferred.
KEEP-XML-NAMES={YES/NO}	specifies whether the statement XML GENERATE is to transfer data item names from the source code to the XML document without any change (YES) or converted to upper case (NO). *SDF option:* SOURCE-PROPERTIES = PARAMETERS(...) XML-NAMES =
LIBFILES= (list-id[,list-id]...)	determines which compilation protocols are to be created and output to a PLAM library. list-id will be one of the following specifications [NO]OPTIONS [NO]DIAG [NO]SOURCE [NO]OBJECT ALL [NO]MAP [NO]XREF NO The requested listings are processed from left to right. The value set last applies to the listing. If XREF is specified, MAP is also automatically assumed to apply. Each requested listing is generated with a standard name as an element of type R. The standard names are as follows: OPTLST.COBOL.program-name (control statement listing) SRCLST.COBOL.program-name (source listing) ERRLST.COBOL.program-name (diagnostics listing) LOCLST.COBOL.program-name (locator map/cross-reference listing) OBJLST.COBOL.program-name (object listing) For details on the possible truncation of names, see section “LISTING option”. The PLAM library must be assigned with the ADD-FILE-LINK command via the link name LIBLINK. If no library name is assigned, the compiler stores the requested listings in the default library PLIB.COBOL.program-name. *SDF option:* LISTING = PARAMETERS(...) OUTPUT = LIBRARY-ELEMENT(...) LIBRARY=<filename 1..54>
LINE-LENGTH=132 / 119..172 LINE-L	specifies the maximum number of characters that are printed per line in the compiler listings. *SDF option:* LISTING = PARAMETERS(...) LAYOUT = PARAMETERS(...) LINE-SIZE =
LINES-PER-PAGE=64 / 20..128 LINES	specifies the maximum number of lines that are printed in the compiler listings per page. A page throw is effected as soon as the specified number of lines is reached. *SDF option:* LISTING = PARAMETERS(...) LAYOUT = PARAMETERS(...) LINES-PER-PAGE =
LISTFILES=(list-id[,list-id]...)	specifies which compiler listings are to be created and output to cataloged files. list-id can be one of the following entries: [NO]OPTIONS [NO]DIAG [NO]SOURCE [NO]OBJECT ALL [NO]MAP [NO]XREF NO

	For more information, read the description of COMOPT LIBFILES, which is similar. *SDF option:* LISTING = PARAMETERS(...) OUTPUT = STD-FILES
MARK-NEW-KEYWORDS={YES/NO} M-N-K	marks keywords from the future standard in the diagnostic listing with a message with severity code I.The value YES can only be specified if ENABLE-COBOL85-KEYWORDS-ONLY is also set to YES. *SDF option:* LISTING=PARAMETERS(...) DIAGNOSTICS=YES MARK-NEW-KEYWORDS=
MAXIMUM-ERROR-NUMBER=integer MAX-ERR	specifies from what error number onwards (depending on the MINIMAL-SEVERITY phrase) compilation should be aborted. *SDF option:* COMPILER-TERMINATION = PARAMETERS(...) MAX-ERROR-NUMBER =
MERGE-DIAGNOSTICS=YES/NO M-DIAG	“merges” all error messages that occurred during compilation with the source listing. For merging to function correctly, the source listing should not contain more than 65535 source lines (see section “Source listing for a compilation unit”). *SDF option:* LISTING=PARAMETERS(...) SOURCE=YES(...) INSERT-ERROR-MSG=
MERGE-REFERENCES={YES/NO} M-REF	extends the source listing with specifications on the address and length of definitions and cross-references to references or definitions. *SDF option:* LISTING=PARAMETERS(...) SOURCE=YES(...) CROSS-REFERENCE=
MERGE-STATEMENT-ADDRESS={YES/NO} M-STMT	causes the address of the first machine instruction which was generated for this purpose to be entered in the source listing for statements . (only if the option MERGE-REFERENCES=YES is set) *SDF option:* LISTING=PARAMETERS(...) SOURCE=YES(...) CROSS-REFERENCE=YES(...) STMT-ADDRESS=
MINIMAL-SEVERITY={I/0/1/2/3} MIN-SEV	suppresses messages in the diagnostic listing if their severity codes are less than the specified value. *SDF option:* LISTING = PARAMETERS(...) DIAGNOSTICS = YES(...) MINIMAL-WEIGHT =
MODULE={*OMF/libname}	specifies where the object module that is generated during compilation is to be output. *OMF initiates output to the temporary EAM file of the current task. libname is the file name of the PLAM library in which the object module is to be placed. libname must be a valid BS2000 file name. *SDF option:* MODULE-OUTPUT = *OMF / *LIBRARY-ELEMENT(...) LIBRARY =
MODULE-ELEMENT=element-name MODULE-ELEM	specifies the name of the element under which an LLM is to be stored in the PLAM library. Max. length of element name: 32 chars. *Note* This compiler option is ignored for object modules and compilation groups (error message severity code I).

	SDF option: MODULE-OUTPUT = *LIBRARY-ELEMENT(...) LIBRARY = <filename>, ELEMENT =
MODULE-VERSION=version MODULE-VERS	enables the assignment of a version designation to the element which contains the module generated during compilation. version can be one of the following entries: *UPPER-LIMIT / *UPPER *HIGHEST-EXISTING / *HIGH *INCREMENT / *INCR <alphanum-name 1..24> *SDF option:* MODULE-OUTPUT = *LIBRARY-ELEMENT(...) LIBRARY = <filename>, ELEMENT = <composed-name> VERSION =
OPTIMIZE-CALL-IDENTIFIER={YES/NO} O-C-I	enables repeated calls for the same subprogram to be processed via CALL identifier without calling system interfaces (this is possible for the first 100 subprograms to be called) *SDF option:* OPTIMIZATION = PARAMETERS(...) CALL-IDENTIFIER =
PERMIT-STANDARD-DEVIATION={YES/NO} P-S-D	specifies whether • the data descriptions in the Linkage section (level number 01 or 77) can be assigned the address of another section or the contents of a pointer by means of a SET statement even in the absence of a BASED specification. (There is then no check to determine whether each parameter that is used has also been specified in the USING clause of the Procedure Division.) • data structures which contain pointer data items or universal object references, or pointer data items or universal object references with level number 01 or 77 are permitted and redefined as receiving items or may be subject to reference modification. *SDF option:* SOURCE-PROPERTIES=PARAMETERS(...) STANDARD-DEVIATION=
PRINT-DIAGNOSTIC-MESSAGES={YES/NO} PRI-DIAG Not available in COBOL2000-BC!	makes it possible to have all COBOL2000 error messages listed. Compilation is not carried out in this case. *SDF option:* COMPILER-ACTION = PRINT-MESSAGE-LIST
REDIRECT-ACCEPT-DISPLAY={YES/NO}	causes the system files SYSDTA and SYSOUT to be assigned instead of system files SYSIPT and SYSLST (defaults) for ACCEPT and DISPLAY statements without FROM and UPON phrases. *SDF option:* RUNTIME-OPTIONS = PARAMETERS(...) ACCEPT-DISPLAY-ASSGN =
REPLACE-PSEUDOTEXT={YES/NO} REP-PSEUDO	determines how COPY elements are to be divided up into individually replaceable text words. If NO is specified, the separators colon, open brackets, close brackets and pseudotext delimiter do not act as separators for text words and are not independent text words. One particular effect of this is that no replacements are made within brackets in mask strings. If NO is specified, the REPLACE statement may not be used. The options in the REPLACING clause are limited to the replacement of a single text word by a text word or an identifier. Hexadecimal and national literals are not recognized as single text words. COPY elements may not contain COPY statements. REPLACE-PSEUDOTEXT=NO is not supported for free format. *SDF option: --*
RESET-PERFORM-EXITS={YES/NO} RES-PERF	specifies whether the control mechanisms for all PERFORM statements are • reset for EXIT PROGRAM according to the ANS85 Standard (default value or YES specification) or

	• to remain active on exiting from the subprogram (NO specification). *SDF option: --*
ROUND-FLOAT-RESULTS-DECIMAL={YES/NO} ROUND-FLOAT	specifies whether floating-point data items are to be rounded to 7 (COMP-1) or 15 (COMP-2) decimal places before being transferred to fixed-point items. The option is only effective if the receiving item has been defined with fewer than 19 decimal digits. *SDF option: --*
SEPARATE-TESTPOINTS={YES/NO} SEP-TESTP Not available in COBOL2000-BC!	specifies whether a separate address is to be generated for all paragraph and section headings in the PROCEDURE DIVISION for debugging with AID. *SDF option:* TEST-SUPPORT = AID(...) PREPARE-FOR-JUMPS =
SET-FUNCTION-ERROR-DEFAULT={YES/NO} S-F-E-D	causes function arguments to be checked for validity and an appropriate return code to be assigned to any function in which a corresponding error occurs. *SDF option:* RUNTIME-OPTIONS = PARAMETERS(...) FUNCTION-ERR-RETURN =
SHORTEN-OBJECT={YES/NO} SHORT-OBJ Not available in COBOL2000-BC!	specifies whether only ESD information should be listed in the requested object listing. *SDF option: --*
SHORTEN-XREF={YES/NO} SHORT-XREF	determines whether the desired cross-reference listing should be shortened by including only data names and procedure names addressed in the program. *SDF option:* LISTING = PARAMETERS(...) NAME-INFORMATION = YES(...) CROSS-REFERENCE =
SORT-EBCDIC-DIN={YES/NO} SORT-E-D	enables selection of the EBCDIC-DIN (ED) format for SORT (see “Sort” manual [6]); among other things, characters with an umlaut, such as ä, ö or ü, are treated as AE, OE or UE during sort operations. *SDF option:* RUNTIME-OPTIONS = PARAMETERS(...) SORTING-ORDER =
SORT-MAP={YES/NO}	enables output of the locator map listing from the compilation unit, sorted by symbolic names in ascending order. The locator map listing comprises lists for data, section, and paragraph names. *SDF option:* LISTING = PARAMETERS(...) NAME-INFORMATION = YES SORTING-ORDER =
SOURCE-ELEMENT=element SOURCE-ELEM	assigns a PLAM library element as the compilation unit to the compiler. The library must be assigned with an ADD-FILE-LINK command (using the link name SRCLIB) prior to compilation. element is the name of the library element. It must be included in a PLAM library under element type S. “element” can have a maximum length of 40 characters. *SDF option:* SOURCE = *LIBRARY-ELEMENT(...) LIBRARY = ELEMENT =
SOURCE-VERSION=version SOURCE-VERS	indicates to the compiler which version of the element assigned with SOURCE-ELEMENT should be compiled. version is one of the following entries: *HIGHEST-EXISTING / *HIGH *UPPER-LIMIT / *UPPER <alphanum-name 1..24>

	SDF option: SOURCE = *LIBRARY-ELEMENT(...) LIBRARY = ,ELEMENT = VERSION =
SUPPRESS-LISTINGS={YES/NO} SUP-LIST	suppresses output of the • object program • locator map and • cross-reference (XREF) listings when an error with a severity code >= 2 occurs. In such cases, only the diagnostic listing and source listing are output (if requested). *SDF option:* LISTING = PARAMETERS(...) NAME-INFORMATION = SUPPRESS-GENERATION =
SUPPRESS-MODULE={YES/NO} SUP-MOD	permits generation of a module and expansion of the parameterized classes/interfaces used to be prevented when an error with a severity code >=2 occurs. SUPPRESS-MODULE=YES has the additional effect of the SUPPRESS-LISTINGS=YES operand. *SDF option:* COMPILER-ACTION = MODULE-GENERATION(...) SUPPRESS-GENERATION =
SYMTEST={ALL/NO} Not available in COBOL2000-BC!	specifies the information that the compiler provides for the advanced interactive debugger AID (see “AID” manual [8]). ALL: The compiler generates LSD information and ESD debugger information. NO: The compiler generates only ESD debugger information. *SDF option:* TEST-SUPPORT = AID(...)
SYSLIST=(list-id[,list-id]...)	defines which compiler listings are to be created and output to the SYSLST system file. list-id can be one of the following entries: [NO]OPTIONS [NO]DIAG [NO]SOURCE [NO]OBJECT ALL [NO]MAP [NO]XREF NO *SDF option:* LISTING = PARAMETERS(...) OUTPUT = SYSLST
TERMINATE-AFTER-SEMANTIC={YES/NO} TERM-A-SEM	causes the compilation group to only be checked for syntax and semantic errors, without a module being generated. Only source and diagnostic listings can be output in this case. *SDF option:* COMPILER-ACTION = SEMANTIC-CHECK
TERMINATE-AFTER-SYNTAX={YES/NO} TERM-A-SYN	causes the compilation group to only be checked for syntax errors, without a module being generated. Only source and diagnostic listings can be output in this case. *SDF option:* COMPILER-ACTION = SYNTAX-CHECK
TEST-VIRTUAL-NAMES={YES/NO} Not available in COBOL2000-BC!	specifies for SYMTEST=ALL whether the virtual names FILLERnn and structure SPECIAL-NAMES are to be generated for debugging with AID. *SDF option:* TEST-SUPPORT = AID(...) VIRTUAL-NAMES=

TEST-WITH-COLUMN1={YES/NO} TEST-W-C Not available in COBOL2000-BC!	defines for SYMTEST=ALL whether the AID source references are to be formed with the help of sequence numbers (columns 1-6) from the compilation group. TEST-WITH-COLUMN1 is not supported with free format. *SDF option:* TEST-SUPPORT = AID(...) STMT-REFERENCE =
UPDATE-REPOSITORY={YES/NO} UPD-R	determines whether or not the interface of the currently compiled source text is placed in the external repository assigned with the link name REPOUT. Repository data is stored in an element of type X. To enable a differentiation, classes are assigned the suffix $CLS, interfaces the suffix $IFC and program prototypes the suffix $PRO. *SDF option:* COMPILER-ACTION=MODULE-GENERATION UPDATE-REPOSITORY=
USE-APOSTROPHE={YES/NO} USE-AP	controls the representation of the figurative “QUOTE” constants. If YES is specified, the figurative QUOTE constant has a single quote as its value. If NO is specified, the value is a double quote.

5 Controlling the compiler with compiler directives

The >>IMP compiler directives enable some compilation options to be specified directly in source text.

In contrast to options which apply for the entire compilation group, directives can be specified separately for each compilation unit.

The values specified by SDF control or COMOPT statements define the default values for the directives described below. The SDF option COMPILER-ACTION=*MODULE-GENERATION(OPTION-DIRECTIVES=*IGNORE) or the COMOPT statement IGNORE-OPTION-DIRECTIVES=YES enables the changing of external control by directives to be prevented.

The notation corresponds to the COBOL notation of the Reference Manual (see the “COBOL2000 Reference Manual” [1]).

5.1 IMP COMPILER-ACTION

This directive controls actions of the compiler during module generation.

Format

```
 >>IMP COMPILER-ACTION {|GENERATE-INITIAL-STATE | UPDATE-REPOSITORY|} {ON | OFF |
DEFAULT}
```

Syntax rule

1. This directive may only be specified before a compilation unit.

General rules

1. The directive applies in the compilation phase.
2. Each operand specified in the directive relates to the compiler option of the same name (see “COBOL2000 Reference Manual” [1]).
3. The GENERATE-INITIAL-STATE specification is rejected if the compiler option RESET-PERFORM-EXITS=NO was specified when the compiler was called.
4. The ON specification causes the value YES to be assumed for the compiler option specified.
5. The OFF specification causes the value NO to be assumed for the compiler option specified.
6. The DEFAULT specification causes the value specified when the compiler was called to be assumed for the compiler option specified.

5.2 IMP LISTING-OPTIONS

This directive enables the values of compiler options which influence the listings generated by the compiler to be modified.

Format

```
>>IMP LISTING-OPTIONS {| EXPAND-COPY | EXPAND-SUBSCHEMA | MERGE-DIAGNOSTICS  | MERGE-
REFERENCES | MERGE-STATEMENT-ADDRESS  | SORT-MAP  | SHORTEN-XREF  |}

       {ON |  OFF  |  DEFAULT }
```

Syntax rule

1. This directive may only be specified before a compilation unit.

General rules

1. The directive applies in the listing generation phase.
2. Each operand specified in the directive relates to the compiler option of the same name (see “COBOL2000 Reference Manual” [1]).
3. The ON specification causes the value YES to be assumed for the compiler option specified. However, the directive applies only if the listing which it affects is also to be generated.
4. The OFF specification causes the value NO to be assumed for the compiler option specified.
5. The DEFAULT specification causes the value specified when the compiler was called to be assumed for the compiler option specified.
6. The last value specified for an operand before a compilation unit is also used for listing generation of the lines specified before a compilation unit (e.g. the last value for EXPAND-COPY also applies for COPY statements which were specified before this directive).

5.3 IMP PRINT-DIRECTIVES

This directive enables the values of directives to be written to the source listing.

Format

```
>>IMP PRINT-DIRECTIVES {ALL | NON-DEFAULT}
```

General rules

1. The directive applies in the listing generation phase.
2. The directive may be used anywhere in a compilation unit.
3. The directive enables the values of directives which apply in the compilation phase and of the >>IMP LISTING-OPTIONS directive to be written to the source listing.
4. The ALL specification causes the values of all directives to be written to the source listing.
5. The NON-DEFAULT specification causes the values of all directives which deviate from the default value to be written to the source listing.
6. The output is generated in the listing directly after the directive is issued.
7. If listing generation is disabled for the line in which the directive is specified (>>LISTING OFF, COPY ... SUPPRESS,...), the directive values are not listed either.

Example 5-1

Source code:

```
      >>IMP PRINT-DIRECTIVES NON-DEFAULT
      >>IMP PRINT-DIRECTIVES ALL
      >>IMP LISTING-OPTIONS MERGE-DIAGNOSTICS
      >>CALL-CONVENTION COBOL
      >>TURN EC-OO-CONFORMANCE EC-OO-NULL CHECKING ON
      >>IMP LISTING-OPTIONS EXPAND-COPY EXPAND-SUBSCHEMA OFF
      >>IMP RUNTIME-ERRORS FUNCTION-DEFAULT-VALUE ON
      >>IMP LISTING-OPTIONS SORT-MAP SHORTEN-XREF ON
      >>IMP PRINT-DIRECTIVES NON-DEFAULT
      >>IMP PRINT-DIRECTIVES ALL
      ...
```

Listing:

OPTIONS BY DEFAULT

```
                       ...
                       EXPAND-COPY = YES
                       ...
                       EXPAND-SUBSCHEMA = YES
                       ...
                       GENERATE-INITIAL-STATE = YES
                       ...
      V    VV
 00001       >>IMP PRINT-DIRECTIVES NON-DEFAULT
#####  ALL DIRECTIVES VALUES ARE SET TO DEFAULT ####
 00002       >>IMP PRINT-DIRECTIVES ALL
##### >>CALL-CONVENTION                                    COMPATIBLE
```

```
##### >>FLAG-85 ZERO-LENGTH                                    OFF
##### >>IMP RUNTIME-ERRORS FUNCTION-DEFAULT-VALUE              OFF
##### >>IMP LISTING-OPTIONS EXPAND-COPY                        ON
##### >>IMP LISTING-OPTIONS EXPAND-SUBSCHEMA                   ON
##### >>IMP LISTING-OPTIONS MERGE-DIAGNOSTICS                  OFF
##### >>IMP LISTING-OPTIONS MERGE-REFERENCES                   OFF
##### >>IMP LISTING-OPTIONS MERGE-STATEMENT-ADDRESS            OFF
##### >>IMP LISTING-OPTIONS SORT-MAP                           OFF
##### >>IMP LISTING-OPTIONS SHORTEN-XREF                       OFF
##### >>IMP COMPILER-ACTION GENERATE-INITIAL-STATE             ON
##### >>IMP COMPILER-ACTION UPDATE-REPOSITORY                  OFF
##### >>TURN EC-DATA-CONVERSION                          CHECKING OFF
##### >>TURN EC-OO-CONFORMANCE                           CHECKING OFF
##### >>TURN EC-OO-METHOD                                CHECKING OFF
##### >>TURN EC-OO-NULL                                  CHECKING OFF
##### >>TURN EC-OO-RESOURCE                              CHECKING OFF
##### >>TURN EC-OO-UNIVERSAL                             CHECKING OFF
##### >>TURN EC-STORAGE-NOT-ALLOC                        CHECKING OFF
##### >>TURN EC-STORAGE-NOT-AVAIL                        CHECKING OFF
##### >>TURN EC-XML-CODESET-CONVERSION                   CHECKING OFF
 00003       >>IMP LISTING-OPTIONS MERGE-DIAGNOSTICS
 00004       >>CALL-CONVENTION COBOL
 00005       >>TURN EC-OO-CONFORMANCE EC-OO-NULL CHECKING ON
 00006       >>IMP LISTING-OPTIONS EXPAND-COPY EXPAND-SUBSCHEMA OFF
 00007       >>IMP RUNTIME-ERRORS FUNCTION-DEFAULT-VALUE ON
 00008       >>IMP LISTING-OPTIONS SORT-MAP SHORTEN-XREF ON
 00009       >>IMP PRINT-DIRECTIVES NON-DEFAULT
##### >>CALL-CONVENTION                                      COBOL
##### >>FLAG-85 ZERO-LENGTH                                  OFF
##### >>IMP RUNTIME-ERRORS FUNCTION-DEFAULT-VALUE            ON
##### >>IMP LISTING-OPTIONS EXPAND-COPY                      OFF
##### >>IMP LISTING-OPTIONS EXPAND-SUBSCHEMA                 OFF
##### >>IMP LISTING-OPTIONS MERGE-DIAGNOSTICS                ON
##### >>IMP LISTING-OPTIONS SORT-MAP                         ON
##### >>IMP LISTING-OPTIONS SHORTEN-XREF                     ON
##### >>TURN EC-OO-CONFORMANCE CHECKING                      ON
##### >>TURN EC-OO-NULL CHECKING                             ON
 00010       >>IMP PRINT-DIRECTIVES ALL
##### >>CALL-CONVENTION                                      COBOL
##### >>IMP RUNTIME-ERRORS FUNCTION-DEFAULT-VALUE            ON
##### >>IMP LISTING-OPTIONS EXPAND-COPY                      OFF
##### >>IMP LISTING-OPTIONS EXPAND-SUBSCHEMA                 OFF
##### >>IMP LISTING-OPTIONS MERGE-DIAGNOSTICS                ON
##### >>IMP LISTING-OPTIONS MERGE-REFERENCES                 OFF
##### >>IMP LISTING-OPTIONS MERGE-STATEMENT-ADDRESS          OFF
##### >>IMP LISTING-OPTIONS SORT-MAP                         ON
##### >>IMP LISTING-OPTIONS SHORTEN-XREF                     ON
##### >>IMP COMPILER-ACTION GENERATE-INITIAL-STATE           ON
##### >>IMP COMPILER-ACTION UPDATE-REPOSITORY                OFF
##### >>TURN EC-DATA-CONVERSION                        CHECKING OFF
##### >>TURN EC-OO-CONFORMANCE                         CHECKING ON
##### >>TURN EC-OO-METHOD                              CHECKING OFF
##### >>TURN EC-OO-NULL                                CHECKING ON
##### >>TURN EC-OO-RESOURCE                            CHECKING OFF
##### >>TURN EC-OO-UNIVERSAL                           CHECKING OFF
##### >>TURN EC-STORAGE-NOT-ALLOC                      CHECKING OFF
##### >>TURN EC-STORAGE-NOT-AVAIL                      CHECKING OFF
##### >>TURN EC-XML-CODESET-CONVERSION                 CHECKING OFF
```

5.4 IMP RUNTIME-ERRORS

This directive controls the checking and handling of runtime errors.

Format

```
>>IMP RUNTIME-ERRORS FUNCTION-DEFAULT-VALUE {ON | OFF | DEFAULT}
```

General rules

1. The directive applies in the compilation phase.
   If the directive is specified in a statement, it applies only for the next clause or statement.

   > The same RUNTIME-ERRORS specification applies for WHEN specifications in EVALUATE and SEARCH statements as for the EVALUATE or SEARCH statement.

2. The directive may be used anywhere in a compilation unit.
3. The directive is rejected if the compiler option CHECK-FUNCTION-ARGUMENTS=YES was specified when the compiler was called.
4. The ON specification causes the value YES to be assumed for the compiler option SET-FUNCTION-ERROR-DEFAULT.
5. The OFF specification causes the value NO to be assumed for the compiler option SET-FUNCTION-ERROR-DEFAULT.
6. The DEFAULT specification causes the value specified when the compiler was called to be assumed for the compiler option SET-FUNCTION-ERROR-DEFAULT.

6 Linking, loading, starting

In the course of compilation, COBOL2000 generates object modules or link-and-load modules (LLMs), which are then available in a PLAM library or in the temporary EAM file of the current task.
The program cannot, however, run in this form because its machine code is not yet complete: each module contains references to external addresses, i.e. to other modules, which must supplement it when execution takes place. The compiler generates these **external references** during compilation for one or more of the following reasons:

- The COBOL program contains statements which
  - require complex routines at machine code level (e.g. SEARCH ALL, INSPECT) or
  - constitute interfaces to other software products or the operating system (e.g. SORT or input/output statements such as READ, WRITE).

  This applies to all COBOL programs, because the routines for program initialization and program termination are also in this category. The machine instruction sequences for these statements are not generated during compilation; they are already available as finished modules in a library, the **runtime system**. An external reference to the corresponding module in the runtime system is entered into the module generated by the compiler for each such COBOL statement.
- The COBOL program calls an external subprogram.

  The CALL statements in the “CALL literal” format cause the compiler to generate external references for the linkage run at the appropriate locations in the generated module.
  CALL statements in the “CALL identifier” format cause the dynamic binder loader to dynamically load the appropriate modules at runtime (see chapter “COBOL2000 and POSIX”).
- The COBOL program has been compiled with
  COMOPT GENERATE-SHARED-CODE=YES (in SDF: SHAREABLE-CODE=YES).The compiler generates a non-shareable data module and a shareable code module (see section “Shareable COBOL programs”). The data module contains an external reference to the associated code module.
- The COBOL program uses language elements for object orientation.The compiler generates external references for all non-parameterized classes /interfaces which are specified in the REPOSITORY paragraph.
- The COBOL program contains data written with external. The compiler creates common areas for this purpose.

During linking and loading, the use of some language elements in programs requires additional modules which are not part of the runtime system(CRTE), but must be available in the system. These must either be linked statically (see section “Static linkage using TSOSLNK” and section “Linking using BINDER”) or assigned using appropriate BLSLIBnn link names during dynamic linkage (see section “Dynamic linking and loading using DBL”). Specifically, this concerns:

- SORT:

  SORT80 module with ILSORT and SORTZM1 entries (among others).
  This module is normally contained in the $TASKLIB library.
- Library member as line sequential file:

  LMSUP1 module.
  This module is normally contained in the $LMSLIB library.
- National data (UTF-16):

  GNLADPT module.
  Where you can find this module is described in the “XHCS (BS2000/OSD)” manual [33].
- XML:

- GNLADPT module.
  See the “XHCS (BS2000/OSD)” manual [33].
- ITCRXFCA module in the ’parser library’, which you must provide.
  See chapter “Processing XML documents”.

6.1 Functions of the linkage editor

The process during which these external references are resolved (i.e. the additionally required modules are linked with the generated module into an executable unit) is called linking; the utility routine which performs this task is called a **linkage editor**.

A linkage editor processes either the result of a compilation (object module or LLM) or a module already prelinked in a linkage run. A prelinked module may consist of one or more object modules or a link-and-load module. Object modules and prelinked modules are referred to collectively using the term “object module”. This term is used in the following whenever the object to be described can be both an object module and a prelinked module.

To enable the unit generated during linking to run, a **loader** must be used to bring it into memory to allow the processor to access and execute the code.

The **Binder-Loader-Starter system** in BS2000 provides the following functional units for linking and loading:

- The binder BINDER

  links modules (object modules, link-and-load modules) to form a logically and physically structured loadable unit. This unit is referred to as a **l**ink-and-**l**oad **m**odule (**LLM**) and is stored by BINDER as an element of type L in a PLAM library.
- The static linkage editor TSOSLNK (**TSOS LINK**age editor)

  links one or more object modules into an object program (also called a load module) and saves it in a cataloged file or as an element of type C in a PLAM library;
  or
  links a number of object modules into a single prelinked module and stores it as an element of type R in a PLAM library or in a temporary EAM file.
- The **d**ynamic **b**inder **l**oader DBL

  combines modules (i.e. object modules and LLMs, possibly generated by an earlier linkage run with BINDER) into a temporary loadable unit, which it then loads into main memory and executes immediately. COBOL programs that call at least one external subprogram with “CALL identifier” can only be executed by this method (see section “Linking and loading subprograms”).
- The static loader ELDE

  loads a program that was linked by TSOSLNK and stored in a file or as an element of type C in a PLAM library.

The COBOL2000 compiler generates object modules or LLMs at compilation. Object modules are placed in the temporary EAM file of the current task or in a PLAM library as an element of type R.
LLMs are stored as elements of type L in a PLAM library.

The following table shows which modules are generated and/or processed by the individual functional units of the Binder-Loader-Starter system.

Module type	**System component**			
	BINDER	DBL	TSOSLNK	ELDE
Object module	yes	yes	yes	no
Link-and-load module (LLM)	yes	yes *)	no	no
Prelinked module	yes	yes	yes	no
Object program (load module)	no	no	yes	yes

*) Only in ADVANCED run mode

The linkage run in the POSIX subsystem is explained in the chapter “COBOL2000 and POSIX”.

The following diagram shows an overview of the various options that are provided to generate and call temporary and permanent executable COBOL programs in BS2000.

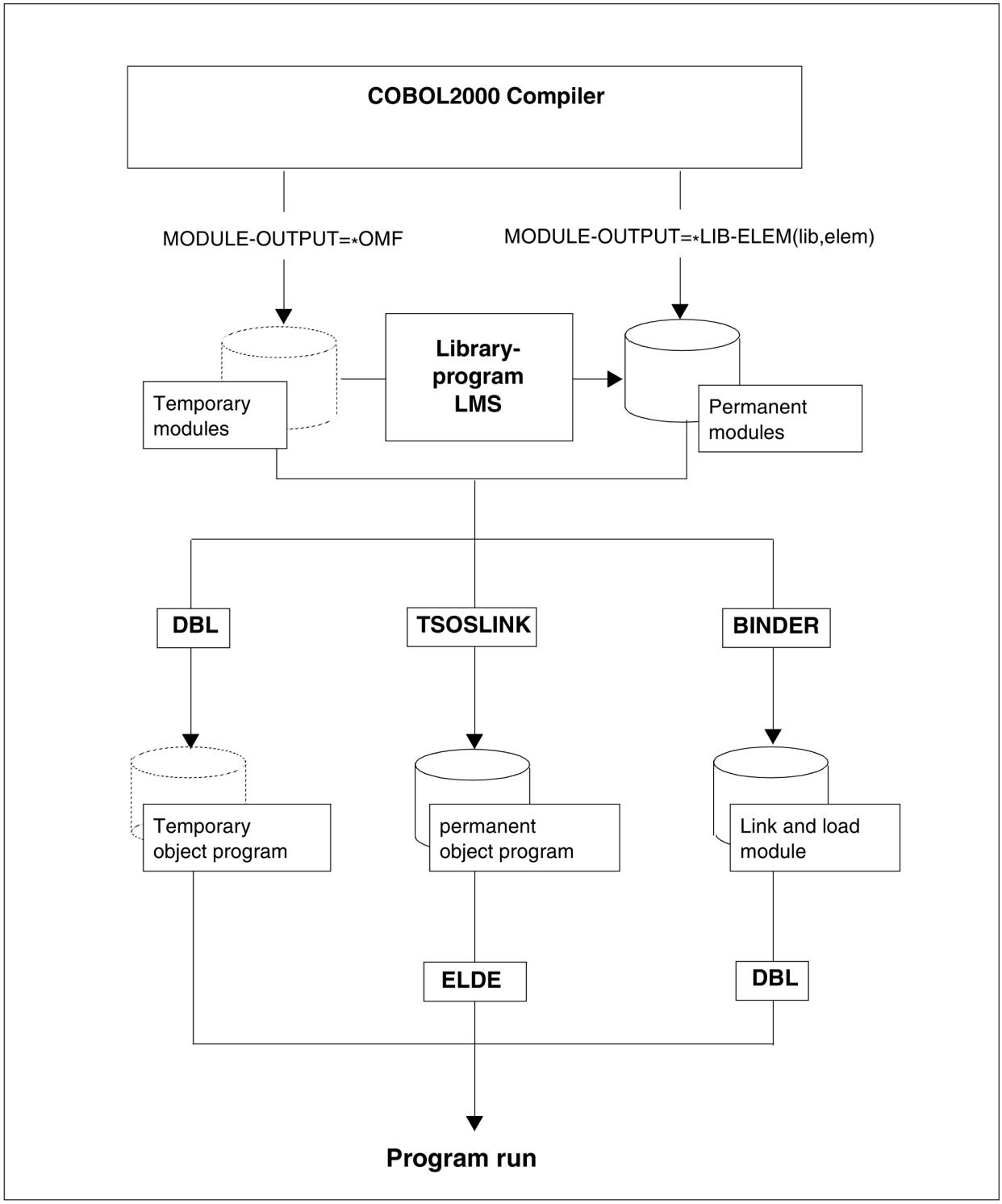


Figure 2: Generating and calling permanent and temporary executable COBOL programs in BS2000

6.2 Static linkage using TSOSLNK

The static linkage editor TSOSLNK processes one or more object modules or prelinked modules to generate either

- an executable program, which it outputs to a separate catalog file or to a PLAM library as a type C library element, or
- a prelinked module, which it stores in the temporary EAM file of the current task, or in a PLAM library as a type R library element.

The TSOSLNK utility routine is called with the START-PROGRAM command. It subsequently expects control statements from SYSDTA that specify

- for output:
  - whether the result of the linkage run is to be an executable program or a prelinked object module, and
  - where the result is to be output
- for input:
  - which object modules it should link in, and
  - which libraries should be used for the resolution of unresolved external address references.

Control statements for TSOSLNK

The TSOSLNK control statements and their operands are described in detail in the “TSOSLNK” manual [9]; the following table merely provides an overview of the most important aspects.

Statement	Short description
PROGRAM PROG	instructs the linkage editor to generate a program from the read object modules, and specifies its characteristics and output location (PLAM library or cataloged file). Among the operands which can be specified are the following: • SYMTEST=MAP or SYMTEST=ALL allows the user to use symbolic names from the compilation unit when debugging with the AID debugger. This is provided that an appropriate control statement was issued to COBOL2000 during compilation in order to have LSD information generated. • SYMTEST=ALL instructs the linkage editor to pass this information on to the program, whereas SYMTEST=MAP allows LSD information from the object module to be loaded dynamically during debugging (see “AID” manual [8] for further information). • LOADPT= * XS defines the load address of the program in the address space above 16 Mbytes. This specification is not possible unless only the XS runtime system and object modules that can be loaded into the upper address space are being linked. • ENTRY/START=entry-point specifies the starting point of the program run. This specification becomes necessary when the COBOL main program is not the first program to be linked into an executable program. entry-point is then the PROGRAM-ID name (abbreviated to 7 positions, if necessary) followed by the suffix “$”. The PROGRAM and MODULE statements (see below) are mutually exclusive.

MODULE MOD	causes the linkage editor to link the read object modules into a prelinked object module and defines its output location. The MODULE and PROGRAM statements (see above) are mutually exclusive.
INCLUDE	specifies individual object modules from which the linkage editor is to create the program or prelinked module.
RESOLVE	assigns PLAM libraries to TSOSLNK for the autolink procedure (which is described on the next page).
EXCLUDE	excludes the specified PLAM libraries from the autolink procedure (described below).
ENTRY	see ENTRY or START operand of the PROGRAM statement.
END	signals the end of the input of linkage editor statements.

Table 9: Control statements for TSOSLNK

TSOSLNK autolink procedure

On finding external address references (in a generated module) that cannot be resolved by modules specified in INCLUDE statements, TSOSLNK proceeds according to the following **autolink procedure**:

1. TSOSLNK first checks whether a library containing a corresponding module was explicitly assigned to the external reference by means of a RESOLVE statement.
2. If TSOSLNK cannot resolve the external reference in the first step, it searches all libraries specified in RESOLVE statements. Libraries can be excluded from this search by means of EXCLUDE statements.
3. If TSOSLNK does not succeed in resolving the external reference in the second step, it searches the library TASKLIB, provided that this has not been prevented by the NCAL statement or a corresponding EXCLUDE statement.
   If there is no file named TASKLIB listed under the user ID of the current task, TSOSLNK uses the system library $.TASKLIB.

If unresolved external references are present even after the autolink procedure, TSOSLNK outputs a list of their names to SYSOUT and SYSLST.

Example 6-1

Static link-editing into an executable program

```
/START-PROGRAM FROM-FILE = $TSOSLNK ———————————————————————— (1)
 % BLS0500 PROGRAM 'TSOSLNK', VERSION 'V21.0E02' OF '1999-03-15' LOADED
 % BLS0552 ...
*PROG COBOLPROG,LIB=PLAM.LIB,ELEM=COBOLLAD ———————————————— (2)
*INCLUDE COBOLMOD,PLAM.LIB ———————————————————————————— (3)
*RESOLVE ,$.SYSLNK.CRTE ——————————————————————————————— (4)
*END ——————————————————————————————————————————————— (5)
 % LNK0500 PROG BOUND
 % LNK0506 PROGRAM LIBRARY : PLAM.LIB
 % LNK0507 PHASE WRITTEN TO ELEMENT 'COBOLLAD'
```

(1) The utility routine TSOSLNK is called.

(2) The PROG statement specifies that TSOSLNK is to generate an executable program whose program name is COBOLPROG, and store it as library element COBOLLAD in the PLAM library PLAM.LIB.

(3) The INCLUDE statement informs the linkage editor that it is to link object module COBOLMOD from the PLAM library PLAM.LIB.

(4) TSOSLNK is to initially resolve external references with modules from the runtime system, which is cataloged on this system under the name $.SYSLNK.CRTE.

(5) END terminates the input of control statements and starts the linkage process; after its completion, TSOSLNK provides information on the program that has been generated.

Linking segmented programs with overlay structure

By using appropriate COBOL language elements (see the “COBOL2000 Reference Manual” [1]), the compiler can be made to output the machine code for a compilation unit in parts, i.e. in the form of a number of object modules rather than just one. This procedure is known as **segmentation**. The program sections created in the process are called **segments**.

An overlay structure can be defined during link-editing of a segmented program (see “TSOSLNK” manual [9]):

With the exception of the root segment, which remains in main memory for the entire program run, it is possible to have the individual segments overlay-loaded under program control whenever they are necessary for execution. Segments can overlay one another under these circumstances, i.e. the segments can occupy a common memory area one after the other. Which segments can overlay one another is determined by means of control statements during linkage-editing of the program.
However, since the Executive of BS2000 automatically subdivides the object program into pages (i.e. 4096-byte sections) and only loads pages into main memory as and when required during execution of the program, segmentation to relieve the load on main memory is not necessary in BS2000. It only becomes essential if virtual storage is not large enough to accept the entire program, including data. For this reason, it is neither practical nor possible to define a true overlay structure for programs that are intended to run in the upper address space.

Overlay structures for segmented programs can be defined with the following TSOSLNK statements.

Statement	Short description
OVERLAY	determines the overlay structure for the program: The OVERLAY statements of a linkage-editor run define • which segments can overlay one another and • at which locations in the object program they are to mutually overlap. OVERLAY statements are only permitted during linkage editing of a load module (PROGRAM statement); in the case of a prelinked object module (MODULE statement), they are rejected with an error message. In the address space above 16 Mbytes, (LOADPT=*XS entry in the PROGRAM or OVERLAY statement) no true overlay structures are possible; although the linkage editor will accept the OVERLAY statement, it will arrange the segments sequentially on the basis of their addresses.
TRAITS	defines for a program segment that it • should be aligned on page boundaries during loading, • can only be read during the program run. (READONLY=Y specified)

6.3 Linking using BINDER

Using BINDER, object modules and LLMs can be linked into a single link-and-load module (LLM) and stored as a type-L element in a PLAM library. BINDER is described in detail in the “BINDER” manual [22]).

> **Important Note**:
> LLMs with a linked runtime system must not be stored in libraries
>
> - from which other LLMs that are not prelinked are also to be directly loaded or
> - which are used by BINDER to resolve external references by means of AUTOLINK.

Example 6-2

Generating an LLM from object modules

```
/START-BINDER ————————————————————————————————————— (1)
%  BND0500 ...
//START-LLM-CREATION INT-NAME=PROG, COPYRIGHT = *NONE ———————— (2)
//INCLUDE-MODULES LIB=*OMF,ELEM=MAIN ———————————————————— (3)
//INCLUDE-MODULES LIB=PLAM.BSP,ELEM=SUB ————————————————— (4)
//RESOLVE-BY-AUTOLINK LIB=$.SYSLNK.CRTE ————————————————— (5)
//SAVE-LLM LIB=PLAM.BSP,ELEM=TESTPROG ——————————————————— (6)
%  BND3101 SOME EXTERNAL REFERENCES UNRESOLVED
%  BND3102 SOME WEAK EXTERNS UNRESOLVED
%  BND1501 LLM FORMAT : '1
//END ——————————————————————————————————————————————— (7)
%  BND1101 BINDER NORMALLY TERMINATED. SEVERITY CLASS: 'UNRESOLVED
EXTERNAL'
/START-PROG *MOD(LIB=PLAM.BSP,ELEM=TESTPROG,RUN-MOD=ADVANCED) ——— (8)
%  BLS0523 ELEMENT 'TESTPROG', VERSION '@' FROM LIBRARY 'PLAM.BSP' IN
PROCESS
%  BLS0524 LLM 'TESTPROG', VERSION ' ' OF '2006-10-26:14:51:46' LOADED
```

(1) BINDER is called.

(2) The START-LLM-CREATION statement generates a new LLM in the work area with the internal name PROG. The generated LLM is subsequently stored as an L-type element in a PLAM library by means of the SAVE-LLM statement (see “SORT” manual [6]).

(3) This INCLUDE-MODULES statement specifies the name of the module containing the main program (MAIN). The module is held in the temporary EAM file (*OMF).

(4) This INCLUDE-MODULES statement specifies the name of the module containing the subprogram (SUB). The module is held in the PLAM library PLAM.LIB.

(5) The RESOLVE-BY-AUTOLINK statement specifies the name of the runtime library from which external references are to be resolved.

(6) The SAVE-LLM statement stores the generated LLM under the name TESTPROG as an L-type element in the PLAM library PLAM.LIB. The BINDER message “SOME WEAK EXTERNS UNRESOLVED” refers to the ILCS module IT0INITS. This module contains weak external references to all languages potentially provided for ILCS. Only the language COBOL2000 is involved in this example and the other references remain unresolved.

(7) The END statement terminates the BINDER run.

(8) The LLM is loaded and started.

With the INCLUDE-MODULES and RESOLVE-BY-AUTOLINK statements, LIB=*BLS-LINK may also be specified instead of the library name (LIB=library). In this case the libraries to be searched must be assigned with the link name BLSLIBnn (00 <= nn <= 99). This is done by means of the SET-FILE-LINK command before BINDER is called, e.g.:

```
/ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME=$.SYSLNK.CRTE
```

An LLM generated by means of BINDER can - provided all external references are resolved - be loaded and started using DBL with no assignment of alternative libraries:

```
START-PROGRAM *MODULE(LIB=library,ELEM=module,RUN-MODE=ADVANCED)
```

When the LLM format is generated, a CSECT is created with the name `program-name&#` and with the following entries:

`program-name`	for the start of the subprogram
`program-name&$`	for the start of the main program
`program-name&A`	for the service entry

When shared-code is generated, the code CSECT `program-name&@` is also created.

6.4 Dynamic linking and loading using DBL

The dynamic binder loader DBL links modules temporarily into a loadable unit, which it then loads into memory and executes immediately. The generated load unit is automatically deleted at the end of the program run. The mode of operation of DBL is described in detail in the “Binder-Loader-Starter” manual [10].

DBL is called implicitly by the commands START-PROGRAM and LOAD-PROGRAM. The following overview summarizes the options of the START-PROGRAM and LOAD-PROGRAM commands that are most relevant to calling DBL; a detailed description of all the available operands is provided in the “BS2000/OSD-BC Commands” manual [3].

```
{START-PROGRAM | LOAD-PROGRAM} FROM-FILE =]
*MODULE (LIBRARY={*OMF,ELEMENT=modul | *OMF [,ELEMENT=*ALL] | bibliothek,
ELEMENT=element} [,RUN-MODE = {*STD | *ADVANCED(ALT-LIB=*YES)}])
```

The START-PROGRAM command instructs DBL to generate an executable program, load it into memory, and start it. Since the program is run immediately after the command, the necessary resources (files) must be assigned to DBL before the START-PROGRAM command is given (see section “Assignment of cataloged files”).

The LOAD-PROGRAM command instructs DBL to generate an executable program and load it into memory without starting it. This makes it possible to enter additional commands prior to program execution, e.g. commands for program monitoring using a debugging aid. The program can subsequently be started by means of

- a %RESUME command, if tests are to be performed using the advanced interactive debugger (AID) or
- a RESUME-PROGRAM command in all other cases.

LIBRARY=*OMF
indicates the temporary EAM file of the current task into which the compiler has output the compiled object module.

ELEMENT=module
is the name of the module that is to be loaded first. The string “module” consists of the first eight characters of the corresponding ID name in the compilation unit. “module” can also be the ENTRY name of the program segment that is to be loaded first.

ELEMENT=*ALL
causes DBL to fetch all the modules from the EAM object module file. If this is what is desired, there is no need to specify it explicitly as this value is preset as the default.

LIBRARY=library
is the name of a PLAM library in which the module is stored as a library element. Using *LINK(LINK-NAME=linkname) it is also possible to specify a predefined file link name for the library.

ELEMENT=element
is the name of the module that is stored as a type-R element in the specified PLAM library. If there is more than one element with the same name in the library, the element with the (alphabetically) highest version designation is used.

RUN-MODE=STD
In this mode, the runtime system CRTE must be assigned as the TASKLIB by using the SET-TASKLIB command before DBL is called.
Apart from the TASKLIB and, if applicable, the library containing the modules, no other libraries can be taken into consideration during link-editing.

RUN-MODE=ADVANCED(ALTERNATE-LIBRARIES=YES)
In this mode, in order to resolve external references DBL searches up to 99 different libraries assigned by means of the link name BLSLIBnn (00 nn 99) prior to invocation of DBL.

Dynamic loading

If other external subprograms are invoked by COBOL modules via “CALL identifiers”, some additional conditions for loading and starting must be observed. See chapter “Program linkage” for more details.

6.5 Loading and starting executable programs

Before a statically linked program can execute, it has to be loaded into main memory. In BS2000 this function is performed by a static loader. This, like the dynamic binder loader, is called with the START-PROGRAM or LOAD-PROGRAM command (see the “BS2000/OSD-BC Commands” manual [3]):

- The START-PROGRAM command instructs the static loader to load the program into main memory and start it. As the program is run immediately after the command is issued, the necessary resources (files) have to be assigned to the loader first (see section “Assignment of cataloged files”).
- The LOAD-PROGRAM command instructs the static loader to load the program into main memory without starting it. This makes it possible to enter additional commands prior to program execution, e.g. commands for program monitoring using a debugging aid. The program can subsequently be started by issuing a RESUME-PROGRAM or %RESUME command.

The following overview summarizes the options of the START-PROGRAM and LOAD-PROGRAM commands that are most relevant for calling the static loader; a detailed description is provided in the “BS2000/OSD-BC Commands” manual [3].

```
{START-PROG | LOAD-PROG} FROM-FILE = {*PHASE(LIB=library,ELEM=element,VERS=version)
| filename}
```

library	specifies the name of a PLAM library in which the program generated by TSOSLNK is stored as a library element.
element	is the name of the library element in which the program is stored. It must be a type C element.
version	specifies the element version as a string of 24 characters or less.
filename	is the name of the cataloged file which contains the program generated by TSOSLNK.

6.6 Program termination

The termination action taken by a program is of special importance when it is invoked within a procedure or is monitored by a job variable.

If error messages to which an internal return code is assigned (see error message COB9119 in chapter “Messages of the COBOL2000 system”) are issued during program execution, this return code is passed to the last two bytes of the return code indicator of a monitoring job variable (see “Job Variables” manual [7]).

The following table provides an overview of

- the possible contents of the return code indicator in job variables,
- the associated error messages, and
- their impact on the further progress of a procedure.

Return code indicator 1)	**Error number** 2)	**Short description of the error**	**Continuation controllable with option** 3)	**Dump**	**Triggers spin-off in procedures** 5)
0**100**	none	No error detected by the runtime system	--	no	no
1**120**	COB9120	Job variables not available	yes		yes
1**121** 1**122**	COB9121 COB9122	End of file during ACCEPT processing	yes yes		
1**123** 1**124** 1**125** 1**126** 1**127**	COB9123 COB9124 COB9125 COB9126 COB9127	Invalid argument in a standard function	yes yes yes yes yes		
1**128**	COB9128	User return code is set	no		
1**131**	COB9131	Job variables: ACCEPT set to empty job variable	yes		
1**132**	COB9132	Wrong number of parameters (CALL)	yes		
1**133**	COB9133	Program execution in BS2000 Version< 10.0	no		
1**134**	COB9134	Sort error	yes		
2**140**	COB9140	Reference modification error	yes	no / yes4)	
2**141**	COB9141	Last XML statement not yet processed	no		
2**142**	COB9142	GO TO has no ALTER	no		
2**143**	COB9143	Purge date for the volume has not yet expired	no		
2**144**	COB9144	Table: Subscript/index range violation	yes		
2**145**	COB9145	Table (with DEPENDING ON element): Subscript/index range violation	yes		
2**146**	COB9146	COBOL2000 runtime system in CRTE is incompatible with the object program	no		
2**148**	COB9148	CALL or ADDRESS OF PROGRAM not executable	no		
2**149**	COB9149	Incompatible data in numeric edited item	no		
2**151**	COB9151	Files: Undetected I/O error (no USE procedure, no INVALID KEY, no AT END)	no		

21**52**	COB9152	Connection to the database could not be established	no		
21**53**	COB9153	Error while converting EBCDIC to UTF-16	yes		
21**54**	COB9154	REPORT WRITER: user error	no		
21**55**	COB9155	Error on exit from a USE procedure	no		
21**56**	COB9156	DML: SUB-SCHEMA module too small for processing an extensive DML statement	no		
21**57**	COB9157	CALL not executable	no		
21**58**	COB9158	More than 9 recursive calls to DEPENDING paragraphs	no		
21**59**	COB9159	Error while quitting an XML PROCESSING procedure	no		
21**60**	COB9160	Runtime unit uses CANCEL, but contains programs compiled with a COBOL85 compiler < V2.0	no		
21**62**	COB9162	The attributes of an external file are not consistent within the programs of a runtime unit	no		
21**63**	COB9163	The storage space for DYNAMIC data could not be set up	no		
21**64**	COB9164	Program called with CALL is not available	no		
21**65** 21**66** 21**67**	COB9165 COB9166 COB9167	Invalid call or invalid exit from USE procedures	no		
21**68** 21**69** 21**71**	COB9168 COB9169 COB9171	REPORT WRITER: user error	no no no		
21**73**	COB9173	SORT run not successful	no		
21**74** 21**75**	COB9174 COB9175	Error handling in the program: user error	no no		
21**76**	COB9176	REPORT WRITER: user error	no		
21**78**	COB9178	Record to be sorted does not match SD description	no		
21**79**	COB9179	Sorted record does not match GIVING file description	no		
21**80**	COB9180	RELEASE/RETURN not under the control of SORT/MERGE	no		
21**81**	COB9181	DATABASE-HANDLER has not yet finished processing the last DML statement	no		
21**82**	COB9182	Invalid inheritance of classes or interfaces	no		
21**84**	COB9184	SORT within the SORT controller	no		
21**85**	COB9185	Error in connection with OO language elements	no		
21**88**	COB9188	XML parser not found	no		
21**89**	COB9189	PARTIAL-BIND runtime system not found	no		
31**91**	COB9191	SUPER class not found	no	yes	
31**92**	COB9192	The end of the program was reached but neither STOP RUN nor EXIT PROGRAM was executed	no		
31**93**	COB9193	DISPLAY error	no		
31**94**	COB9194	Error during input from SYSDATA	no		
31**95**	COB9195	Error during output to SYSLST	no		
31**96**	COB9196	ACCEPT or DISPLAY statement error at the runtime system/operating system interface	no		
31**97**	COB9197	Job variables: access failed	yes		

31**98**	COB9198	Hardware interrupt	no
31**99**	none	WROUT error: No further messages can be output	no

Table 10: Return code indicators in job variables

1) The first digit indicates the weight of the message (0: note, 1: warning, 2: error, 3: fatal error).
The second digit (always 1) identifies the program as a COBOL object.
The final two digits (in bold print) represent the internal return code.

2) For content and meaning of messages see chapter “Messages of the COBOL2000 system”.

3) Program abortion can be induced with RUNTIME-OPTIONS=PAR(ERROR-REACTION = TERMINATION) or COMOPT CONTINUE-AFTER-MESSAGE=NO. After the program has been aborted, the associated return code is set in the job variable monitoring the program.

4) Batch processing: no
Interactive processing: query yes/no

5) When a spin-off is triggered, all subsequent commands are ignored with the exception of the SET-JOB-STEP, EXIT-JOB, LOGOFF, CANCEL-PROCEDURE, END-PROCEDURE and EXIT-PROCEDURE commands. The SET-JOB-STEP command terminates the spin-off, and processing is continued with the next command.

6.7 Shareable COBOL programs

In large programs it may be advantageous to make individual program segments shareable if they are to be accessed by several users (tasks).

For this, the following control statement must be specified at compilation time:

COMOPT GENERATE-SHARED-CODE=YES

or

SHAREABLE-CODE=YES
in the MODULE-GENERATION parameter of the COMPILER-ACTION option

The compiler then generates two object modules, one of which contains the nonshareable section and the other the shareable section of the object. These are referred to in the following as the “nonshareable” and “shareable” module, respectively. The shareable and nonshareable modules can themselves be linked into prelinked modules.

The shareable modules must be stored in a PLAM library either directly by the compiler (via a COMOPT MODULE statement or the SDF option MODULE-LIBRARY) or by means of the LMS utility routine (see “LMS” manual [11]).

All nonshareable sections of a program are loaded separately for each task and user into class 6 memory.
Program systems with shareable modules can only be called using DBL. The call always uses the name of the nonshareable (data) module. This contains external references to its shareable code module as well as to any other nonshareable modules.

Sample call:

```
/SET-TASKLIB $.SYSLNK.CRTE ————————————————————————————(1)
/START-PROGRAM *MOD(library,element) ——————————————————(2)
```

(1) The SET-TASKLIB command is used to assign the library that contains the COBOL runtime system.

(2) element is the name of the data module or prelinked module which must contain at least the nonshareable section of the main program. library is the library containing the user-written modules.

The following figure illustrates program runs with and without shared code.

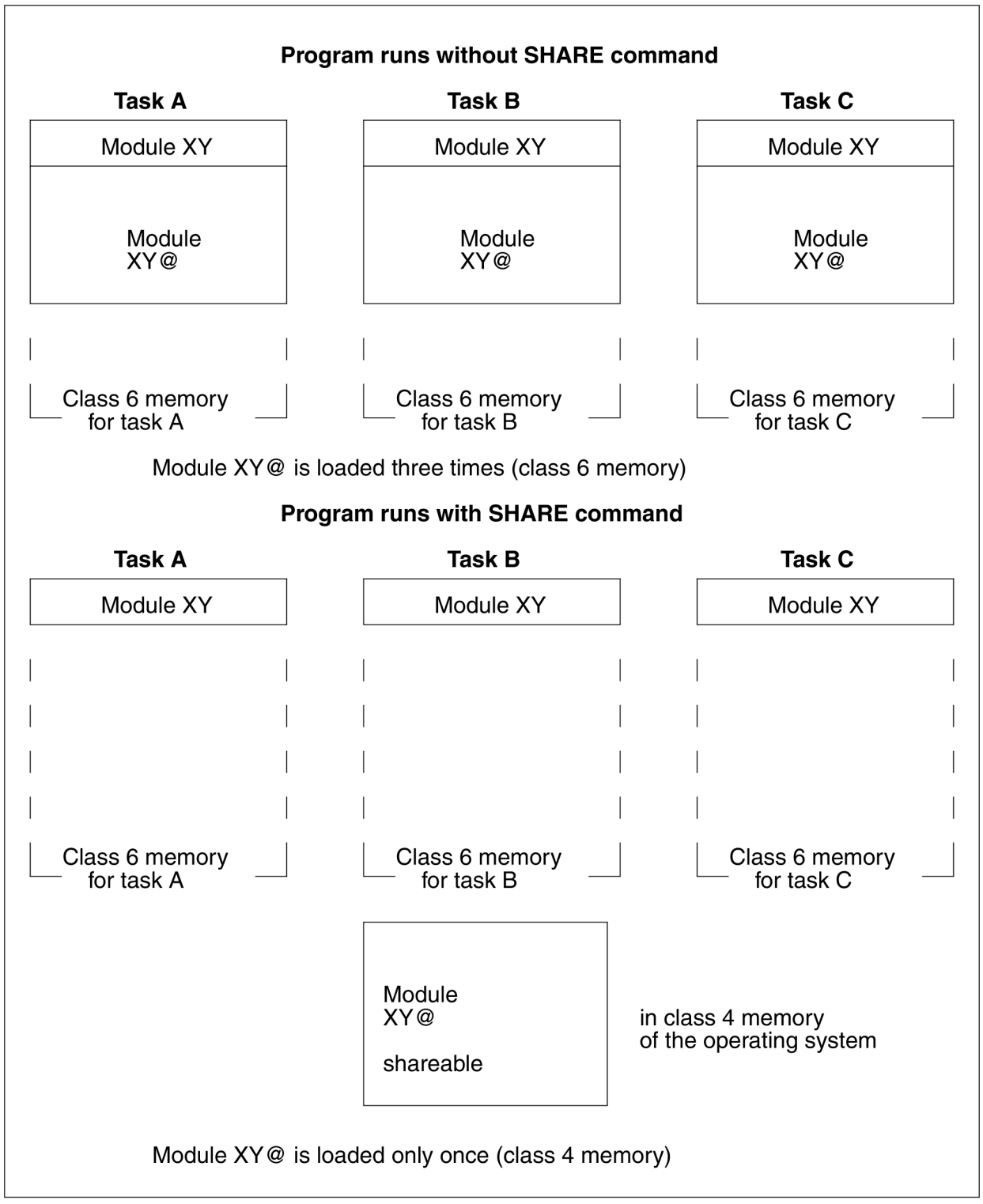


Figure 3: Shared code

7 Debugging aids for program execution

Even a syntactically correct COBOL program may still have logic errors and therefore not run as intended. A number of different aids are available to the COBOL programmer for detecting and correcting such errors:

- The programmer can use the **A**dvanced **I**nteractive **D**ebugger (AID) during program execution. This requires no special programming provisions and permits errors to be detected and corrective action to be taken while the loaded program is being executed.
- Debugging lines can be inserted in the compilation unit and activated by the programmer as the need arises. This assumes that potential error conditions were anticipated and provided for when the compilation unit was written. The diagnosis of an unexpected error can therefore make it necessary to modify or add debugging lines and then recompile the entire compilation unit. Debugging lines are described in “COBOL2000 Reference Manual” [1] and in section “Debugging lines”.

The debugging aids can be used analogously in the POSIX subsystem (see the chapter “COBOL2000 and POSIX”).

7.1 Advanced Interactive Debugger (AID)

Not supported in the COBOL2000-BC !

Only a short introduction to AID is given in this User Guide. For a detailed description of this debugger, refer to “AID” manuals [8], [20] and [21]. Knowledge of the information in the “AID” manual [8] is required here.

AID has the following features:

1. It makes it possible to test “symbolically”, i.e. to specify symbolic names from the compilation unit in commands rather than absolute addresses. For this purpose, the required LSD information must be generated at compile time and passed to the loaded program at a later stage (see section “Symbolic debugging with AID”).

   However, with respect to the program in its entirety, it is not always necessary to load this information together with the program. Instead, AID allows LSD information to be dynamically loaded for each compilation unit, provided the associated modules (with LSD information) reside in a PLAM library. In this way, more efficient use of resources is achieved:

   - Program memory space is saved, as LSD information has to be loaded only when it is required for debugging (memory space for a program increases by about a factor of 5 if the information is loaded at the same time as the program).
   - A program that runs without errors at debugging time does not have to be recompiled (without LSD information) or relinked for the production run.
   - When the results of a production run make a test run desirable, the necessary LSD information is available and can be used without any need for the program to be recompiled and relinked.
2. It provides functions permitting
   - program execution to be traced at symbolic level and logged (TRACE function)
   - program execution to be interrupted at specified points or when defined events occur, in order to initiate AID or BS2000 commands (referred to as “subcommands”)
   - a section or a paragraph in the PROCEDURE DIVISION to be specified after a program interrupt. Debugging is continued with the section or paragraph specified, irrespective of the coded program logic (%JUMP statement (see “AID” [8])); this is possible only if the program was compiled with PREPARE-FOR-JUMPS=YES in the AID parameter of the TEST-SUPPORT option or using COMOPT SEPARATE-TESTPOINTS=YES (see section “TEST-SUPPORT option” and section “Table of COMOPT operands” respectively)
   - the contents of fields to be output in a form that takes account of the data definitions of the compilation unit
   - the contents of fields to be changed, with AID performing the necessary moves according to the specifications of the COBOL MOVE statement.
3. It supports the analysis of dumps in disk files as well as the diagnosis of loaded programs.
4. It can be used in batch mode as well as in interactive mode. For program testing, however, interactive mode is recommended as it does not require the sequence of the commands to be defined in advance and allows this sequence to be tailored to suit the current debugging environment.

7.1.1 Conditions for symbolic debugging

For debugging at a symbolic level, AID permits data items, sections, and paragraphs to be addressed using the names defined in the compilation unit. It also permits statement lines and individual COBOL verbs in the Procedure Division to be referenced. Consequently, AID must be provided with the appropriate information on these symbolic names. This information can be subdivided into two parts:

- the List for Symbolic Debugging (LSD), in which the symbolic names and statements defined in the module are cataloged, and
- the External Symbol Dictionary (ESD), which records a module's external references.

Generation or transfer of this information is initiated or suppressed by means of appropriate operands in the call command or control statement at each of the following stages:

- compilation with COBOL2000
- linking and loading with the dynamic binder loader (DBL) or
- linking with the static linkage editor (TSOSLNK) and
- loading with the static loader (ELDE)

The ESD information is generated and transferred as standard, whereas the LSD information can be made accessible to AID in two ways. After it has been generated at compile time, this information can be:

- loaded together with the entire program, or
- dynamically loaded for each compilation unit as necessary, provided the associated object modules are available in a PLAM library.

For each of these cases, the following table provides an overview of the operands that need to be specified in order to generate and transfer the LSD information.

Stages in the development of the program	**Operands to be specified**	
	if the LSD information is to be loaded jointly with the entire program	if the LSD information is to be dynamically loaded by AID at a later stage [1)]
Compile with COBOL2000[2)]	TEST-SUPPORT=AID() or COMOPT SYMTEST=ALL	TEST-SUPPORT=AID() or COMOPT SYMTEST=ALL
Link and load with the dynamic binder loader	LOAD-PROGRAM ..., TEST-OPTIONS=AID or START-PROGRAM ..., TEST-OPTIONS=AID	LOAD-PROGRAM ..., [TEST-OPTIONS=NONE] or START-PROGRAM ..., [TEST-OPTIONS=NONE]
Link with TSOSLNK	PROGRAM...,SYMTEST=ALL	PROGRAM...[,SYMTEST=MAP]
Load or load and start with the static loader	LOAD-PROGRAM ..., TEST-OPTIONS=AID or START-PROGRAM ..., TEST-OPTIONS=AID	LOAD-PROGRAM ..., [TEST-OPTIONS=NONE] or START-PROGRAM ..., [TEST-OPTIONS=NONE]

Table 11: Operands for generating LSD information

1) This is possible only if the associated modules reside in a PLAM library.

2) If the COMOPT GEN-SHARE=YES or the SDF option SHARE-CODE=YES is specified, only statements from the code or data module are listed for the trace when debugging.

7.1.2 Symbolic debugging with AID

Symbolic debugging with AID permits data items, compilation units, sections, and paragraphs to be addressed using the names defined in the source text.

However, in order to reference a line in the Procedure Division, the programmer must specify a name in the form

- S'n' (for a line with a section or paragraph name) or
- S'nverbm' (for a line with COBOL verbs).

Such an **LSD name** is created by COBOL2000 for each line in the Procedure Division and for each COBOL verb in a statement line (see Example 7-1). Its components have the following meaning:

n	is the number (5 digits at most) of the line in the Procedure Division. The number, which is assigned by COBOL2000 at compile time, must be specified without leading zeros. If the sequence number of the compilation unit (max. 6 positions) is to be used as the line number, then this must be requested by the user with the SDF operand STMT-REFERENCE=COLUMN-1-TO-6 in the TEST-SUPPORT option or with COMOPT TEST-WITH-COLUMN1.
verb	is the predefined abbreviation of a COBOL verb in the line concerned. A list of the abbreviations is given below.
m	is a one- or two digit number specifying which of several identical verbs within a line n is to be indicated.If k is equal to 1, it is omitted.

Example 7-1

Creation of LSD names

```
000026              IF A = B MOVE A TO D MOVE B TO E.
```

In this statement line

- the first verb has the LSD name S'26IF',
- the second verb has the LSD name S'26MOV', and
- the third verb has the LSD name S'26MOV2'.

A detailed example explaining how a COBOL program can be debugged with AID is provided in the "AID" [8].

List of COBOL verbs and their abbreviations:

ACC	ACCEPT	INI	INITIATE
ADD	ADD	INSP	INSPECT
ADDC	ADD CORRESPONDING	INV	INVOKE
ALLO	ALLOCATE	KEE	KEEP
ALT	ALTER	MOD	MODIFY
CALL	CALL	MOV	MOVE
CANC	CANCEL	MOVC	MOVE CORRESPONDING
CLO	CLOSE	MRG	MERGE

COM	COMPUTE	MUL	MULTIPLY
CON	CONNECT	OPE	OPEN
CONT	CONTINUE	PER	PERFORM oder EXIT PERFORM
DEL	DELETE		end of main part of loop [2]
DIS	DISPLAY	PERT	TEST OF PERFORM
DIV	DIVIDE	RAIS	RAISE
DSC	DISCONNECT	REA	READ
END	END-xxx [1] [2]	REDY	READY
ENTR	ENTRY	REL	RELEASE
ERA	ERASE	RES	RESUME
EVAL	EVALUATE	RET	RETURN
EXI	EXIT	REW	REWRITE
EXI	EXIT PARAGRAPH	SEA	SEARCH
EXI	EXIT SECTION	SET	SET
EXIT	EXIT METHOD	SOR	SORT
EXIT	EXIT PROGRAM	STA	START
FET	FETCH	STO	STOP
FIN	FINISH	STOR	STORE
FND	FIND	STRG	STRING
FRE	FREE	SUB	SUBTRACT
GEN	GENERATE	SUBC	SUBTRACT CORRESPONDING
GET	GET	TER	TERMINATE
GO	GOBACK	UNST	UNSTRING
GOT	GO TO	WRI	WRITE
IF	IF	XML	XML
INIT	INITIALIZE		

1) Explicit scope terminator (e.g. END-ADD)

2) The point at which END is to stop comes after the scope terminator; for ENDPERFORM, in particular, this point comes after the PERFORM has been completed. A further stopping point exists prior to END-PERFORM, after a single loop. You can use PER to address this second stopping point.

7.1.3 Predefined information

Information about the object being debugged

You can use the AID command

```
%D[ISPLAY] {_COMPILER | _COMPILATION_DATE | _COMPILATION_TIME | _PROGRAM_NAME}
```

to call up general information on the object being debugged:

`_Compiler`	the compiler that compiled the object
`_Compilation_Date`	the date of compilation
`_Compilation_Time`	the time of compilation
`_Program_Name`	ID name of the object
`_EBCDIC_CCSN`	Name of the EBCDIC variant which is assumed in the case of conversions between alphanumeric and national data

Information about the exception condition

You can use the AID command

```
%D[ISPLAY] _LAST_EXCEPTION
```

to request general information on the last exception condition.

Format of the output:

```
01 _LAST-EXCEPTION.
   02 _EXCEPTION_NAME PIC X(31).
```

`_EXCEPTION_NAME`	Name of the exception condition which has led to the exception condition (blank if no exception condition exists).

7.1.4 Notes on symbolic debugging of nested programs

- Setting test points
  - Paragraphs and sections of the contained program in which the interrupt point lies can be referenced without qualification.
  - Sections and paragraphs in a different program, which may also lie in a different compilation unit, are accessed via the S and PROC qualification:

    %INSERT [S=program-id.]PROC=program-id-contained.paragraph [IN section]
  - The S qualification must be specified whenever the test point is to be set in a different, separately compiled program.
  - A test point at the start of the Procedure Division of the outermost containing program can be set by means of a PROG qualification:

    %INSERT PROG=program-id.program-id

    or written out in full:

    %INSERT S=program-id.PROC=program-id.program-id

    This method is only meaningful if the program-id does not exceed 8 characters or if an LLM was generated, since otherwise the source name, but not the procedure name, would be truncated to 8 characters.
  - It is not possible to set a test point at the start of a contained program by using a PROG qualification, since S and PROC are different. This can, however, be achieved as follows:

    %INSERT [S=program-id.]PROC=program-id-contained.program-id-contained
  - Names that are unique in the current compilation unit can also be addressed without any qualification.
- Accessing data
  - %D locates the data of the current nested program and also data having the GLOBAL attribute that is not locally concealed, i.e. it is possible to access the same data that the program itself can also access at this point.
  - %SD can be used to give the data of all the surrounding programs, in accordance with the current call hierarchy.
  - The PROC qualification can be used to specifically access one item of data from a different program.

    %D PROC=program-id-contained.data-item

    %SD is also possible here instead of %D provided the item of data lies in a calling program.
- Depending on how the program is nested, the PROC qualification can be repeated more than once when accessing both test points and data.
- The %TRACE command logs all statements of the current CSECT, i.e. including all statements of the called contained programs, but not including the statements in separately compiled programs.
- If the statement types are indicated in the trace, additional LABEL specifications are occasionally reported by AID on account of internally generated paragraphs.

7.1.5 Notes on debugging object-oriented COBOL programs

- Addressing
  - **Classes** are addressed by a source qualification: S=<class>, where <class> is the name specified in the CLASS-ID paragraph.
  - **Methods** are addressed by a procedure qualification:
    PROC={FACTORY | OBJECT}.PROC=<method>, where <method> is the name specified in the METHOD-ID paragraph.

    A source qualification is required whenever the current program location is not in (a method of) the class. Procedure qualifications are only needed to the extent required for unique identification. Consequently, PROC={FACTORY | OBJECT} can always be dropped for methods, since the method name must be unique in the class.
- Commands
  - ***Setting test points***

    Test points can be set in methods by using a source and procedure qualification:

    %INSERT [S=<class>.] [PROC=<method>.] srcref

    Write monitoring can be set on an object reference with:
    %ON %WRITE(objref)
    However, an object reference modified by NEW can only be displayed after returning to the calling point.
  - ***Tracing***

    Classes and methods can be specified as the trace area with %TRACE as follows:

    %TRACE <n> IN S=<class>.[PROC={FACTORY | OBJECT}.PROC=<method>]
  - ***Displaying data***

    %DISPLAY

    The data of an object is only visible if the interrupt point lies in a method of that object. No qualification is specified in such cases.

    The data in a method is only visible within that method.

    An object reference is displayed as follows:

    ```
    <level> objref
            <level+1> FACTORY | OBJECT | NULL
            <level+1> class-name
    ```

    The first component indicates whether the reference points to the factory object or a normal object or whether a null reference is involved. The second component shows the class name of the currently referenced object and is dropped for null references.

    %SD

    %SD shows the data in the current dynamic call hierarchy of programs and methods. In the case of methods, only the local data of the method is displayed, not the data of the surrounding object.
    In addition, the global data for a source module such as the _COMPILATION_DATE, for example, is output per class.
  - ***Editing data***

    %SET, %MOVE

High-level assignments to object references are rejected by AID with an error message (Types are not convertible...). Low-level access to object references is possible, but entirely at the user’s own risk.

7.1.6 Information on testing programs with user-defined types

AID V3.1A supports the TYPEDEF clause and type-specific pointers in COBOL2000.

A dereferencing operator and an address operator now supplement the familiar AID operators (refer to the “AID” manual [8]).

The dereferencing operator is used to access the date addressed by a pointer. It is represented by an asterisk and can be combined with the COBOL qualification (IN, OF) and the COBOL subscripting.

The address operator supplies the address of a date for providing a value of a pointer or for further use in low-level AID. For this purpose AID supports the COBOL syntax ADDRESS OF.

- **Access to data names**

  The data names of the TYPEDEF clause are not used in the AID command.

  - The input of simple, qualified and indexed symbols in AID takes place in the same way as in COBOL. This means that partial qualifications in particular are permitted provided they are unambiguous. For performance reasons it is advisable to qualify an input symbol completely for programs with a very large number of groups (structures). This expedites the search process and obviates the need for a uniqueness test. Complete qualification is always required for symbols with a dereferenced component (e.g. %D NAME IN *ADDRESS-START). Furthermore, index information must always be specified exactly.
  - The following rules apply for complex data accesses arising from the combination of qualification, subscripting and dereferencing:
    - Processing always takes place from right to left. The operator on the farthest right is processed first.
    - If an operand is grouped with an operator, the operator in parentheses has priority when processing from right to left.

- **Adress selector**

  Just as in COBOL, the keyword of the address selector is **ADDRESS OF**. It is reserved and does **not** apply in the setting %AID SYMCHARS=NOSTD.

- **Assignments and comparisons**

  - Assignments and comparisons of variables with the same TYPEDEF clause without a STRONG specification can, similarly to groups, only be performed at low level, i.e. through explicit conversion of the groups to hexadecimal strings.
  - In the case of assignments and comparisons of variables with the same TYPEDEF clause and with a STRONG specification, explicit conversion to hexadecimal strings in the AID command entry is not required. AID checks whether the source and target have the same[1] TYPEDEF clause with the STRONG specification and then carries out the assignment or comparison. However, during execution the string is converted internally to the variable as a whole and not to the individual components.
  - In the case of assignments and comparisons of type-specific pointers, a check is made to see whether the pointers have the same reference type. If the reference type is a group (structure) with the TYPEDEF clause, the STRONG specification is also required in the declaration of the type. If the pointer is assigned an address via the address selector, or if a pointer is compared with an address selector, the analogous type check is performed between the reference type of the pointer and the argument type of the address selector.

Example 7-2

```
01 PT-TYP TYPEDEF USAGE POINTER TO PERSON.
01 PERSON TYPEDEF STRONG.
   02 NAME PIC X(30).
```

```
   02 VORNAME PIC X(30).
01 VERWEIS TYPE PT-TYP.
01 PERS1 TYPE PERSON.
01 PERS2 TYPE PERSON.
```

Possible entry in AID:

```
%SET ADDRESS OF PERS1 INTO VERWEIS.
%D *VERWEIS
%D NAME IN *VERWEIS                              (1)
%D ADDRESS OF PERS1                              (2)
%SET PERS1 INTO PERS2                            (3)
```

(1) Shows the content of the data item to which VERWEIS refers as a group of the type PERSON or the element field NAME in this group.

(2) Specifies the address of PERS1 (in hexadecimal format) for further use in low-level AID.

(3) In COBOL corresponds to: MOVE PERS1 TO PERS2.

[1] Please note: AID does not treat equivalent types as “identical” types (see the “COBOL2000 Reference Manual” [1]).

7.2 Debugging lines

At the compilation unit level, COBOL2000 offers debugging lines for the diagnosis of logical errors. These are specially identified lines in the compilation unit which

- contain only COBOL statements for test purposes and
- at compile time can be treated as statement lines or comment lines, as necessary.

COBOL2000 supports the use of debugging lines with the following language elements (see “COBOL2000 Reference Manual” [1]):

- The WITH DEBUGGING MODE clause in the SOURCE-COMPUTER paragraph of the ENVIRONMENT DIVISION:

  This clause defines how debugging lines are to be treated by the compiler: If the clause is specified, debugging lines are compiled as normal statement lines; if it is not specified, the compiler treats debugging lines as comment lines.

  This feature allows debugging lines to be left untouched in the compilation unit after the test phase. Only the WITH DEBUGGING MODE clause has to be removed before the program is compiled for productive use.
- The identification of debugging lines by means of a “D” in the indicator area (column 7): A “D” in column 7 of a line specifies that the line is to be treated either as a statement line or as a comment line by the compiler, depending on whether or not the WITH DEBUGGING MODE clause is present.

  When defining debugging lines, the following should be noted:

  - In the compilation unit, debugging lines are permitted only after the OBJECT-COMPUTER paragraph.
  - The COBOL compilation unit must be syntactically correct, regardless of whether or not the debugging lines are taken into account.
  - Debugging lines are only permitted in fixed format.

8 Interface between COBOL programs and BS2000/OSD

The interface between COBOL programs and the POSIX subsystem is described in the chapter “COBOL2000 and POSIX”.

8.1 Input/output via system files

System files are standardized input/output areas of the system to which particular terminals or files can be assigned. They are available to any task and require no prior declaration.
They include

- the logical input files of the operating system SYSDTA and SYSIPT
- the logical output files of the operating system
  SYSOUT, SYSLST, SYSLSTnn (nn = 01...99) and SYSOPT

8.1.1 COBOL language elements

COBOL programs can use system files to input or output low-volume data (e.g. control statements). COBOL2000 supports access to system files and the console with the following language elements (see “COBOL2000 Reference Manual” [1]):

- Program-internal mnemonic names for system files, declared in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION:
  PROCEDURE DIVISION statements can reference the assigned system files via these mnemonic names (see below). Among other things, mnemonic names can be declared for the following files:
  - input files:

SYSDTA	with TERMINAL IS mnemonic-name
SYSIPT	with SYSIPT IS mnemonic-name

  - output files:

SYSOUT	with TERMINAL IS mnemonic-name
SYSLST	with PRINTER IS mnemonic-name
SYSLSTnn	with PRINTERnn IS mnemonic-name (nn = 01...99)
SYSOPT	with SYSOPT IS mnemonic-name

- The statements ACCEPT, DISPLAY and STOP literal of the PROCEDURE DIVISION:

  These access system files or the console according to the following rules:
  - ACCEPT...FROM mnemonic-name

    reads data from the **input file** that is associated (in the SPECIAL-NAMES paragraph) with mnemonic-name

    This causes the data to be transferred left-justified to the receiving item specified in the ACCEPT statement, its length being determined by this item as follows:
    If the item is longer than the value to be transferred, it is padded with spaces on the right; if it is shorter, the value is truncated on the right during the transfer to conform to the length of the item.

    If the input file has record format F (fixed-length records, see section “System files: primary assignments, reassignments, record formats”), the following also applies:
    If the length of the receiving item of the ACCEPT statement is greater than the logical record length of the system file, additional data is automatically requested, i.e. additional read operations (macro calls) are initiated.

    If the program detects the end-of-file condition while reading the system file, it issues message COB9121 or COB9122.
    Depending on the COMOPT operand CONTINUE-AFTER-MESSAGE or ERROR-REACTION in the RUNTIME-OPTIONS option (SDF), the program run is subsequently continued (default) or terminated.

    When the program run is continued the string “/*” is stored in the first two positions of the receiving item (“/” is stored if the receiving item is only one character long) and processing continues with the statement following ACCEPT.
  - ACCEPT (without FROM phrase)

    reads data by default from the system input file SYSIPT.

    With COMOPT REDIRECT-ACCEPT-DISPLAY=YES or
    ACCEPT-DISPLAY-ASSGN= *TERMINAL in the SDF option RUNTIME-OPTIONS, it is possible to switch the assignment to system file SYSDTA.
  - DISPLAY...UPON mnemonic-name

writes data into the **output file** that is associated (in the SPECIAL-NAMES paragraph) with mnemonic-name.

The size of the data transfer is determined by the length of the sending items orliterals specified in the DISPLAY statement:
If the total number of characters to be transferred is greater than the maximum record length for the output file (see table 14 in section “System files: primary assignments, reassignments, record formats”), additional records are output until all characters are transferred. In the case of files with fixed-length records, if the number of characters is smaller than the record length, the records are space-filled on the right.

- DISPLAY (without UPON phrase)

  writes data by default to the system output file SYSLST.

  With COMOPT REDIRECT-ACCEPT-DISPLAY=YES or
  ACCEPT-DISPLAY-ASSGN= *TERMINAL in the SDF option RUNTIME-OPTIONS, it is possible to switch the assignment to system file SYSOUT.

- STOP literal

  outputs a literal (with a maximum length of 122 characters) on the **console**.

Example 8-1

Accessing a system file via a declared mnemonic name

```
IDENTIFICATION DIVISION.
    ...
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
    ...
SPECIAL-NAMES.
    SYSIPT IS SYS-INPUT ──────────────────────────────────────── (1)

    ...
PROCEDURE DIVISION.
    ...
    ACCEPT CONTROL-FIELD FROM SYS-INPUT.──────────────────────── (2)

    ...
```

(1) The program-internal mnemonic name SYS-INPUT is declared for the system file SYSIPT.

(2) ACCEPT reads (via the mnemonic name SYS-INPUT) a value from SYSIPT into the item CONTROL-FIELD.

8.1.2 System files: primary assignments, reassignments, record formats

Primary assignments

At the start of a task, the system files in BS2000 are assigned to particular input/output devices. This is known as the **primary assignment** of the files and is dependent on the type of job (interactive mode or batch mode). The various options are summarized in the following table:

System file	Primary assignment	
	in interactive mode	in batch mode
SYSDTA	Terminal	Spoolin file or ENTER file
SYSIPT	No primary assignment	Spoolin file or ENTER file
SYSOUT	Terminal	Temporary spoolout file (EAM file) which is output on the printer at task end and then deleted
SYSLST SYSLSTnn	Temporary spoolout files (EAM files), which are output on the **printer** at task end and then deleted	
SYSOPT	Temporary spoolout file (EAM file), which is output on **floppy disk** at task end and then deleted.	

Table 12: Primary assignments of the system files

Reassignments

The assignment of the system files can be changed in the course of a task by using the ASSIGN-*system-file* command, i.e. they can be redirected to other devices, system files, or even cataloged files.

A detailed description of the command can be found in the “BS2000/OSD-BC Commands” manual [3].

System file	Reassigned to ...	using the command
SYSDTA	cataloged disk file (SAM or ISAM) or PLAM library	ASSIGN-SYSDTA filename ASSIGN-SYSDTA *LIBRARY(library, element)
	floppy disk	ASSIGN-SYSDTA *DISKETTE(...)
SYSIPT	cataloged disk file (SAM or ISAM)	ASSIGN-SYSIPT filename
SYSOUT	cataloged disk file (tape or disk)	ASSIGN-SYSOUT filename (in batch mode only)
SYSLST SYSLSTnn	cataloged disk file (SAM)	ASSIGN-SYSLST filename ASSIGN-SYSLST *SYSLST-NUMBER(...)
	dummy file (*DUMMY)	ASSIGN-SYSLST *DUMMY
SYSOPT	cataloged disk file (SAM)	ASSIGN-SYSOPT filename or ASSIGN-SYSOPT filename, OPEN-MODE = EXTEND

	dummy file (*DUMMY)	ASSIGN-SYSOPT *DUMMY

Table 13: Reassignments of system files

Record formats

The system files process fixed-length records (record format F) or variable-length records (record format V). The following table provides an overview of the record formats and record lengths permissible in each case.

System file	Record format	Record length
SYSDTA	V	When input via terminal or disk file: max. 32 Kbytes
	F	When input via card reader: max. 80 bytes
SYSIPT	F, V	Max. 80 bytes of data if SAM file. Max. 80 bytes: 72 bytes of data if ISAM file, bytes 73-80 contain ISAM key.
SYSOUT	V	In batch mode: max. 132 bytes (+ 1 line-feed character)
		In interactive mode: max. 32 Kbytes
SYSLST SYSLSTnn	V	Max. 133 bytes: 1 byte control information and 132 bytes data
SYSOPT	F	Max. 80 bytes: 72 bytes of data; bytes 73-80 contain the first 8 characters of the name from the PROGRAM-ID

Table 14: Record formats and record lengths for system files

8.2 Job switches and user switches

BS2000 makes 32 job switches (numbered from 0 to 31) available to each job (task) and 32 user switches (numbered from 0 to 31) available to each user identification (see “Commands” manual [3]). Each switch is able to assume an ON or an OFF status. They may be used to control activities within a task or to coordinate the activities of several tasks. Thus, for example:

- job switches can be used when two or more (COBOL) programs within one job need to communicate, e.g. when the execution of one program is dependent on the processing steps of another program that was invoked earlier;
- user switches can be used when two or more jobs need to communicate. When the communicating jobs belong to different user IDs, user switches associated with one ID can be interrogated by the jobs of another ID, but cannot be modified by these jobs.

Job and user switches can be accessed and modified at operating system level (by means of commands) or at program level (via COBOL statements). COBOL2000 supports access to job and user switches with the following language elements (see “COBOL2000 Reference Manual” [1]):

- Program-internal mnemonic names for job and user switches and their status, declared in the SPECIAL-NAMES paragraph of the Environment Division:

  These mnemonic names allow Procedure Division statements to reference the assigned switches and their status (see below). The mnemonic names can be declared as described below:

  - For the job switches via the implementor-names TSW-0, TSW-1,..., TSW-31, where the additional phrase ON IS... and OFF IS... allow the user to define condition names for the respective switch status.
    It is thus possible, for example, to declare the mnemonic name and status for task switch 17 with the phrases

    ```
    TSW-17 IS mnemonic-name-17
       ON  IS switch-status-on-17
       OFF IS switch-status-off-17
    ```

  - For the user switches via the implementor-names USW-0, USW-1,..., USW-31, where the additional phrases ON IS... and OFF IS... allow the user to define condition names for the respective switch status.
    It is thus possible, for example, to declare the mnemonic name and status for user switch 18 with the phrases

    ```
    USW-18 IS mnemonic-name-18
       ON  IS switch-status-on-18
       OFF IS switch status-off-18
    ```

- Interrogation and modification of switches in the Procedure Division:
  - Conditions (e.g. in the IF, PERFORM or EVALUATE statement) can contain the names (declared in the SPECIAL-NAMES paragraph) of switch status conditions and in this way evaluate them for the control of program execution.
  - SET (format 3; see “COBOL2000 Reference Manual” [1]) can access switches (via the mnemonic names declared in the SPECIAL-NAMES paragraph) and change their status.

Example 8-2

Use of job switches

In the following extract from an interactive task, a procedure provides for different processing paths, depending on the status of job switches 12 and 13. The switches are evaluated and changed both at operating system level and at program level:

First, job switch 12 can be set at operating system level in order to control processing within the succeeding procedure, where its status will be evaluated at program level. Job switch 13 is then set, depending on the status of program execution. This switch is subsequently evaluated at operating system level.

```
/MODIFY-JOB-SWITCHES ON=12,OFF=13  ————————————————————— (1)
...
/CALL-PROC PROG.SYSTEM
```

```
The PROG.SYSTEM file contains  ————————————————————— (2)
following commands:
/BEGIN-PROC ...
  ...
/START-PROGRAM PROG-1 ————————————————————————————— (3)
```

```
Extract from PROG-1:
...
SPECIAL-NAMES.                              (4)
TSW-12 IS SWITCH-12                          |
    ON IS ON-12                              |
TSW-13 IS SWITCH-13                          |
    ON IS ON-13                             (4)
...
PROCEDURE DIVISION.
...
    IF ON-12 PERFORM A-PAR. ————————————— (5)
    PERFORM B-PAR.
    ...
    IF FIELD = 99 SET SWITCH-13 TO ON. ———— (6)
    STOP RUN.
A-PAR.
    ...
B-PAR.
    ...
```

```
...
/SKIP-COMMANDS TO-LABEL .END,IF=JOB-SWITCHES (OFF=13) —— (7)
/START-PROGRAM PROG-2
/.END MODIFY-JOB-SWITCHES OFF=(12,13) ————————————— (8)
/END-PROC
```

```
/...
```

(1) Job switch 12 is set to ON, job switch 13 to OFF at the operating system level.

(2) Extract from a procedure.

(3) The COBOL program PROG-1 is called.

(4) The internal names SWITCH-12 and SWITCH-13 are declared in the program for job switches 12 (TSW-12) and 13 (TSW-13) respectively, and the condition names ON-12 or ON-13 for their respective ON status.

(5) If job switch 12 is ON (see (1)), the statement PERFORM A-PAR is executed before PERFORM B-PAR.

(6)

If the indicator FIELD contains the value 99 at the end of program execution, PROG-1 sets the job switch to ON.

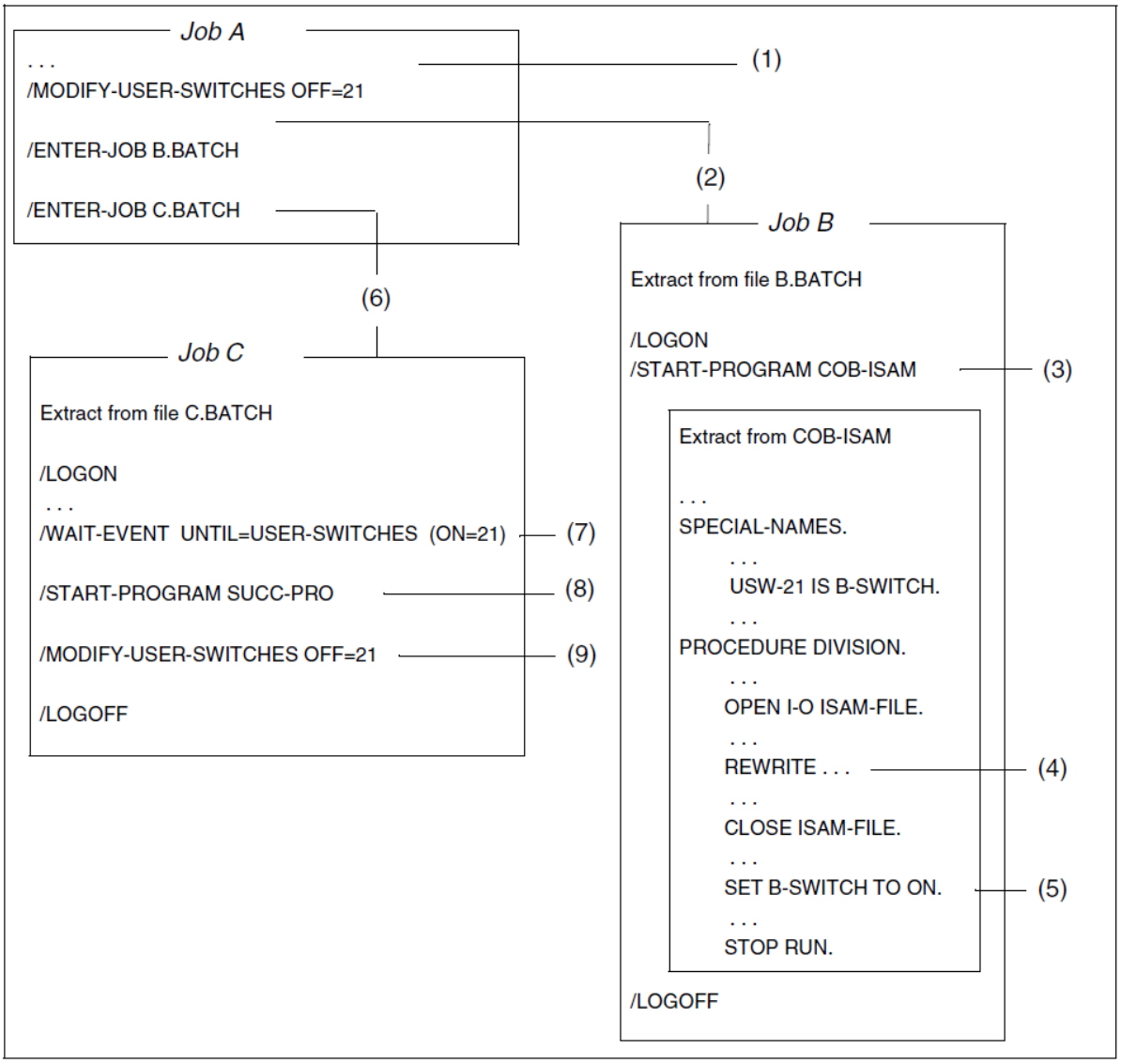
(7) The procedure evaluates the status of job switch 13: if it was not set to ON by PROG-1, the procedure branches to the end. Otherwise, PROG-2 is executed in addition to PROG-1.

(8) At the operating system level, job switches 12 and 13 are reset.

Example 8-3

Use of user switches

In the following extract, interactive job A generates two batch jobs, B and C. In job B, an ISAM file is updated. Job C can only execute after this takes place. User switch 21 is used in three different jobs. It is set at program level, and evaluated and reset at operating system level.



(1) User switch 21 is initialized to OFF.

(2) The ENTER procedure B.BATCH is called; it generates batch job B.

(3) Batch job B calls the COBOL program COB-ISAM.

(4) COB-ISAM updates the file ISAM-FILE.

(5) When updating is completed, COB-ISAM sets user switch 21 to ON.

(6) The ENTER procedure C.BATCH is called; it generates batch job C.

(7) Job C waits until the user switch is set to ON in job B.

(8) As soon as user switch 21 is set to ON, job C calls the COBOL program SUCC-PRO; it can then access the ISAM-FILE updated in job B.

(9) User switch 21 is set to OFF, in order to mark the (normal) end of job C.

8.3 Job variables

Job variables are available as a separate software product. Like job and user switches, they too are useful in information exchange

- between user programs and the operating system, or
- between different user programs.

Compared with switches, however, job variables offer the following additional facilities:

- They can be declared as monitoring job variables when a program is called. As such, they are automatically supplied with status and return codes, which provide information on program status and termination action, as well as on potential runtime errors.
- At the operating system or program level, they can be loaded with records of up to 256 bytes in length (128 bytes in the case of monitoring job variables). Because of this, they are able to communicate more detailed information than job or user switches, which are only capable of switching between ON and OFF status.
- In contrast to job and user switches, they can also be modified by jobs running under different user IDs.

Before a COBOL program can access a job variable, the variable must first be assigned to it via a link name, in a similar way to a file. The SET-JV-LINK command performs this function for job variables. Its format is described in the “Commands” [3] and “Job Variables” [7] manuals, and an example of its usage is contained in the following section. The link name to be specified in the command is apparent from the declarations in the COBOL program (see below).

COBOL2000 supports access to job variables with the following language elements (see “COBOL2000 Reference Manual” [1]):

- Link names and program-internal mnemonic names for job variables, declared in the SPECIAL-NAMES paragraph of the Environment Division:
  Job variables can be assigned via link names, and the statements of the PROCEDURE DIVISON can refer to them via their mnemonic names (see below). Link names and mnemonic names for job variables can be declared with phrases in the following format:

  JV-jvlink IS mnemonic-name

jvlink	specifies the link name for the job variable. When the link name is formed, an “*” is prefixed to jvlink as its first character; the resulting link name is therefore *jvlink. For this reason, the string jvlink must not be longer than 7 bytes.
mnemonic-name	declares the program-internal mnemonic name for the job variable.

- The ACCEPT and DISPLAY statements of the Procedure Division:
  - ACCEPT...FROM mnemonic-name

    reads the contents of the job variable that is associated (in the SPECIAL-NAMES paragraph) with mnemonic-name. The data is transferred left-justified into the receiving item specified in the ACCEPT statement, according to the length of this item. If the item is longer than 256 bytes (128 bytes in the case of monitoring job variables), it is space-filled on the right; if it is shorter, the contents of the job variable are truncated on the right during the transfer to conform to the length of the item.
  - DISPLAY...UPON mnemonic-name

    writes data in the job variable that is associated (in the SPECIAL-NAMES paragraph) with mnemonic-name. The size of the data transfer is determined by the length of the sending items or literals of the DISPLAY

statement, provided the maximum record length of 256 bytes (128 bytes for monitoring job variables) is not exceeded. If the total number of characters to be transferred is greater than the maximum record length, the record is truncated to the maximum length when it is transferred.
When data is written to a monitoring job variable it should be noted that the first 128 bytes of the job variable are protected by the system against write access. Thus, only that part of the record starting at position 129 is written to the job variable, beginning at position 129 of the variable.

If a COBOL program which includes statements for job variables is run on a BS2000 installation that does not support job variables, these statements are not executed. After an ACCEPT statement, the receiving item contains the characters “/*”, starting in column 1. The first attempt to access a job variable causes message COB9120 to be output to SYSOUT.

A failed access to a job variable in a BS2000 installation that does support job variables causes message COB9197 to be output to SYSOUT (see table 10 in section “Program termination”).

Example 8-4

Communication via a job variable

In the following interactive job, the job variable CONTROL.RUN is used both by a COBOL program and at command level. Depending on the contents of the job variable, the program can go through various processing branches, updating the contents of the job variable if required. A different job - even one under another user identification - can access this job variable, provided the job variable was cataloged with the command CREATE-JV ...,USER-ACCESS=ALL-USERS.

```
/SET-JV-LINK LINK-NAME=UPDATE,JV-NAME=CONTROL.RUN ———————————— (1)
/START-PROGRAM PROG.WORK-1
```

```
Extract from the program:
...
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS T
    JV-UPDATE IS FIELDJV. ————————————————————————————— (2)
     ...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  DATE-TODAY      PIC X(6). ———————————————————————— (3)
01  CONTENTS-JV. ———————————————————————————————————— (4)
  05  DATE-UPDATE   PIC X(6).
  05  FILLER        PIC X(20).
  05  NUM-UPDATE    PIC 9(4).
     ...
PROCEDURE DIVISION.
    ACCEPT CONTENTS-JV FROM FELDJV. ———————————————— (5)
    ACCEPT DATE-TODAY FROM DATE.
    IF DATE-UPDATE NOT EQUAL DATE-TODAY ———————————— (6)
      PERFORM WORK
        ELSE PERFORM ALREADY-UPDATED.
     ...
WORK.
     ...
    MOVE DATE-TODAY TO AKT-DAT.                      (7)
    ADD 1 TO NUM-UPDATE.                              |
    DISPLAY CONTENTS-JV UPON FIELDJV.                (7)
     ...
```

```
ALREADY-UPDATED.
    DISPLAY "END OF UPDATE"
      UPON T.
     ...
```

```
/SHOW-JV JV-NAME(CONTROL.RUN) ————————————————————————————— (8)
%930629 UPDATE NR. 1679
```

(1) The job variable CONTROL.RUN is assigned to the COBOL program called after it (PROG.WORK-1) via the link name *UPDATE.

(2) The link name *UPDATE and the (program-internal) mnemonic name FIELDJV are declared for the job variable in the SPECIAL-NAMES paragraph of PROG.WORK-1.

(3) DATE-TODAY is reserved as the receiving item for the date.

(4) The receiving item for the contents of the job variable is declared. It contains subordinate items for recording the most recent update date (DATE-UPDATE) and an updating counter (NUM-UPDATE).

(5) ACCEPT transfers the contents of the job variable FIELDJV to CONTENTS-JV.

(6) Depending on whether the update date (DATE-UPDATE) of the job variable corresponds to the current date (DATE-TODAY), different processing procedures are executed in the program.

(7) At the end of processing, the items DATE-UPDATE and NUM-UPDATE are updated and written back to the job variable with DISPLAY CONTENTS-JV... .

(8) The job variable is read at operating system level: it contains the date and the number of the most recent update.

8.4 Accessing an environment variable

You can access an environment variable using the ACCEPT or DISPLAY statements.
The names of environment variables are defined using format 4 in the DISPLAY statement. Format 5 of the ACCEPT statement is required in order to access the contents of environment variables.
At system level, the environment variables must be set up using an SDF-P variable.

Example 8-5

Accessing an environment variable

```
/SET-VAR TSTENV=’AAAA BBB CC D’
/START-PROGRAM ...
Program excerpt:
IDENTIFICATION DIVISION.
...
SPECIAL-NAMES.
    ENVIRONMENT-NAME IS ENV-NAME
    ENVIRONMENT-VALUE IS ENV-VAR
    TERMINAL IS T
...
WORKING-STORAGE SECTION.
01  A   PIC X(15).
...
PROCEDURE DIVISION.
...
    DISPLAY "TSTENV" UPON ENV-NAME
    ACCEPT A FROM ENV-VAR
      ON EXCEPTION DISPLAY "ENVIRONMENT ’TSTENV’ IS UNKNOWN!" UPON T
                   END-DISPLAY
      NOT ON EXCEPTION DISPLAY "VALUE IS:" A UPON T
                       END-DISPLAY
    END-ACCEPT
```

The exception condition appears with every failed access. Causes of a failed access can be, for example:

- missing SET-VAR command
- content of the variable is longer than the receiving field

8.5 Compiler and operating system information

COBOL programs can access information held by the compiler and the operating system. This includes information on

- the compilation of the source
- CPU time used since LOGON
- the task in which the object program is running, and
- the terminal from which the program was called.

COBOL2000 supports access to this information with the following language elements:

- Program-internal mnemonic names for the individual types of information, declared in the SPECIAL-NAMES paragraph of the Environment Division:
  The ACCEPT statement of the Procedure Division can access the relevant information via these mnemonic names. Mnemonic names can be declared for information on
  - the compilation with COMPILER-INFO IS mnemonic-name
  - the CPU time used with CPU-TIME IS mnemonic-name
  - the task with PROCESS-INFO IS mnemonic-name
  - the terminal with TERMINAL-INFO IS mnemonic-name
  - the date with DATE-ISO4 IS mnemonic-name (with century)
- The ACCEPT statement in the Procedure Division:

  ACCEPT...FROM mnemonic-name

  moves the information that is associated (in the SPECIAL-NAMES paragraph) with mnemonic-name into the specified receiving item.

  The size of the data transfer is determined by the length of the receiving item specified in the ACCEPT statement; the data is aligned on the left in the item, as follows:
  If the item is longer than the value to be transferred, it is padded with spaces on the right; if it is shorter, the value is truncated on the right during transfer to conform to the length of the item. This does not apply to CPU TIME: here a numerically correct transfer is always carried out.
  The length (and possibly the structure) to be declared for the receiving item is determined by the type of information that the item is to receive. The formats of the various types of information are listed in the following section.

Contents and structure of the information

The following table explains the structure of the information that is made available to a COBOL program via the implementor-names COMPILER-INFO, CPU-TIME, PROCESS-INFO, TERMINAL-INFO and DATE-ISO4.

Character position(s)	Information for COMPILER-INFO
1-10	Compiler name (COBOL2000'BLANK', COBOL2000B, COBOL2000R)
11-20	Compiler version Format: Vdd.dldddd d = digit or blank l = letter or blank (e.g. “V02.2A ”)

21-30	Date of compilation Format: YYYY-MM-DD (e.g. “1999-12-31”)
31-38	Time of compilation Format: HH-MM-SS (e.g. “23-59-59”)
39-68	The first eight characters of the PROGRAM-ID name

	Information for CPU-TIME
PIC 9(6)V9(4)	CPU time in ten thousandths of a second

Character position(s)	**Information for PROCESS-INFO**
1	Type of job Contents: B for Batch D for Dialog (= interactive)
2-5	Job sequence number
6-13	User identification
14-21	Account number
22	Privilege class of the job Contents: U for user S for system administrator
23-32	Operating system version Format: Vdd.dldddd (e.g. “V11.2 ”)
33-40	Name of the next processor to which the terminal is connected.

Character position(s)	**Information for PROCESS-INFO**
41-120	System administrator privileges; the fields contain 8 blanks if the privilege is not present
41-48	SECADM
49-56	USERADM
57-64	HSMSADM
65-72	SECOLTP
73-80	TAPEADM
81-88	SATFGMMF
89-96	NETADM
97-104	FTADM
105-112	FTACADM
113-120	TSOS

Character position(s)	Information for TERMINAL-INFO
1-8	Terminal name
9-13	Number of characters per line
14-18	Number of physical lines that can be output without activating the information overflow control.
19-23	Number of characters that can be output without activating the information overflow control.
24-27	Device type If a device type is not known to the runtime system, these positions are filled with blanks.

Character position(s)	Information for DATE-ISO4
1-14	Current date (including century YYYY and numbered dayNNN of the current year) Format: YYYY-MM-DDNNN'BLANK'

Table 15: structure of compiler and operation system information

Example 8-6

Data structures for acceptance of compiler and operating system information by means of the ACCEPT statement

```
*
 01  COMPILER-INFORMATION.
     02  COMPILER-NAME                    PIC X(10).
     02  COMPILER-VERSION                 PIC X(10).
     02  DATE-OF-COMPILATION              PIC X(10).
     02  TIME-OF-COMPILATION              PIC X(8).
     02  PROGRAM-NAME                     PIC X(30).
*
 01 CPU-TIME-IN-SECONDS                   PIC 9(6)V9(4).
*
 01 TASK-INFORMATION.
     02  TASK-TYPE                        PIC X.
         88 BATCH-TASK                            VALUE "B".
         88 INTERACTIVE-TASK                      VALUE "D".
     02  TASK-SEQUENCE-NUMBER             PIC X(4).
     02  USER-IDENTIFICATION              PIC X(8).
     02  ACCOUNT-NUMBER                   PIC X(8).
     02  PRIVILEGE-IDENTIFIER             PIC X.
         88  SYSTEM-ADMINISTRATOR                 VALUE "S".
         88  USER                                 VALUE "U".
     02  OPERATING-SYSTEM-VERSION         PIC X(10).
     02  PROCESSOR-NAME                   PIC X(8).
     02  SYSTEM-ADMINISTRATOR-PRIVILEGE   PIC X(80).
*
 01  TERMINAL-INFORMATION.
     02  TERMINAL-NAME                    PIC X(8).
     02  CHARS-PER-LINE                   PIC 9(5).
     02  LINES-PER-SCREEN                 PIC 9(5).
```

```
      02  CHARS-PER-SCREEN                PIC 9(5).
      02  DEVICE-TYPE                     PIC X(4).
*
 01  CURRENT-DATE.
      05  YEAR                            PIC X(4).
      05  FILLER                          PIC X.
      05  MONTH                           PIC X(2).
      05  FILLER                          PIC X.
      05  DAY-OF-THE-MONTH                PIC X(2).
      05  DAY-OF-THE-YEAR                 PIC X(3).
      05  FILLER                          PIC X.
```

9 Processing of cataloged files

The processing of POSIX files is described in the chapter “COBOL2000 and POSIX”.

9.1 Basic information on the structure and processing of cataloged files

- Basic concepts relating to the structure of files
- Assignment of cataloged files
- Definition of file attributes
- Disk and file formats

9.1.1 Basic concepts relating to the structure of files

From the viewpoint of a COBOL application program, a file is a named collection of data records that is provided with a logical structure (**file organization**), has specific **record formats**, and is stored on one or more data storage media, which are referred to as volumes.

COBOL programs access files by making use of functions provided by the Data Management System (DMS). The particular **DMS access method** used for this purpose is determined by the file organization.
As seen from the perspective of DMS, the accessing of a file always represents a transfer of **data blocks** between a peripheral storage device and a part of the main memory, called the **buffer**, which is an area set up by the application program for accommodating the data blocks.

File organization and DMS access methods

The logical structure of a file, and thus the method by which it is accessed, is defined by its type of organization. The file organization is specified at file creation and cannot be changed subsequently.
In COBOL, files may be organized as sequential, relative, or indexed files. The features and characteristics of the various types of file organization are detailed in section “Sequential file organization”, section “Relative file organization” and section “Indexed file organization”. Each of the above organization types corresponds to an access method of the DMS.

The relationship is shown in the following table:

File organization	DMS access method
sequential	SAM
relative	ISAM/UPAM
indexed	ISAM

Table 16: File organization and DMS access methods

Data records and record formats

A (logical) data record represents the unit of a file that can be accessed by a COBOL program with a single I-O statement. Each read operation makes a record available to the program, while each write statement creates a record in the file.

The records of a file may be classified according to their record format. Depending on the type of file organization, the following record formats are permitted in COBOL:

- Fixed-length records (RECFORM=F)

  All records of a file are of the same length and contain no information regarding the record length.
- Variable-length records (RECFORM=V)

  The records of a file can have different lengths; each record contains a specification of its length in the first word, called the record length field.
  In a COBOL program, this record length field does not form a part of the record description entry. This means that its content cannot be explicitly accessed unless a RECORD clause with the DEPENDING ON phrase is specified for the file (see “COBOL2000 Reference Manual” [1]).
- Records of undefined length (RECFORM=U)

  The records of a file may vary in length but include no record length information.

Data blocks and buffers

A (logical) data block is the unit of a file that is transferred (by the DMS) between peripheral storage and main memory during a file access operation. In order to such data blocks, the program reserves a storage area in its address space. This reserved area is called the buffer.

A logical block may consist of one or more records; however, a record must not be distributed over more than one logical block.

If a logical block contains more than one data record, these records are said to be blocked. Only records of fixed or variable length can be blocked. Blocking is not possible with records of undefined length.

In terms of size, a logical block (and thus a buffer) may be defined:

- for disk files, as a standard block, i.e. a physical block (PAM block) of 2048 bytes or integral multiples thereof (up to 16 PAM blocks) and
- for magnetic tape files, additionally as a non-standard block of any length up to 32767 bytes.

To simply adaptations to future disk formats, only even-numbered multiples of 2048 bytes should be used as the block size in the ADD-FILE-LINK command or in the program specifications.

During compilation, the compiler calculates a value for the buffer size for each file on the basis of record and block length specifications given in the compilation unit. This default value can be modified during the assignment of the file by specifying the BUFFER-LENGTH operand in the ADD-FILE-LINK command. It must be noted, however, that

- the buffer must be at least as large as the longest data record, and
- there must be space for the management information (PAM key) in the buffer when processing in non-key format (BLKCTRL = DATA) (see section “Disk and file formats”).

Except in the case of newly created files (OPEN OUTPUT), the block size entered in the catalog always takes priority over block size specifications in the program or ADD-FILE-LINK command.

9.1.2 Assignment of cataloged files

A program-internal name is specified in the SELECT clause (see “COBOL2000 Reference Manual” [1]) for each file to be processed by a COBOL program. The COBOL statements for a given file reference the file via this name. During program execution, each of the specified file-names must be assigned an actual file.

This assignment can be defined before the program call by means of a ADD-FILE-LINK or ASSIGN system-file command. Which of the two commands should be used is determined by the entry in the ASSIGN clause (see “COBOL2000 Reference Manual” [1]) for the file. If no file is explicitly assigned, the default values generated during compilation come into effect.

The individual methods used for the assignment of files are summarized below:

Assignment via the ADD-FILE-LINK command

Assignment via the ADD-FILE-LINK command is therefore possible only if the link name of the file is specified in the form “literal” or “data-name” in the ASSIGN clause. “literal” specifies the link name statically for the program. The “data-name” data item can be used to modify the link name dynamically, i.e. during program execution.

To assign a cataloged file, the user must issue an ADD-FILE-LINK command for this file before the program call; this ADD-FILE-LINK command must include a LINK-NAME operand specifying the declared link name. File attributes can also be specified at the same time using other operands of the ADD-FILE-LINK command.

Any link name must conform to BS2000 requirements in respect of link names (see also “Introductory Guide to DMS” [4]). More specifically, this means that

- it must be alphanumeric,
- it may consist of a maximum of eight characters, and
- it must not contain any lowercase letters.

Example 9-1

Assignment of a cataloged file via the ADD-FILE-LINK command

Entry in the FILE-CONTROL paragraph of the COBOL program LINKLIT:	`SELECT MASTER-FILE ASSIGN TO "MASTLNK".`
Link name generated at compilation:	`MASTLNK`

Assignment of file STORE.INVENTORY and program call:	`/ADD-FILE-LINK LINK-NAME=MASTLNK,`
	`  FILE-NAME=STORE-INVENTORY`
	`/START-PROGRAM LINKLIT`

In the case of COBOL programs using SORT (see chapter “Sorting and merging”), the following link names are reserved for the SORT utility routine and are thus not available for other files:

MERGEnn (nn=01,...99)
SORTIN
SORTINnn (nn=01,...99)
SORTOUT
SORTWK
SORTWKn (n=1,...9)
SORTWKnn (nn=01,...99)
SORTCKPT

A file assignment remains in effect until it is

- released explicitly by a REMOVE-FILE-LINK command or implicitly at end of task, or
- modified by means of a subsequent ADD-FILE-LINK command.

This should be noted particularly where several different files are to be successively assigned to one program-internal file-name within the same task.

Information regarding currently assigned cataloged files can be obtained by using the SHOW-FILE-LINK command (see “Commands” manual [3]).

Example 9-2

Changing file assignments :

```
/ADD-FILE-LINK INOUTFIL,FILE.UPDATE.1 ————————————————————————(1)
/START-PROGRAM UPDPROG
   ...
/ADD-FILE-LINK INOUTFIL,FILE.UPDATE.2 ————————————————————————(2)
/START-PROGRAM UPDPROG
   ...
/REMOVE-FILE-LINK INOUTFIL ———————————————————————————————————(3)
```

The COBOL program UPDPROG specifies the link name INOUTFIL for a input/output file. This program successively actualizes the cataloged files FILE.UPDATE.1 and FILE-UPDATE.2.

(1) The file named FILE.UPDATE.1 is assigned via link name INOUTFIL to the program UPDPROG for subsequent processing.

(2) After processing, a further ADD-FILE-LINK command terminates the previously valid file assignment for the link name INOUTFIL and assigns the file named FILE.UPDATE.2 as the new file.

(3) The file assignment for link name INOUTFIL is released by means of the REMOVE-FILE-LINK command.

Assignment via the default settings

If a cataloged file has not been explicitly assigned at program runtime to an internal file name (with the link name “linkname”), the following defaults come into effect:

- In the case of an output file and ENABLE-UFS-ACCESS = NO, the program attempts to access a cataloged file with the name from the SELECT clause. If no catalog entry is found under this name, the program writes to a previously created file with the name FILE.COBOL.linkname.
  In the case of ENABLE-UFS-ACCESS = YES, the program writes directly to the file FILE.COBOL.linkname.
- For an input file whose SELECT clause contains the OPTIONAL phrase, the first read access attempt causes an AT END condition and passes control to the procedures declared in the program for such a case.
- In the case of an input file (without the OPTIONAL phrase in the SELECT clause) and ENABLE-UFS-ACCESS = NO or of an input-output file, the program attempts to access a cataloged file with the name from the SELECT clause. If no catalog entry is found under this name, execution is interrupted with error message COB9117 and may be continued with the RESUME-PROGRAM command after a correct file assignment is made.

Assignment via the ASSIGN-*systemfile* command

The prerequisite for this is that the name of a system file must not have been specified in the ASSIGN clause. The system files are designated vendorname-1 (PRINTER) or vendorname-2 (PRINTER01...PRINTER99, SYSIPT, SYSOPT).

By issuing a ASSIGN-*systemfile* command for the specified system file,

- a cataloged file or
- another system file

can be assigned before the program is called. The permissible assignments for each system file are given in the description of the ASSIGN-*systemfile* command in the “Commands” manual [3].

Example 9-3

Assigning a cataloged file via the ASSIGN *systemfile* command

Entry in the FILE-CONTROL paragraph of the COBOL program LISTPROG:	`SELECT PRINT-FILE ASSIGN TO PRINTER.`
Assignment of the LIST.FILE and program call:	`/ASSIGN-SYSLST LIST.FILE` `/START-PROGRAM LISTPROG`

If no file is explicitly assigned at program runtime, the program will execute its I-O operations on the specified system file.

A file assignment remains in effect until it is

- released at the end of the task, or
- modified by means of a subsequent ASSIGN-*systemfile* command.

This should be noted particularly where several different files are to be successively assigned to one program-internal file name within the same task.

Information regarding current file assignments can be obtained by using the SHOW-SYSTEM-FILE-ASSIGNMENTS command.

9.1.3 Definition of file attributes

BS2000 provides the CREATE-FILE command for creating files. A task file table entry with further file attributes is created by means of the ADD-FILE-LINK command. The complete format for this command and a detailed description are provided in “Commands” manual [3] and “Introductory Guide to DMS” [4].

Catalog entry

When file attributes are defined with the CREATE-FILE command it must be ensured that the primary memory allocation is sufficient.
This is particularly true in the case of files in which the block size exceeds the primary allocation which the operating system assumes by default. The block size is calculated from the specification in the COBOL program (see “COBOL2000 Reference Manual” [1]). If the COBOL program yields a block length of (STD, n), the PRIMARY-ALLOCATION in the SPACE operand of the CREATE-FILE command should be at least

- 2n for sequential files.
- 2n+2 for indexed and relative files.

Task file table (TFT)

Whenever a ADD-FILE-LINK command with the operand

LINK-NAME=linkname

is issued for a file, the DMS creates an entry for the file under this link name in the task file table (TFT) and stores all file attributes explicitly defined in the ADD-FILE-LINK command under this entry.

Each of these entries is retained in the TFT until it is

- removed by a REMOVE-FILE-LINK command for the assigned file link name, or deleted together with the TFT at the end of the task, or
- overwritten by a new ADD-FILE-LINK command for the same file link name.

Information on the current contents of the TFT can be obtained by using the SHOW-FILE-LINK command.

When a COBOL program attempts to open a file, the DMS first checks whether the TFT contains the link name that was defined for the file at compilation (see section “Assignment of cataloged files”). If such an entry is found, the program takes over file attributes from

- the TFT entry under this link name,
- the file attributes that were explicitly or implicitly specified in the program, and
- the catalog entry of the associated file.

The specifications from the TFT entry (i.e. file attributes explicitly defined in the ADD-FILE-LINK command) overwrite file specifications from the COBOL program. The catalog entry is referred to only for file attributes that are defined neither in the program nor in the TFT entry or those that were specified as null operands in the ADD-FILE-LINK command.

This approach could lead to conflicts during file access, especially when file attributes specified in the ADD-FILE-LINK command are not compatible with the (explicitly or implicitly) defined characteristics in the COBOL program or in the catalog entry of the assigned file. This is especially applicable in the following situations:

- Conflicting entries on the **open mode**

COBOL program	ADD-FILE-LINK command

OPEN INPUT...[REVERSED]	OPEN-MODE=OUTPUT or OPEN-MODE=EXTEND
OPEN OUTPUT	OPEN-MODE=INPUT or OPEN-MODE=REVERSE
OPEN EXTEND	OPEN-MODE=INPUT or OPEN-MODE=REVERSE

- Conflicting entries on the **organization type of the file**

COBOL program	ADD-FILE-LINK command
ASSIGN clause ORGANIZATION clause	ACCESS-METHOD operand

- Conflicting entries on the **record format**

COBOL program	ADD-FILE-LINK command
RECORD clause RECORDING MODE clause	RECORD-FORMAT operand

- Conflicting entries on the **record length**

COBOL program	ADD-FILE-LINK command
RECORD clause record description entry	RECORD-SIZE operand

- Conflicting entries on the **record key**

COBOL program	ADD-FILE-LINK command
RECORD KEY clause record description entry	KEY-POSITION operand KEY-LENGTH operand

- Conflicting entries on the **disk format** or **file format**

Catalog entry	ADD-FILE-LINK command
BLK-CONTR = BUF-LEN =	BLOCK-CONTROL-INFO operand BUFFER-LENGTH operand

Example 9-4

Creating and displaying a TFT entry
(shown in BS2000/OSD V5.0)

```
/ADD-FILE-LINK INOUTFIL,ISAM.UPDATE, -                                (1)
/              BUFFER-LENGTH=*BY-CATALOG,                              |
               SUPPORT=*DISK(SHARED-UPDATE=*YES)                      (1)
/SHOW-FILE-LINK INOUTFIL,INFORMATION=*ALL  ————————————————————————— (2)
```

```
LINK-NAME ————————— FILE-NAME ————————————————————————————————————————
INOUTFIL            :N:$F2190202.ISAM.UPDATE
————————————————————— STATUS ————————————————————————————————————————
STATE      = INACTIVE    ORIGIN      = FILE
————————————————————— PROTECTION ————————————————————————————————————
RET-PER    = *BY-PROG    PROT-LEV    = *BY-PROG
BYPASS     = *BY-PROG    DESTROY     = *BY-CAT
————————————————————— FILE-CONTROL-BLOCK - GENERAL ATTRIBUTES ————————
ACC-METH   = *BY-PROG    OPEN-MODE   = *BY-PROG   REC-FORM   = *BY-PROG
REC-SIZE   = *BY-PROG    BUF-LEN     = *BY-CAT    BLK-CONTR  = *BY-PROG
F-CL-MSG   = STD         CLOSE-MODE  = *BY-PROG
————————————————————— FILE-CONTROL-BLOCK - DISK FILE ATTRIBUTES ——————
SHARED-UPD = YES         WR-CHECK    = *BY-PROG   IO(PERF)   = *BY-PROG
IO(USAGE)  = *BY-PROG    LOCK-ENV    = *BY-PROG
————————————————————— FILE-CONTROL-BLOCK - TAPE FILE ATTRIBUTES ——————
LABEL      = *BY-PROG   (DIN-R-NUM   = *BY-PROG,  TAPE-MARK  = *BY-PROG)
CODE       = *BY-PROG    EBCDIC-TR   = *BY-PROG   F-SEQ      = *BY-PROG
CP-AT-BLIM = *BY-PROG    CP-AT-FEOV  = *BY-PROG   BLOCK-LIM  = *BY-PROG
REST-USAGE = *BY-PROG    BLOCK-OFF   = *BY-PROG   TAPE-WRITE = *BY-PROG
STREAM     = *BY-PROG                      |
————————————————————— FILE-CONTROL-BLOCK - ISAM FILE ATTRIBUTES ——————
KEY-POS    = *BY-PROG    KEY-LEN     = *BY-PROG   POOL-LINK  = *BY-PROG
LOGIC-FLAG = *BY-PROG    VAL-FLAG    = *BY-PROG   PROPA-VAL  = *BY-PROG
DUP-KEY    = *BY-PROG    PAD-FACT    = *BY-PROG   READ-I-ADV = *BY-PROG
WR-IMMED   = *BY-PROG
————————————————————— VOLUME ————————————————————————————————————————
DEV-TYPE   = *NONE       T-SET-NAME = *NONE
VSN/DEV    = PUBN03/D3480
```

(1) The ADD-FILE-LINK command assigns the link name INOUTFIL to the file ISAM.UPDATE and defines

- that the value from the catalog entry for ISAM.UPDATE will be assigned for the BUFFER-LENGTH operand when the file is opened, and
- SHARED-UPDATE=YES, i.e. ISAM.UPDATE can be updated by more than one user simultaneously (see section “Shared updating of files (SHARED-UPDATE)”).

The DMS creates a TFT entry under the name INOUTFIL and stores these specifications in it.

(2) The SHOW-FILE-LINK command outputs the contents of the TFT entry for INOUTFIL with the operand values. Note that the values

- BUF-LEN = *CAT and
- SHARUPD = YES

are derived from the specifications in the ADD-FILE-LINK command. The remaining operands were not explicitly defined and thus have the default values *BY-PROG or *NONE.

9.1.4 Disk and file formats

Disk formats

BS2000 now supports disks with different formats:

- **K**eyed volumes (or K disks) are used for files in which block control information is stored in a separate field (“Pamkey”) for each 2K data block. These files have the PAMKEY block format.
- **N**on-**K**ey volumes (or NK disks) are used for files in which no separate Pamkey fields exist, i.e. files that have no block control information (block format NO) or files in which the block control information is stored in each respective data block (block format DATA).

NK volumes are differentiated on the basis of the minimum transfer unit. NK2 volumes have the usual transfer unit of 2K; NK4 volumes have a transfer unit of 4K.

When using NK4 volumes, it is important to ensure that the record lengths correspond to an even blocking factor.

The block format of a COBOL file can be defined with the BLOCK-CONTROL-INFO operand of the ADD-FILE-LINK command:

```
/ADD-FILE-LINK ..., -
/   BLOCK-CONTROL-INFO = BY-PROGRAM / BY-CATALOG / WITHIN-DATA-BLOCK / PAMKEY / NO
```

Two additional operand values are available for NK-ISAM files:

WITHIN-DATA-2K-BLOCK / WITHIN-DATA-4K-BLOCK

A detailed description of the BLOCK-CONTROL-INFO operand, the various file and volume formats, and the conversion of K file formats to NK file formats can be found in the “Introductory Guide to DMS” [4].

If the values specified in the BLOCK-CONTROL-INFO or BUFFER-LENGTH operands of the ADD-FILE-LINK command are not compatible with

- the block format of the file or
- the volume on which the file is stored or
- the required blocking factor,

file processing is aborted without success, and the runtime system reports the fact with I-O status (File Status) 95.

If no ADD-FILE-LINK command is used for a COBOL file, the default value to be set by the system administrator in the BLKCTRL operand of the CLASS2-OPTION applies.

K-ISAM and NK-ISAM files

ISAM files in K format that use the maximum record length become longer than the usable area of the data block when converted to NK format. But they can still be handled in NK format because the DMS extends the data block by creating so-called “overflow blocks”.

The creation of overflow blocks is attended by the following problems:

- The overflow blocks increase space requirements on the disk and thus the number of input/output operations during file processing.
- The ISAM key must never be located within an overflow block.

Overflow blocks can be avoided by ensuring that the longest record in the file is shorter than the usable area of a logical block in NK-ISAM files.

The following table shows how to calculate how much space per logical block is available for data records in ISAM files.

File format	RECORD-FORMAT	Maximum usable area
K-ISAM	VARIABLE	BUF-LEN
	FIXED	BUF-LEN - (s*4) where s = number of records per logical block
NK-ISAM	VARIABLE	BUF-LEN - (n*16) - 12 - (s*2) (rounded down to next number divisible by 4) where n = blocking factor s = number of records per logical block
	FIXED	BUF-LEN - (n*16) - 12 - (s*2) - (s*4) (rounded down to next number divisible by 4) where n = blocking factor s = number of records per logical block

Table 17: Maximum usable block area in ISAM files

Explanation of the formulae:

With RECORD-FORMAT=FIXED, every record of both K-ISAM and NK-ISAM files contains a 4-byte record length field, but this is not included in the RECSIZE value. In these cases, therefore, it is necessary to deduct 4 bytes per record.
In NK-ISAM files, each PAM page of a logical block contains 16 bytes of management information. The logical block additionally contains a further 12 bytes of management information and a 2-byte record pointer per record.

Example 9-5

Maximum record length for an NK-ISAM file (fixed-length records)

File declaration:

```
/ADD-FILE-LINK ...,RECORD-FORMAT=FIXED,BUFFER-LENGTH=STD(SIZE=2), -
/                              BLOCK-CONTROL-INFO=WITHIN-DATA-BLOCK
```

maximum record length (according to the formula in table 17):
4096 - (2*16) - 12 - 1*2 - 1*4 = 4046,
rounded down to next number divisible by four: 4044 (bytes).

K-SAM and NK-SAM files

SAM files do not have overflow blocks. This means that SAM files in K format that use the maximum record length are not converted to NK-SAM files. COBOL programs that work with records having the maximum length possible for K-SAM files are not executable with NK-SAM files.

The following table shows how much space per logical block is available for data records in SAM files.

File format	RECORD-FORMAT	Maximum usable area

K-SAM	VARIABLE	BUF-LEN - 4
	FIXED / UNDEFINED	BUF-LEN
NK-SAM	VARIABLE / FIXED / UNDEFINED	BUF-LEN - 16

Table 18: Maximum usable block area in SAM files

The reason for deducting 4 bytes from the block size in K-SAM files with variable-length records is that the logical blocks of such files contain a 4-byte block length field that is not counted in the BUF-LEN value.

9.2 Sequential file organization

Two types of file are organized sequentially: record-sequential and line-sequential files. The following general description refers to record-sequential files.
Differences to and restrictions for line-sequential files are described in section “Line-sequential files”.

9.2.1 Characteristics of sequential file organization

Records of a sequentially organized file are always arranged logically in the order in which they were written to the file. Consequently,

- every record (except for the last) has a unique successor, and
- every record (except for the first) has a unique predecessor.

This relationship between predecessor and successor cannot be modified during the entire life of the file.

It is thus not possible to

- insert records,
- delete records, or
- change the position of a record within the specified order

in a sequential file.

However, sequential file organization does permit the

- updating of records already in existence
  (provided that their lengths remain the same and that the file is a disk storage file), and
- the appending of new records to the end of the file.

Individual records of a file cannot be directly (randomly) accessed; they can only be processed in the same order in which they are stored in the file.

To process sequential files, COBOL programs make use of the SAM access method, which is provided by DMS for this purpose. Further details on this topic are provided in “Introductory Guide to DMS” [4].

Sequential files can be set up on magnetic tape devices or direct access devices (disk storage units).

9.2.2 COBOL language tools for the processing of sequential files

The following program skeleton summarizes the most important clauses and statements provided in COBOL2000 for the processing of sequential files. The most significant phrases and entries are briefly explained thereafter:

```
IDENTIFICATION DIVISON.
    .
    .
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT internal-file-name
    ASSIGN TO external-name
    ORGANIZATION IS SEQUENTIAL
    ACCESS MODE IS SEQUENTIAL
    FILE STATUS IS status-items.
    .
    .
DATA DIVISION.
FILE SECTION.
FD  internal-file-name
    BLOCK CONTAINS block-length-spec
    RECORD record-length-spec
    RECORDING MODE IS record-format
    ...
01  data-record.
    nn item-1              type&length.
    nn item-2              type&length.
    ...
PROCEDURE DIVISION.
    ...
    OPEN open-mode internal-file-name.
    ...
    WRITE data-record.
    ...
    READ internal-file-name
    ...
    REWRITE data-record.
    ...
    CLOSE internal-file-name.
    ...
    STOP RUN.
```

`SELECT internal-file-name`

specifies the name by which the file is to be referenced in the compilation unit.

internal-file-name must be a valid user-defined word.

The SELECT clause format also permits use of the OPTIONAL phrase for input files whose presence is not essential at program runtime.
If a file name declared with SELECT OPTIONAL is not assigned any file during program execution, then:

- in the case of OPEN INPUT, the program run is interrupted with message COB9117 and an ADD-FILE-LINK command is requested (in dialog mode), or the AT-END condition is initiated (in batch mode)
- in the case of OPEN I-O or OPEN EXTEND, a file with the name FILE.COBOL.link-name is created.

ASSIGN TO external-name

specifies the system file associated with the file and defines the name via which a cataloged file can be assigned.

external-name must be either

- a permissible literal or
- a permissible data name defined in the DATA DIVISION
- a valid implementor name

from the ASSIGN clause format (see “COBOL2000 Reference Manual” [1]).

ORGANIZATION IS SEQUENTIAL

specifies that the file is sequentially organized.

The ORGANIZATION clause may be omitted in the case of sequential files, since the compiler assumes sequential file organization by default.

ACCESS MODE IS SEQUENTIAL

specifies that the records of the file can only be accessed sequentially.
The ACCESS MODE clause is optional and only serves for documentation purposes in the case of sequential files. This is because sequential access is the default value assumed by the compiler and is the only permitted access mode for sequential files.

FILE STATUS IS status-items

specifies the data items in which the runtime system stores status information after each access operation on a file. This information indicates

- whether the I-O operation was successful and
- the type of any errors that may have occurred.

The status items must be declared in the Working-Storage Section or the Linkage Section. Their format and the meanings of individual status codes are described in section “Processing magnetic tape files”.
The FILE STATUS clause is optional. If it is not specified, the information mentioned above is not available.

BLOCK CONTAINS block-length-spec

defines the maximum size of a logical block. It determines how many records are to be transferred together by each I-O operation into/from the buffer of the program.

block-length-spec must be a permissible value from the format of BLOCK CONTAINS clause.

The blocking of records reduces

- the number of accesses peripheral storage and thus the runtime of the program;
- the number of interblock gaps on the storage medium and thus the physical storage space required by the file.

During compilation, the compiler calculates a value for the buffer size on the basis of the record and block length entries given in the compilation unit. In the case of disk files, the runtime system rounds up this value for the DMS to the next multiple of a PAM block (2048 bytes). This default value can be modified during the file assignment by specifying the BUFFER-LENGTH operand in the ADD-FILE-LINK command (see section “Definition of file attributes”).

In this case, it must be noted that

- the buffer must be at least as large as the longest data record, and
- there must be space for the management information (PAM key) in the buffer when processing in non-key format (BLKCTRL = DATA) (see section “Disk and file formats”).

Except in the case of newly created files (OPEN OUTPUT), the block size entered in the catalog always takes priority over block size specifications in the program or ADD-FILE-LINK command.

The BLOCK CONTAINS clause is optional. If it is omitted, the compiler assumes BLOCK CONTAINS 1 RECORDS, i.e. unblocked records.

`RECORD record-length-spec`

- specifies whether records of fixed or variable length are to be processed and
- defines, for variable-length records, a range of permissible values for the record length. If provided for in the format, a data item is additionally specified for the storage of current record length information.

record-length-spec must conform to one of the three RECORD clause formats provided in COBOL2000. It must not be in conflict with the record lengths computed by the compiler from the specifications in the associated record description entry or entries.

The RECORD clause is optional. If it is not specified, the record format is obtained from the phrase in the RECORDING MODE clause (see below). Should this clause also be omitted, records of variable length are assumed by the compiler (see section “Permissible record formats and access modes” for the relationship between the RECORD and RECORDING MODE clauses).

`RECORDING MODE IS U`

defines the format of the logical records as “undefined”; i.e. the file may contain an optional combination of fixed or variable records.

The RECORDING MODE clause is optional and is only required when declaring records of undefined length, since fixed- and variable-length records can also be specified in the RECORD clause (see section “Permissible record formats and access modes”).

```
01 data-record.
   nn item-1      type&length
   nn item-2      type&length
```

represents a record description entry for the associated file. It describes the logical format of data records.

At least one record description entry is required for each file. If more than one record description entry is specified for a file, the declared record format must satisfy the following rules:

- for fixed-length records, all record description entries must specify the same record size
- for variable-length records, they must not be in conflict with the record length specified in the RECORD clause.

The subdivision of data-record into data items (item-1, item-2, ...) is optional. For type&length, the required length and format declarations (PICTURE and USAGE clauses etc.) must be entered.

`OPEN open-mode internal-file-name`

opens the file for processing in the specified open mode. The following phrases can be entered for open-mode:

INPUT	opens the file as an input file; it can only be read.
OUTPUT	opens the file as an output file; it can only be written.
EXTEND	opens the file as an output file; it can be extended.
I-O	opens the file as an I-O file; it can be read (one record at a time), updated and rewritten.

The open-mode entry determines with which I-O statements a file may be accessed (see section “Open modes and types of processing (sequential processing)”).

```
WRITE data-record
READ internal-file-name
REWRITE data-record
```

are I-O statements for the file that

- write or
- read or
- rewrite

one record at a time.

The open-mode declared in the OPEN statement determines which of these statements is admissible for the file. The relationship between access mode and open mode is described in section “Open modes and types of processing (sequential processing)”.

`CLOSE internal-file-name`

terminates processing. Depending on the entry in the format, it may apply to

- the file (no further phrase) or
- a disk storage unit (phrase: UNIT) or
- a magnetic tape reel (phrase: REEL).

This clause can optionally be used to prevent

- a tape from being rewound (phrase: WITH NO REWIND) or
- a file from being opened again (phrase: WITH LOCK) in the same program run.

9.2.3 Permissible record formats and access modes

Record formats

Sequential files may contain records of fixed length (RECFORM=F), variable length (RECFORM=V), or undefined length (RECFORM=U). However, blocking is possible only in the case of fixed- or variable-length records.

In the COBOL compilation unit, the format of records to be processed is defined in the RECORD or RECORDING MODE clause (see “COBOL2000 Reference Manual” [1]). The following table lists the phrases associated with each record format:

	Phrase in the	
Record format	**RECORD clause**	**RECORDING MODE clause**
Fixed-length records	RECORD CONTAINS...CHARACTERS (format 1)	
Variable-length records	RECORD IS VARYING IN SIZE... (format 2) or RECORD CONTAINS...TO... (format 3)	
Records of undefined length	Declaration not possible with the RECORD clause	RECORDING MODE IS U

Table 19: Specification of record formats in the RECORD or RECORDING MODE clause

If neither of the two clauses is specified, the compiler assumes that the records are of variable length.

Access modes

Records of a sequential file can only be accessed sequentially, i.e. the program can only process them in the order in which they were written to the file during its creation.

The mode of access is specified in the ACCESS MODE clause in the COBOL compilation unit. In the case of sequential files, ACCESS MODE IS SEQUENTIAL is the only admissible entry. Since this is also the default value assumed by the compiler, the ACCESS MODE clause may be omitted in this case.

9.2.4 Open modes and types of processing (sequential processing)

The various language elements available for use in a COBOL program can be sequential files to be

- created,
- read,
- extended (by adding new records at the end of the file), and
- updated (by making changes in existing records)

The individual I-O statements that may be used in the program to process a given file are determined by its open mode, which is specified in the OPEN statement.

OPEN OUTPUT

WRITE is permitted as an I-O statement in the following format:

```
WRITE... [FROM...] [{BEFORE | AFTER} ...]
         [AT END-OF-PAGE...]
         [NOT AT END-OF-PAGE...]
         [END-WRITE]
```

In this mode, it is possible to create new sequential files (on tape or disk). Each WRITE statement places one record in the file. See section “Line-sequential files” for notes on the creation of print files.

OPEN INPUT or
OPEN INPUT...REVERSED

READ is permitted as an I-O statement in the following format:

```
READ...[NEXT]
       [INTO...]
       [AT END...]
       [NOT AT END...]
       [END-READ]
```

In this mode sequential files can be read (from disk or tape). Each READ statement reads one record from the file.

The OPEN INPUT...REVERSED phrase causes the records to be read in reversed order, beginning with the last record in the file.

OPEN EXTEND

WRITE is permitted as an I-O statement in the following format:

```
WRITE... [FROM...] [{BEFORE | AFTER} ...]
         [AT END-OF-PAGE...]
         [NOT AT END-OF-PAGE...]
         [END-WRITE]
```

In this mode, new records can be appended to the end of a sequential file. Already existing records are not overwritten.

OPEN I-O

READ and REWRITE are permitted as I-O statements in the following formats:

```
READ   [NEXT]
       [INTO...]
       [AT END...]
       [NOT AT END...]
       [END-READ]

REWRITE...[FROM...]
          [END-REWRITE]
```

In this mode, records of a sequentially organized disk file can be retrieved (READ), updated by the program and subsequently written back (REWRITE) to disk. It must be noted, however, that a record can only be written back with REWRITE if

- it was previously retrieved by a successful READ statement, and
- its record length was not changed during the update.

The OPEN I-O phrase is only permitted for disk files.

9.2.5 Line-sequential files

The line-sequential organization used in COBOL files is a language element defined by the X/Open standard. The corresponding language format is as follows:

```
FILE-CONTROL.
...
[ORGANIZATION IS] LINE SEQUENTIAL
...
```

In BS2000, a line-sequential file can be stored

- as a cataloged SAM file
- or as an element in a PLAM library.

This provides the option of processing both cataloged files and files in the form of library elements in a COBOL program.
Restrictions by comparison with record-sequential files:

- Only variable-length records are permissible (RECORD-FORMAT=V).
  Does not apply to ENABLE-UFS-ACCESS=YES.
- These files can only be opened with OPEN INPUT and OPEN OUTPUT, without the specifications REVERSED and NO REWIND.
- The only permissible input/output statements are READ (for OPEN INPUT) and WRITE (for OPEN OUTPUT).
- The only specification that can be made in the CLOSE statement is WITH LOCK.

As with record-sequential files, a line-sequential file can be linked to a current SAM file using the ADD-FILE-LINK command (see section “Assignment of cataloged files”).
They can be linked to a library element with the SDF-P command SET-VARIABLE, which must have the following structure:

```
/SET-VAR SYSIOL-name='*LIBRARY-ELEMENT(library,element[version],type)'
```

SYSIOL-name	S variable. name must be the external name of the file in the ASSIGN clause.
library	Name of the PLAM library
element	Name of the element
version	Version indicator. Permissible values are: <alphanum-name 1..24> / *UPPER[-LIMIT] / *HIGH[EST-EXISTING] / *INCR[EMENT] (only for write accesses) If no version is specified, the highest possible version is generated during write accesses; read operations access the highest available version.
type	Element type. Permissible values: S, M, J, H, P, U, F, X, R, D.

Line-sequential COBOL files can only be processed in library elements if the LMSLIB library exists under the TSOS ID.
When linking the program that is to process the line-sequential files, the $LMSLIB library must be specified in addition to CRTE in order to satisfy the requirements of external references.

Example 9-6

Generating a line-sequential file in a library element

Entries in the COBOL compilation unit:

```
...
FILE-CONTROL.
SELECT AFILE ASSIGN TO "LIBELEM"
    ORGANIZATION IS LINE SEQUENTIAL
...
PROCEDURE DIVISION.
...
    OPEN OUTPUT AFILE.
...
```

Assignment of library and element before the program is called:

```
/SET-VAR SYSIOL-LIBELEM=’*LIBRARY-ELEMENT(CUST.LIB,MEYER,S)’
```

i

1. The SET-VARIABLE command can be inserted in a BS2000 procedure if it is a structured SDF-P procedure. This kind of structured procedure is described in the “SDF-P” manual [28].
2. The phrases within the quotes in the SET-VARIABLE command must all be written in uppercase.

9.2.6 Creating print files

COBOL language elements for print files

COBOL2000 provides the following language elements for the creation of files that are to be printed:

- Specification of the symbolic device names in the ASSIGN clause
- The LINAGE clause in the file description entry
- The ADVANCING phrase and the END-OF-PAGE phrase in the WRITE statement.

The use of these language elements is detailed in the "COBOL2000 Reference Manual” [1]. The following table shows the use of the symbolic device names in conjunction with the WRITE statement and the generation of the associated control characters:

Symbolic device name	WRITE statement without ADVANCING phrase	WRITE statement with ADVANCING phrase	Comments
PRINTER literal	Standard spacing when ADVANCING is omitted as if AFTER 1 LINE had been specified; the first character of the record is available for user data.	The first character of the record is available for user data.	The place for the carriage control character is reserved by the compiler and is not accessible to the user.This type of printer supports specification of the LINAGE clause in the file description entry. Write statements both with and without the ADVANCING phrase specified are allowed for a given file.
PRINTER PRINTER01 - PRINTER99	As above.	As above.	The place for the carriage control character is reserved by the compiler and is not accessible to the user. The LINAGE clause is not permitted for this file. Use of WRITE statements with and without the ADVANCING phrase for the same file is not permitted. If this does occur, a WRITE AFTER ADVANCING is executed implicitly for the records without the ADVANCING phrase.
literal	Spacing is controlled by the first character in each logical record; the user must therefore supply the appropriate control character before every execution of such a WRITE statement.	Spacing is controlled by the first character in each logical record; the user must therefore supply the appropriate control character before every execution of such a WRITE statement.	Mixed use of WRITE statements both with and without specifications of the ADVANCING phrase is permitted. In either case, however, the user information of the printer record begins only with the second character of the record.

Table 20: Use of symbolic device names in conjunction with the WRITE statement

Line-feed control characters for print files

When a WRITE statement is executed, the control byte of all print files (whose ASSIGN clauses do not contain the “literal” specification) is automatically supplied with a feed control character, which causes the page to be advanced as specified in the ADVANCING phrase (see the two following tables). Should the ADVANCING phrase be omitted, single-line spacing is assumed. The place for the carriage control character is reserved by the compiler and is inaccessible to the user.

When “literal” is specified in the ASSIGN clause for a file, a line-feed control character can be supplied to the control byte in two ways:

- A WRITE statement with the ADVANCING phrase generates a feed control control character on execution causing the printer to be advanced as specified in the ADVANCING phrase.
- A WRITE statement without the ADVANCING phrase does not supply a value to the control byte; the required control character must be explicitly transferred to it prior to the execution of the statement.

  This allows the user not only to use feed control characters but also to define other feed control characters in the program (e.g. for special printers). Information concerning the validity of individual characters and how they are interpreted during printing is available in the relevant printer manuals.

Since carriage control characters are usually not printable, they must be defined in the program by means of the SYMBOLIC CHARACTERS clause, so as to ensure that they can be referenced in MOVE statements (see Example 9-7).

Depending on the output destination different feed control characters are generated:

	Feed with output to BS2000	**Feed with output to POSIX file system**
PRINTER literal	BS2000 feed control characters as per table 21 and table 22	Feed control characters and lines as per UNIX conventions
PRINTER	as above	as above
PRINTER01-99	as above	not supported
literal	as above	BS2000 feed control characters as per table 21, table 22

The following tables list feed control characters:

Advance by number of lines	Control characters for line spacing		
	After printing	Before printing	
		Hex code[1]	Printed form
1	01	40	(space)
2	02	41	non-printable
3	03	42	non-printable
. .	. .	. .	. .
11	0B	4A	c (CENT)
12	0C	4B	. (period)
13	0D	4C	< (less than)
14	0E	4D	((parenthesis
15	0F	4E	+ (plus sign)

Table 21: Feed control characters

[1] Due to hardware characteristics, the values of the second half-byte are 1 less than the desired number of lines.

Skip to punched tape channels[1]	Printer control character		
	After printing	Before printing	
		Hex. code	Printed form
1	81	C1	A
2	82	C2	B

3	83	C3	C
4	84	C4	D
5	85	C5	E
6	86	C6	F
7	87	C7	G
8	88	C8	H
10	8A	CA	non-printable
11	8B	CB	non-printable

Table 22: Feed control characters for feed via punched tape channels

[1] Skipping to channel 9 or 12 is not possible as these are reserved for an end-of-form condition.

The SPECIAL-NAMES paragraph of the Environment Division enables the user to assign a symbolic name to any hexadecimal value, thus ensuring that all such values (including non-printable line-feed control characters) can be addressed in the COBOL compilation unit (see “COBOL2000 Reference Manual” [1]). The example that follows illustrates how linefeed control characters can be defined in this way.

Example 9-7

Supplying a hexadecimal control character to the control byte

In this example, the hexadecimal value 0A is to be transferred to the control byte of the print record. This causes the printer to advance 10 lines after printing.

```
IDENTIFICATION DIVISION.
   ...
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT PRINT-FILE ASSIGN TO "OUTPUT".
CONFIGURATION SECTION.
   ...
SPECIAL-NAMES.
   ...
    SYMBOLIC CHARACTERS HEX-0A IS 11 ———————————————————————— (1)
   ...
DATA DIVISION.
FILE SECTION.
FD PRINTER-FILE
   ...
01 PRINT-RECORD.
    02 CONTROL-BYTE            PIC X.
    02 PRINT-LINE              PIC X(132).
   ...
PROCEDURE DIVISION.
   ...
    MOVE "CONTENT" TO PRINT-LINE.
    MOVE HEX-0A TO CONYTROL-BYTE. ————————————————————————— (2)
    WRITE PRINT-RECORD.
   ...
```

(1) The eleventh character of the EBCDIC character set - corresponding to the hexadecimal value 0A - is assigned the symbolic name HEX-0A.

(2) The MOVE statement refers to this symbolic name in order to transfer hexadecimal value 0A to the control byte.

Using ASA line-feed control characters

ASA line-feed control characters can only be used in files that are assigned with ASSIGN TO literal or ASSIGN TO data-name.
In addition, the following ADD-FILE-LINK command is required for the file to be processed:

ADD-FILE-LINK filename, REC-FORM=*VAR(*ASA)

The ASA control characters and the corresponding WRITE statements that can be used under these conditions are listed in the table below::

ASA line-feed control characters	Format of the WRITE statement
+	`WRITE ... BEFORE ADVANCING 0`
0	`WRITE ... AFTER ADVANCING 0 or 1`
-	`WRITE ... AFTER ADVANCING 2`
1	`WRITE ... AFTER ADVANCING PAGE or C01`
2	`WRITE ... AFTER ADVANCING C02`
3	`WRITE ... AFTER ADVANCING C03`
4	`WRITE ... AFTER ADVANCING C04`
5	`WRITE ... AFTER ADVANCING C05`
6	`WRITE ... AFTER ADVANCING C06`
7	`WRITE ... AFTER ADVANCING C07`
8	`WRITE ... AFTER ADVANCING C08`
A	`WRITE ... AFTER ADVANCING C10`
B	`WRITE ... AFTER ADVANCING C11`

Table 23: ASA line-feed control characters and corresponding WRITE statements

9.2.7 Processing files in ASCII or in ISO 7-bit code

COBOL2000 supports the processing of sequential files in ASCII or ISO 7-bit code by means of the following clauses (see “COBOL2000 Reference Manual” [1]):

- ALPHABET alphabet-name-1 IS STANDARD-1(for ASCII code) or
  ALPHABET alphabet-name-1 IS STANDARD-2 (for ISO 7-bit code)
  in the SPECIAL-NAMES paragraph of the Configuration Section and
- CODE-SET IS alphabet-name-1 in the file description entry of the File Section.

ASCII code

The following program skeleton indicates the phrases that must be entered in the COBOL compilation unit in order to process a file in ASCII code.

```
IDENTIFICATION DIVISION.
...
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
...
SPECIAL-NAMES.
...
   ALPHABET alphabetname-1 IS STANDARD-1 ————————————————————————(1)
    ...
DATA DIVISION.
FILE SECTION.
FD file
   CODE-SET IS alphabetname-1 ——————————————————————————————————(2)
    ...
```

(1) The ALPHABET clause links code type STANDARD-1 (i.e. ASCII code) to the name alphabet-name-1.

(2) The CODE-SET clause specifies the code type connected with alphabet-name-1 as the character set for the file.

ISO 7-bit code

The declarations to be made in the compilation unit for the processing of a file in ISO 7-bit code are similar to those specified for the ASCII code above.
The only difference is that the keyword entered in the ALPHABET clause must be STANDARD-2 instead of STANDARD-1.
For magnetic tape files in ISO 7-bit code there is a further alternative (see also section “Processing magnetic tape files”): namely, to specify SUPPORT=TAPE(CODE=ISO7) for the file assignment in the ADD-FILE-LINK command.

9.2.8 Processing magnetic tape files

COBOL2000 supports the processing of magnetic tape files by means of the following language elements (see “COBOL2000 Reference Manual” [1]):

- The INPUT...REVERSED and WITH NO REWIND phrases in the OPEN statement:

  Each of these phrases prevents the file position indicator being set to the start of the file when the file is opened.

  INPUT...REVERSED causes a file to be positioned at its last record when the file is opened. Records of the file can then be read in reversed (i.e. descending) order.

  WITH NO REWIND can be specified for OPEN INPUT as well as OPEN OUTPUT and prevents the file from being repositioned when the OPEN statement is executed.

- The REEL, WITH NO REWIND and FOR REMOVAL phrases in the CLOSE statement:

  REEL is permitted only for multi-volume files, i.e. files that are distributed over more than one data volume (magnetic tape reels in this case). Depending on the open mode of each file, this phrase initiates the execution of different volume-closing operations at the end of the current reel (see CLOSE statement in “COBOL2000 Reference Manual” [1]). If the WITH NO REWIND or FOR REMOVAL has also been specified, the actions associated with these phrases (see below) are also performed at the end of the volume.

  The WITH NO REWIND phrase causes the current reel to be left in its current position (i.e. not repositioned to the start of the reel) when processing of the file or reel is closed.

  FOR REMOVAL indicates that the current reel is to be unloaded at the end of the file or at the end of the magnetic tape reel.

Assignment of magnetic tape files

Like disk files, magnetic tape files can also be assigned via the ADD-FILE-LINK command and supplied with attributes (see section “Assignment of cataloged files” and section “Definition of file attributes”). A detailed description of the command format for tape files is provided in “Commands” manual [3] and “Introductory Guide to DMS” [4].

Example 9-8

Assigning a tape file

```
/SEC-RESOURCE-ALLOC,TAPE=PAR(VOL=CA176B,TYPE=T6250,ACCESS=WRITE) ————(1)
/CREATE-FILE STOCK.NEW,SUPPORT=TAPE(VOLUME=CA176B,DEVICE-TYPE=T6250)——(2)
/ADD-FILE-LINK OUTFILE,STOCK.NEW ————————————————————————————————(3)
/START-PROGRAM *LIB(PLAM.LIB,UPDATE) ————————————————————————————(4)
...
/REMOVE-FILE-LINK OUTFILE,UNLOAD-RELEASED-TAPE=YES————————————————(5)
```

(1) For batch operations in particular, it is recommended that the required private volumes and devices be reserved with SECURE-RESOURCE-ALLOCATION before processing begins. In the above example, the tape with VSN CA176B is requested on a tape device with a recording density of 6250 bpi (TYPE=T6250) and a mounted write-permit ring (ACCESS=WRITE).

(2) The CREATE-FILE command

- catalogs the file STOCK.NEW as a tape file and
- links the volume serial number (VOLUME) and the tape device (DEVICE-TYPE)

(3) The ADD-FILE-LINK command links the file name STOCK.NEW with the link name OUTFILE.

(4) START-PROGRAM calls the problem program that is stored as a program under the element name UPDATE in the PLAM library PLAM.LIB.

(5) On completion of processing, the REMOVE-FILE-LINK command

- releases the link between the file STOCK.NEW and the link name OUTFILE and
- causes the tape CA176B to be unloaded; the tape device is released by default.

9.2.9 I-O status

The status of each access operation performed on a file is stored by the runtime system in specific data items which can be assigned to every file in the program. These items, which are specified by using the FILE STATUS clause, provide information on

- whether the I-O operation was successful, and
- the type of any errors that may have occurred.

This data can be evaluated (by USE procedures in the DECLARATIVES, for example) and used by the program to analyze I-O errors. As an extension to Standard COBOL, COBOL2000 provides the option of including the keys of the DMS error messages in this analysis, thus allowing a finer differentiation between different causes of errors.

The FILE STATUS clause is specified in the FILE-CONTROL paragraph of the Environment Division. Its format is (see “COBOL2000 Reference Manual” [1]):

```
FILE STATUS IS data-name-1 [data-name-2]
```

where data-name-1 and data-name-2 (if specified) must be defined in the WORKING-STORAGE SECTION or the LINKAGE SECTION. The following rules apply with regard to the format and possible values for these two items:

data-name-1

- must be declared as a two-byte alphanumeric data item, e.g.

```
01 data-name-1    PIC X(2).
```

- contains a two-character numeric status code following each access operation on the associated file. The table provided at the end of this section lists all such codes together with their meanings.

data-name-2

- must be declared as a 6-byte group item with the following format:

```
01 data-name-2.
   02 data-name-2-1     PIC 9(2) COMP.
   02 data-name-2-2     PIC X(4).
```

- is used for storing the DMS error code for the relevant I-O status. Following each access operation on the associated file, data-name-2 contains a value that directly depends on the content of data-name-1. The relationship between the values is shown in the table below:

Contents of data-name-1 not equal 0?	DMS code not equal 0?	Value of data-name-2-1	Value of data-name-2-2
no	no	undefined	undefined
yes	no	0	undefined
yes	yes	64	DMS code of the associated error message

The DMS codes and the associated error messages are given in “Introductory Guide to DMS” [4].

Caution

For *line*-sequential files, the only I-O status available is the one represented by data-name 1.

The status values and their meanings generally refer to *record*-sequential files. When *line*-sequential files are being processed, due consideration must be given to the peculiarities of line-sequential organization with regard to the interpretation of status values (see section “Line-sequential files”).

I-O status	Meaning
	Execution successful
00	The I-O statement terminated normally. No further information regarding the I-O operation is available.
04	Record length conflict: A READ statement terminated normally. However, the length of the record read lies outside the limits defined in the record description entry for this file.
05	Successful execution of an OPEN INPUT/I-O/EXTEND on a file; however, the referenced file indicated by the OPTIONAL phrase was not present at the time the OPEN statement was executed.
07	1. Successful OPEN statement with NO REWIND clause on a file that is on a UNIT-RECORD medium. 2. Successful CLOSE statement with NO REWIND, REEL/UNIT, or FOR REMOVAL clause on a file that is on a UNIT-RECORD medium.
10	Execution unsuccessful: AT END condition 1. An attempt was made to execute a READ statement. However, no next logical record existed, because the end-of-file was encountered. 2. A sequential READ statement with the OPTIONAL phrase was attempted for the first time on a nonexistent file.
30	Execution unsuccessful: unrecoverable error 1. No further information regarding the I-O operation is available (the DMS code provides further information). 2. During line-sequential processing: access to a PLAM element was unsuccessful
34	An attempt was made to write outside the sequential file boundaries set by the system.
35	An OPEN statement with the INPUT/I-O phrase was attempted on a nonexistent file.
37	OPEN statement on a file that cannot be opened in any of the following ways: 1. OPEN OUTPUT/I-O/EXTEND on a write-protected file (password, RETENTION-PERIOD, ACCESS=READ in catalog) 2. OPEN I-O on a tape file 3. OPEN INPUT on a read-protected file (password)

38	An attempt was made to execute an OPEN statement for a file previously closed with the LOCK phrase.
39	The OPEN statement was unsuccessful as a result of one of the following conditions: 1. One or more of the operands ACCESS-METHOD, RECORD-FORMAT or RECORD-SIZE were specified in the ADD-FILE-LINK command with values deviating from the corresponding explicit or implicit program specifications. 2. Record length errors occurred for input files (catalog check if RECFORM=F). 3. The record size is greater than the BLKSIZE entry in the catalog (in the case of input files). 4. The catalog entry of one of the FCBTYPE, RECFORM or RECSIZE (if RECFORM=F) operands for an input file is in conflict with the corresponding explicit or implicit program specifications or with the specifications in the ADD-FILE-LINK command.
	Execution unsuccessful: logical error
41	An attempt was made to execute an OPEN statement for a file which was already open.
42	An attempt was made to execute a CLOSE statement for a file which was not open.
43	While accessing a disk file opened with OPEN I-O, the most recent I-O statement executed prior to a REWRITE statement was not a successfully executed READ statement.
44	Boundary violation: 1. An attempt was made to execute a WRITE statement. However, the length of the record is outside the range allowed for this file. 2. An attempt was made to execute a REWRITE statement. However, the record to be rewritten did not have the same length as the record to be replaced.
46	An attempt was made to execute a READ statement for a file in INPUT or I-O mode. However, there is no valid next record since: 1. the preceding READ statement was unsuccessful without causing an AT END condition 2. the preceding READ statement resulted in an AT END condition.
47	An attempt was made to execute a READ statement for a file not in INPUT or I-O mode.
48	An attempt was made to execute a WRITE statement for a file not in OUTPUT or EXTEND mode.
49	An attempt was made to execute a REWRITE statement for a file not open in I-O mode.

	Other conditions with unsuccessful execution
90	System error; no further information available regarding the cause.
91	System error; a system call terminated abnormally; either an OPEN error or no free device; the actual cause is evident from the DMS code (see “FILE STATUS clause”)
95	Incompatibility between values specified in the BLOCK-CONTROL-INFO or BUFFER-LENGTH operand of the ADD-FILE-LINK command and the file format, block size, or the format of the used volume.

Table 24: I-O status for sequential files

9.3 Relative file organization

- Characteristics of relative file organization
- COBOL language tools for processing relative files
- Permissible record formats and access modes
- Open modes and types of processing (relative files)
- Random creation of a relative file
- I-O status

9.3.1 Characteristics of relative file organization

Each record in a relatively organized file is assigned a number which indicates its position in the file: the first record is assigned number 1, the second, number 2 etc.

By specifying a relative key data item in the program, the user can access any record of the file directly (randomly) via its relative record number. In addition to the options provided by sequential file organization, relatively organized files

- can be randomly created, i.e. records can be placed on the file in any order,
- can be randomly processed, i.e. records can be retrieved and updated in any order,
- permit the insertion of records, provided the desired position (relative record number) is not yet occupied,
- permit the logical deletion of existing records.

For the processing of relative files, COBOL programs use the DMS access methods ISAM and UPAM (see “Introductory Guide to DMS” [4]), which permit several users to update the file simultaneously (see section “Shared updating of files (SHARED-UPDATE)”).

Existing files have a defined FCBTYPE. For files to be created, FCBTYPE ISAM is always set unless FCBTYPE PAM has been defined with the ACCESS-METHOD operand of the ADD-FILE-LINK command of FCBTYPE *UPAM (SAM is rejected with an error message).

In the following cases, only FCBTYPE ISAM can be specified:

- if a variable record length is explicitly specified in the RECORD clause and/or
- if OPEN EXTEND is specified and/or
- if READ REVERSED is specified

When a relative file is mapped to ISAM, the 8-byte record key (hexadecimal) is inserted before the start of the record. The relative record key is mapped onto the key of the indexed file.

Relative files can only be stored on disk.

File structure

A description of the file structure of an ISAM file is given in section “Characteristics of indexed file organization”.

The following considerations apply to PAM files:

In terms of its logical structure, a PAM file may be seen as a sequence of areas of equal length, each capable of holding one record (only fixed-length records are allowed for PAM files). Each of these areas can be accessed by means of its relative record number.If the file is created sequentially, each of these areas - starting with the first - is filled in turn with a data record; no area can be skipped.

In the case of random creation, each record is written to the area whose relative record number was supplied in the relative key field of the record prior to the output statement. The program computes the associated position in the file on the basis of the specified record number and the record length. Unoccupied areas that are skipped during output are then created as empty records, i.e. the program reserves storage areas equal to the record length and supplies the first byte of each such area with the hexadecimal value FF (HIGH-VALUE), which identifies it as an empty record (see section “Open modes and types of processing (relative files)”).
PAM files can only be created on “key disks”, i.e., the BLOCK-CONTROL=PAMKEY entry is needed in the ADD-FILE-LINK command (see section “Disk and file formats”).

9.3.2 COBOL language tools for processing relative files

The following program skeleton summarize the most important clauses and statements provided in COBOL2000 for the processing of relative files. The most significant phrases and entries are briefly explained thereafter:

```
IDENTIFICATION DIVISION.
...
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT internal-file-name
    ASSIGN TO external-name
    ORGANIZATION IS RELATIVE
    ACCESS MODE IS mode RELATIVE KEY IS key
    FILE STATUS IS status-items.
    ...
DATA DIVISION.
FILE SECTION.
FD  internal-file-name
    BLOCK CONTAINS block-length-spec
    RECORD CONTAINS record-length-spec
    ...
01  data-record.
    nn item-1                type&length.
    nn item-2                type&length.
    ...
WORKING-STORAGE SECTION.
    ...
    nn key                   type&length
    ...
PROCEDURE DIVISION.
    ...
    OPEN open-mode internal-file-name
    ...
    START internal-file-name
    ...
    READ internal-file-name
    ...
    REWRITE data-record
    ...
    WRITE data-record
    ...
    DELETE internal-file-name
    ...
    CLOSE internal-file-name
    ...
    STOP RUN.
```

`SELECT internal-file-name`

specifies the name by which the file is to be addressed in the compilation unit.
internal-file-name must be a valid user-defined word.

The format of the SELECT clause also permits the OPTIONAL phrase to be specified for input files that need not necessarily be present during the program run.

If, during program execution, no file has been assigned to a file name declared with SELECT OPTIONAL, then:

- in the case of OPEN INPUT, program execution is interrupted with message COB9117 and an ADD-FILE-LINK command is requested (in dialog mode), or the AT END condition is initiated (in batch mode);
- in the case of OPEN I-O or OPEN EXTEND, a file named FILE.COBOL.linkname is created.

`ASSIGN TO external-name`

specifies the system file associated with the file and defines the name via which a cataloged file can be assigned.

external-name must be either

- a permissible literal,
- a permissible data name defined in the DATA division, or
- a valid implementor name

from the ASSIGN clause format (see “COBOL2000 Reference Manual” [1]).

`ORGANIZATION IS RELATIVE`

specifies that the file is organized as a relative file.

`ACCESS MODE IS mode`

specifies the mode in which the records in the file can be accessed.

The following may be specified for mode (see also section “Open modes and types of processing (relative files)”):

SEQUENTIAL	specifies that the records can only be processed sequentially.
RANDOM	declares that the records can only be accessed in random mode.
DYNAMIC	allows the records to be accessed in either sequential or random mode.

The ACCESS MODE clause is optional. If it is not specified, the compiler assumes the default value ACCESS MODE IS SEQUENTIAL.

`RELATIVE KEY IS key`

specifies the relative key data item for holding the relative record numbers in the case of random access to the records.

key must be declared as an unsigned integer data item and must not be a part of the associated record description entry.
In the case of random access, the relative record number of the record to be processed must be supplied in key before each I-O operation.

The RELATIVE KEY phrase is optional for files for which ACCESS MODE IS SEQUENTIAL is declared; it is mandatory when ACCESS MODE IS RANDOM or DYNAMIC is specified.

`FILE STATUS IS status-items`

specifies the data items in which the runtime system will store status information after each access to a file. This information indicates

- whether the I-O operation was successful and
- the type of any errors that may have occurred.

The status items must be declared in the WORKING-STORAGE SECTION or the LINKAGE SECTION. Their format and the meaning of the various status codes are described in section “I-O status”.

The FILE STATUS clause is optional. If it is not specified, the information mentioned above is not available

BLOCK CONTAINS block-length-spec

defines the maximum size of a logical block. It determines how many records are to be transferred together by each I-O operation into/from the buffer of the program.

block-length-spec must be an integer and must not be shorter than the record length of the file or greater than 32767. It specifies the size of the logical block in bytes. The blocking of records reduces

- the number of accesses to peripheral storage and thus the runtime of the program;
- the number of interblock gaps on the storage medium and thus the physical storage space required by the file.

On the other hand, access employing the locking mechanism during shared update processing (see section “Shared updating of files (SHARED-UPDATE)”) cause the entire block containing the current record to be locked. In each case, therefore, a large blocking factor would lead to a reduction in processing speed.

During compilation, the compiler calculates a value for the buffer size on the basis of the record and block length entries given in the compilation unit. The runtime system rounds up this value for DMS to the next multiple of a PAM block (2048 bytes). This default value can be modified during the file assignment by specifying the BUFFER-LENGTH operand in the ADD-FILE-LINK command. It must be noted, however, that the specified buffer needs to be at least as large as the longest data record.

Except in the case of newly created files (OPEN OUTPUT), the block size entered in the catalog always takes priority over block size specifications in the program or ADD-FILE-LINK command.

The BLOCK CONTAINS clause is optional. If it is not specified, the compiler assumes the record length of the file as the block size.

RECORD record-length-spec

- specifies whether fixed or variable length records are to be processed and
- defines, for variable length records, a range for the valid record sizes and, if specified in the format, a data item to contain the current record length information.

record-length-spec must match one of the three formats in the RECORD clause supported by COBOL2000. It must not conflict with the record lengths the compiler computes from the specifications in the associated record description entry or entries.

The RECORD clause is optional. If it is not specified, the compiler assumes that the records are of variable length.

```
01 data-record.
   nn item-1      type&length
   nn item-2      type&length
```

represents a record description entry for the associated file. It describes the logical format of data records.

At least one record description entry is required for each file. If more than one record description entry is specified for a file, the declared record format must be considered:

- For records of fixed length all record description entries must be of the same size,
- for records of variable length they must not conflict with the record length specification in the RECORD clause.

The subdivision of data-record into data items (item-1, item-2, ...) is optional. For type&length, the required length and format declarations (PICTURE and USAGE clauses etc.) must be entered.
The relative key data item declared in the RELATIVE KEY phrase must not be subordinate to data-record.

`nn key type&length`

defines the relative key data item specified in the RELATIVE KEY phrase.

When specifying values for type&length, it should be noted that key must be an unsigned integer data item.
In the case of random access, the relative record number of the record to be processed must be supplied in key before each I-O operation.

`OPEN open-mode internal-file-name`

opens the file for processing in the specified open mode.

The following phrases can be entered for open-mode:

INPUT	opens the file as an input file; it can only be read.
OUTPUT	opens the file as an output file; it can only be written.
EXTEND	opens the file as an output file; it can be extended.
I-O	opens the file as an I-O file; it can be read (one record at a time), updated and rewritten.

The open-mode entry determines with which I-O statements a file may be accessed (see section “Open modes and types of processing (relative files)”).

```
START internal-file-name
READ internal-file-name
REWRITE data-record
WRITE data-record
DELETE internal-file-name
```

are I-O statements for the file that

- position to a record in the file
- read a record
- rewrite a record
- write a record, and
- delete a record.

The open mode declared in the OPEN statement determines which of these statements is admissible for the file. The relationship between access mode and open mode is described in section “Open modes and types of processing (relative files)”.

`CLOSE internal-file-name`

terminates processing of the file.

The WITH LOCK phrase can be additionally specified to prevent the file from being opened again in the same program run.

9.3.3 Permissible record formats and access modes

Record formats

Relative files may contain fixed-length records (RECFORM=F) or variable-length records (RECFORM=V). In either case the records may be blocked or unblocked. COBOL compilation units permit the format of records to be processed to be declared in the RECORD clause. The phrases that are associated with the record format concerned are summarized in the following table:

Record format	Phrase in the RECORD clause	
Fixed length records	RECORD CONTAINS...CHARACTERS	(Format 1)
Variable length records	RECORD IS VARYING IN SIZE... or RECORD CONTAINS...TO...	(Format 2) (Format 3)

Table 25: Record format and RECORD clause

Access modes

Access to records of a relative file may be sequential, random, or dynamic.

In the COBOL compilation unit, the access mode is defined with the ACCESS MODE clause. The following table lists the possible phrases together with their effect on the access mode.

Phrase in the ACCESS MODE clause	Access mode
SEQUENTIAL	Sequential access: The records can be processed only in the order in which they appear in the file. This order is determined by the relative record number. In other words: When reading records, the next or previous record is made available. When writing records, each successive record output to the file is assigned the next relative record number; no empty records are written.
RANDOM	Random access: The records can be accessed in any order via the relative record number. For this purpose, the relative record number of the record to be processed must be supplied in the RELATIVE KEY item prior to each I-O operation.
DYNAMIC	Dynamic access: The records can be accessed sequentially as well as randomly. The applicable access mode is selected via the format of the I-O statement in this case.

Table 26: ACCESS MODE clause and access mode

9.3.4 Open modes and types of processing (relative files)

The various language elements available for use in a COBOL program allows relative files to be

- created,
- read,
- extended (by adding new records), and
- updated (by changing or deleting existing records).

The individual I-O statements that may be used in the program to process a given file are determined by its open mode, which is specified in the OPEN statement:

OPEN OUTPUT

Regardless of the phrase in the ACCESS MODE clause, WRITE is permitted as an I-O statement in the following format:

```
WRITE...[FROM...]
        [INVALID KEY...]
        [NOT INVALID KEY...]
        [END WRITE...]
```

In this mode, it is only possible to create (load) new relative files. Depending on the specified access mode, the WRITE statement has the following effect:

- ACCESS MODE IS SEQUENTIAL

  allows a relative file to be created sequentially i.e. WRITE writes records to the file successively with ascending relative record numbers (starting with 1).
  The RELATIVE KEY key item - if specified - is not evaluated by WRITE; it is loaded with the value of the (automatically incremented) relative record number of the last record to be written.
- ACCESS MODE IS RANDOM or DYNAMIC

  (both phrases have the same meaning here) enables the random creation of a file. WRITE causes each record to be written to the position in the file that is indicated by its relative record number.
  Thus, before each WRITE statement is executed, the RELATIVE KEY item must be supplied with the relative record number which the record to be written is to receive in the file. If the number of an already existing record is specified, the INVALID KEY condition occurs, and WRITE branches to the INVALID KEY statement or to the declared USE procedure without writing the record. It is thus not possible to overwrite records in this case.

OPEN EXTEND

OPEN EXTEND is used to extend an existing file. Access is only possible in sequential mode.

- ACCESS MODE IS SEQUENTIAL

  permits a relative file to be extended sequentially. Starting with the highest key+1, WRITE writes the records with continuous, ascending record numbers to the file.The RELATIVE KEY key item, if specified, is not interpreted by WRITE; it contains the (automatically incremented) relative record number of the last written record.

OPEN INPUT

The phrase specified in the ACCESS MODE clause determines which I-O statements or statement formats are permitted. The following table lists the options available for OPEN INPUT:

Statement	Entry in the ACCESS MODE clause

	SEQUENTIAL	RANDOM	DYNAMIC
START	START... [KEY IS...] [INVALID KEY...] [NOT INVALID KEY...] [END-START]	Statement not permitted	START... [KEY IS...] [INVALID KEY...] [NOT INVALID KEY] [END-START]
READ	READ...[NEXT | PREVIOUS] [INTO...] [AT END...] [NOT AT END...] [END-READ...]	READ... [INTO...] [KEY IS...] [INVALID KEY...] [NOT INVALID KEY...] [END-READ]	For sequential access: READ...{NEXT | PREVIOUS} [INTO...] [AT END...] [NOT AT END...] [END-READ] For random access: READ... [INTO...] [KEY IS...] [INVALID KEY...] [NOT INVALID KEY...] [END-READ]

Table 27: Permitted I-O statements for OPEN INPUT

Relative files can be read in this mode. Depending on the access mode specified, the READ statement has the following effect:

- ACCESS MODE IS SEQUENTIAL

  permits the file to be read sequentially only. READ retrieves records in ascending (NEXT) or descending (PREVIOUS) order, based on the value of the relative record numbers.
  The RELATIVE KEY key data item - if specified - is not evaluated by READ; it contains the relative record number of the last record that was read. However, if a RELATIVE KEY key data item was declared, a START statement can be used to position to any record of the file before execution of a READ statement: START uses a relation condition to establish the relative record number of the first record to be read and thus defines the starting point for sequential read operations that follow.
  If the comparison relation condition cannot be satisfied by any relative record number of the file, an INVALID KEY condition exists, and START transfers control to the imperative statement specified in the INVALID KEY phrase or to the declared USE procedure.
- ACCESS MODE IS RANDOM

  enables records of the file to be read randomly. READ retrieves the records in the specified order; access to each record is effected via its relative record number.Accordingly, the relative record number of each record to be read must be supplied in the RELATIVE KEY data item prior to each READ operation. If the number of a unavailable record (e.g. an empty record) is specified, the INVALID KEY condition exists, and READ transfers control to the INVALID KEY statement or to the USE procedure declared for this condition.
- ACCESS MODE IS DYNAMIC

  permits random and sequential reading of the file. The applicable access mode is selected via the format of the READ statement (see table 27).
  In this case, a START statement is meaningful for sequential reading only.

`OPEN I-O`

The phrase specified in the ACCESS MODE clause determines which I-O statements or statement formats are permitted. The following table lists the options available for OPEN I-O:

Statement	**Entry in the ACCESS MODE clause**		

	SEQUENTIAL	RANDOM	DYNAMIC
START	START... [KEY IS...] [INVALID KEY...] [NOT INVALID KEY...] [END-START]	Statement not permitted	START... [KEY IS...] [INVALID KEY...] [NOT INVALID KEY] [END-START]
READ	READ...[NEXT | PREVIOUS] [INTO...] [AT END...] [NOT AT END...] [END-READ...]	READ... [INTO...] [KEY IS...] [INVALID KEY...] [NOT INVALID KEY...] [END-READ]	For sequential access: READ...{NEXT | PREVIOUS} [INTO...] [AT END...] [NOT AT END...] [END-READ]
			For random access: READ... [INTO...] [KEY IS...] [INVALID KEY...] [NOT INVALID KEY...] [END-READ]
REWRITE	REWRITE... [FROM...] [INVALID KEY...] [NOT INVALID KEY...] [END-REWRITE]	REWRITE... [FROM...] [INVALID KEY...] [NOT INVALID KEY...] [END-REWRITE]	REWRITE... [FROM...] [INVALID KEY...] [NOT INVALID KEY...] [END-REWRITE]
WRITE	Statement not allowed	WRITE... [FROM...] [INVALID KEY...] [NOT INVALID KEY...] [END-WRITE]	WRITE... [FROM...] [INVALID KEY...] [NOT INVALID KEY...] [END-WRITE]
DELETE	DELETE... [END-DELETE]	DELETE... [INVALID KEY...] [NOT INVALID KEY...] [END-DELETE]	DELETE... [INVALID KEY...] [NOT INVALID KEY...] [END-DELETE]

Table 28: Permitted I-O statements for OPEN I-O

When a file is opened in this mode, records can be

- read,
- added,
- updated by the program, and
- overwritten or
- deleted.

OPEN I-O assumes that the file to be processed already exists. It is therefore not possible to create a new relative file in this mode.

The specified access mode determines the type of processing that can be performed, as well as the effect of the individual I-O statements:

- ACCESS MODE IS SEQUENTIAL

  As in the case of OPEN INPUT, this access mode permits the sequential reading of a file with READ and the use of a preceding START to position to any record as the starting point.

  In addition, the record that is read by a successful READ operation can be updated by the program and then rewritten with REWRITE or logically deleted with DELETE. However, no other I-O statement should be executed for this file between the READ and REWRITE or DELETE statements.

- ACCESS MODE IS RANDOM

  enables records to be randomly retrieved with READ (as in OPEN INPUT).

  In addition, new records can be inserted into the file with WRITE, and existing records in the file can be rewritten or deleted with REWRITE or DELETE (regardless of whether they were read earlier).

  Prior to each WRITE, REWRITE, or DELETE statement, the RELATIVE KEY data item must be supplied with the relative number of the record that is to be added, rewritten, or deleted. If the following cases apply the number of an already existing record is specified for WRITE, or the number of a unavailable record (e.g. an empty record) is specified for REWRITE or DELETE, an INVALID KEY condition exists, and WRITE, REWRITE, or DELETE branches to the INVALID KEY statement or to the USE procedure declared for this event.

- ACCESS MODE IS DYNAMIC

  allows a file to be processed sequentially or randomly. Here, the desired access mode is selected via the format of the READ statement.

9.3.5 Random creation of a relative file

The following example illustrates a simple COBOL program with which a relative file can be randomly created. The records may be written to the file in any order.

Example 9-9

Program for the random creation of a relative file

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. RELATIV.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SPECIAL-NAMES.
     TERMINAL IS T.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT RELATIVE-FILE
     ASSIGN TO "RELFILE"
     ORGANIZATION IS RELATIVE
    ACCESS MODE IS RANDOM ————————————————————————————(1)
    RELATIVE KEY IS REL-KEY ——————————————————————————(2)
    FILE STATUS IS FS-CODE DMS-CODE. ———————————————————(3)
 DATA DIVISION.
 FILE SECTION.
 FD RELATIVE-FILE.
 01 RELATIVE-RECORD               PIC X(33).
 WORKING-STORAGE SECTION.
 01 REL-KEY                       PIC 9(3).
    88 END-OF-INPUT                          VALUE ZERO.
 01 I-O-STATUS.
    05 FS-CODE                    PIC 9(2).
    05 DMS-CODE.
       06 DMS-CODE-1              PIC 9(2) COMP.
          88 DMS-CODE-2-DEFINED              VALUE 64.
       06 DMS-CODE-2              PIC X(4).
 01 CLOSE-SCWITCH                 PIC X      VALUE "0".
    88 FILE-OPEN                             VALUE "1".
    88 FILE-CLOSED                           VALUE "0".
 01 RELATIVE-TEXT.
    05                           PIC X(24)
      VALUE „******HERE IS RECORD NO. „.
    05 REC-NO                    PIC 9(3).
    05                           PIC X(6)   VALUE "$$$$$$".
 PROCEDURE DIVISION.
 DECLARATIVES.
 OUTPUT-ERROR SECTION.
   USE AFTER STANDARD ERROR PROCEDURE ON RELATIVE-FILE.
UNRECOVERABLE-ERROR. ————————————————————————————————(4)
     IF FS-CODE NOT LESS THAN 30
        DISPLAY „UNRECOVERABLE ERROR ON RELATIVE-FILE“
         UPON T
      DISPLAY „FILE STATUS: „ FS-CODE UPON T
      IF DMS-CODE-2-DEFINED
         DISPLAY „DMS-CODE: „ DMS-CODE-2 UPON T
      END-IF
      IF FILE-OPEN
         CLOSE RELATIVE-FILE
```

```
       END-IF
      DISPLAY „PROGRAM TERMINATED ABNORMALLY“ UPON T
       STOP RUN
   END-IF.
OUTPUT-ERROR-END.
       EXIT.
END DECLARATIVES.
INITIALIZATION.
    OPEN OUTPUT RELATIVE-FILE
    SET FILE-OPEN TO TRUE.
LOAD-FILE.
    PERFORM INPUT-RELATIVE-KEY
       WITH TEST AFTER
       UNTIL REL-KEY IS NUMERIC
    PERFORM WITH TEST BEFORE UNTIL END-OF-INPUT
       WRITE RELATIVE-RECORD FROM RELATIVE-TEXT
           INVALID KEY ————————————————————————————————(5)
             DISPLAY „RECORD NO. „ REL-KEY
                „ALREADY EXISTS IN FILE“ UPON T
        END-WRITE
        PERFORM INPUT-RELATIVE-KEY
           WITH TEST AFTER
           UNTIL REL-KEY IS NUMERIC
     END-PERFORM.
TRAILER.
    SET FILE-CLOSED TO TRUE
    CLOSE RELATIVE-FILE
    STOP RUN.
INPUT-RELATIVE-KEY.
   DISPLAY „PLEASE ENTER RELATIVE KEY: THREE-DIGIT WITH L
-  „EADING ZEROES“ UPON T
   DISPLAY „TERMINATE PROGRAM ENTERING ‚000‘“ UPON T
   ACCEPT REL-KEY FROM T
   IF REL-KEY NUMERIC
      THEN MOVE REL-KEY TO REC-NO
      ELSE DISPLAY „INPUT MUST BE NUMERIC“ UPON T
   END-IF.
```

(1) The ACCESS MODE clause specifies random access for records of the file named RELATIVE-FILE. They may thus be written to the file in any order during creation.

(2) The RELATIVE KEY clause defines REL-KEY as the relative key data item for the relative record number. It is declared in the Working-Storage Section as a three-digit numeric data item.

(3) The FILE STATUS clause is defined so as to make the DMS code available to the program in addition to the FILE STATUS code. The data items required for storing this information are declared in the Working-Storage Section and evaluated in the DECLARATIVES.

(4) The DECLARATIVES provide only one procedure for unrecoverable I-O errors (FILE STATUS >= 30), since an AT END condition cannot occur for output files and record key errors can be caught via INVALID KEY.

(5) An INVALID KEY condition occurs in the case of a random WRITE operation when the record with the associated relative record number is already present in the file.

9.3.6 I-O status

The status of each access operation performed on a file is stored by the runtime system in specific data items, which can be assigned to every file in the program. These items, which are specified by using the FILE STATUS clause, provide information on

- whether the I-O operation was successful, and
- the type of errors that may have occurred.

This data can be evaluated (by USE procedures in the DECLARATIVES, for example) and used by the program to analyze I-O errors. As an extension to Standard COBOL, COBOL2000 provides the option of including the key of the DMS error messages in this analysis, thus allowing a finer differentiation between different causes of errors. The FILE STATUS clause is specified in the FILE-CONTROL paragraph of the Environment Division. Its format is (see “COBOL2000 Reference Manual” [1]):

```
FILE STATUS IS data-name-1 [data-name-2]
```

where data-name-1 and data-name-2 (if specified) must be defined in the WORKING-STORAGE SECTION or the LINKAGE SECTION. The following rules apply with regard to the format and possible values for these two items:

data-name-1

- must be declared as a two-byte alphanumeric data item, e.g.

```
01 data-name-1      PIC X(2).
```

- contains a two-character numeric status code following each access operation on the associated file. The table provided at the end of this section lists all such codes together with their meanings.

data-name-2

- must be declared as a 6-byte group item with the following format:

```
01 data-name-2.
   02 data-name-2-1        PIC 9(2) COMP.
   02 data-name-2-2        PIC X(4).
```

- is used for storing the DMS error code for the relevant I-O status. Following each access operation on the associated file, data-name-2 contains a value that directly depends on the content of data-name-1. The relationship between the values is shown in the table below:

Is contents of data-name-1 not equal to 0?	Is DMS code not equal to 0?	Value of data-name-2-1	Value of data-name-2-2
no	no	undefined	undefined
yes	no	0	undefined
yes	yes	64	DMS code of the associated error message

The DMS codes and the associated error messages are given in “Introductory Guide to DMS” [4].

	Meaning

I-O status	
	Execution successful
00	The I-O statement terminated normally. No further information regarding the I-O operation is available.
04	Record length conflict: A READ statement was executed successfully, but the length of the record which was read does not lie within the limits specified in the record description for the file.
05	Successful OPEN INPUT/I-O/EXTEND for a file with the OPTIONAL phrase in the SELECT clause that was not present at the time of execution of the OPEN statement.
	Execution unsuccessful: AT END condition
10	An attempt was made to execute a READ statement. However, no next logical record was available, since the end-of-file was encountered (sequential READ). A first attempt was made to execute a READ statement for a non-existent file with the specification OPTIONAL. A first attempt was made to execute a READ statement for a non-existent file which is specified as OPTIONAL.
14	An attempt was made to execute a READ statement. However, the data item described by RELATIVE KEY is too small to accommodate the relative record number. (sequential READ).
	Execution unsuccessful: invalid key condition
22	Duplicate key An attempt was made to execute a WRITE statement with a key for which there is already a record in the file.
23	Record not located or zero record key An attempt was made (using a READ, START, DELETE or REWRITE statement with a key) to access a record not contained in the file, or the access was effected with a zero record key.
24	Boundary values exceeded. An attempt was made to execute a WRITE statement beyond the system-defined boundaries of a relative file (insufficient secondary allocation in the FILE command), or a WRITE statement is attempted in sequential access mode with a relative record number so large that it does not fit in the data item defined with the RELATIVE KEY phrase.
	Execution unsuccessful: permanent error
30	No further information regarding the I-O operation is available.
35	An attempt was made to execute an OPEN INPUT/I-O statement for a nonexistent file.
37	An OPEN statement is attempted on a file that cannot be opened due to the following conditions: 1. OPEN OUTPUT/I-O/EXTEND on a write-protected file (password, RETPD in catalog, ACCESS=READ in catalog) 2. OPEN INPUT on a read-protected file (password)

38	An attempt was made to execute an OPEN statement for a file previously closed with the LOCK phrase.
39	The OPEN statement was unsuccessful as a result of one of the following conditions: 1. One or more of the operands ACCESS-METHOD, RECORD-FORMAT or RECORD-SIZE were specified in the ADD-FILE-LINK command with values which conflict with the corresponding explicit or implicit program specifications. 2. The catalog entry of the FCBTYPE operand for an input file does not match the corresponding explicit or implicit program specification or the specification in the ADD-FILE-LINK command. 3. Variable record length has been defined for a file that is to be processed using the UPAM access method of the DMS.
	Execution unsuccessful: logical error
41	An attempt was made to execute an OPEN statement for a file which was already open.
42	An attempt was made to execute a CLOSE statement for a file which was not open.
43	For ACCESS MODE IS SEQUENTIAL: The most recent I-O statement executed prior to a DELETE or REWRITE statement was not a successfully executed READ statement.
44	Record length limits exceeded: An attempt was made to execute a WRITE or REWRITE statement, but the length of the record does not lie within the limits defined for the file.
46	An attempt was made to execute a sequential READ statement for a file in INPUT or I-O mode. However, there is no valid next record since: 1. the preceding START statement terminated abnormally, or 2. the preceding READ statement terminated abnormally without causing an AT END condition. 3. the preceding READ statement caused an AT END condition.
47	An attempt was made to execute a READ or START statement for a file not in INPUT or I-O mode.
48	An attempt was made to execute a WRITE statement for a file that • on sequential access is not in OUTPUT or EXTEND mode • on random or dynamic access is not in OUTPUT or I-O mode.
49	An attempt was made to execute a DELETE or REWRITE statement for a file not in I-O mode.
	Other conditions with unsuccessful execution
90	System error; no further information regarding the cause is available.
91	System error; OPEN error

93	For shared update processing only (see section “Shared updating of files (SHARED-UPDATE)”): The I-O statement could not terminate normally because a different task is accessing the same file, and the access operations are incompatible.
94	For shared update processing only (see section “Shared updating of files (SHARED-UPDATE)”): deviation from call sequence READ - REWRITE/DELETE.
95	Incompatibility between values specified in the BLOCK-CONTROL-INFO or BUFFER-LENGTH operand of the ADD-FILE-LINK command and the file format, block size, or the format of the used volume.
96	READ PREVIOUS is not supported for modules that were compiled with COBRUN ENABLE-UFS-ACCESS=YES or the file should be processed with the DMS UPAM access mode.

Table 29: I-O status for relative files

9.4 Indexed file organization

- Characteristics of indexed file organization
- COBOL language tools for the processing of indexed files
- Permissible record formats and access modes
- Open modes and types of processing (indexed files)
- Positioning with START
- I-O status

9.4.1 Characteristics of indexed file organization

Each record in an indexed file is assigned a key, i.e. a sequence of arbitrary (including non-printable) characters that uniquely identify the record in the file. The starting position (KEYPOS) and the length (KEYLEN) of the keys are identical for all records of a file.

The position and length of the key in the record are specified by means of a key data item defined in the program. An index is maintained for this item, enabling any record in the file to be accessed directly (randomly) via its record key. In addition to the options available for sequentially organized files, this facility allows the following operations in an indexed file:

- records can be created randomly,
- records can be read and updated randomly,
- records may be inserted at a later stage, and
- existing records can be logically deleted.

COBOL programs use the indexed sequential access method (ISAM) of DMS (see “Introductory Guide to DMS” [4]) for processed indexed files. This method allows several users to update a file simultaneously (see section “Shared updating of files (SHARED-UPDATE)”).

Indexed files can be created on disk storage only.

File structure

A detailed description of the structure of ISAM files is provided in “Introductory Guide to DMS” [4]; the following description merely provides a quick review of the most important facts:

An ISAM file consists of two components, each of which serve different functions:

- index blocks and
- data blocks.

If private volumes are used, index and data blocks may reside on different volumes.

- Data blocks contain the user’s records. These records are logically linked together in ascending order of their keys; their physical sequence on the volume is arbitrary.

  Data blocks can have a length of one PAM block (2048 bytes) or an integer multiple thereof (up to 16 PAM blocks).
- Index blocks serve to locate data records via the record key. They can be classified into various index levels:

  Index blocks of the lowest level contain pointers to data blocks, while index blocks of higher levels have pointers to the index blocks of the next lower level.

  The highest-level index block is always created in the file, even if the file does not hold any records. In addition to the pointers, this block contains a 36-byte area for ISAM label information.

  Entries in index blocks are always arranged physically in the order of ascending record keys; they must therefore be reorganized whenever new index or data blocks are generated in the level below.

  Index blocks have a fixed length of one PAM block.

Block splitting

When an ISAM file is extended, each new record is inserted into the data block to which it belongs, based on the value of its record key.
At times, the space available in this block may prove insufficient for the inclusion of a further record. In such cases,

the old block is split, and the resulting halves are entered into new (empty) blocks. The old block remains allocated to the file, but is now marked as free (see the “Introductory Guide to DMS” [4]).

Frequent splitting of blocks slows down processing. However, this can be largely avoided by reserving space in the data blocks for later extensions when the file is first created. This is achieved by using the PADDING-FACTOR operand in the ADD-FILE-LINK command when the output file is assigned. The PADDING-FACTOR operand serves to specify the percentage of a data block that is to remain free for subsequent extensions when loading the file.

Example 9-10

Use of the PADDING-FACTOR operand when assigning an ISAM file

When the file ISAM.OUTPUT is created, only about one in four records is to be initially available, with 75% of each data block being reserved for future extensions. This is accomplished by means of the following ADD-FILE-LINK command:

```
/ADD-FILE-LINK OUTFILE,ISAM.OUTPUT,ACCESS-METHOD=ISAM,PADDING-FACTOR=75
```

9.4.2 COBOL language tools for the processing of indexed files

The following program skeleton summarizes the most important clauses and statements provided in COBOL2000 for the processing of indexed files. The most significant phrases and entries are briefly explained thereafter:

```
IDENTIFICATION DIVISION.
...
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT internal-file-name
    ASSIGN TO external-name
    ORGANIZATION IS INDEXED
    ACCESS MODE IS mode
    RECORD KEY IS primary-key
    ALTERNATE RECORD KEY IS secondary-key
    FILE STATUS IS status-items.
    ...
DATA DIVISION.
FILE SECTION.
FD  internal-file-name.
    BLOCK CONTAINS block-length-spec
    RECORD record-length-spec
    ...
01  data-record.
    nn item-1 type&length
    ...
    nn primary-key-item type&length
    nn secondary-key-item type&length
    ...
PROCEDURE DIVISION.
    ...
    OPEN open-mode internal-file-name
    ...
    START internal-file-name
    ...
    READ internal-file-name
    ...
    REWRITE data-record
    ...
    WRITE data-record
    ...
    DELETE internal-file-name
    ...
    CLOSE internal-file-name
    ...
STOP RUN.
```

`SELECT internal-file-name`

specifies the name by which the file is to be addressed in the compilation unit.

internal-file-name must be a valid user-defined word.

The format of the SELECT clause also permits the OPTIONAL phrase to be specified for input files that need not necessarily be present during the program run.

If, during program execution, no file has been assigned to a file name declared with SELECT OPTIONAL, then:

- in the case of OPEN INPUT, program execution is interrupted with message COB9117 and an ADD-FILE-LINK command is requested (in dialog mode), or the AT END condition is initiated (in batch mode);
- in the case of OPEN I-O or OPEN EXTEND, a file named FILE.COBOL.linkname is created.

`ASSIGN TO external-name`

specifies the system file associated with the file and defines the name via which a cataloged file can be assigned.

external-name must be either

- a valid literal
- a valid data-name defined in the Data Division, or
- a valid implementor-name

from the ASSIGN clause format (see “COBOL2000 Reference Manual” [1]).

`ORGANIZATION IS INDEXED`

specifies that the file is organized as an indexed file.

`ACCESS MODE IS mode`

specifies the mode in which the records can be accessed.
The following may be specified for mode (see also section “Open modes and types of processing (indexed files)”):

SEQUENTIAL	specifies that the records can only be processed sequentially.
RANDOM	declares that the records can only be accessed in random mode.
DYNAMIC	allows the records to be accessed in either sequential or random mode.

The ACCESS MODE clause is optional. If it is not specified, the compiler assumes the default value ACCESS MODE IS SEQUENTIAL.

`RECORD KEY IS primary-key`

specifies which field in the record holds the primary record key.

primary-key must be declared as a data item in the associated record description entry (see below).

Except in sequential read operations, the primary record key of the record to be processed must be entered for primary-key before the execution of each I-O statement.

`ALTERNATE RECORD KEY IS secondary-key`

COBOL programs can also be used for processing files with records containing one or more secondary keys (ALTERNATE RECORD KEY) in addition to the mandatory primary record key (RECORD KEY).
If secondary keys are defined in a file, the user can access the records either via the primary key or via the secondary key(s).

The secondary key must be declared as a data item within the associated record description entry (see below).

`FILE STATUS IS status-items`

specifies the data items in which the runtime system stores status information after each access to a file. This information indicates

- whether the I-O operation was successful and

- the type of any errors that may have occurred.

The status items must be declared in the Working-Storage Section or the Linkage Section. Their format and the meaning of the various status codes are described in section “I-O status”.

The FILE STATUS clause is optional. If it is not specified, the information mentioned above is not available to the program.

`BLOCK CONTAINS block-length-spec`

defines the maximum size of a logical block. It determines how many records are to be transferred together by each I-O operation into/from the buffer of the program.

block-length-spec must be a permissible specification from the BLOCK CONTAINS clause.

The blocking of records reduces

- the number of accesses to peripheral storage and thus the runtime of the program;
- the number of interblock gaps on the storage medium and thus the physical storage space required by the file.

On the other hand, accesses employing the locking mechanism during shared update processing (see section “Shared updating of files (SHARED-UPDATE)”) cause the entire block containing the current record to be locked. In such a case, therefore a large blocking factor would lead to a reduction in processing speed.

During compilation, the compiler calculates a value for the buffer size on the basis of the record and block length entries given in the compilation unit. The runtime system rounds up this value for DMS to the next multiple of a PAM block (2048 bytes). This default value can be modified during the file assignment by specifying the BUFFER-LENGTH operand in the ADD-FILE-LINK command. It must be noted, however, that the specified buffer needs to be at least as large as the longest data record (see section “Definition of file attributes”).

Except in the case of newly created files (OPEN OUTPUT), the block size entered in the catalog always takes priority over block size specifications in the program or ADD-FILE-LINK command.

The BLOCK CONTAINS clause is optional. If it is omitted, the compiler assumes the BLOCK CONTAINS 1 RECORDS, i.e. unblocked records.

`RECORD record-length-spec`

- specifies whether records of fixed or variable length are to be processed and
- defines, for variable-length records, a range of permissible values for the record length. If provided for in the format, a data item is additionally specified for the storage of current record length information.

The record-length-spec must conform to one of the three RECORD clause formats provided in COBOL2000. It must not be in conflict with the record lengths computed by the compiler from the specifications in the associated record description entry or entries.

The RECORD clause is optional. If it is not specified, records of variable length are assumed by the compiler.

```
01 data-record.
   nn item-1         type&length
   ...
   nn primary-key    type&length
   ...
   nn secondary-key  type&length
```

represents a record description entry for the associated file. It describes the logical format of data records.

At least one record description entry is required for each file. If more than one record description entry is specified for a file, the declared record format must satisfy the following rules:

- for fixed-length records, all record description entries must specify the same size;
- for variable-length records, the entries must not conflict with the record length specified in the RECORD clause, and even the record description entry with the shortest record length must still be capable of containing the entire record key.

At least one of the record description entries must explicitly declare the primary record key data item as a subordinate item of data-record. For type&length, the required length and format declarations (PICTURE and USAGE clauses etc.) must be entered (primary-key may have a maximum length of 255 bytes).

secondary-key is the data-name from the corresponding ALTERNATE RECORD KEY clause. Each secondary key item can be up to 127 bytes long. Overlaps with the primary key or other secondary keys are permissible provided two key items do not start at the same position. The COBOL2000 compiler also allows secondary keys defined as pure numeric (PIC 9) or alphabetic (PIC A) items.

The subdivision of data-record into subordinate data items (item-1, item-2, ...) is optional for all other record description entries.

`OPEN open-mode internal-file-name`

opens the file for processing in the specified open-mode.

The following phrases can be entered for open-mode:

INPUT	opens the file as an input file; it can only be read.
OUTPUT	opens the file as an output file; it can only be written.
EXTEND	opens the file as an output file; it can be extended.
I-O	opens the file as an I-O file; it can be read (one record at a time), updated and rewritten.

The open-mode entry determines with which I-O statements a file may be accessed (see section “Open modes and types of processing (indexed files)”).

```
START internal-file-name
READ internal-file-name
REWRITE data-record
WRITE data-record
DELETE internal-file-name
```

are I-O statements for the file that

- position to a record in the file
- read a record
- rewrite a record
- write a record, and
- delete a record.

The open mode declared in the OPEN statement determines which of these statements is admissible for the file. The relationship between access mode and open mode is described in section “Open modes and types of processing (indexed files)”.

`CLOSE internal file-name`

terminates processing of the file.

The WITH LOCK phrase can be additionally specified to prevent the file from being opened again in the same program run.

9.4.3 Permissible record formats and access modes

Record formats

Indexed files may contain records of fixed length (RECFORM=F) or variable length (RECFORM=V). In both of these formats, the records may be blocked or unblocked.In the COBOL compilation unit, the format of the records to be processed can be defined in the RECORD clause. The following table lists the phrases associated with each record format:

Record format	Phrase in the RECORD clause	
Fixed-length records	RECORD CONTAINS...CHARACTERS	(format 1)
Variable-length records	RECORD IS VARYING IN SIZE... or RECORD CONTAINS...TO...	(format 2) (format 3)

Table 30: Specification of record formats in the RECORD clause

Access modes

Access to records of an indexed file may be sequential, random, or dynamic.

In the COBOL compilation unit, the access mode is defined by means of the ACCESS MODE clause. The following table lists the possible phrases together with their effect on the access mode.

ACCESS MODE clause	Access mode
SEQUENTIAL	Sequential access: The records can be processed only in the order of their record keys. This means: • when reading records, the next or previous logical record is made available (for primary or secondary keys). • when writing records, the next logical record (with ascending primary key) is output to the file.
RANDOM	Random access: The records can be accessed in any order via their record keys. For this purpose, the (primary or secondary) key of the record to be processed must be supplied in the (ALTERNATE) RECORD KEY item prior to each I-O statement.
DYNAMIC	Dynamic access: The records can be accessed sequentially and/or randomly. The applicable access mode is selected via the format of the I-O statement in this case.

Table 31: Indexed files: ACCESS MODE clause and access mode

9.4.4 Open modes and types of processing (indexed files)

The language elements available for use in a COBOL program allow indexed files to be

- created,
- read,
- extended (by adding new records), and
- updated (by changing or deleting existing records).

The individual I-O statements that may be used in the program to process a given file are determined by its open mode, which is specified in the OPEN statement:

OPEN OUTPUT

Regardless of the phrase in the ACCESS MODE clause, WRITE is permitted as an I-O statement in the following format:

```
WRITE...[FROM...]
        [INVALID KEY...]
        [NOT INVALID KEY...]
        [END-WRITE]
```

In this mode, it is only possible to create (load) new indexed files.

- ACCESS MODE IS SEQUENTIAL

  enables the sequential creation of an indexed file. The records must be sorted in ascending order of their record keys before they are made available to the WRITE statement.
  Before each WRITE statement, the RECORD KEY data item must therefore be supplied with the record key of the record to be output. Each new key must be greater than the preceding one. If this is not the case, an INVALID KEY condition exists, and WRITE branches to the INVALID KEY statement or to the declared USE procedure without writing the record. It is thus not possible to overwrite records in this case.
- ACCESS MODE IS DYNAMIC or RANDOM

  enables the random creation of an indexed file. Note that file creation based on ascending record keys is performed more efficiently.

OPEN EXTEND

OPEN EXTEND enables an existing file to be extended. The ACCESS MODE clause is used in the same way as for OPEN OUTPUT.

OPEN INPUT

The phrase specified in the ACCESS MODE clause determines which I-O statements or statement formats are permitted. The following table lists the options available for OPEN INPUT:

Statement	**Entry in the ACCESS MODE clause**		
	SEQUENTIAL	RANDOM	DYNAMIC
START	START... [KEY IS...] [INVALID KEY...] [NOT INVALID KEY...] [END-START]	Statement not permitted	START... [KEY IS...] [INVALID KEY...] [NOT INVALID KEY] [END-START]
READ			For sequential access:

	READ...[NEXT | PREVIOUS] [INTO...] [AT END...] [NOT AT END...] [END-READ...]	READ... [INTO...] [KEY IS...] [INVALID KEY...] [NOT INVALID KEY...] [END-READ]	READ...{NEXT | PREVIOUS} [INTO...] [AT END...] [NOT AT END...] [END-READ]
			For random access READ... [INTO...] [KEY IS...] [INVALID KEY...] [NOT INVALID KEY...] [END-READ]

Table 32: Permitted I-O statements for OPEN INPUT

Indexed files can be read in this mode. Depending on the access mode specified, the READ statement has the following effect:

- ACCESS MODE IS SEQUENTIAL

  permits the file to be read sequentially only. READ retrieves records in ascending (NEXT) or descending (PREVIOUS) order based on the record keys.
  The (ALTERNATE) KEY item is not evaluated by READ. However, a START statement can be used to position to any record of the file before execution of a READ statement: START uses a relation condition to determine the record key of the first record to be read and thus establishes the starting point for subsequent sequential read operations (see also section “I-O status”). If the relation condition cannot be satisfied by any record key of the file, an INVALID KEY condition exists, and START makes a branch to the INVALID KEY statement or to the declared USE procedure.
- ACCESS MODE IS RANDOM

  enables records of the file to be randomly read. READ retrieves the records in a user-specified order; access to each record is effected via its record key.
  Accordingly, the record key of each record to be read must be supplied in the (ALTERNATE) KEY data item prior to each READ operation.
  If the record key of an unavailable record is specified, an INVALID KEY condition exists, and READ branches to the INVALID KEY statement or to the USE procedure declared for this condition.
- ACCESS MODE IS DYNAMIC

  permits random and/or sequential reading of the file. The desired access mode is selected via the format of the READ statement (see table 33).
  In this case, a START statement is meaningful for sequential reading only.

OPEN I-O

The phrase specified in the ACCESS MODE clause determines which I-O statements or statement formats are permitted.

Records in an indexed file opened in OPEN I-O mode can be

- read,
- added,
- updated by the program, and
- overwritten or
- deleted.

The following table lists the options available for OPEN I-O

Statement	Entry in the ACCESS MODE clause		
	SEQUENTIAL	RANDOM	DYNAMIC
START	START... [KEY IS...] [INVALID KEY...] [NOT INVALID KEY...] [END-START]	Statement not permitted	START... [KEY IS...] [INVALID KEY...] [NOT INVALID KEY] [END-START]
READ	READ...[NEXT | PREVIOUS] [INTO...] [AT END...] [NOT AT END...] [END-READ...]	READ... [INTO...] [KEY IS...] [INVALID KEY...] [NOT INVALID KEY...] [END-READ]	For sequential access: READ...{NEXT | PREVIOUS} [INTO...] [AT END...] [NOT AT END...] [END-READ] For random access: READ... [INTO...] [KEY IS...] [INVALID KEY...] [NOT INVALID KEY...] [END-READ]
REWRITE	REWRITE... [FROM...] [INVALID KEY...] [NOT INVALID KEY...] [END-REWRITE]	REWRITE... [FROM...] [INVALID KEY...] [NOT INVALID KEY...] [END-REWRITE]	REWRITE... [FROM...] [INVALID KEY...] [NOT INVALID KEY...] [END-REWRITE]
WRITE	Statement not permitted	WRITE... [FROM...] [INVALID KEY...] [NOT INVALID KEY...] [END-WRITE]	WRITE... [FROM...] [INVALID KEY...] [NOT INVALID KEY...] [END-WRITE]
DELETE	DELETE... [END-DELETE]	DELETE... [INVALID KEY...] [NOT INVALID KEY...] [END-DELETE]	DELETE... [INVALID KEY...] [NOT INVALID KEY...] [END-DELETE]

Table 33: Permitted I/O statements for OPEN I-O

OPEN I-O assumes that the file to be processed already exists. It is therefore not possible to create a new indexed file in this mode.

The specified access mode determines the type of processing that can be performed, as well as the effect of the individual I-O statements:

- ACCESS MODE IS SEQUENTIAL

  As in the case of OPEN INPUT, this access mode permits the sequential reading of a file with READ and the use of a preceding START to position to any record as the starting point.

  In addition, the record that is read by a successful READ operation can be

  - updated by the program and rewritten with REWRITE, or
  - logically deleted with DELETE.

  However, it must be noted that

  - no further I-O statement must be executed for this file, and
  - the RECORD KEY data item must not be changed

  between the READ statement and the REWRITE or DELETE statement.

- ACCESS MODE IS RANDOM

  enables records to be randomly retrieved with READ (as in OPEN INPUT). In addition, new records can be inserted into the file with WRITE, and existing records in the file can be rewritten or deleted with REWRITE or DELETE (regardless of whether they were read earlier).

  Prior to each WRITE, REWRITE, or DELETE statement, the RECORD KEY data item must be supplied with the key of the record that is to be added, rewritten, or deleted. If the following conditions apply the number of an already existing record is specified for WRITE, or the number of an unavailable record is specified for REWRITE or DELETE, an INVALID KEY condition exists, and WRITE, REWRITE, or DELETE branches to the INVALID KEY statement or to the USE procedure declared for this event.

- ACCESS MODE IS DYNAMIC

  allows a file to be processed sequentially or randomly. Here, the desired access mode is selected via the format of the READ statement.

9.4.5 Positioning with START

Any record in an indexed (or relative) file can be selected as the starting point for subsequent sequential read operations by means of START. START sets up a comparison via a relation condition in order to establish the (primary or secondary) key of the first record to be read.

The following example illustrates how the language extension (to ANS85) START...KEY LESS... and READ... PREVIOUS can be used to sequentially process an indexed file in reverse order, i.e. in the order of descending record keys, beginning with the highest key in the file. The precise location within a file is identified by means of a conceptual entity called the file position indicator.

Example 9-11

Processing an indexed file in reverse order

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. INDREV.
*    INDREV PROCESSES THE RECORDSOF AN INDIVIDUAL FILE
*    IN DESCENDING RECORD KEY ORDER.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SPECIAL-NAMES.
    TERMINAL IS T.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT IND-FILE
     ASSIGN TO "INDFILE"
     ORGANIZATION IS INDEXED
     ACCESS IS DYNAMIC
     RECORD KEY IS REC-KEY.
 DATA DIVISION.
 FILE SECTION.
 FD  IND-FILE.
 01  IND-REC.
     05  REC-KEY                   PIC X(8).
     05  REC-TEXT                  PIC X(72).
 WORKING-STORAGE SECTION.
 01  PROCESSING-SWITCH             PIC X.
     88  END-OF-PROCESSING                   VALUE "1".
 PROCEDURE DIVISION.
 INITIALIZTION.
    OPEN I-O IND-FILE ———————————————————————————————————(1)
    MOVE HIGH-VALUE TO REC-KEY ——————————————————————————(2)
    MOVE "0" TO PROCESSING SWITCH.
 PROCESS FILE.
     START IND-FILE KEY LESS OR EQUAL REC-KEY
        INVALID KEY
           DISPLAY "FILE IS EMPTY" UPON T
           SET END-OF-PROCESSING TO TRUE
        NOT INVALID KEY
          READ IND-FILE PREVIOUS ———————————————————————————(3)
               AT END
                   SET END-OF-PROCESSING TO TRUE
               NOT AT END
                  DISPLAY "HIGHEST RECORD NUMBER: " REC-KEY
                     UPON T
                  PERFORM PROCESS-RECORD
            END-READ
```

```
      END-START

      PERFORM WITH TEST BEFORE UNTIL END-OF-PROCESSING
       READ IND-FILE PREVIOUS ──────────────────────────────────────(4)
           AT END
              SET END-OF-PROCESSING TO TRUE
           NOT AT END
              DISPLAY "NEXT RECORD KEY: " REC-KEY
                 UPON T
              PERFORM PROCESS-RECORD
         END-READ
      END-PERFORM.
  TERMINATION.
      CLOSE IND-FILE
      STOP RUN.
  PROCESS-RECORD.
*
*    PROCESSING OF CURRENT RECORD ──────────────────────────────────(5)
*
```

(1) The file IND-FILE is opened for processing with OPEN I-O.

(2) To obtain the record with the highest key in the file,

- the RECORD KEY is preset to the highest possible value (HIGH-VALUE in the NATIVE alphabet), and
- START...KEY LESS OR EQUAL sets the file position indicator to it.

(3) READ...PREVIOUS reads the record to which the file position indicator was previously set by START.

(4) READ...PREVIOUS reads the record preceding the last record to be read.

(5) The read record is processed. If its RECORD KEY is changed during processing, the original value must be restored before the next START statement.

9.4.6 I-O status

The status of each access operation performed on a file is stored by the runtime system in specific data items, which can be assigned to every file in the program. These items, which are specified by using the FILE STATUS clause, provide information on

- whether the I-O operation was successful, and
- the type of errors that may have occurred.

This data can be evaluated (by USE procedures in the DECLARATIVES, for example) and used by the program to analyze I-O errors. As an extension to Standard COBOL, COBOL2000 provides the option of using DMS codes to include error messages in this analysis, thus allowing a finer differentiation between different causes of errors.

The FILE STATUS clause is specified in the FILE-CONTROL paragraph of the Environment Division. Its format is (see “COBOL2000 Reference Manual” [1]):

```
FILE STATUS IS data-name-1 [data-name-2]
```

where data-name-1 and data-name-2 (if specified) must be defined in the Working-Storage Section or the Linkage Section. The following rules apply with regard to the format and possible values for these two items:

data-name-1

- must be declared as a two-byte alphanumeric data item, e.g.

```
01 data-name-1    PIC X(2).
```

- contains a two-character numeric status code following each access operation on the associated file. The table provided at the end of this section lists all such codes together with their meanings.

data-name-2

- must be declared as a 6-byte group item with the following format:

```
01 data-name-2.
   02 data-name-2-1      PIC 9(2) COMP.
   02 data-name-2-2      PIC X(4).
```

- is used for storing the DMS error code for the I-O status. Following each access operation on the associated file, data-name-2 contains a value that directly depends on the content of data-name-1. The relationship between the values is shown in the table below:

Contents of data-name-1 not equal 0?	DMS code not equal 0?	Value of data-name-2-1	Value of data-name-2-2
no	no	undefined	undefined
yes	no	0	undefined
yes	yes	64	DMS code of the associated error message

The DMS codes and the associated error messages are given in “Introductory Guide to DMS” [4].

I-O status	Meaning

	Execution successful
00	The I-O statement terminated normally. No further information regarding the I-O operation is available.
02	A record was read with ALTERNATE KEY and subsequent sequential reading with the same key has found at least one record with an identical key. A record was written with ALTERNATE KEY WITH DUPLICATES and there is already a record with an identical key value for at least one alternate key.
04	Record length conflict: A READ statement terminated normally. However, the length of the record read lies outside the limits defined in the record description entry for the given file.
05	An OPEN statement was executed for an OPTIONAL file which does not exist.
	Execution unsuccessful: AT END condition
10	An attempt was made to execute a sequential READ operation. However, no next logical record was available, as the end-of-file was encountered.
	Execution unsuccessful: invalid key condition
21	File sequence error in conjunction with ACCESS MODE IS SEQUENTIAL: 1. The record key value was changed between the successful execution of a READ statement and the execution of the next REWRITE statement for a file, or 2. the ascending sequence of record keys was violated in successive WRITE statements.
22	Duplicate key An attempt was made to execute a WRITE statement with a primary key for which there is already a record in the indexed file. An attempt was made to create a record with ALTERNATE KEY, but without WITH DUPLICATES, and there is already an alternate key with the same value in the file.
23	Record not located An attempt was made (using a READ, START, DELETE or REWRITE statement with key) to access a record not contained in the file.
24	Boundary values exceeded An attempt was made to execute a WRITE statement beyond the system-defined boundaries of an indexed file.
	Execution unsuccessful: unrecoverable error
30	No further information regarding the I-O operation is available (the DMS code provides further information).
35	An OPEN statement with the INPUT, I-O or EXTEND phrase was issued for a non-optional file which does not exist.
37	OPEN statement on a file that cannot be opened due to the following violations: 1. OPEN OUTPUT/I-O/EXTEND on a write-protected file (password, RETENTION-PERIOD, ACCESS=READ in catalog) 2. OPEN INPUT on a read-protected file (password)

38	An attempt was made to execute an OPEN statement for a file previously closed with the LOCK phrase.
39	The OPEN statement was unsuccessful as a result of one of the following conditions: 1. One or more of the operands ACCESS-METHOD, RECORD-FORMAT, RECORD-SIZE or KEY-LENGTH were specified in the ADD-FILE-LINK command with values that conflict with the corresponding explicit or implicit program specifications. 2. Record length error occurred for an input file (catalog check, if RECFORM=F). 3. The record size is greater than the BLKSIZE entry in the catalog of an input file. 4. The catalog entry of one of the FCBTYPE, RECFORM, RECSIZE (if RECFORM=F), KEYPOS, or KEYLEN operands for an input file is in conflict with the corresponding explicit or implicit program specifications or with the corresponding specifications in the FILE command. 5. An attempt was made to open a file whose alternate key does not match the key values specified in the ALTERNATE RECORD KEY clause in the program.
	Execution unsuccessful: logical error
41	An attempt was made to execute an OPEN statement for a file which was already open.
42	An attempt was made to execute a CLOSE statement for a file which was not open.
43	For ACCESS MODE IS SEQUENTIAL: The most recent I-O statement executed prior to a DELETE or REWRITE statement was not a successfully executed READ statement.
44	Record length limits exceeded: An attempt was made to execute a WRITE or REWRITE statement. However, the length of the record is outside the range allowed for this file.
46	An attempt was made to execute a sequential READ statement for a file in INPUT or I-O mode. However, no valid next record is available since: 1. the preceding START statement was unsuccessful, or 2. the preceding READ statement was unsuccessful without leading to an AT END condition, or 3. an attempt was made to execute a READ statement after the AT END condition was encountered.
47	An attempt was made to execute a READ or START statement for a file that is not open in INPUT or I-O mode.
46	An attempt was made to execute a WRITE statement for a file that • on sequential access is not in OUTPUT or EXTEND mode • on random or dynamic access is not in OUTPUT or I-O mode.
49	An attempt was made to execute a DELETE or REWRITE statement for a file that is not in I-O mode.
	Other conditions with unsuccessful execution
90	System error; no further information regarding the cause is available.

91	OPEN error: the actual cause is evident from the DMS code (see “FILE STATUS clause” specifying data-name-2).
93	For shared update processing only (see section "Shared updating of files (SHARED-UPDATE)"): The I-O statement could not terminate normally because a different task is accessing the same file, and the access operations are incompatible.
94	1. For shared update processing only (see section “Shared updating of files(SHARED-UPDATE)”): deviation from call sequence READ - REWRITE/DELETE. 2. The record size is greater than the block size.
95	Incompatibility between values specified in the BLOCK-CONTROL-INFO or BUFFER-LENGTH operand of the ADD-FILE-LINK command and the file format, block size, or the format of the used volume.
96	READ PREVIOUS is not supported for modules which were compiled with COBRUN ENABLE-UFS-ACCESS=YES.

Table 34: I-O status values for indexed files

9.5 Shared updating of files (SHARED-UPDATE)

- ISAM files
- PAM files

9.5.1 ISAM files

ISAM files with indexed or relative file organization can be shared by several users simultaneously by means of the SHARED-UPDATE operand in the SUPPORT parameter of the ADD-FILE-LINK command:

```
/ADD-FILE-LINK linkname,filename,SUPPORT=DISK(SHARED-UPDATE=YES)
```

The following table shows which OPEN statements are available to user B after the file has already been opened by user A.

		Options permitted for user B					
		SHARED-UPDATE =YES			SHARED-UPDATE=NO		
Options selected by user A	**OPEN statement**	INPUT	I-O	OUTPUT/ EXTEND	INPUT	I-O	OUTPUT/ EXTEND
SHARED-UPDATE=YES	INPUT	X	X		X		
	I-O	X	X				
	OUTPUT/EXTEND						
SHARED-UPDATE=NO	INPUT	X			X		
	I-O						
	OUTPUT/EXTEND						

Table 35: OPEN statements permitted for shared updating

X = permitted combinations of OPEN statement and SHARED-UPDATE value

It is clear from the table that the SHARED-UPDATE=YES option is superfluous for INPUT files if all users use OPEN INPUT. If SHARED-UPDATE=YES must nevertheless be specified for input files because at least one user uses OPEN I-O, the locks described below will not be set or released.

The SHARED-UPDATE=YES option makes sense, and is also necessary, only for the simultaneous updating of one or more ISAM files (OPEN I-O) by two or more interactive users.

Updates in batch processing mode must be executed successively in order to avoid both logic errors and excessive runtimes (unnecessary specification of SHARED-UPDATE=YES increases both runtime and CPU time).

If SHARED-UPDATE=YES is specified, WRITE-CHECK=YES will automatically be set also, i.e. the ISAM buffers will be rewritten immediately after each change. This is necessary for reasons of data security and consistency, but causes a considerable increase in the number of I-O operations.

In order to ensure data consistency during simultaneous updating of an ISAM file by several users, the COBOL2000 runtime system utilizes the locking and unlocking mechanism of the DMS access method ISAM. This mechanism locks or unlocks the data blocks that contain the data records referenced by the COBOL statements READ, WRITE, REWRITE or DELETE.

A data block is the multiple of a PAM page (2048 bytes) defined implicitly or explicitly by the BUFFER-LENGTH parameter in the ADD-FILE-LINK command when the file was created (see section “Basic concepts relating to the structure of files”).

In the following, a record lock is to be understood as the lockout of the entire block that contains the record.

To permit shared updating of ISAM files, a format extension to the READ or START statement is provided. However, this extension is effective only if SHARED-UPDATE=YES is specified in the ADD-FILE-LINK command and the file was opened with OPEN I-O.

Format extension (for all READ/START formats):

```
READ  file-name [WITH NO LOCK]...
START file-name [WITH NO LOCK]...
```

Rules for shared updating

1. READ or START statement with the WITH NO LOCK phrase:

   If user A specifies the WITH NO LOCK phrase and the appropriate record is present, this record is read or selected irrespective of any lock set by any other user. The record is not locked. A REWRITE or DELETE statement cannot be performed on a record thus read.

   Simultaneously, a user B may read or update the same record.

2. READ or START statement without the WITH NO LOCK phrase:

   If user A does not specify the WITH NO LOCK phrase and the appropriate record is present, a READ or START statement can only be executed successfully if that record is not already locked by user B. If execution of the statement is successful, the record will be locked. Before the lock is released, user B can only read or select the same or another record in the same block by specifying WITH NO LOCK, but will not be able to update any record in this block. (If user B opened the file using OPEN INPUT, he will still be able to read records in the locked block.)

3. Updating of records:

   If a record is to be updated using a REWRITE or a DELETE statement, the appropriate record must be read immediately before this by a READ statement (without the WITH NO LOCK phrase). Between this READ statement and the REWRITE or DELETE statement, no further I-O statement may be executed for the same ISAM file. Between these two statements, only READ or START statements with the WITH NO LOCK phrase can be executed for other ISAM files whose ADD-FILE-LINK command contains SHARED-UPDATE=YES and which are open at the same time in the OPEN I-O mode. Statements for other ISAM files (without SHARED-UPDATE=YES and OPEN I-O) can be executed.
   Any violation of these rules will lead to an unsuccessful REWRITE or
   DELETE statement with FILE STATUS 94.

4. Waiting times in the event of a lock:

   If user A has locked a data record following a successful READ or START statement, and user B attempts to perform a READ or START statement without the WITH NO LOCK phrase on the same data record or any other in the same data block, user B will not immediately be unsuccessful. User B must wait in a queue until the lock is released by user A. Only if the maximum waiting time elapses without the lock being released will the statement be considered unsuccessful and FILE STATUS 93 be set. If the lock is released before the waiting time has elapsed, user B can continue with the successful call.

5. Releasing a locked record:

   A user maintains a record lock until he executes one of the following statements:

   - Successful REWRITE or DELETE statement for the locked data record
   - WRITE statement for an ISAM file whose ADD-FILE-LINK command contains SHARED-UPDATE=YES and which is open as OPEN I-O (i.e., for the same file that contains the locked record, or for another ISAM file; unlocking is also effected if the INVALID KEY condition occurs)

- READ or START statement with the WITH NO LOCK phrase for the same file (unlocking is also effected if the AT END or INVALID KEY condition occurs)
- READ or START statement without the WITH NO LOCK phrase for a record in another data block of the same file (unlocking is also effected if the AT END or INVALID KEY condition occurs)
- READ or START statement without the WITH NO LOCK phrase for another ISAM file whose ADD-FILE-LINK command contains SHARED-UPDATE=YES and which is open in OPEN I-O mode (unlocking is also effected if the AT END or INVALID KEY condition occurs)
- CLOSE statement for the same file

Thus, a statement for an ISAM file can release the record lock on another ISAM file.

Notes

1. If an ISAM file (with SHARED-UPDATE=YES and OPEN I-O) is to be processed in a program, a USE AFTER STANDARD ERROR procedure should be provided for this file. In this procedure, the file status values relating to shared update processing, i.e. 93 (record has been locked by a simultaneous user) and 94 (REWRITE or DELETE statement without preceding READ statement), can be interrogated and then processed accordingly.
2. It should be noted that when a record is locked using the ISAM block lockout mechanism, all records in the same block are locked to all concurrent users.
3. A user cannot lock more than one block at a time, i.e. can protect only one block against updating by other users. This is also true if the user has opened several ISAM files (all SHARED-UPDATE=YES) in OPEN I-O mode.
4. A deadlock (mutual locking of data blocks by different users) is excluded, because only one block of all ISAM files (all SHARED-UPDATE=YES) can be locked for each user.This does not, however, apply if a PAM file is accessed simultaneously with SHARED-UPDATE=YES in I-O mode.
5. If a READ statement for a record is not immediately followed by a REWRITE or DELETE statement but by an attempt to access some other data block (of the same or any other ISAM file) instead, the record will need to be read again before the REWRITE or DELETE statement is executed. Since the affected data block was unlocked to other users in the period between the READ and REWRITE or DELETE statements, the contents of the original record may have been changed (see (9-12 a)).

   If an access attempt is made to another block or another file without the locking mechanism, the data made available might meanwhile have been changed by concurrent users before the REWRITE or DELETE statement is executed (see (9-12 b)).
6. In order to prevent a user from working with data that is no longer up-to-date, the WITH NO LOCK phrase should only be used when absolutely necessary.
7. A locked record (block) will give rise to waiting times when concurrent users try to access the same record or another in the same block. To keep waiting times as short as possible, the lock should be released as soon as possible. If a lock is not released in due time, the waiting time will have elapsed, and the program branches to the appropriate USE procedure, if available (see Example 9-13), or is aborted (message COB9151, FILE STATUS 93 and DMS error code DAAA).

Example 9-12

Reading and rewriting in file ISAM1 when data from file ISAM2 is required before rewriting:

(a) Without the WITH NO LOCK phrase: Two READ statements for the same file are necessary, but the locking times are shorter:

```
...
READ ISAM1 INTO WORK1 ————————————————————————————————————(1)
  INVALID KEY...
...
READ ISAM2 ————————————————————————————————————————————————(2)
  INVALID KEY...
...
```

Processing of WORK1 taking into account ISAM2REC:

```
...
READ ISAM1 ————————————————————————————————————————————————(3)
  INVALID KEY...
```

Check for changes to ISAM1REC in meantime; repeat processing if necessary:

```
...
REWRITE ISAM1SATZ FROM WORK1———————————————————————————————(4)
  INVALID KEY...
```

(1) Reads a record from ISAM1 and buffers it in WORK1; relevant block in ISAM1 locked

(2) Reads a record from ISAM2, releases lock in ISAM1, locks relevant block in ISAM2

(3) Rereads record from ISAM1, releases lock in ISAM2, locks relevant block in ISAM1

(4) Rewrites record into ISAM1, releases lock in ISAM1

(b) With the WITH NO LOCK phrase: Only one READ statement is required for this file, but the locking times are consequently longer:

```
...
READ ISAM1 ————————————————————————————————————————————————(1)
  INVALID KEY...
...
READ ISAM2 WITH NO LOCK ———————————————————————————————————(2)
  INVALID KEY...
...
```

Processing of ISAM1REC taking into account ISAMREC:

```
...
REWRITE ISAM1SATZ FROM WORK1 ——————————————————————————————(3)
  INVALID KEY...
```

(1) Reads a record from ISAM1; relevant block locked

(2) Reads a record from ISAM2; relevant block not locked

(3) Rewrites record into ISAM1; lock is released.

Example 9-13

Branch to USE AFTER STANDARD ERROR procedure

```
   ...
FILE-CONTROL.
    SELECT ISAM1
     ...
    FILE STATUS IS FILESTAT1.
WORKING-STORAGE SECTION.
77  FILESTAT1 PIC 99.
    ...
PROCEDURE DIVISION.
DECLARATIVES.
ISAM1ERR SECTION.
    USE AFTER STANDARD ERROR PROCEDURE ON ISAM1.
LOCK-PAR.
    IF FILESTAT1 = 93
       THEN DISPLAY "RECORD CURRENTLY LOCKED" UPON T
       ELSE DISPLAY "DMS-ERROR ISAM1, FILE-STATUS="
                    FILESTAT1 UPON T.
ISAM1ERR-EX.
    EXIT.  ————————————————————————————————————————————(1)
END DECLARATIVES.
CNTRL SECTION.
    ...
```

(1) Control is transferred to the statement following the statement that caused the error. Suitable error recovery measures depend on the application involved.

9.5.2 PAM files

Like ISAM files, files with relative organization and FCBTYPE=PAM can also be updated simultaneously by several users if the ADD-FILE-LINK command contains SHARED-UPDATE=YES and the file was opened with OPEN I-O.

To ensure data consistency during shared updating, the COBOL2000 runtime system uses the locking and unlocking mechanism of the DMS access method UPAM. Unlike ISAM, access coordination in this case is file-specific. Accordingly, statements for a specific file have no impact on any other file.

As with ISAM, the lock affects not just one specific record in a block, but the entire block in which that record is contained (see section “Indexed file organization”).

For PAM files, as with ISAM files, the format extension WITHNO LOCK may also be used in all formats of the READ or START statement provided SHARED-UPDATE=YES and OPEN I-O were specified for these files.

Rules for shared updating

1. Reading and positioning with or without the WITH NO LOCK phrase is effected in the same manner as for ISAM files.
2. Updating of records

   If a record is to be updated by a REWRITE or DELETE statement, the record must be read (as with ISAM files) immediately before this statement by means of a READ statement (without the WITH NO LOCK phrase). No other statement for this file must be executed between these statements. Unlike the conventions for ISAM files, statements for other PAM files are allowed (on account of the file-oriented access coordination).
3. Waiting times in the event of a lock

   The maximum waiting time for the release of a locked block is 999 seconds. After this time has elapsed, control is transferred to the USE AFTER STANDARD ERROR procedure, if present, or the program is terminated with error message COB9151 (FILE STATUS 93 and DMS error code D9B0 or D9B1).
4. Releasing a locked record

   The release of a locked block can be accomplished with the same statements as for ISAM files, but all statements must refer to the same file.

   In contrast to ISAM files, a statement for a PAM file does not release blocks of another PAM file.

Notes

1. If a PAM file (with SHARED-UPDATE=YES, OPEN I-O) is to be processed in a program, a USE AFTER STANDARD ERROR procedure should be defined for this file (see “Indexed files”).
2. Unlike ISAM files (with SHARED-UPDATE=YES, OPEN I-O), the simultaneous processing of two or more files (all with SHARED-UPDATE=YES, OPEN I-O), of which at least one is a PAM file, allows one record per user to be locked simultaneously in any number of files (but only one record in one file). This may result in a deadlock situation.
3. As with ISAM files, any locks on records (blocks!) in PAM files should be released as soon as possible in order to minimize waiting times for other users.

Example 9-14

Deadlock

User A:	User B:

`READ file1 (record n)`	`READ file2 (record m)`
`.`	`.`
`.`	`.`
`READ file2 (record m)`	`READ file1 (record n)`
(Block in file1 not unlocked)	(Block in file2 not unlocked)

Both users are waiting for the particular block to be released (deadlock).

The maximum waiting time for the release of a locked block is 999 seconds. After this time has elapsed, the USE AFTER STANDARD ERROR procedure, if present, is activated or the program is terminated with error message COB9151 (FILE STATUS 93 and DMS error code D9B0 or D9B1).

10 Processing XML documents

- Making XML documents available
- Using XML language elements in programs
- Linking, loading, starting programs with XML language elements
- Encoding identification
- Obtaining the parser
- Extended I-O status for XML statements (CBX code)

10.1 Making XML documents available

Depending on the type of processing, a COBOL program can process an XML document which is made available in working memory or in a file:

Processing type	XML document	
	in memory	in file
Structure-oriented	X	X
Event-oriented	X	–

If the XML document is made available in a file, the file's BS2000/OSD access method is irrelevant. The breakdown of the XML document into records is visible to the processing program: each record change in he XML document is replaced by an end-of-line character, i.e. the equivalent of the ASCII character X'0A' in the encoding in which the program received data from the XML document. However, the insertion of additional end-of-line characters can also be suppressed if the relevant control mechanism XML-LINE-FEED=IGNORED is set during compilation, see section “RUNTIME-OPTIONS option”.

10.2 Using XML language elements in programs

A prerequisite for compiling programs which use language elements to process XML documents is that the new keywords connected to the language elements are recognized. For this purpose the relevant control mechanism XML-SUPPORT=YES must be set when compilation takes place, see section “SOURCE-PROPERTIES option”.

When such programs run, a separate open source program package - the **parser** - analyzes and parses the XML documents. This program package is not a component part of the COBOL compiler or CRTE, but can be downloaded from the Internet.

> ¡ As it is open source software, the use of this program package is subject to its own license conditions and regulations which you must **accept** when you download it from the Internet.

At execution time this program package must be available as a module library in BS2000/OSD. How you obtain this module library if it is not already provided in BS2000/OSD is described in detail in section “Obtaining the parser”.

10.3 Linking, loading, starting programs with XML language elements

Programs which use language elements to process XML documents are in principle linked, loaded and started as described in chapter “Linking, loading, starting”.

Processing XML documents also always requires encodings to be converted. Connection module GNLAPDT must consequently be linked for corresponding XHCS functions:

When TSOSLNK is used for linking, see section “Static linkage using TSOSLNK”, the following additional statement is required in the linkage run:

```
*RESOLVE ,$.SYSOML.XHCS-SYS.020    ————————————————————————— (1)
```

(1) XHCS connection module: it is implied that the library which contains the module is available in the system with the name SYSOML.XHCS-SYS.020.

When BINDER is used for linking, see section “Linking using BINDER”, the following additional statement is required in the linkage run:

```
//RESOLVE-BY-AUTOLINK LIB=$.SYSOML.XHCS-SYS.020    ———————————— (1)
```

(1) XHCS connection module: it is implied that the library which contains the module is available in the system with the name SYSOML.XHCS-SYS.020.

When dynamic linking and loading takes place with DBL, see section “Dynamic linking and loading using DBL”, the following additional command is required before the START PROGRAM or LOAD-PROGRAM command:

```
/ADD-FILE-LINK BLSLIB02,$.SYSOML.XHCS-SYS.020    ————————————— (1)
```

(1) XHCS connection module: it is implied that the library which contains the module is available in the system with the name SYSOML.XHCS-SYS.020

When a COBOL Program with XML statement **executes**, the executing program must be able to access the module library which contains the parser program package so that it can load modules from it dynamically. To permit this, use the link name COBPRSXM to assign the library with the parser modules before the COBOL program executes. In the case of dynamic linking and loading or when loading linked programs, the following additional commands are required before the START PROGRAM or LOAD-PROGRAM command:

```
/ADD-FILE-LINK COBPRSXM,$XYZ.SYSLIB.UTM-XML.030.RT     ——————————————— (1)
/ADD-FILE-LINK BLSLIB01,$.SYSLNK.CRTE          ———————————————————————— (2)
```

(1) XML parser: it is implied that the library which contains the relevant modules is available on the XYZ ID with the name SYSLIB.UTM-XML.030.RT.

(2) The parser’s modules require the C runtime system. You must therefore also assign CRTE.

10.4 Encoding identification

To ensure an XML document is processed correctly, it is essential that the encoding which is used to present the document is identified. XML permits this encoding to be specified in an encoding declaration within the document. When data is transferred between different data processing systems, the encodings used are generally also converted, but no changes are made to the contents. This can result in the specification of the encoding in the XML document no longer matching the encoding which is actually used for the presentation.

In order to identify the encoding declaration in the XML document, an assumption must have been made beforehand regarding the encoding used to permit the document to be read. This is roughly possible because a well-formed XML document must always begin with the string <?xml. The encoding currently used for the XML document can be derived by comparing the start of the document with the presentation of this characteristic string in the various encodings supported by the parser.

Furthermore, BS2000/OSD enables a file attribute to be assigned for files; this names an encoding (CODED-CHARACTER-SET), but does not force the file content to be presented in this encoding. When XML documents are made available in working memory (which is also possible in COBOL), the specifications in the program also allow an encoding to be derived which is used to present the document, see the “COBOL 2000 Compiler” manual [1], “ASSIGN clause” section.

Three sources consequently exist from which the same encoding which is used to present the document can be derived:

Z1 from examining the start of the document; closed, assumed encoding

Z2 external specification of the encoding as a file attribute or specifications in the program

Z3 encoding declaration in the XML document

In order to as far as possible prevent manual intervention from being required before an XML document is processed, the COBOL system also to some extent accepts missing or contradictory specifications regarding encodings from these three sources.

The decision on the encoding which is ultimately assumed for processing purposes or the decision on the I-O status in the case of contradictions which cannot be resolved is taken in accordance with the table below. A dash (–) means that the existence or compatibility plays no part in this decision.

Existing situation						Decision taken	
Z1 identified *	**Z2 exists ****	**Z3 exists ****	**Z2 compatible with Z1 *****	**Z3 compatible with Z1 *****	**Z3 compatible with Z2 *****	**Encoding used**	**I-O status**
yes	yes	yes	yes	–	yes	Z3	
yes	yes	yes	yes	–	no	Z2	
yes	yes	yes	no	yes	–		3D
yes	yes	yes	no	no	–		3D
yes	yes	no	yes	–	–	Z2	
yes	yes	no	no	–	–		3D
yes	no	yes	–	yes	–	Z3	
yes	no	yes	–	no	–	Z1	

yes	no	no	–	–	–	Z1	
no	yes	–	–	–	–	Document in file: Z2	Document in memory: 3D
no	no	–	–	–	–		3D
–	Unknown encoding	–	–	–	–		3D
–	–	Unknown encoding	–	–	–		3D

* Only UTF-16, EBCDIC or UTF can be identified as encoding Z1. Here EBCDIC stands for a(n) (imprecise) superset for all special variants (such as EDF03IRV, EDF041, etc.) and UTF as a(n) (imprecise) superset for UTF-8 and all ISO variants supported by XHCS.

** Only UTF-8, UTF-16, EBCDIC, ISO646 and the special EBCSIC variants and ISO variants under the term 'exists', i.e. all those which also know XHCS, are understood as Z2 for documents in files and as Z3. All other encodings as regarded as 'unknown'. Only EBCDIC (for alphanumeric data items) and UTF-16 (for national data items) are possible as Z2 for documents in memory.

*** 'Encoding Zx compatible with encoding Zy' means that Zx and Zy designate the same encoding, or that Zx is a more precisely named encoding from the (imprecise) superset Zy.

If the encoding ultimately selected only designates the imprecise superset EBCDIC, the special variant available at the time the program is compiled is used.

If the encoding ultimately selected only designates the imprecise superset UTF, UTF-8 is used.

This encoding identification takes place in every OPEN DOCUMENT statement (without an AT phrase), and during an XML PARSE statement both for the primary XML document and for the external entities or DTDs in this document which are addressed.

10.5 Obtaining the parser

Proceed as follows to download the XML parser from the Internet and to make it available as a module library (see also the “XML for openUTM” manual [27]).

The software required is provided on the Internet under 'openUTM'. However, it does **not** require that COBOL programs which you want to use must run under openUTM.

1. Go to *http://ts.fujitsu.com/openUTM* on the Internet.
2. Follow the instructions for downloading, accept the license conditions if required and download the latest version of the BS2000 library to your PC.
3. Unpack the SYSLIB.UTM-XML.nnn.RT member (nnn: version identifier, at least 030) from the archive and transfer it to the BS2000/OSD system. There are two ways of doing this:
   a. Unpack the member from the ftp subdirectory and transfer it in binary format using ftp.
   b. Unpack the member from the openft subdirectory and transfer it using openFT (binary file type, transparent transfer mode).

A PLAM library is now available in BS2000/OSD: this is the module library which is required for executing the COBOL programs.

10.6 Extended I-O status for XML statements (CBX code)

The simple I-O statuses for XML files are described in the “COBOL2000 Reference Manual” [1].

The errors which are indicated by italics in the table below occur only when DTDs are parsed.

I-O status	Meaning
0003	Same name used more than once for attributes
0004	Less than sign ('<') in the attribute value
0005	Opening and closing tags do not match
0010	Double hyphen in comment
0011	Processing instruction not terminated
0012	Name of the processing instruction begins with the reserved string 'xml' (not case-sensitive)
0013	Invalid hexadecimal character specification in numeric character entity
0014	Invalid decimal character specification in numeric character entity
0016	Numeric character entity is not a valid UTF-8 character
0017	Invalid character in entity reference name
0102	Empty document
0110	Value of an attribute not terminated correctly
0119	CDATA section not terminated correctly
0127	Equals sign ('=') missing in attribute
0128	Value for attribute missing
0138	Target name for processing instruction invalid
0139	Invalid character in processing instruction
0142	Version identifier missing in the XML declaration
0148	Equals sign ('=') missing after keyword 'encoding'
0149	Encoding name missing
0150	Encoding name not terminated correctly
0151	Invalid character after the encoding declaration
0155	Value for standalone declaration neither 'yes' nor 'no'
0160	The end of the document is followed by something illegal
0315	'UTF-16LE' encoding is not supported

0317	Encoding cannot be identified
0320	EBCDIC character in a national data item
0321	ASCII/UTF-8 character in a national data item
1001	Parameter entity reference at the end of the document
1002	Parameter entity reference in the prologue
1003	Parameter entity reference in the epilogue
1004	*Parameter entity reference in the markup declaration in internal DTD*
1005	Entity not declared
1006	Reference to unparsed entity
1007	Reference to external entity in attribute value
1008	Invalid reference to a parameter entity
1009	Expected string begins with quotes
1010	Name space declaration false
1012	*Value in an entity declaration is incorrect*
1013	*Expected literal not found*
1014	Blank missing
1015	Expected name missing
1016	Expected greater than sign ('>') missing
1017	Expected equals sign ('=') missing
1018	Entity is not balanced (Something was begun which has not been terminated or something was terminated which had not been begun.)
1019	*Illegal character ('&' or '%') in the value of the entity declaration*
1020	*Parameter entity reference in value or entity*
1021	*Invalid URI in value or entity*
1022	*URI begins with hash character ('#')*
1023	Encoding declaration missing in XML declaration of an external entity
1024	External DTD required although excluded by standalone declaration
1025	External entity could not be loaded
1098	Contradiction between external encoding identification and encoding declaration
1099	Contradiction between internal encoding and encoding declaration
2001	String not terminated correctly

2003	*Literal not terminated correctly*
2004	Comment not terminated correctly
2005	*Name missing in notation declaration*
2006	*Notation declaration not terminated correctly*
2007	*Error in attribute list*
2008	*Attribute list not terminated correctly*
2009	*Error in element declaration for mixed content*
2010	*Element declaration for mixed content not terminated correctly*
2011	*Error in element declaration*
2012	*Element declaration not terminated correctly*
2013	*XML declaration incorrect or missing*
2014	*XML declaration not terminated correctly*
2015	*Error in conditional section*
2016	*Conditional section not terminated correctly*
2017	*Invalid content in external DTD*
2018	*Document type definition not terminated correctly*
2019	String ’]]>’ not permitted in a value for end CDATA section
2020	*Separator missing*
2021	*nmtoken missing in attribute list*
2022	*String '#PCDATA' missing in element declaration for mixed content*
2023	*URI missing*
2024	*Public identifier missing*
2026	*Conditional section incorrect*
2027	*Value missing in entity declaration*
2028	There is something behind a balanced entity
2029	Nesting depth of entity references is greater than 40
2030	*Keyword 'INCLUDE' or 'IGNORE' missing in a conditional section*
2031	Invalid character in content
2096	UTF-16 character in alphanumeric data item
2097	ASCII/UTF-8 character in alphanumeric data item

2098	Contradiction between external and internal encoding identification
2099	Encoding from encoding declaration is not supported
2994	ILCS error
2995	Unexpected EOF
2996	Error while reading
2997	Preview buffer too small
2998	Memory error
2999	Internal error / System error
3000	Unused parser codes - please notify system administrator

Table 36: Extended I-O statuses for XML statements

11 Sorting and merging

- COBOL language elements for sorting and merging files
- Files for the sort program
- Checkpointing and restart for sort programs
- Sorting tables
- Sorting with extended character sets

11.1 COBOL language elements for sorting and merging files

COBOL2000 supports sorting and merging with the following language elements (see “COBOL2000 Reference Manual” [1]):

- Specification of the literal “SORTWK” in the ASSIGN clause:

  This explicitly declares the link name SORTWK for the sort file.

  The format of the ASSIGN clause for sort files also permits other specifications, but these are treated as comment entries by the compiler. The link name for the sort file is always SORTWK.

- The sort file description entry (SD) in the Data Division

  This corresponds to the file description entry (FD) for other files and defines the physical structure, the format and the size of the records.

- The SORT and MERGE statements in the Procedure Division:

  input procedure. The sorted records are written to a file or transferred SORT sorts records by one or more data items (up to 64), specified as sort keys.
  These data records can be made available to SORT from a file or via an to an output procedure.

  For sorting, COBOL2000 uses the sorting function of the BS2000 utility SORT for sorting (see “Sort” manual [6]).

  MERGE merges records from two or more sorted input files in a sort file on the basis of a number of data items (up to 64) that have been specified as sort keys.The merged records are written to a file or transferred to an output procedure.

- Declaration of input and output procedures

  An input procedure (INPUT PROCEDURE phase) can be declared for any SORT statement. The procedure allows the records which are to be sorted to be generated or processed before they are passed to the sort file via a RELEASE statement.

  An output procedure (OUTPUT PROCEDURE phase) can be declared for any SORT or MERGE statement. The procedure allows the sorted or merged records to be processed further, after they have been made available to it with a RETURN statement.

> **i** If desired, text can be sorted according to the DIN standard for EBCDIC. To this end, sort format ED of the SORT utility routine is selected for all SORT statements in a program by compiling the program with the SDF option RUNTIME-OPTIONS=PAR(SORTING-ORDER=BY-DIN) or, alternatively, with COMOPT SORT-EBCDIC-DIN=YES (see “Sort” manual [6]).
> This means that
>
> - lowercase letters are equated with the corresponding uppercase letters and
> - the character
>
> “ä” / “Ä” is identified with “AE”
>
> “ö” / “Ö” is identified with “OE”
>
> “ü” / “Ü” is identified with “UE”
>
> “ß” is identified with “SS”
> - digits are sorted before letters

11.2 Files for the sort program

The following files are required for a sort operation:

Sort file

Data records are sorted in this file (work area). Its name is declared, for example, via the clause

```
SELECT sort-file ASSIGN TO "SORTWK"
```

In addition, the file must be described in the sort file description entry (SD) of the Data Division. The file is accessed with the statement

```
SORT sort-file ...
```

Without the user having to issue a SET-FILE-LINK command, this file will be cataloged under the name `SORTWORK.tsn.yymmdd.hhmmss` (where tsn = task sequence number, yy = year, mm = month, dd = day, hhmmss = time in six digits). The link name is SORTWK. After a normal termination of the sort operation the file is deleted.
By default, the size of the sort file when created without the SET-FILE-LINK command is 24 * 16 = 384 PAM pages (this value can be modified by supplying values to special registers for SORT). Accordingly, the primary allocation is 384 PAM pages; the secondary allocation is 1/4 of this, i.e. 96 PAM pages.

Using the command

```
/MODIFY-FILE-ATTRIBUTES filename,–
/      SUPPORT=PUBLIC-DISK(SPACE=RELATIVE(PRIMARY-ALLOCATION=.., –
/                    SECONDARY-ALLOCATION=..))
```

the user can define the sort file size independently (see “Sort” manual [6]). This is recommended for large files. After normal termination of the sort operation this file will be closed, but **not** deleted.

Special registers for SORT (see “COBOL2000 Reference Manual” [1]):

The programmer can load the following special registers for SORT before the sort operation:

- SORT-FILE-SIZE: This register is loaded with the total number of records.
- SORT-MODE-SIZE: This register is loaded with the average record size.

The SORT utility routine uses these two registers to calculate the file size. This implies that the programmer can indirectly affect the SPACE operand.

- SORT-CORE-SIZE: This register is loaded with the desired size of the internal work areas, expressed in bytes. These entries can be used to influence program execution.
  If they are omitted, 24 * 4096 bytes (i.e. 24 4-Kb pages) is assumed by default. For further information see “Sort” manual [6] on sort run optimization.

After SORT, RELEASE and before RETURN statements, the programmer may interrogate the SORT special register SORT-RETURN:

“0” indicates that sorting was successful,

“1” that sorting was errored.

This interrogation is recommended because the program is not terminated in the event of an errored sort operation.

If an invalid value is loaded into a SORT special register, error message COB9134 is issued (see chapter “Messages of the COBOL2000 system”).

Input file(s)

If no input procedure has been defined, COBOL2000 generates an OPEN INPUT and a READ...AT END for the specified file. Each input file must be defined in the COBOL program.

The link names SORTIN and SORTINnn (01 <= nn <= 99) must not be used within a sort program.

Output file

If no output procedure has been defined, COBOL2000 generates an OPEN OUTPUT and a WRITE for the specified file. The output file must be defined in the COBOL program.

The link name SORTOUT must not be used within a sort program.

SORT parameter files

SORT permits predefined values to be specified and modified for some parameters (see MODIFY-SORT-DEFAULTS statement in the “Sort” manual [6]).

Most of these values have no effect on SORT statements in COBOL programs. Consequently such parameter files are analyzed only in the first SORT statement in a COBOL runtime unit. Subsequent changes have no effect for the rest of the program run. This speeds up COBOL programs, which dynamically execute a large number of SORT statements with few records that need to be sorted.

11.3 Checkpointing and restart for sort programs

Specifying the RERUN clause (format 2) causes special checkpoints to be issued for sort files. Checkpoint records contain information concerning the status of the sort operation. They are necessary in order to enable a program which has been interrupted by the user or because of a computer malfunction to be restarted without having to repeat the entire program run up to that point. The taking of sort checkpoints is especially recommended when dealing with large quantities of sort data, because a successful presort will then not be lost in the event of an abnormal program termination.

Format 2 of the RERUN clause:

```
RERUN ON implementor-name EVERY SORT OF sort-file-name
```

implementor-name: SYSnnn (000 <= nnn <= 200)

Checkpoint records are written to a checkpoint file (see chapter “Program linkage”) which is created by the sort program using the default file name `SORTCKPT.Dyyddd.Tnnnn` (where yy = year, ddd = current day of the year, nnnn = task sequence number of the current task) and the standard link name SORTCKPT (see “Sort” manual [6]). Using the SPACE operand in the SUPPORT parameter of the MODIFY-FILE-ATTRIBUTES command, the user can independently determine the size of this file. Checkpoint output is logged on SYSOUT (message E301; see chapter “Program linkage”). The timing of the checkpoint output cannot be determined by the user independently.

When the sort operation terminates normally, the checkpoint file is closed, released and deleted, i.e. the user has no access to it.

If a sort program is terminated abnormally, program execution can be resumed from the last checkpoint taken. To do this, the user issues a RESTART-PROGRAM command which makes use of the information logged on SYSOUT. See chapter “Checkpointing and restart” and “Commands” manual [3].

11.4 Sorting tables

The BS2000 sort function SORT can also be used for sorting tables. The equivalent COBOL language feature is the SORT statement (see “COBOL2000 Reference Manual” [1]).

11.5 Sorting with extended character sets

In sorting with extended character sets, the TRANSLATE-CHARACTER format of SORT (see “Sort” manual [6]) is used in BS2000/OSD.

The special register SORT-CCSN (see the “COBOL2000 Reference Manual” [1]) is available with the SORT statement (file and table sorting) as a language element for sorting with extended character sets[1].

The contents of the special register SORT-CCSN are transferred to SORT as the name of a module from the table module library (SYSLNK.SORT.nnn.TAB[2]).

This library currently contains the modules EDF03DRV, EDF03IRV and EDF041. In order to define additional tables, you need authorization to modify this library.

For the purpose of defining separate modules, SORT provides the source code element MUSTER in the table module library (see also the notes on creating
TRANSLATE-CHARACTER tables in “Sort” manual [6]).

Example 11-1

Creating files with the extended character set EDF041

To create a file in the extended character set using an editor in BS2000, the following steps are necessary:

- Emulation settings: Configuration ... display terminal
  DSS mode: 8 bit
  Character set: Lat. alphabet Nr. 9 ISO8859-15
  Display terminal type: DSS9763

- Modify the logical properties of the terminal (see [3])

```
/MODIFY-TERMINAL-OPTIONS CODED-CHARACTER-SET=EDF041
```

- Set the code in EDT for a new file (see [23]):

```
@CODENAM EDF04
```

Example 11-2

Assigning an output file with an extended character set:

```
/CREATE-FILE SORT-AUSGABE,CODED-CHARACTER-SET=EDF041 ——————————— (1)
/ADD-FILE-LINK LINK-NAME=AUSGABE,FILE-NAME=SORT-AUSGABE ——————————— (2)
```

(1) Instructs the DMS to create the file SORT-AUSGABE with the CODED-CHARACTER-SET EDF041.

(2) Establishes the relationship with the program.

[1] The CODED-CHARACTER-SET attribute of SORT input or output files is not evaluated by COBOL-SORT

[2] *nnn* stands for the current SORT version

12 Checkpointing and restart

Checkpoint records are output by COBOL2000 objects to an external checkpoint file (or two checkpoint files if necessary: see below). A checkpoint record comprises identification information, program status, associated system status and virtual memory contents. All this is required for any subsequent restart which might be effected.

By writing such checkpoint records, it is possible for a program to be continued at any time at the point where the last checkpoint record was written before the program was interrupted (whether intentionally or because of system malfunction). Checkpointing is especially recommended for programs with a fairly long execution time. However, it is only meaningful if the original data can be restored for a possible restart.

This functionality is not available in the POSIX subsystem (see the chapter “COBOL2000 and POSIX”).

12.1 Checkpointing

The writing of checkpoint records is initiated by the user with the aid of the RERUN clause. The user can determine the point in time when the checkpoint records are to be written; for a given file, checkpointing is possible at each reel swapping, or after the processing of a specific number of records of that file.

Format 1 of the RERUN clause (extract; for a complete description, see “COBOL2000 Reference Manual” [1]):

```
RERUN [ON implementor-name] EVERY {END OF {REEL | UNIT} | integer-1 RECORDS} OF file-name
```

implementor-name

Specified as SYSnnn (0 <= nnn <= 244)
COBOL2000 generates either one or two checkpoint files:

a. One checkpoint file if nnn <= 200.
   COBOL2000 forms the standard name `progid.RERUN.SYSnnn` as well as the link name SYSnnn.
   Checkpoint records are written to this file continuously. At end of file, additional storage space is requested internally.
b. Two checkpoint files, if nnn > 200.
   COBOL2000 forms the standard names `progid.RERUN.SYS.nnnA`,
   progid.RERUN.SYS.nnnB and the link names SYSnnnA and SYSnnnB.Checkpoint records are written to each of the two files alternately; any previously written checkpoint record will be overwritten.

Format 2 of the RERUN clause is permitted for sort files only; it is therefore described in section “Checkpointing and restart for sort programs”.
After each successful output of a checkpoint record, information necessary for a possible restart will be displayed on SYSOUT.

Message format:

```
E301 CHECKPOINT#aa, HALF PAGE#=bb, DATE=cc, TIME=dd:ee
```

aa	checkpoint number
bb	PAM page number
cc	mm/dd/yy:month/day/year
dd	hour
ee	minute

12.2 Restart

The RESTART-PROGRAM command enables the user to restart an executable program at a point at which a checkpoint was taken.

Format of the RESTART-PROGRAM command (see “Commands” manual [3]):

```
/RESTART-PROGRAM filename,REST-OPT=START-PROG(CHECKPOINT=NUMBER(...)
```

filename	Name of the checkpoint file (COBOL standard name; see section "Checkpointing")
NUMBER(...)	Number of the PAM page in which the checkpoint records begin

i

1. Before restart the user must assign all resources that are required by the program to be restarted, because data concerning the required resources is not saved when a checkpoint is taken.
2. The status of user data is **not** automatically restored at restart. Therefore, the user has to provide the data in an appropriate form as at the time the checkpoint was taken.

13 Program linkage

A program system consists of a main program (the program that is called at system level) and one or more external subprograms, which may be written in the same language as the main program or in other programming languages.

Consequently, a means of linking the various programs is required; this function is performed by the Inter-Language Communication Services (ILCS). ILCS is a component of the Common Runtime Environment (CRTE) and is described in the “CRTE User Guide” [2].

13.1 Linking and loading subprograms

The name of a subprogram can be specified in the CALL statement either as a literal or as the identifier of a data item that contains the subprogram name or subprogram address. Depending on the kind of subprogram call, a program system is linked and loaded in different ways.

Subprogram call “CALL literal” or program address identifier “ADDRESS OF PROGRAM literal”

The name of the subprogram is already defined at compilation time. The compiler sets up external references to these subprograms; these are resolved by the relevant linkage editor in subsequent linkage runs. If a program system contains only calls in the form “CALL literal” or “ADDRESS OF PROGRAM literal”, it may be linked into a permanent or temporary executable run unit and loaded subsequently, as described in chapter “Linking, loading, starting”.

Subprogram call “CALL identifier” or program address identifier “ADDRESS OF PROGRAM identifier”

The name of the subprogram need not be known before runtime (e.g. upon input at the terminal). For subprograms called as required by means of “CALL identifier” and program address identifier “PROGRAM ADDRESS identifier” there are no external references; they are therefore loaded dynamically by DBL during program execution. Program systems with subprograms of this kind can only be run in one of the following ways:

1. Use DBL to dynamically link the modules generated during the compilation, and dynamically load the subprograms without external references (in the main program).
2. Use TSOSLNK to create a prelinked module which contains the main program as well as the subprograms with external references. Call the prelinked module with DBL and dynamically load the subprograms without external references (in the main program).
3. Use BINDER to (pre-)link one or more link-and-load modules (LLMs). Then use DBL to call the (prelinked) LLM or the LLM containing the main program, and dynamically load the subprograms without external references (in the main program).

Generally, before calling DBL, the following assignment should be made:

```
/ADD-FILE-LINK BLSLIBnn,runtime-library
```

for the Common Run-Time Environment (CRTE), which includes the COBOL runtime system (the library SYSLNK.CRTE.PARTIAL-BIND may not be used for this purpose; see the “CRTE User Guide” [2]).

In addition, the the following assignment must be made:

```
/ADD-FILE-LINK COBOBJCT, library
```

for a library which contains the subprograms to be dynamically loaded.

Note that the link name BLSLIBnn is only effective if

```
RUN-MODE=ADVANCED(ALTERNATE-LIBRARIES=YES)
```

is specified in the invocation command for DBL (see section “Dynamic linking and loading using DBL”).

If a load unit contains unresolved weak external references WXTRNSs (which is the case, for example, when the subprogram contains files with a different file organization than those in the main program), then the operands UNRESOLVED-EXTERNS=DELAY and LOAD-INFORMATION=REFERENCES must be specified:

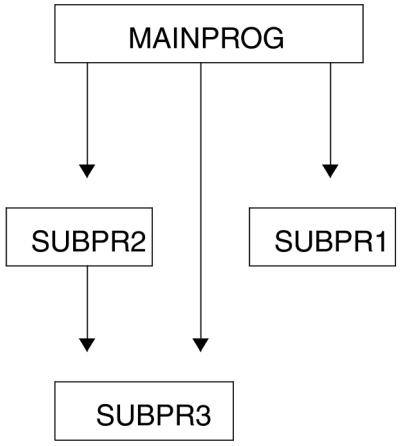
```
RUN-MODE=ADVANCED (ALTERNATE-LIBRARIES=YES, UNRES-EXT=DELAY, LOAD-INF=REF
```

Names of link-and-load modules (LLMs) can be specified in CALL, CANCEL and ADDRESS OF PROGRAM as identifiers with a length of up to 30 characters. For object modules the program names must not exceed eight characters in length. In the case of CANCEL identifier statements for programs in object module format, the first seven characters of these names must be unique in the run unit, and the eighth character must not be a hyphen '-'.

Example 13-1

Linking and loading techniques for program systems containing dynamically loadable subprograms

Program constellation and types of calls



MAINPROG:

```
...
CALL "SUBPR1" USING ...
MOVE "SUBPR2" TO identifier-1
MOVE "SUBPR3" TO identifier-2
CALL identifier-1 USING identifier-2
...
```

SUBPR2:

```
    ...
    PROCEDURE DIVISION USING identifier-2
    CALL identifier-2
```

```
...
CALL "SUBPR3"
...
```

SUBPR1 is called only in the form "CALL literal".
SUBPR2 is called only in the form "CALL identifier".
SUBPR3 is called in either way.

This means that external references are set up for SUBPR1 and SUBPR3; SUBPR2 is loaded dynamically.

The ways of initiating the program run for this program constellation are shown below.

The individual programs are stored as object modules under the element names MAINPROG, SUBPR1, SUBPR2 and SUBPR3 in the library USER-PROGRAMS.

1. Using DBL (dynamic linking)

```
/ADD-FILE-LINK BLSLIB00,$.SYSLNK.CRTE ————————————————————————————(1)
/ADD-FILE-LINK COBOBJCT,USER-PROGRAMS ————————————————————————————(2)
/START-PROGRAM *MODULE(LIB=USER-PROGRAMS,ELEM=MAINPROG,-    ————————————(3)
/RUN-MODE=ADVANCED(ALT-LIB=YES,UNRES-EXT=DELAY,-
/LOAD-INF=REFERENCES))
```

(1) Assigns the runtime library.

(2) Assigns the library from which subprogram SUBPR2 will be dynamically loaded.

(3) Calls the object module containing the main program MAINPROG. DBL searches the library specified here (USER-PROGRAMS) to resolve the external references to SUBPR1 and SUBPR3.

2. Using TSOSLNK (linking prelinked modules)

```
/START-PROGRAM $TSOSLNK
MODULE PRLNKMOD,LET=Y,UNSAT=N, LIB=MODUL.LIB ————————————————————(1)
INCLUDE MAINPROG,USER-PROGRAMS ——————————————————————————————————(2)
RESOLVE,USER-PROGRAMS ———————————————————————————————————————————(3)
LINK-SYMBOLS *KEEP ———————————————————————————————————————————————(4)
END

/ADD-FILE-LINK BLSLIB00,$.SYSLNK.CRTE ————————————————————————————(5)
/ADD-FILE-LINK COBOBJCT,USER-PROGRAMS ————————————————————————————(6)
/START-PROGRAM *MODULE(LIB=MODUL.LIB,ELEM=PRLNKMOD,-    ——————————————(7)
/RUN-MODE=ADVANCED(ALT-LIB=YES,UNRES-EXT=DELAY,-
/LOAD-INFO=REFERENCE))
```

(1) The prelinked module PRLNKMOD is stored in the MODUL.LIB library.

(2) Links in the MAINPROG module from the USER-PROGRAMS library.

(3) Links in the USER-PROGRAMS library to resolve external references to SUBPR1 and SUBPR3.

(4) Symbols for the entry points and the program segments for subsequent execution under DBL are kept visible by means of this statement.

(5) Assigns the runtime library.

(6) Assigns the library from which subprogram SUBPR2 will be dynamically loaded.

(7) Calls the prelinked module PRLNKMOD.

3. Using BINDER (linking LLMs)

Unlike TSOSLNK, BINDER keeps all external references and entry points visible by default; this is essential for the succeeding DBL run.

Moreover, using BINDER enables external references to remain unresolved. This means that the runtime system does not have to be linked in. This is an advantage if a shareable runtime system is to be used for program execution.

a) Generating a single link-and-load module

```
/START-PROGRAM $BINDER
//START-LLM-CREA PRLNKMOD————————————————————————————————————————(1)
//INCLUDE-MODULES LIB=USER-PROGRAMS,ELEM=MAINPROG————————————————————(2)
//INCLUDE-MODULES LIB=USER-PROGRAMS,ELEM=SUBPR2——————————————————————(3)
//RESOLVE-BY-AUTOLINK LIB=USER-PROGRAMS——————————————————————————————(4)
//SAVE-LLM LIB=MODUL.LIB——————————————————————————————————————————(5)
//END

/ADD-FILE-LINK BLSLIB00,$.SYSLNK.CRTE ——————————————————————————————(6)
/START-PROGRAM *MODULE(LIB=MODUL.LIB,ELEM=PRLNKMOD,-   ————————————————(7)
/RUN-MODE=ADVANCED(ALT-LIB=YES,UNRES-EXT=DELAY,-
/LOAD-INFO=REFERENCE))
```

(1) Creates a link-and-load module named PRLNKMOD.

(2) Explicitly links in the main program module MAINPROG from the USER-PROGRAMS library.

(3) Explicitly links in the module SUBPR2 from the USER-PROGRAMS library in order to avoid dynamic loading. This makes it unnecessary to assign the USER-PROGRAMS library with the link name COBOBJCT in the ensuing link-and-load operation.

(4) Links in all other modules required (SUBPR1, SUBPR3) from the USER-PROGRAMS library.

(5) Stores the generated link-and-load module as a type L element in the program library MODUL.LIB.

(6) Assigns the runtime library.

(7) Calls the link-and-load module PRLNKMOD.

b) Converting object modules into single link-and-load modules

```
/START-PROGRAM $BINDER
...
//START-LLM-CREATION INTERNAL-NAME=MAINPROG                          (1)
//INCLUDE-MODULES LIB=USER-PROGRAMS,ELEM=MAINPROG                     |
//SAVE-LLM LIB=MODULE.LLM                                           (1)
...
//START-LLM-CREATION INTERNAL-NAME=SUBPR1                           (2)
//INCLUDE-MODULES LIB=USER-PROGRAMS,ELEM=SUBPR1                       |
//SAVE-LLM LIB=MODULE.LLM,ENTRY-POINT=SUBPR1                        (2)
...
//START-LLM-CREATION INTERNAL-NAME=SUBPR2                           (3)
//INCLUDE-MODULES LIB=USER-PROGRAMS,ELEM=SUBPR2                       |
//SAVE-LLM LIB=MODULE.LLM,ENTRY-POINT=SUBPR2                        (3)
...
//START-LLM-CREATION INTERNAL-NAME=SUBPR3                           (4)
//INCLUDE-MODULES LIB=USER-PROGRAMS,ELEM=SUBPR3                       |
//SAVE-LLM LIB=MODULE.LLM                                           (4)
//END
...
/ADD-FILE-LINK BLSLIB00,$.SYSLNK.CRTE ———————————————————————————— (5)
/ADD-FILE-LINK COBOBJCT,MODULE.LLM ———————————————————————————————— (6)
/START-PROGRAM *MODULE(LIB=MODULE.LLM,ELEM=MAINPROG,- ————————————————— (7)
/RUN-MODE=ADVANCED(ALT-LIB=YES,UNRES-EXT=DELAY,-
/LOAD-INFO=REFERENCE))
```

(1)

Creates an LLM named MAINPROG; the name of the LLM is freely selectable. Links in the object module MAINPROG from the USER-PROGRAMS library. As the element name is omitted from the SAVE-LLM command, the name specified in the START-LLM-CREATION command is used; that is, MAINPROG.

(2) Creates an LLM named SUBPR1. Links in the object module SUBPR1 from the USER-PROGRAMS library. ENTRY-POINT=SUBPR1 defines this LLM as a subprogram.

(3) Creates an LLM named SPROG2. Links in the object module SUBPR2 from the USER-PROGRAMS library. ENTRY-POINT=SUBPR2 defines this LLM as a subprogram.

(4) Creates an LLM named SUBPR3. Links in the object module SUBPR3 from the USER-PROGRAMS library. As no ENTRY-POINT is specified in the SAVE-LLM command, SUBPR3 can be used both as a subprogram and as a main program.

(5) Assigns the runtime library.

(6) Assigns the library containing the LLMs by means of the link name COBOBJCT so that the unresolved external references of the previously created LLMs can be resolved.

(7) Calls the LLM containing the main program MAINPROG.

13.2 COBOL special register RETURN-CODE

The COBOL special register RETURN-CODE can be used for communication purposes between separately compiled COBOL programs of a run unit. The special register exists only once in the program system and is defined internally as a four-digit binary data item (PIC S9(9) COMP-5 SYNC).
RETURN-CODE can be interrogated or modified as required at run time by the separately compiled programs. On termination of the program run, the runtime system checks whether RETURN-CODE is zero. If this is not the case, the error message COB9119 (if the COBOL return code > 0) or COB9128 (if the user return code > 0) is output. If the program was called within a procedure, the program branches to the next SET-JOB-STEP, EXIT-JOB, LOGOFF, CANCEL-PROCEDURE, END-PROCEDURE or EXIT-PROCEDURE command.

When a COBOL subprogram is exited, the value of the special register RETURN-CODE is also loaded in registers 0 and 1. This makes the value available to the calling program in the form of a function value, in accordance with the ILCS conventions.

In order to use a function value from a C program, the calling COBOL program must be compiled with the control statement RETURN-CODE=FROM-ALL-SUBPROGRAMS for the RUNTIME-OPTIONS option, or with the COMOPT operand ACTIVATE-XPG4-RETURNCODE=YES (Attention: the function value cannot be obtained with the “RETURNING” specification in the CALL statement).

To avoid the abnormal termination of the program, the user must ensure that RETURN-CODE contains the value 0 before the STOP RUN statement is reached.

13.3 Passing parameters to programs in other languages

Thanks to the use of COBOL prototypes, it is also possible to write programs in other languages. In such cases, all the possibilities of the extended CALL Format 3 are available when these programs are called (see the “COBOL2000 Reference Manual” [1]). Otherwise only the restricted possibilities of Format 1 and Format 2 can be used.

More details on parameter passing can be found in the “CRTE User Guide” [2].

13.4 Unloading COBOL subroutines

COBOL offers no language resources for unloading subroutines. Users must provide assembler programs themselves to do this (see section "UNBIND macro" in the “Binder-Loader-Starter” manual [10]).

In the event of such unloading processes, dependencies between the modules and on the COBOL runtime system in the context of CRTE must be taken into account:

- While other COBOL modules are still loaded, the COBOL runtime system may not be unloaded, e.g. if the module to be unloaded is linked as an LLM or prelinked module and contains all or part of the runtime system.
- When external files of COBOL programs are addressed, the COBOL runtime system must remain loaded even if all COBOL programs have been unloaded but COBOL subroutines are to be loaded dynamically again to permit further processing of the external files.
- If the COBOL module to be unloaded is a class of interface definition, all modules which use or inherit from these class or interface definitions must be unloaded.
- If the module to be unloaded was loaded dynamically using the "CALL identifier" from a COBOL module which was compiled with the OPTIMIZE-CALL-IDENTIFIER=YES option, the unloaded module may not be called again with the "CALL identifier".

In the case of program exchange of modules under openUTM, the requirements must also be taken into account.

The COBOL compiler does not check whether the rules have been observed.

14 COBOL2000 and POSIX

Not supported in COBOL2000-BC !

The COBOL compiler can be invoked and passed control options in the POSIX environment (POSIX shell).

In addition, COBOL programs compiled in the POSIX environment or in BS2000 can be run in the POSIX environment.

Finally, if the POSIX subsystem is available, it is possible to access the POSIX file system even when running the compiler or a program in BS2000.

For further information on POSIX you can refer to the following publications:

- "POSIX Basics" manual [30]
- "POSIX Commands" manual [29]

14.1 Overview

The following three subsections summarize how the compiler is used in the POSIX subsystem.

14.1.1 Compiling

COBOL compilation units can be compiled by using the POSIX command `cobol`. This command is described in detail in section “Controlling the compiler”.

Generating an LLM object file (“.o” file)

For each source file it compiles, the compiler generates an LLM and stores it as a POSIX object file with the default name *basename*.o.
basename is the name of the source file without its directory components and without the .cob or .cbl extension.

When compiling compilation groups, the compiler generates an LLM for each compilation unit and stores each LLM in a POSIX object file. In this case *basename* is the associated ID-name for the second to the last compilation unit. Lowercase letters are converted to uppercase if required.
Unless otherwise specified, a linkage run is started once the compilation run is completed.You can use the `-c` option (see section “General options”) to suppress the linking phase.

Generating a compiler listing

You can use the `-P` option (see section “Option for compiler listing output”) to request various compiler listings (source listing, diagnostic listing, cross-reference listing and so forth). The compiler writes the listings you request to a listing file with the default name *basename*.lst and stores it in the current directory. *basename* is the name of the source file without the directory components and without the suffix .cob or .cbl. In these cases, the name of the source file can also be specified with the option *-k file-name*.

To print listing files you can use the POSIX `lp` command (see the “POSIX Commands” manual [29]).

Example of printing a compiler listing

```
lp -o control-mode=*physical cobbsp.lst
```

Output locations and output code

The compiler stores the output files in the current directory, i.e. in the directory from which the compiler run was started.
Character and character string constants in the program (object file) are always stored in EBCDIC.

If you store the POSIX file system on a mounted UNIX file system or edit the POSIX files in ASCII code with UNIX system tools, you must store error files (ERRFIL or piped screen output) outside the POSIX file system in BS2000, since code conversion is available only in limited form for these files.

Using compiler variables under POSIX

When the compiler is called under POSIX in BS2000/OSD the values of the compiler variables can be taken over from environment variables. In this case, there is no need to prefix the “SYSDIR-” name part (see also section “Assignment to compiler variables to control source text manipulation”).

In POSIX, environment variables are untyped and their content is interpreted as a string in the program during conditional compilation.

14.1.2 Linking

In the POSIX shell you use the `cobol` command to link a COBOL program into an executable file.

A linkage run is automatically started, provided the `-c` option is not specified (see section “General options”), and no serious errors occurred in a preceding compilation, if any.
Once linked, the program is written to an executable POSIX file in the form of an LLM. The name of this file and the directory it is stored in are defined by the `-o` option. If this option is not specified, the executable POSIX file is stored in the current directory under the default name a.out.

When performing linking in the POSIX shell it is not possible to generate linkage editor listings. If errors occur, error messages are written to stderr.

Linking user modules

User-written modules can be linked in statically and dynamically (i.e. at runtime). Programs containing unresolved external references to user modules cannot be started in the POSIX shell.

The possible input sources for the linkage editor are:

- object files generated by the compiler (“.o” files)
- archives created with the `ar` utility (“.a” files)
- LLMs copied from PLAM libraries to POSIX object files using the POSIX `bs2cp` command. These may be LLMs directly generated by a compiler in a BS2000 environment or object modules written to an LLM using the BINDER linkage editor.
- LLMs and object modules stored in BS2000 PLAM libraries. This entails assigning the PLAM libraries using the BLSLIB*nn* environment variables (see the operand `-l BLSLIB` on section “Options for the linkage run”).

The modules can be modules generated by any ILCS-capable BS2000 compiler (such as COBOL85, COBOL2000, C, C++, ASSEMBH or Fortran90).

If modules generated in a BS2000 environment by the COBOL2000 compiler are to be linked, they must have been compiled using the ENABLE-UFS-ACCESS option.

During the linkage run, INCLUDE-MODULES statements are issued internally for POSIX object files, while RESOLVE-BY-AUTOLINK statements are issued for ar archives and PLAM libraries. The modules are linked in the order described below.

When linking you must use the `-M` option to specify the name of the COBOL main program (PROGRAM-ID name). If you fail to do so, the linkage editor will assume the main program is a C program.

Linking CRTE runtime libraries

The linkage editor resolves open external references to the COBOL2000 runtime system automatically from the CRTE library $.SYSLNK.CRTE.PARTIAL-BIND.

Linking order

1. All compiler-generated object files with INCLUDE-MODULES statements
2. All explicitly specified object files (“.o“ files) with INCLUDE-MODULES statementsand, where appropriate, all explicitly specified ar libraries (“.a“ files). A separate RESOLVE-BY-AUTOLINK statement is issued for each ar library.

3. All ar libraries specified with the options `-l` and `-L`, and PLAM libraries assigned with -l BLSLIB. A separate RESOLVE-BY-AUTOLINK statement is issued for each ar library. The PLAM libraries assigned with `-l BLSLIB` are passed to the BINDER linkage editor in list form in a single RESOLVE-BY-AUTOLINK.
4. The CRTE libraries ($.SYSLNK.CRTE.PARTIAL-BIND) and, if appropriate, the SORT library ($.SORTLIB)

The object files and libraries processed in steps 1 to 3 are linked in the order in which they appear in the command line. In the case of object files generated by the compiler (see “COBOL2000 Reference Manual” [1]), the order of linking depends on the order of the associated source files.

Example 14-1

```
export BLSLIB99=‘$MYTEST.LIB2‘
export BLSLIB01=‘$MYTEST.LIB1‘
cobol -M COBBSP -o cobbsp cobupro1.cob cobupro2.cob cobbsp.o cobupro3-5.a \
      -L /usr/private -l xyz -l BLSLIB
```

Linking order:

1. cobupro1.o
2. cobupro2.o
3. cobbsp.o
4. cobupro3-5.a
5. /usr/private/libxyz.a
6. $MYTEST.LIB1
7. $MYTEST.LIB2
8. Runtime libraries

14.1.3 Debugging

Linked programs can be debugged with the AID Advanced Interactive Debugger. This is conditional on the availability of debugging information (LSD) generated by the compiler when invoked with the `-g` option (see section “Debugger option”).

To activate the AID debugger, you use the POSIX command `debug` *program-name [arguments]* at a BS2000 terminal.
After you enter this command, the BS2000 environment will be your current environment, as indicated by the % DEBUG/ prompt. In this mode you can enter debugging commands as described in the “AID” manual [8] . When you terminate the program, your current environment will again be the POSIX shell.

The `debug` command is described in the “POSIX Commands” manual [29].

14.2 Reading in the compilation unit

The COBOL2000 compiler identifies COBOL source files by the presence of one of the following standard file name extensions:

`.cob` or `.cbl`

COBOL source files with file names not ending in a standard extension can still be compiled, as long as the file names are specified with the **-k** option (see section “General options”).

Compilation units stored in BS2000 files or PLAM libraries cannot be processed by the compiler in the POSIX subsystem.

To transfer BS2000 files and PLAM library elements to the POSIX file system and back you can use the POSIX `bs2cp` command.

The POSIX `edt` command allows you to edit POSIX files at a BS2000 terminal.If the POSIX shell was accessed with rlogin, the POSIX command `vi` is available for editing purposes. Refer to the “POSIX Commands” manual [29].

Input of program segments (COPY elements)

The format of the COPY statement for extracting COPY texts from POSIX files is as follows:

```
COPY text-name [IN/OF library-name]
```

text-name is the name of the POSIX file (without the directory components) containing the COPY text. The name may not contain lowercase letters.

library-name is the name of an environment variable containing one or more absolute path names of directories to search. The name may not contain lowercase letters.

If the IN/OF *library-name* argument is not included in the COPY statement, the compiler evaluates the contents of an environment variable named COBLIB.

Before the compiler is invoked, the environment variables must have been assigned the path names of the directories to search and exported with the POSIX `export` command.

Example 14-2

COPY statements in the compilation unit:

```
...
COPY TEXT1 IN COPYLNK
COPY TEXT2 IN COPYLNK
...
```

Defining and exporting the environment variable in the POSIX shell:

```
export COPYLNK=/USERIDXY/copy1:/USERIDXY/copy2
```

The colon causes the COPYLNK environment variable to be initialized with the names of two directories to be searched for the POSIX files containing the COPY texts (TEXT1, TEXT2). The directory /USERIDXY/copy1 is first searched and then the directory /USERIDXY/copy2.

14.3 Controlling the compiler

In the POSIX shell you can use the **cobol** command to invoke the COBOL2000 compiler and pass it control options.

Command-line syntax

cobol 'BLANK'*option*'BLANK' ... *input-file*'BLANK' ...

Input rules

1. Options and input files may be specified in any order.
2. Options without arguments (e.g. `-c, -v, -g`) may be grouped together (e.g. `-cvg`).
3. An option (such as `-C`) and its arguments must not be grouped in this way. There must always be a blank ('BLANK') between the option and the argument (e.g. `-C EXPAND-COPY=YES`).
4. The following options may appear more than once in the command line:

   -C, -k, -L, -P, -I

   All other options are only allowed once. If one of these options is specified more than once, the last instance in the command line will apply.
5. Options that are unknown to the compiler, i.e. which begin with an unknown character after the hyphen (“-”), are passed through to the linkage editor cobld. If a blank appears between the unknown option and an argument, the option is interpreted as one without an argument and passed accordingly.

By default, i.e. if the `-c` option is not used to terminate the compiler run after compilation and if the program compiled without serious errors, a linkage run with the linkage editor cobld automatically follows compilation.

The options for controlling the compilation and linkage run are described below.

14.3.1 General options

–c

This option terminates the compiler run once an LLM has been generated and stored in an object file named *basename*.o for each source file compiled. *basename* is the name of the source file without its directory components and without the `.cob` or `.cbl` extension. The object file is written to the current directory.

If a compilation unit is compiled without this option, a linkage run is started once compilation is complete.

–k *filename*

This option allows you to specify a COBOL source file which does not have the extension `.cbl` or `.cob`.

If the source file name specified with `-k` does however end with the suffix `.cbl` or `.cob`, this suffix is overwritten with the suffix `.o` or `.lst` when the basename for the object and listing files is formed.

–v

This option causes the following information to be displayed on the screen:

- Copyright and version strings of the driver for the COBOL2000 compiler and the cobol command
- Messages of the COBOL2000 compiler relating to accepted control statements
- All the information and error messages of the compilation run
- CPU time consumed
- the full command line for the call to the linkage editor

This option affects only the output of the COBOL2000 compiler.

–W *err-level*

This option is mapped internally to COMOPT MINIMAL-SEVERITY = err-level. The COMOPT MINIMAL-SEVERITY should not therefore be passed with -C.
As a result of this option, the diagnostic listing excludes any messages with an error level lower than the specified value. The possible values for *err-level* are:

`I` information (default)

`0` warning

`1` error

`2` unrecoverable error

`3` system error

14.3.2 Option for compiler statements

–C *comopt*

comopt can be replaced by all the COMOPT statements listed below, either in full or abbreviated form. The functions of the various COMOPTs are described in chapter “Controlling the compiler with COMOPT statements”.

Example 14-3

-C SET-FUNCTION-ERROR-DEFAULT=YES or -C S-F-E-D=YES

Overview: COMOPTs that can be passed with the –C option

COMOPT	Possible abbreviation
ACCEPT-LOW-TO-UP={YES/NO}	ACC-L-T-U
ACTIVATE-WARNING-MECHANISM={YES/NO}	ACT-W-MECH
ACTIVATE-XPG4-RETURNCODE={YES/NO}	
ALIGN-LLM-PAGE={YES/NO}	A-L-P
CHECK-CALLING-HIERARCHY={YES/NO}	CHECK-C-H
CHECK-DATE={YES/NO}	CHECK-D
CHECK-FUNCTION-ARGUMENTS={YES/NO}	CHECK-FUNC
CHECK-PARAMETER-COUNT={YES/NO}	CHECK-PAR-C
CHECK-REFERENCE-MODIFICATION={YES/NO}	CHECK-REF
CHECK-SCOPE-TERMINATORS={YES/NO}	CHECK-S-T
CHECK-SOURCE-SEQUENCE={YES/NO}	CHECK-S-SEQ
CHECK-TABLE-ACCESS={YES/NO}	CHECK-TAB
CONTINUE-AFTER-MESSAGE={YES/NO}	CON-A-MESS
DEFAULT-CALL-CONVENTION= {COBOL/COMPATIBLE}	DEF-C-C
ENABLE-COBOL85-KEYWORDS-ONLY={YES/NO}	
EXPAND-COPY={YES/NO}	EXP-COPY
FLAG-ABOVE-INTERMEDIATE={YES/NO}	
FLAG-ABOVE-MINIMUM={YES/NO}	
FLAG-ALL-SEGMENTATION={YES/NO}	
FLAG-INTRINSIC-FUNCTIONS={YES/NO}	
FLAG-NONSTANDARD={YES/NO}	
FLAG-OBSOLETE={YES/NO}	
FLAG-REPORT-WRITER={YES/NO}	
FLAG-SEGMENTATION-ABOVE1={YES/NO}	

GENERATE-INITIAL-STATE={YES/NO}	GEN-INIT-STA
GENERATE-LINE-NUMBER={YES/NO}	GEN-L-NUM
GENERATE-SHARED-CODE={YES/NO}	GEN-SHARE
IGNORE-COPY-SUPPRESS={YES/NO}	IGN-C-SUP
IGNORE-OPTION-DIRECTIVES={YES/NO}	IGN-O-DIR
INHIBIT-BAD-SIGN-PROPAGATION={YES/NO}	
LINE-LENGTH=132 / 119..172	LINE-L
LINES-PER-PAGE=64 / 20..128	LINES
MARK-NEW-KEYWORDS={YES/NO}	M-N-K
MAXIMUM-ERROR-NUMBER=1..100	MAX-ERR
MERGE-DIAGNOSTICS={YES/NO}	M-DIAG
MERGE-REFERENCES={YES/NO}	M-REF
PERMIT-STANDARD-DEVIATION={YES/NO}	P-S-D
RESET-PERFORM-EXITS={YES/NO}	RES-PERF
ROUND-FLOAT-RESULTS-DECIMAL={YES/NO}	ROUND-FLOAT
SEPARATE-TESTPOINTS={YES/NO}	SEP-TESTP
SET-FUNCTION-ERROR-DEFAULT={YES/NO}	S-F-E-D
SHORTEN-OBJECT={YES/NO}	SHORT-OBJ
SHORTEN-XREF={YES/NO}	SHORT-XREF
SORT-EBCDIC-DIN={YES/NO}	SORT-E-D
SORT-MAP={YES/NO}	
SUPPRESS-LISTINGS={YES/NO}	SUP-LIST
SUPPRESS-MODULE={YES/NO}	SUP-MOD
TERMINATE-AFTER-SEMANTIC={YES/NO}	TERM-A-SEM
TERMINATE-AFTER-SYNTAX={YES/NO}	TERM-A-SYN
TEST-WITH-COLUMN1={YES/NO}	TEST-W-C
UPDATE-REPOSITORY={YES/NO}	UPD-R
USE-APOSTROPHE={YES/NO}	USE-AP

14.3.3 Option for compiler listing output

–P “(*listing, ...*)”

This option controls which listings are generated by the compiler.
This option is mapped internally to COMOPT SYSLIST=(listing,...). The COMOPT SYSLIST should not therefore be passed with -C.
The *listing* argument takes the form of a list of any of the following values (as with COMOPT SYSLIST in BS2000):

```
OPTIONS
NOOPTIONS
SOURCE
NOSOURCE
MAP
NOMAP
OBJECT
NOOBJECT
DIAG
NODIAG
XREF
NOXREF
ALL
NO
```

By default (NO), no compiler listings are generated.

The compiler writes listings requested with the `-P` option to a listing file named *basename*.lst, where *basename* is the name of the source file without its directory components and without the .cob or .cbl extension. The listing file is stored in the current directory.

Example 14-4

```
-P "(ALL,NOXREF)"
```

14.3.4 Options for the linkage run

The following linkage editor options have no effect if the `-c` option is used to terminate the compiler run once compilation is complete. The `cobol` command issues a warning for any such unused option.
General information on linking and the linking order is provided in section “Linking”.

–L *directory*

You use this option to specify path names of directories that the linkage editor is to search for libraries named `lib`*name*`.a`. These libraries must be specified with the `-l` *name* option. By default, only `/usr/lib` and `/usr/ccs/lib` are searched for libraries.The order of the `-L` options is significant. The directories specified with `-L` are always searched with higher priority, i.e. before the default directories.
The -L options must be specified before the -l options to which they apply.

–M *name*

name must be the PROGRAM-ID name of the COBOL main program in uppercase letters. This option must always be specified if the main program is a COBOL program.

–o *output-file*

The executable file generated by the linkage editor is written to *output-file*.
If *output-file* does not include any directory components, the file will be stored in the current directory; otherwise it will be stored in the directory specified as part of *output-file*.
By default, the executable is stored in the current directory under the name `a.out`. The following must be noted here: both write and read access rights are required for the output file.

–l *name*

This option causes the linkage editor to search the library named `lib`*name*.a when resolving external references using the AUTOLINK mechanism.
If no other directory is specified with the linkage editor’s `-L` option, the linkage editor looks for the specified library in `/usr/lib` and `/usr/ccs/lib`.
The sort library lib*sort.a* (for example) is not in the default directories, but is installed as a PLAM library in BS2000. This also applies to the runtime system library libc.a.
The linkage editor searches the libraries in the order in which they appear in the command line.

–l BLSLIB

This option causes the linkage editor to scan PLAM libraries that have been assigned using the shell environment variables named BLSLIB*nn* (00 <= *nn* <= 99). The environment variables must have been assigned the library names and exported with the POSIX `export` command before the compiler is invoked. The libraries are scanned in ascending order of *nn*.
All the libraries assigned with the BLSLIB*nn* environment variables are internally passed to the BINDER in list form in a single RESOLVE statement.

Example 14-5

```
export BLSLIB00=‘$RZ99.SYSLNK.COB.999‘
export BLSLIB01=‘$MYTEST.LIB‘
cobol mytest.o -l BLSLIB -M MYTEST
```

14.3.5 Debugger option

−g

The compiler generates additional information (LSD) for the AID debugger.
By default no debugging information is generated.
This option is mapped internally to COMOPT SYMTEST=ALL.
The COMOPT SYMTEST should not therefore be passed with -C.

14.3.6 Input files

The compiler deduces the contents of a file from its file name extension and performs the appropriate compilation actions. Hence the file name must have the extension which is appropriate to the file's contents in accordance with POSIX conventions. The choices are:

Extension	Meaning
.cob/.cbl	COBOL source file
.o	Object file generated in an earlier compilation
.a	Object file archive generated by the `ar` utility

Files with a .cob or .cbl extension are the input sources for the COBOL2000 compiler. The COBOL2000 compiler also recognizes COBOL source files with names which do not have one of these standard extensions, but in this case the names of the source files must be specified with the `-k` *filename* option (see section "General options"), and not as operands.

Files with a .o or .a extension are the input sources for the linkage editor.

File names with other extensions are passed through to the linkage editor cobld.

14.3.7 Output files

The following files are generated with default names and stored in the current directory. You can select a different file name and a different directory for the linkage editor output file (a.out) by using the –o option (see section “Options for the linkage run”). *basename* is the name of the source file without its default extension and without its directory components.

basename.lst	file containing all the compiler listings
basename .o	LLM object file generated by the compiler, suitable for further processing by the linkage editor
a.out	executable file generated by the linkage editor

When compilation groups are compiled, the names of the LLM object files for the second through to the last compilation unit are formed from the ID name and the .o extension (see also section “Compiling”).

14.4 Introductory examples

Compiling and linking with the cobol command

```
cobol -M BSPPROG hugo.cob
```

compiles `hugo.cob` and generates an executable named `a.out`.
The program with the PROGRAM-ID name BSPPROG becomes the main program

```
cobol -o hugo -M BSPPROG hugo.cob
```

compiles `hugo.cob` and generates an executable named `hugo`.
The program with the PROGRAM-ID name BSPPROG becomes the main program

```
cobol -c -P "(SOURCE,DIAG)" hugo.cob upro.cob
```

compiles `hugo.cob` and `upro.cob` and generates the object files `hugo.o` and `upro.o` and a source and diagnostic listing for each compilation unit. The listings are stored in the listing files `hugo.lst` and `upro.lst` respectively.

```
cobol -M BSPPROG -o hugo hugo.o upro.o
```

links the main program `hugo.o` and the module `upro.o` into an executable named `hugo`. The program with the PROGRAM-ID name BSPPROG becomes the main program

14.5 Comparison with COBOL2000 in BS2000

Due to the system-specific differences between POSIX and BS2000, when developing programs to run under POSIX you need to allow for a number of special features related to the scope of the language and its runtime behavior, as discussed in the following subsections.

14.5.1 Restrictions on the functionality of the language

The following language tools of the COBOL2000 compiler are not supported when the program is run in the POSIX subsystem.

Dynamic subprogram call

Subprogram calls with the COBOL statement CALL *identifier* are illegal in POSIX and could abort the program run.

Program address identifier

Like “CALL identifier ...”, “ADDRESS OF PROGRAM identifier” requires dynamic loading at run time and is therefore not possible in POSIX.

ENTRY statement

The ENTRY statement is not allowed when the program is run under POSIX because it can only be used to define entry points to object modules, and under POSIX the compiler always generates link-and-load modules (LLMs).

Segmentation

As the compiler under POSIX always generates link-and-load modules (LLMs), COBOL program segmentation is not possible in POSIX.

File processing

- Label handling when processing magnetic tapes is not possible in POSIX.
- Checkpointing for restart of magnetic tapes is not possible in POSIX.
- Shared updating of files is not possible in POSIX.
- The usual LOCKING mechanism of UNIX is implemented in POSIX, so multiple instances of the same file could, for example, be opened concurrently for output.
- The STANDARD-2 character set (International Reference Version of the ISO 7-Bit Code) specified in the ALPHABET clause is not supported in the CODE-SET clause. Any OPEN of this type will be currently rejected with FILE STATUS 30.
- The messages COB9151 and COB9175 for errors on accessing POSIX files do not contain the DMS codes, but the corresponding SIS message numbers. This also applies to the “extended” file status returned to the COBOL object. Even the returned file status could deviate from the usually expected value (see section “I-O status”).
- READ PREVIOUS is not supported and is rejected with file status 96.

XML documents

The new language elements for reading XML documents need to be locaded dynamically at the program’s execution time and are therefore not possible in POSIX.

14.5.2 Extensions to the functionality of the language

Access to the command line

Under POSIX it is possible to access the command line from within a program by means of ACCEPT/DISPLAY statements in conjunction with the special names ARGUMENT-NUMBER and ARGUMENT-VALUE (see “COBOL2000 Reference Manual” [1]).

Example 14-6

```
    IDENTIFICATION DIVISION.
    ...
    SPECIAL-NAMES.
        ARGUMENT-NUMBER IS NO-OF-CMD-ARGUMENTS
        ARGUMENT-VALUE  IS CMD-ARGUMENT
    ...
    WORKING-STORAGE SECTION.
    01  I   PIC  99   VALUE 0.
    01  J   PIC  99   VALUE 0.
    01  A   PIC  X(5) VALUE ALL "x".
    ...
    PROCEDURE DIVISION.
    ...
      ACCEPT I FROM NO-OF-CMD-ARGUMENTS
      DISPLAY "no.ofcommandarguments=" I
      PERFORM VARYING J FROM 1 BY 1 UNTIL J > I
        ACCEPT A FROM CMD-ARGUMENT
        DISPLAY "cmd argument-" J " <" A ">"
      END-PERFORM
    ...
     DISPLAY 2 UPON NO-OF-CMD-ARGUMENTS
     ACCEPT A FROM CMD-ARGUMENT
     DISPLAY "argument-2" " :" A ":"
    ...
```

Calling the program

```
/a.out AAAA BBB CC D
```

Runtime log

```
no. of command arguments=4
cmd argument-1 <AAAA >
cmd argument-2 <BBB  >
cmd argument-3 <CC   >
cmd argument-4 <D    >
argument-2 :BBB  :
```

14.5.3 Differences in the program/operating system interfaces

COBOL programs running in POSIX behave differently in some respects to when running in BS2000:

Low-volume data I/O

In POSIX, the COBOL2000 implementor names in ACCEPT/DISPLAY statements for lowvolume data I/O are assigned the following standard input and output streams:

COBOL2000	BS2000	POSIX
TERMINAL	SYSDTA	stdin
SYSIPT	SYSIPT	undefined
TERMINAL	SYSOUT	stdout
PRINTER	SYSLST	stdout
PRINTER01..99	SYSLST01..99	undefined
SYSOPT	SYSOPT	undefined
CONSOLE	CONSOLE	undefined

Sorting and merging

The sort file is automatically stored in the BS2000 file system, and the POSIX user has no access to it.

Job variable

The use of BS2000 job variables is not possible for programs run under POSIX.

Job switches and user switches

The use of BS2000 job switches and user switches is not meaningful for programs run under POSIX.

File processing

- The link between the external file name in the ASSIGN clause and the file name in the POSIX file system is established using an environment variable whose name is identical to the external file name in the ASSIGN clause. The name of the environment variable must always be in uppercase letters. Detailed information is provided in section “Program execution in the POSIX shell”.
- Program execution is not interrupted after an unsuccessful OPEN INPUT on a file for which OPTIONAL has not been specified.
- Some I/O status values are different in POSIX:

BS2000	POSIX
37	30
93, 94, 95	90

- In the extended I/O status that can be requested with filename-2 in the FILE STATUS clause, the (POSIX) SIS code is output instead of the (BS2000) DMS code.
- The file attributes are finally defined when the file is opened for the first time and cannot be modified later.

- Relative files which use the BS2000 access method UPAM cannot be processed.
- COB90xx messages are written in POSIX to stderr.

Repository usage

It is not possible to assign one or more repositories for input and one repository for output. Only the default repository SYS.PROG.LIB in BS2000 is available for this purpose (no assignment is required).

14.6 Processing POSIX files

- Program execution in the BS2000 environment
- Program execution in the POSIX shell
- I-O status

14.6.1 Program execution in the BS2000 environment

A COBOL program developed and executed in BS2000 can, in certain circumstances, access files from the POSIX file system as well as cataloged (BS2000) files.

Requirements

- When compiling, the compiler option ENABLE-UFS-ACCESS=YES or the SDF option RUNTIME-OPTIONS=PAR (ENABLE-UFS-ACCESS=YES) must be specified.
- When linking, the POSIX link option module contained in the CRTE library SYSLNK.CRTE.POSIX must be linked **with higher priority** ahead of the modules in the library SYSLNK.CRTE or SYSLNK.CRTE.PARTIAL-BIND.
  When linking using TSOSLNK or BINDER, this library should be linked using an INCLUDE or INCLUDE-MODULES statement (without specifying the module name).
  When linking dynamically using the DBL, the library must be assigned a BLSLIB*nn* with a lower *nn* than the CRTE libraries to be linked with lower priority.
  When developing programs in the POSIX shell using the cobol command, the CRTE library is automatically linked.

Restrictions

The processing of a POSIX file is subject to the following restrictions:

- no label processing
- no checkpointing for restart
- no shared updating
- no support of pseudo files (see ADD-FILE-LINK in BS2000/OSD “Commands” manual [3].
- The file attributes are finally defined when the file is opened for the first time and cannot be modified later.
- Relative files which use the BS2000 access method UPAM cannot be processed.
- The STANDARD-2 character set (International Reference Version of the ISO 7-Bit Code) specified in the ALPHABET clause is not supported in the CODE-SET clause. Any OPEN of this type will be currently rejected with FILE STATUS 30.
- The messages COB9151 and COB9175 for errors on accessing POSIX files do not contain the DMS codes, but the corresponding SIS message numbers. This also applies to the “extended” file status returned to the COBOL object. Even the returned file status could deviate from the usually expected value (see section “I-O status”).
- The use of dummy files is not supported.
- Files larger than 32 Gbytes can be processed without the need to activate this in the /ADD-FILE-LINK command.

Assigning a POSIX file

A POSIX file is assigned using an S variable named SYSIOL-*external-name*, where SYSIOL is a fixed component of the name and *external-name* must contain the link name from the program’s ASSIGN clause.

The S variable is set up with the command DECLARE-VARIABLE, which has the following format:

```
/[SET-VAR] SYSIOL-external-name= {  ’*POSIX(filename)‘
                                 | ’*POSIX(relative-pathname)’
                                 | ’*POSIX(absolute-pathname)’
                                }
```

`filename` identifies the requested file if it resides in the home directory of the POSIX file system.

`relative-pathname` is the file name with the directory components as of the home directory.

`absolute-pathname` is the file name with all directory components including the root directory (beginning with /).

Example 14-7

mixed file processing

COBOL compilation unit:

```
...
FILE-CONTROL.
    SELECT POSFILE ASSIGN TO "CUST1"
    SELECT BS2FILE ASSIGN TO "CUST2"
...
```

Linkage with the POSIX file before calling the program:
`/SET-VAR SYSIOL-CUST1='*POSIX(/USERIDXY/customers/cust1)'`

Linkage with the BS2000 file before calling the program:
`/ADD-FILE-LINK CUST2,CUST.FILE`

14.6.2 Program execution in the POSIX shell

A COBOL program developed and executed in the POSIX shell or in BS2000 can process POSIX files without any preparatory steps when compiling and linking (cf. program execution in BS2000).
It is not possible to process BS2000 files from the POSIX shell.

When processing POSIX files, the same functionality restrictions apply as for file processing in BS2000 (see section “Program execution in the BS2000 environment”).

Assigning a POSIX file

A POSIX file is assigned using a shell environment variable named `external-name`. `external-name` is a file name from the program’s ASSIGN clause.
The environment variable must be initialized with the name of the POSIX file and exported using the POSIX export command.
The environment variable is initialized as follows:

```
external-name= {filename | relative-pathname | absolute-pathname}
```

`filename` identifies the requested POSIX file if it is in the current directory.

`relative-pathname` is the file name with the directory components as of the current directory.

`absolute-pathname` is the file name with all directory components including the root directory (beginning with /).

Example 14-8

COBOL compilation unit:

```
...
FILE-CONTROL.
SELECT AFILE ASSIGN TO "CUST1"
...
```

Linkage with the POSIX file cust1 before calling the program:

```
export CUST1=/USERIDXY/customers/cust1
```

14.6.3 I-O status

The status of each access operation performed on a file is stored by the runtime system in specific data items which can be assigned to every file in the program. These items, which are specified by using the FILE STATUS clause, provide information on

- whether the I/O operation was successful, and
- the type of any errors that may have occurred.

This data can be evaluated (by USE procedures in the DECLARATIVES, for example) and used by the program to analyze I-O errors. As an extension to Standard COBOL, COBOL2000 provides the option of including the keys of the POSIX error messages in this analysis, thus allowing a finer differentiation between different causes of errors. The FILE STATUS clause is specified in the FILE-CONTROL paragraph of the Environment Division. Its format is described in section “I-O status”.

The functions of the two data items definable in the FILE STATUS clause are as follows:

data-name-1

contains a two-character numeric status code following each access operation on the associated file.

data-name-2

is broken down into data-name-2-1 and data-name-2-2 and is used for storing the (POSIX) SIS codes for the relevant I-O status. Following each access operation on the associated file, it contains a value that directly depends on the content of data-name-1. The value can be derived from the table below:

Content of data-name-1 not equal 0?	SIS code not equal 0?	Value of data-name-2-1	Value of data-name-2-2
no	not relevant	undefined	undefined
yes	no	0	undefined
yes	yes	96	SIS code of the associated error message

For program execution in BS2000, the meaning text of each SIS code can be output using the command HELP-MSG-INFORMATION SIS<data-name-2-2>.

The base and extended I-O status values are described in the two tables that follow.

Base I-O status

Value	Org *)	Meaning
0x		Execution successful
00 02 04 05 07	SRI I SRI SRI S	No further information Successful READ, allowable duplicate key Successful READ, but record length error Successful OPEN on nonexistent OPTIONAL file - Successful OPEN with NO REWIND - Successful CLOSE with NO REWIND, REEL/UNIT or FOR REMOVAL
1x		Execution unsuccessful: AT END condition

10 14	SRI R	Unsuccessful READ - end of file reached Unsuccessful READ - key item length error
2x		Execution unsuccessful, key error
21 22 23 24	I RI RI RI	Incorrect key sequence on sequential access WRITE for existing record READ for nonexistent record Key item length error
3x		Execution unsuccessful, unrecoverable error
30 34 35 38 39	SRI S SRI SRI SRI	No further information (check SIS code) Insufficient secondary allocation in CREATE-FILE or MODIFY-FILE-ATTRIBUTES command OPEN INPUT/I-O on nonexistent file OPEN for file closed using CLOSE WITH LOCK OPEN error due to incorrect file attributes
4x		Execution unsuccessful, logical error
41 42 43 44 46 47 48 49	SRI SRI S RI SRI S RI S RI SRI S RI	OPEN for a file which is already open CLOSE for a file which is not open REWRITE not preceded by successful READ DELETE/REWRITE not preceded by successful READ WRITE/REWRITE with invalid record length Repeated READ after unsuccessful READ or after detection of AT END Sequential READ after unsuccessful READ/START or after detection of AT END READ for file not opened for reading READ/START for file not opened for reading WRITE for file not opened for writing REWRITE for file not opened in I-O mode DELETE/REWRITE for file not opened in I-O mode
9x		Other conditions with unsuccessful execution
90 91 96	SRI SRI R1	System error, no further information OPEN error or no free device READ PREVIOUS is not supported

*) S = sequential organization, R = relative organization, I = indexed-sequential organization

Extended I-O status - (SIS code)

I/O status	Meaning
0601	End-of-file detected
0602	Specified record does not exist
0603	Specified record exists
0604	Start-of-file detected
0605	Specified link does not exist

0606	File name longer than P_MAXFILENAME
0607	Path string longer than P_MAXPATHSTRG
0608	Path name longer than P_MAXPATHNAME
0609	Link name longer than P_MAXLINKNAME
0610	Out of memory
0611	Number of path elements exceeds P_MAXHIERARCHY
0612	Function not supported
0613	File name missing or syntactically incorrect
0614	Number of secondary keys exceeds P_MAXKEYS
0615	Too many files open at once
0616	Specified file does not exist
0617	Write access not allowed
0618	No file name specified
0619	File is locked
0620	Invalid combination of file attributes
0621	Invalid file handling specified
0622	Current record shorter than MINSIZE
0623	Current record longer than MAXSIZE
0625	No sequential READ before sequential REWRITE
0626	Invalid record format
0627	MINSIZE larger than MAXSIZE
0628	Invalid file organization
0629	File exists though declared as nonexistent
0630	Specified access function not allowed
0631	Specified key
0632	Key duplication not allowed
0633	Current record is locked
0634	Current key out of sequence
0635	Specified path undefined
0636	System-specific error occurred

0637	End-of-line reached
0638	Record truncated
0640	No space available to extend file
0643	Invalid file open mode
0644	Length of link exceeds P_MAXLINKSTRG
0645	Invalid version string specified
0646	Specified file lifespan invalid
0647	Syntax error in file, link or path string
0649	File close mode invalid
0650	Access denied
0651	Parameter error
0652	Invalid pointer to I/O area
0653	Invalid record length detected
0654	Storage limits reached on device
0655	Specified feed control invalid
0656	Specified code invalid
0657	Invalid combination of open mode and file lifespan
0658	I/O aborted
0659	Length of key identifier exceeds P_MAXKEYWORD
0660	Key identifier ambiguous
0661	Number of exits exceeds P_MAXEXITS
0662	New line detected
0663	New page detected
0664	Not all paths closed
0665	Next indexed record has same secondary key
0666	Secondary key of written record already exists
0667	Current record number exceeds MAX_REC_NR
0668	Path name already exists
0669	Link name already exists
0670	Specified value for positioning condition invalid

0671	Unknown control character detected
0672	A unique file name could not be generated
0673	Last record incomplete; function not executed
0674	Specified value for positioning invalid
0675	Unidentifiable record format
0676	Unidentifiable MAXSIZE
0677	Internal PROSOS-D error
0678	Specified file is a file container
0679	Specified file cannot be reached on given path
0680	Version not incrementable
0681	Defective reopen after implicit close
0682	Defective PROSOS-D initialization
0683	Number of link indirections exceeds P_MAXLINKNESTING

15 Useful software for COBOL users

- Advanced Interactive Debugger (AID)
- Library Maintenance System (LMS)
- Job variables
- Database interface ESQL-COBOL
- Universal Transaction Monitor openUTM
- Net Express® development environment with the BS2000/OSD option

15.1 Advanced Interactive Debugger (AID)

Product characteristics

AID is an efficient and powerful debugging system which allows users to diagnose errors, test programs, and provisionally correct programming errors under the BS2000 operating system.

AID supports symbolic debugging of programs written in COBOL, C, C++, Assembler, FORTRAN, and PL/1 as well as non-symbolic debugging at machine-code level of programs written in any BS2000 programming language.

Symbolic debugging of a COBOL program means that symbolic names from a COBOL compilation unit can be used for addressing. Non-symbolic debugging at machine-code level is generally required whenever symbolic testing proves insufficient or impossible as a result of lacking diagnostic information.

Some of the basic functions that can be called using special AID commands are listed below:

- execution monitoring
  - specific types of source statements in the compilation unit
  - selected events in the program run
  - declared program addresses
- access to data items and modification of item contents
- management of AID output files and libraries
- detection of global declarations
- control of
  - output data sets
  - AID output volumes

There is also an additional HELP function

- for all AID commands and operands
- for the meaning of AID messages and possible actions to be taken.

The user can control program execution by instructing AID to interrupt a program run and execute certain subcommands at defined addresses, during the execution of selected types of statements, or when specific events take place. A subcommand is an individual command or a sequence of AID and BS2000 commands. It is defined as an operand of an AID command. Starting with version V2.0, the execution of subcommands can be made dependent on conditions. This enables dynamic monitoring of program states and the values of variables.

AID also provides facilities for the modification of data items and the output of elementary data items, group items, or entire Data Divisions of COBOL programs.

An AID command can be used to display the level of the call hierarchy at which the program was interrupted and the modules which are contained in the CALL or INVOKE nesting.

AID can be used to process a running program as well as to analyze a memory dump in a disk file. It is possible to switch between these two options within a single debugging session, e.g. in order to compare data in an executing program with the data obtained from a memory dump.

Description of functions

AID is a diagnostic and debugging tool for testing application programs at source language level (high level language debugger).

The debugging and diagnostic functions available for the testing of COBOL compilation units compiled with COBOL2000 are:

- Output and setting of user-defined data:

  Data defined in the user program can be addressed interactively, subject to the COBOL rules pertaining to qualification, uniqueness, indexing, and scope.
  The data itself is converted and edited in accordance with the attributes specified in the user program.
- Symbolic dump:

  All or selected data from dynamically nested programs can be edited and output according to the current program nesting.
- Setting of test points:

  Test points at which specific actions are to be executed can be set or reset via source references or markers in the program (paragraphs, sections, etc.). Markers are referenced according to the qualification rules applicable in COBOL.
- Tracing of the program at statement level:

  Dynamic tracing of the program is controllable via statement classification (e.g. procedure trace, control flow trace, assignment trace...). AID outputs the source reference of executing statements that correspond to the statement classification.

Documentation

“AID - Core Manual” [20]
“AID - Debugging of COBOL Programs” [8]
“AID - Debugging on Machine Code Level” [21].

15.2 Library Maintenance System (LMS)

Product characteristics

The library maintenance system LMS can be used to create and maintain program libraries and to process the elements contained in them.

Program libraries are BS2000 PAM files that are processed with the program library access method PLAM (and hence also referred to as PLAM libraries).

The main advantages of LMS are as follows:

- All element types in a library can be processed by means of uniform statements.
- Elements may have identical names and be differentiated by type or version.
- Versions are incremented automatically.
- The library can be accessed simultaneously by many users and also be written to concurrently.
- Different access rights can be assigned for each element.
- Access to elements can be monitored.
- LMS provides uniform data management and common access functions for most of the data elements involved in the development of software.
- Utility routines and compilers can access the data repository and also process individual elements directly.

LMS thus provides vital support in the creation, maintenance, and documentation of programs.

Structure of libraries

A program library is a file with a substructure. It contains elements as well as a directory of the elements stored in it.

An element is a logically related data set, e.g. a file, a procedure, an object module, or a compilation unit. Each element in the library can be addressed independently. Each library has an entry in the system catalog. The user can define the library name and other file attributes such as retention periods and shareability.

The collective storage of several files in a library reduces the load on the system catalog, since only the library is entered there and not each element. It also saves storage space, since library elements are stored in compressed form.

Support for multiple versions

If the delta technique is used for multiple versions of an element, only the differences (deltas) with respect to the previous version are stored. This also helps to save storage space.

When such delta versions are read, LMS inserts the required deltas at the appropriate positions and thus provides the user with a complete element.

LMS supports symbolic version identifiers and automatically increases the version ID in accordance with the selected version format.

Embedding in the program environment

Utility routines of the programming environment such as EDT, compilers, etc., can directly access program libraries.

Documentation

“LMS” user guide [11]

15.3 Job variables

Product characteristics

Job variables are data objects that are used for the exchange of information between individual users, and between the operating system and users.

Job variables can be created and modified by the user. The user can also instruct the operating system to set specific job variables to predefined values when certain events occur.

Job variables represent a flexible tool for job control under user supervision. They offer the option of defining the interrelationships in a complex production run in simple terms and form the basis for event-driven job processing.

Description of functions

Job variables are objects that are managed by the operating system. They can be addressed via names and can each hold up to 256 bytes of data. They are used for the exchange of information between individual users, as well as between the operating system and users. Job variables can be accessed via the command and macro interfaces. When the SDF component of BS2000-BC is used, job variables can serve as global parameters on the command level.

In conditional statements, job variables can be linked via Boolean operations. In this way, actions can be made dependent on the truth value of the condition. In addition, user job variables and monitoring job variables (see below) offer the option of synchronous or asynchronous event control at command and program level.

Different job variables are available for different functions:

- User job variables

  In their most general form, job variables are available as user job variables. The name, life, and data to be stored in such a variable is determined exclusively by the user. These job variables can be supplied with protection attributes such as passwords, writeprotection, and retention period. Access to a user job variable can be restricted to a particular user ID or be universally granted.

  User job variables are particularly suitable for the exchange of information. However, they can also be used for job control.

- Monitoring job variables

  The monitoring job variable is a special form of the user job variable. It is assigned to a job or a program. The name, lifespan, and protection attributes are defined by the user. However, in contrast to the user job variable, these variables are supplied with predetermined values by the operating system and reflect the status of the assigned job or program.

  Monitoring job variables are particularly suitable for job control, such as is required, for example, for managing interdependencies in production runs.

Documentation

“Job Variables” manual [7]

15.4 Database interface ESQL-COBOL

Product characteristics

ESQL-COBOL (BS2000/OSD) V3.0 implements the “embedded SQL” application program interface to the SESAM /SQL Server V5.0 database management system for COBOL applications in BS2000/OSD.

ESQL-COBOL (BS2000/OSD) V3.0 allows unrestricted use of the extended SQL functionality of SESAM/SQL Server V5.0

ESQL-COBOL (BS2000/OSD) V3.0 is required purely as an SQL precompiler for program development. The SQL runtime system is a component of SESAM/SQL Server.

The SQL runtime system is always required in order to use ESQL-COBOL (BS2000/OSD) V3.0. SESAM/SQL Server V5.0 is consequently required to precompile embedded SQL-COBOL programs even if precompilation takes place with no check against the database schemata.

Scope of SQL functions

- searches for data records (SELECT statement) including higher functions such as join, arithmetic, aggregate functions (e.g. averaging)
- adding, modifying, deleting records.

SESAM/SQL enables data to be manipulated and functions for administering databases to be executed (see the SESAM/SQL-Server (BS2000/OSD) manual [17].

Technical notes

The SQL statements of an ESQL-COBOL program are embedded in the COBOL code and are replaced with COBOL source code by a precompiler. The output from the precompiler is regular COBOL source code that has to be compiled with the COBOL2000 compiler. In addition, the precompiler extracts the SQL statements and transforms them into “SQL objects”.

The compiled COBOL program is linked with the SQL objects, the COBOL and DBMS runtime modules, and with a runtime system for the SQL objects; the result is an executable program.

Documentation

SQL/ESQL manuals [16] - [18]

15.5 Universal Transaction Monitor openUTM

openUTM simplifies the task of developing and running transaction applications.

A standardized programming interface (KDCS, DIN 66265) that is supported by most programming languages is available for program creation.

Format-driven input/output is supported for terminals in conjunction with the FHS formatting system.

openUTM guarantees that a transaction is executed either in its entirety, with all data updates, or not at all. It also ensures consistency of user data in combination with UDS/SQL, SESAM/SQL, LEASY and PRISMA.

openUTM offers restart functions in the event of application abortion, power failure/disruption or screen malfunctions. openUTM supports both interactive (dialog) and asynchronous processing, with the option of determining the start time of the programs.

Control facilities for distributing resources (tasks) are also offered.

Secure print processing is offered by virtue of built-in control functions for print outputs to remote printers.

Inquiry-and-transaction processing means that a large number of terminals can work with openUTM applications.

Accounting requirements are catered for by an accounting procedure specially tailored to inquiry-and-transaction processing.

openUTM offers comprehensive data protection mechanisms for access to applications and for selection of subfunctions of an application.

openUTM serves as a basis for a number of other software products.

Documentation

openUTM manuals [23] - [29]

15.6 Net Express® development environment with the BS2000/OSD option

Micro Focus, a long-standing partner, offers Net Express with the BS2000/OSD option. This is a development environment that runs on Windows systems and allows you to develop BS2000 COBOL applications.

With its BS2000/OSD option, the Net Express development environment offers all the functions required to develop BS2000 applications rapidly and efficiently. In addition to pure batch and openUTM applications with access to the LEASY, SESAM/SQL and UDS/SQL data storage systems, it is also possible to develop client/server applications and test them on a PC.

Batch and interactive applications that use the openUTM transaction monitor can be developed and tested on the Windows workstation using Net Express before being put into productive operation on a BS2000 platform. Transferring development activities to the PC in this way results in decisive improvements in terms of productivity and the quality of the software.

Client/server applications that use a BS2000/OSD server can be developed and tested under Net Express using the BS2000/OSD option. Net Express provides a uniform development environment for both the client and server components of the application as well as a runtime environment for productive deployment on the client and permits the client and server to be tested on a single platform with the help of the BS2000/OSD option.

Integrated development environment

Net Express offers a highly efficient development environment, and together with its BS2000/OSD option it represents a complete suite of tools and wizards for application development. A wizard guides the user right from the outset when creating a project, thus allowing BS2000-specific options to be generated automatically.

By using project management functions, it is possible to maintain even very large applications with ease. The various generation steps are defined once and can then be started with a simple mouse click using the Rebuild function. Designed for COBOL programmers, the editor also makes it easier to edit source code. The control functions for managing the source files permit people to work together in teams without one programmer overwriting the changes made by another.

Modern COBOL compiler

Net Express contains a modern compiler based on the proven strenths of COBOL and offers the following key features for the development of BS2000 applications:

- compatibility with BS2000’s COBOL2000 compiler can be set by means of a directive. All constructs that do not meet requirements are marked as incompatible.
- full support of the BS2000 EBCDIC code with the BS2000/OSD option
- support for object-oriented COBOL development and debugging
- functions for simulating the BS2000 system environment:

  For development purposes on the PC the Net Express BS2000/OSD option simulates specific functions of the BS2000 runtime environment. These include user and task switches, whose settings can be stored in files so that they are available to the applications at runtime. A tool sets job variables in the Net Express development environment and makes them available for the application.

openUTM simulation

The openUTM simulation of the BS2000/OSD options is based on the definitions for the KDCDEF utility in BS2000. In this way, an openUTM application is described with all its parameters such as transactions and subprograms. The KDCCECK tool allows KDCDEF files designed or edited on the PC to be checked for syntax. The BS2000 offloading wizard sets all the parameters required for the compilation and the linkage run of an openUTM

application and automatically ensures the application starts correctly in testing. The KDCS interface is supported for communication between the application programs and openUTM. Its parameters can be displayed and modified interactively at runtime of the application by means of the TRACE function.

In this way, the application logic and the associated transaction limits, the links of the various subprograms via openUTM and the control of mask output are visualized and rapid error localization enabled. The FHS formatting system is simulated to support formatted dialogs. IFG format libraries can be unloaded on the host and transferred to the PC.

There they can be edited using a special mask editor (SMSEDX) and then transferred to the host. The mask library is accessed at runtime of the application on the PC. The corresponding mask is formatted in accordance with the rules of the BS2000 formatting system, and color settings can be configured.

The keyboard inputs and screen outputs of formatted openUTM dialogs can be logged with the dialog test recorder. These recordings can be used for the automatic repetition of a logged test run. A special viewer (DTRVIEW) allows the logged test results to be compared and permits rapid error analysis if there are differences.

Simulation of the openUTM client

For the purpose of developing client/server applications with an openUTM application as the server, the Net Express BS2000/OSD option contains a simulation of the openUTM client. The client can thus also be developed and tested with Net Express. During testing, the client application can communicate with a server application running under the openUTM simulation of the BS2000/OSD option. In this way, the interaction between the two parts of the application can be tested on a single platform. Both parts of the application can be animated simultaneously, and the server can be monitored with the openUTM TRACE. For communication with a “real” openUTM application, the corresponding software is required.

Simulation of the LEASY, SESAM/SQL and UDS/SQL data storage systems

The Net Express BS2000/OSD option permits simulation modules to be selected for the BS2000 data storage systems LEASY, SESAM/SQL and UDS/SQL at installation. All DB simulations contain different utilities that are added to the integrated development environment of Net Express at installation. They enable structural information and test data to be taken from the host’s databases. The DB simulation modules allow BS2000 aplications that use these database systems to be fully maintained and further developed or implemented anew, regardless of whether they are batch or interactive applications. When a project like this is created, the BS2000 offloading wizard generates the corresponding database-specific settings for the compiler and the linkage editor.

The database simulation modules perform the database accesses on the PC and behave at the interface to the COBOL program completely analogously to the original database on the BS2000. This applies, in particular, to the returned parameters, such as the database status, thus allowing error situations to be tested in the simulation as well. In openUTM applications, the transactions of the databases are also coordinated with openUTM transactions in the simulation. Consequently, this interaction can be tested with the BS2000/OSD option as well.

TRACE function for database accesses

All database simulations are equipped with a convenient TRACE function with a graphical user interface. At the interface between the COBOL program and the database, all the parameters that are passed can be displayed and modified interactively. They can be displayed both before and after the call, thus allowing the effects of each action to be analyzed immediately. Help functions explain the possible causes of database or access errors.

The TRACES allow you to switch between different forms of representation for the database accesses. Whereas the current parameter contents of the CALL interface can be displayed for individual accesses to the database, the current history of database accesses can be followed in a table at any time. This type of representation is also possible for UDS/SQL applications that use the COBOL DML.

In addition, the TRACE function for UDS/SQL, which displays all currency tables, provides a detailed impression of the internal connections within the UDS/SQL database, which are of decisive importance for programming and error analysis.

16 Messages of the COBOL2000 system

The COBOL2000 compiler and the COBOL2000 runtime system comprehensively log all errors that occur during compilation and execution of a COBOL program. The messages which are output when errors are encountered can be divided into two groups:

1. Messages which refer to errors in the COBOL compilation unit: These are output in a diagnostic listing and/or an error file by the compiler at the end of compilation and have the following structure:

```
Msg-Index      Source Seq. No      Severity Code      Error Text
```

where:

Msg-Index	designates a 5-digit (hexadecimal) error message number (the first two characters indicate the compiler module which detected the error)
Source Seq. No	is the sequence number of the source line (in the compilation unit) containing the error
Severity Code	is the error class of the error, and
Error Text	is the text of the error message, which contains a more detailed description of the error and possibly a recovery measure.

Message texts can be output in English or German; the language can be selected via the SDF command MODIFY-MSG-ATTRIBUTES TASK-LANGUAGE=E/D.

A current list of all error messages of the COBOL2000 compiler can be requested with COMOPT PRINT-DIAGNOSTIC-MESSAGES=YES or with the SDF option COMPILER-ACTION=PRINT-MESSAGE-LIST (see section “Table of COMOPT operands” and section “COMPILER-ACTION option”).

Attention

Errors for which the message text begins with SE-1 or S.E. must always to reported to the system administrator /supervisor.

Severity code	Meaning	
F	Information	The compiler has identified language elements in the compilation unit which • represent an extension to the COBOL standard ANS85, • will not be supported by future COBOL standards, • as per FIPS (Federal Information Processing Standard), must be assigned to a particular language set. COBOL2000 issues severity code F messages only if they are explicitly requested with COMOPT ACTIVATE-WARNING-MECHANISM=YES or ACTIVATE-FLAGGING=ALL-FEATURES (SDF).

I	Information	The compiler has identified control statements or COBOL language elements that should be brought to the user's attention but do not justify issuing a warning or diagnostic message.
0	Warning	There may be an error in the compilation unit, but the program can still be executed.
1	Diagnostic	The compiler has detected an error; it will normally assume a corrective option. The program may be executed for test purposes.
2	Unrecoverable	Normally the compiler will not assume a corrective erroroption; the erroneous statement will not be generated.
3	System error	The error is so severe that the compiler is incapable of continuing the compilation.

Table 37: Severity codes and their meaning

2. The following messages:
   - Messages generated by the compiler on the execution and termination of the compilation run (CBL90nn)
   - Messages generated by the COBOL2000 runtime system on the execution and termination of the user program (COB91nn)
   - Messages of the POSIX driver for COBOL (CBL92nn)
   - Messages generated when writing object-oriented programs (COB93nn)

   These messages are output to SYSOUT during compilation or program execution and have the following structure:

   ```
   { CBL90nn | COB91nn | CBL92nn | COB93nn } Text
   ```

   where:

   CBL90nn
   COB91nn
   CBL92nn
   COB93nn
   : is the message identification number

   Text
   : is the text of the message. It contains
     - a note on the execution of the compiler or user program run or
     - a more detailed description of the error that has occurred and
     - in some cases, the request for user input which would make the error recoverable.

     Error messages can be output either in English or in German; the language can be selected via the SDF command MODIFY-MSG-ATTRIBUTES TASK-LANGUAGE=E/D.

   The program name specified in “COMPILATION UNIT IS program-name” in the messages COB9101 and COB9102 always identifies a separately compiled program. This may be an individual program or the outermost containing program of a nested program.

   Specifying the COMOPT operand GENERATE-LINE-NUMBER=YES or ERR-MSG-WITH-LINE-NR=YES in the SDF option RUNTIME-OPTIONS causes message COB9102 to be output with each program message instead

of COB9101. The COB9102 message also contains the number of the source line being executed when the message was issued.

Messages CBL9004, CBL9017, CBL9095, CBL9097 and CBL9099 (which are output during compilation) are suppressed if job switch 4 is set before the compiler is invoked.

The HELP-MSG-INFORMATION command enables you to display the complete message text for a message. In particular it contains the following information:

- Type (of message)
- Meaning (of variables in the message)
- Action (of the program)
- If required, response (of the user)

17 Appendix

- Structure of the COBOL2000 system
- Database operation (UDS/SQL)
- Description of listings

17.1 Structure of the COBOL2000 system

The COBOL2000 system consists of compiler modules and runtime modules. The structure of the compiler and the names of the modules are described below in detail. The runtime modules for COBOL2000 are included in the Common Runtime Environment (CRTE). Their names and individual functions are described in the “CRTE User Guide” [2].

17.1.1 Structure of the COBOL2000 compiler

The COBOL2000 compiler consists of a number of modules that are linked in linear sequence.

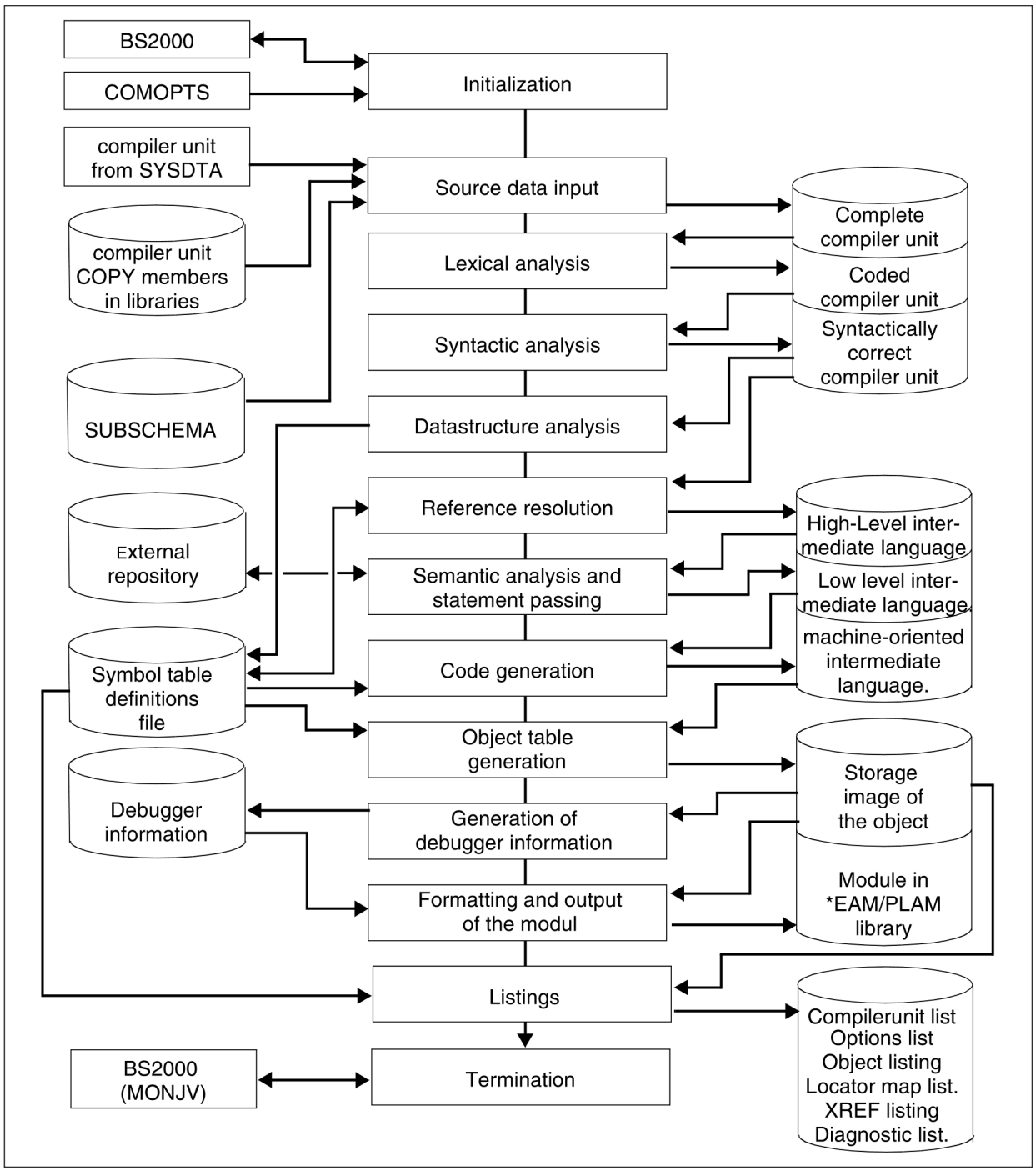
The individual modules constitute functional units that have been formed by a COBOL compilation run and by the structuring of the COBOL program into different Divisions.

The compilation process is divided into the following functional units:

1. Initialization
2. Source data input
3. Lexical analysis
4. Syntactic analysis
5. Semantic analysis
6. Code generation
7. Assembly run
8. Module generation
9. Report generation

The structure of the compiler and the arrangement of the individual function units in working storage are presented in the following diagram.

Structure of the compiler



17.1.2 The COBOL2000 runtime system

The COBOL2000 runtime system is a component of the **C**ommon **R**un**T**ime **E**nvironment (CRTE) for COBOL2000 and C/C++ programs.

CRTE is described in a separate “CRTE User Guide” [2].

The COBOL2000 runtime routines are subprograms that are known to the COBOL2000 compiler. They can basically be divided into three groups:

1.Subprograms for complex COBOL statements

Examples of complex COBOL statements are given in “COBOL2000 Reference Manual” [1] (e.g. SEARCH ALL...). However, even seemingly simple functions (e.g. COMPUTE A = B ** C), for which no corresponding machine instructions exist, are broken down by forming subprograms which are then stored externally in precompiled modules.

2.Subprograms for connecting the generated module to operating system functions

The main purpose of these subprograms is to ensure that the code generation of the compiler is totally independent of the operating system. The resulting loss in efficiency is largely offset by greater operating system independence. If interfaces to the operating system are changed, it is generally sufficient to relink the existing modules with the new runtime system.

Essential functions in this context are:

- Connection of COBOL programs to the input-output system
- Connection of COBOL programs to SORT
- Connection of COBOL programs to UDS/SQL
- Connection of COBOL programs to Executive functions

The following table contains a list of the names and functions of the COBOL2000 runtime modules. The table does not include those runtime modules that must be present in the COBOL2000 runtime system for compatibility reasons only, nor those modules that are used for access to the POSIX file system.

Name	Function
ITCMADPT *)	Adapter module for the partial bind technique
ITCMAID1 *)	AID connection module (Data Division)
ITCMECE1 *)	ENTRY, CANCEL, EXIT work area
ITCMECE2 *)	Table for COMOPT OPTIMIZE-CALL-IDENTIFIER and SDF CALL-IDENTIFIER =OPTIMIZE
ITCMERF1 *)	Error analysis routine for input/output
ITCMINIT *)	ILCS initialization
ITCMMAT1 *)	Data for math (IML...-) functions
ITCMMDP0 *)	OCCURS DEPENDING (recursive)
ITCMMEM1	ALLOCATE and FREE work area
ITCMOBAS *)	OO: BASE CLASS

ITCMOWK0 *)	OO: work area for OO runtime routines
ITCMPOVH *)	COBOL2000 program manager
ITCMSMG0 *)	SORT/MERGE statement
ITCMSTBO *)	Table sort
ITCMTOM0 *)	TOM adapter routine
ITCMXCAB *)	XML: callback adapter for basic functions
ITCMXCAS *)	XML: callback adapter for XML PARSE statement
ITCMXDT1 *)	Data module for exception handling
ITCMXHC2 *)	XHCS conversion work area
ITCMXMD1 *)	XML: data area for XML PARSE special registers
ITCMXWK0 *)	XML: data area for XML runtime routines
ITCRACA0	ACCEPT statement
ITCRACX0	ACCEPT statement for environment variable/command line
ITCRAID2	AID connection module (Procedure Division)
ITCRBCT0	Binary constant table
ITCRBEG0	Program system initialization routine
ITCRCCL1	CLOSE for INITIAL / CANCEL
ITCRCHP0	RERUN clause with integer RECORDS phrase
ITCRCHP2	RERUN clause for SORT files and END OF REEL
ITCRCLA0	Compare ALL literal
ITCRCLI0	CLOSE statement for indexed files
ITCRCLL0	CLOSE statement for line-sequential files
ITCRCLR0	CLOSE statement for relative files
ITCRCLS0	CLOSE statement for sequential files
ITCRCLX0	CLOSE statement for XML files
ITCRCMD0	Execution of a BS2000 command
ITCRCNA0	Comparison national figurative constant
ITCRCVB0	Conversion of packed decimal to binary (10 to 18 digits)
ITCRCVD0	Conversion of binary to packed decimal (10 to 18 digits)
ITCRCVF0	Conversion to and from floating point

ITCRCVL0	Conversion of packed + decimal to/from binary (>18 digits)
ITCRDFE0	Division external floating point
ITCRDPL0	Division of decimal numbers > 15 positions
ITCRDSA0	DISPLAY statement
ITCRDSI1	Storage assignment DYNAMIC data
ITCRDSX0	DISPLAY statement for environment variable
ITCRDYF1	Reservation of storage space for the functions REVERSE, UPPER-CASE, and LOWER-CASE
ITCRECE0	ENTRY, CANCEL, EXIT for separately compiled programs
ITCREND0	Program termination routine (normal and abnormal)
ITCREPL2	Output line with line-feed control character
ITCREV0	Event handling (return from subroutine of another language)
ITCREV1	Event handling (“recoverable interrupts”)
ITCREV2	Event handling (“unrecoverable interrupts”)
ITCREV3	Event handling (remaining events)
ITCRFAT0	Table for FACTORIAL function
ITCRFCH0	Message output for function argument check
ITCRFCT1	Floating point constants
ITCRFDT0	Date conversion functions
ITCRFMD0	MEDIAN function
ITCRFMX0	Functions: MAX, MIN, ORD-MAX, ORD-MIN, RANGE, MIDRANGE
ITCRFNM0	Functions: NUMVAL, NUMVAL-C
ITCRFPV0	PRESENT-VALUE function
ITCRFRN0	RANDOM function
ITCRFST0	Functions: REVERSE, UPPER-CASE, LOWER-CASE, DISPLAY-OF, NATIONAL-OF
ITCRFVR0	VARIANCE function
ITCRHSW0	Set and check job/user switches
ITCRIFA0	FCB initialization; control routine
ITCRIFC1	RERUN clause FCB generation
ITCRIFI1	ISAM FCB generation for indexed files
ITCRIFL1	SAM FCB generation for line-sequential files

ITCRIFR1	ISAM FCB generation for relative files
ITCRIFS1	SAM FCB generation for sequential files
ITCRIFX1	FCB generation for XML files
ITCRINI0	INITIALIZE statement
ITCRINS0	INSPECT statement
ITCRLHS2	User label handling for sequential files
ITCRLNL1	LINKAGE clause with WRITE for line-sequential files
ITCRLNS1	LINKAGE clause with WRITE for record-sequential files
ITCRMAT0	Connection module for math (IML...-) functions
ITCRMEM0	ALLOCATE and FREE statement
ITCRMEV2	Interrupt message for event handling routine
ITCRMPL0	Multiplication of decimal numbers > 15 positions
ITCRMSG0	Output of error messages, level 0
ITCRMSG3	Output of error messages
ITCRMVE0	MOVE for numeric-edited items
ITCRNED0	De-editing MOVE
ITCRNSP0	CALL, CANCEL, ENTRY, EXIT in nested program
ITCROCA0	Check match between current / formal methods parameters
ITCROFP2	OO:Formal parameter description
ITCROIF1	OO: Initialization routine for files in objects
ITCROIS0	OO: Initialization routine for classes / interfaces
ITCROMD0	OO: Check routine for object view
ITCROMS1	OO: Output of OO error messages
ITCRONW1	OO: Create and initialize new object
ITCROOI0	Control routine for object-oriented initialization and termination
ITCROPI0	OPEN statement for indexed files
ITCROPL0	OPEN statement for line-sequential files
ITCROPR0	OPEN statement for relative files
ITCROPS0	OPEN statement for sequential files
ITCROPX0	OPEN statement for XML files

ITCROSM0	OO: Select method
ITCROTC2	OO: Conformance test
ITCROVC1	OO: Check interface conformance
ITCROVI2	OO: Check class inheritance
ITCRPAM1	Physical read/write routine for relative files (PAM)
ITCRPBND	Partial-bind large module
ITCRPCA0	Comparisons under PROGRAM COLLATING SEQUENCE
ITCRPCS0	Comparisons under PROGRAM COLLATING SEQUENCE
ITCRRCH0	Check table limits
ITCRRDI0	READ/START statement for indexed files
ITCRRDL0	READ/START statement for line-sequential files
ITCRRDR0	READ/START statement for relative files
ITCRRDS0	READ statement for sequential files
ITCRRDX0	READ/START statement for XML files
ITCRRPW0	REPORT-WRITER control module
ITCRSCH0	SEARCH ALL statement
ITCRSEG0	Activation of segmented COBOL programs
ITCRST11	CODE SET table for ASCII
ITCRST21	CODE SET table for ISO-7
ITCRSTG0	STRING statement
ITCRSTP0	STOP literal statement
ITCRTCA1	Class test table for ALPHABETIC test
ITCRTCD1	Class test table for NUMERIC (COMP-3 with sign) test
ITCRTCE1	Class test table for NUMERIC (COMP-3 without sign) test
ITCRTCL1	Class test table for ALPHABETIC-LOWER test
ITCRTCN0	Class test for national operands
ITCRTCP1	Class test table for ALPHABETIC-UPPER test
ITCRTCS1	Class test table for NUMERIC test (with sign)
ITCRTCU1	Class test table for NUMERIC test
ITCRTCV0	Class test for data items > 256 bytes or variables

ITCRTOD3	Time of day / Date (SVC-free)
ITCRUDS0	DML link to database handler
ITCRUPC0	Processing declaratives
ITCRUPC1	Processing declaratives
ITCRUPC2	Processing declaratives
ITCRUST0	UNSTRING statement
ITCRVCL0	Comparison for items of variable length/address or > 256 bytes
ITCRVCN0	Comparison for national items of variable length/address or > 256 bytes
ITCRVMA0	MOVE ALL literal
ITCRVMN0	MOVE for national items of variable length/address or > 256 bytes
ITCRVMP0	Padding for items > 256 bytes in case of MOVE
ITCRVMV0	MOVE for items of variable length/address or > 256 bytes
ITCRWRI0	WRITE/REWRITE statement for indexed files
ITCRWRL0	WRITE/REWRITE statement for line sequential files
ITCRWRR0	WRITE/REWRITE statement for relative files
ITCRWRS0	WRITE statement for sequential files
ITCRXBND	XML: parser dynamic loading routing
ITCRXCFB	XML: callback functions for basic functionality
ITCRXCFS	XML: callback functions for XML PARSE statement
ITCRXDP1	XML: document tree processing
ITCRXFS0	Extended file status
ITCRXGE0	XML GENERATE statement
ITCRXHC0	XHCS conversions
ITCRXIT0	FILE STATUS and error handling routine
ITCRXLC0	Exception handling
ITCRXPA0	XML PARSE statement
ITCRXPC1	XML: document file processing
ITCRXPF0	Exponentiation (floating point)
ITCRXPI0	Exponentiation (integer)

*) Module not shareable

17.2 Database operation (UDS/SQL)

Not supported in COBOL2000-BC !

A description of the universal database management system UDS/SQL is given in the UDS/SQL manuals: “Design and Definition” [13], “Creation and Restructuring” [14], and “Application Programming” [15].

UDS/SQL databases are processed by user programs via

- COBOL-DML language elements (DML is an integral component of COBOL) and
- CALL DML (database handling via subprogram call).

The following text concentrates on COBOL-DML. Furthermore, it is assumed that schema and subschema have already been generated. The individual steps towards generating a UDS/SQL user program are briefly explained.

The Database Handler (DBH), the main component of the UDS/SQL database system, is responsible for communication between the user program and the database (via the subschema). A distinction is made between:

- Linked-in DBH: It is linked into the user program and is therefore suitable for cases where only one user program is to work with the database.
- Independent DBH: This is not linked into the user program and is therefore capable of controlling more than one user program (independent task).

Structure of a COBOL-DML program

```
DATA DIVISION.
    .
    .
    .
SUB-SCHEMA SECTION.
    DB subschema-name WITHIN schema-name.
PROCEDURE DIVISION.
    .
    .
    Sequence of COBOL-DML statements
    ...   .
```

The formats of the COBOL-DML statements are described in “Application Programming” manual [15].

schema-name/subschema-name are defined at schema/subschema generation time.

Compiling a COBOL-DML program

The COBOL2000 compiler generates a program module and a subschema module from a COBOL-DML program. When the user program is compiled, the COBOL compiler must read the COSSD file of the database concerned. The following options are available here:

1. The COSSD file is assigned explicitly to the COBOL compiler using the command

```
/ADD-FILE-LINK       LINK-NAME=UDSCOSSD, -
/                    FILE-NAME=[:catid:][$userid.]dbname.COSSD
```

   Here `:catid:` and `$userid` are the catalog ID and the user ID under which the COSSD file is cataloged. If the `:catid:` or `$userid` specification is missing, the file name is complemented in accordance with the standard BS2000 rules.

   The COSSD file must be cataloged under the name specified in the command since in the event of an error no search is made for a COSSD file at a different location. This procedure is mandatory when multiple COSSD files with the corresponding database name exist in all catalogs which can be accessed locally from the user ID.

   Example of a command sequence:

```
/ADD-FILE-LINK UDSCOSSD,dbname.COSSD
/START-PROGRAM $COBOL2000
COMOPT MODULE=module-library
END compilation-unit-file
```

2. The COBOL compiler is notified of the database name using the following command.

```
/SET-FILE-LINK       LINK-NAME=DATABASE, -
/                    FILE-NAME=[:catid:][$userid.]dbname
```

   Any `:catid:` specified in the SET-FILE-LINK command is ignored. The COBOL compiler then searches for a COSSD file with the name `dbname.COSSD` in all catalogs which can be accessed locally from the user ID which was specified explicitly in the SET-FILE-LINK command or was complemented by BS2000. This procedure can be used only when just one COSSD file with the corresponding database name exists in the specified catalog environment.

   Example of a command sequence:

```
/SET-FILE-LINK DATABASE,dbname
/START-PROGRAM $COBOL2000
COMOPT MODULE=module-library
END compilation-unit-file
```

If ADD-FILE-LINK or SET-FILE-LINK commands exist for both LINK=DATABASE and for LINK=UDSCOSSD, only the procedure for LINK=UDSCOSSD is used.

Linking a COBOL-DML program

The link-editing of COBOL programs is described at length in the chapter “Generating and calling executable programs”.

Note that for COBOL-DML programs a suitable UDS/SQL connection module must be linked in, depending on the DBH version used (linked-in/independent). For further information see “Application Programming” manual [15].

Example of a linkage editor run:

```
/START-BINDER
//START-LLM-CREATION INT-NAME=programmname
//INCLUDE-MODULES LIB=module-library, ELEM=cobol-dml-programm
```

```
//INCLUDE-MODULES LIB=uds-module-library, ELEM=uds-connection-modul
//RESOLVE-BY-AUTOLINK LIB=$.SYSLNK.CRTE
//SAVE-LLM LIB=module-library, ELEM=uds-test-prog
//END
```

Execution of a UDS/SQL user program

When the independent DBH is used, execution of a UDS/SQL user program is possible only within a UDS/SQL session. The connection to this session or to the database is provided by the SET-FILE-LINK command (see the UDS/SQL (BS2000/OSD) manual [15]).

Execution with linked-in DBH:

```
/SET-FILE-LINK DATABASE,dbname
/START-PROGRAM filename
[DBH parameters]
PP END
[user-program-parameters]
```

Execution with independent DBH:

```
/SET-FILE-LINK DATABASE,dbname
/START-PROGRAM filename
[user-program-parameters]
```

17.3 Description of listings

In this section, the formats of the following listings output by COBOL2000 during compilation are explained briefly, using a programming example as the basis for the description:

- Control statement listing
- Source listing
- Locator map/ cross-reference listing
- Diagnostic listing

To save space, the blanks after the last printed character are removed from the individual data records of a list.

17.3.1 Header line

Each page of a listing starts with a header line (see below), which, regardless of the type of listing, contains the following information:

(1) Name and version of the compiler

(2) PROGRAM-ID nameType of listing Time of compilation Date of compilation Page number

(3) Type of listing

(4) Time of compilation

(5) Date of compilation

(6) Page number

(1)	(2)	(3)	(4)	(5)	(6)
COBOL2000 V01.4A02	COPYING	LIBRARY LISTING	09:58:34	2006-08-04	PAGE 0003

17.3.2 Control statement listing

In this listing, COBOL2000 logs

(1) the environment of the compilation run,

(2) the selected compiler options (OPTIONS IN EFFECT; COMOPTs),

(3) the compiler options (COMOPTs) set by default (OPTIONS BY DEFAULT) at compilation time, and

(4) information for maintenance and diagnostic purposes.

```
COBOL2000  V01.5A00  COPYING                    COMOPT LISTING              09:58:34 2009-02-12  PAGE 0001


ENVIRONMENT           (1)
-----------
                         PROCESSOR                     : 7.500- S170-30
                         OPERATING SYSTEM              : BS2000 V16.0
                         COMPILER                      : COBOL2000  V01.5A00
                         TASK-SEQUENCE NUMBER          : 1k59
                         USER-ID                       : CAC21
                         Copyright (C) Fujitsu Technology Solutions 2009
                                       All Rights Reserved


OPTIONS IN EFFECT     (2)
-----------------
                         MODULE                        = COB
                         SYSLIST                       = (OPTIONS,DIAG,MAP,SOURCE,XREF)
                         GENERATE-LLM                  = YES
                         SOURCE-ELEMENT                = KOPIEREN.COB
                         MERGE-REFERENCES              = YES
                         MERGE-DIAGNOSTICS             = YES
                         ENABLE-XML-PROCESSING         = NO


OPTIONS BY DEFAULT    (3)
------------------
                         CHECK-DATE                    = YES
                         EXPAND-COPY                   = YES
                         LINE-LENGTH                   = 132
                         ALIGN-LLM-PAGE                = YES
                         LINES-PER-PAGE                = 064
                         MODULE-ELEMENT                = *STD
                         MODULE-VERSION                = *UPPER-LIMIT
                         SOURCE-VERSION                = *HIGHEST-EXISTING
                         EXPAND-SUBSCHEMA              = YES
                         MINIMAL-SEVERITY              = I
                         REPLACE-PSEUDOTEXT            = YES
                         RESET-PERFORM-EXITS           = YES
                         CONTINUE-AFTER-MESSAGE        = YES
                         GENERATE-INITIAL-STATE        = YES
                         DEFAULT-CALL-CONVENTION       = COMPATIBLE
                         INHIBIT-BAD-SIGN-PROPAGATION  = YES


FOR CUSTOMER SERVICE  (4)
--------------------
REV# = A
REV# = B
```

17.3.3 Source listing for a compilation unit

Each line of a source listing is subdivided into the following areas:

(1) Indicator field

Column 1 gives information about errors in the user-defined numbering of the input records (S flag) and about any violations of the maximum line length of 80 characters in fixed format resp. 248 characters in free format (T flag). Furthermore, it identifies records copied from a COPY library (C flag), declared by a REPLACING or REPLACE (R flag) or associated with the Sub-schema Section (D flag). Column 3 shows the nesting depth for expanded COPY elements.

A minus sign (-) in column 1 identifies lines that have been ignored because of compiler directives.

(2) Sequence number field

Contains a number (max. 5 digits) assigned by COBOL2000 to identify the input compiler unit record. This number uniquely identifies the source code lines. It appears in all the listings generated by COBOL2000 as a cross-reference number. It is also used for reference in any error messages. Its maximum value is 65535. If a compilation unit exceeds this number, consecutive numbering starts again from 0.

(3) At the beginning of each new page of a source listing, a line containing column markers (V) is generated after the heading. These markers conform to the COBOL reference format and make it easier for the user to recognize any violations of the column format required by COBOL.

(4) Area that can be used by programmers to mark source lines

(5) Compilation unit area

Contains the record entered by the user. Note that only printable characters are shown.

The following exist only in a “compressed” list (see the parameter SOURCE=YES(CROSS-REFERENCE=YES) in the section “LISTING option”).

(6) If a line contains more than one definition, or if there are implicit definitions in a compilation unit, these are shown in the compressed listing in additional lines containing only the right-aligned name of this definition instead of the source text.

(7) REL LOC

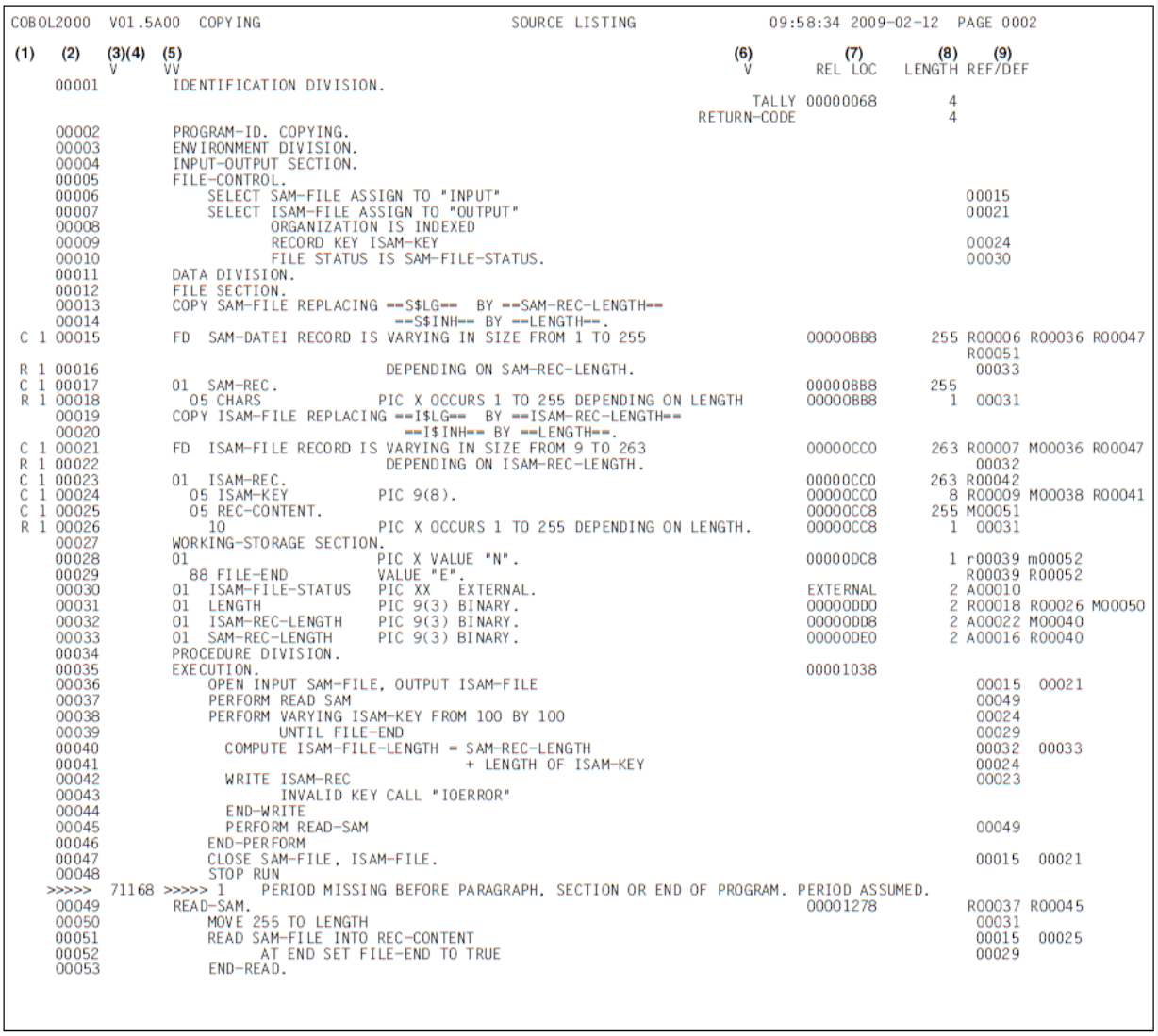
Contains the position of a data definition or of a chapter or paragraph name relative to the start of the module.

(8) LENGTH

Contains the (decimal) length of the area in the module which has been assigned to a data definition.

(9) REF/DEF
Contains the sequence numbers of the lines that refer to a definition, together with the type of the reference (see section “Locator map listing” for an explanation of this reference type). In the case of the referring line, it is the sequence number of the definition line. If there are more cross-references than fit into a line, continuation lines are formed (see the LINE-SIZE parameter in the section “LISTING option”).

The compilation messages are "merged" in the sample listing (see the INSERT-ERROR-MSG parameter in the section “LISTING option”).



```
COBOL2000  V01.5A00  COPYING                          SOURCE LISTING              09:58:34 2009-02-12  PAGE 0002

(1)   (2)   (3)(4)  (5)                                                          (6)          (7)        (8)    (9)
            V       VV                                                            V        REL LOC   LENGTH REF/DEF
     00001          IDENTIFICATION DIVISION.
                                                                                TALLY 00000068        4
                                                                          RETURN-CODE                 4
     00002          PROGRAM-ID. COPYING.
     00003          ENVIRONMENT DIVISION.
     00004          INPUT-OUTPUT SECTION.
     00005          FILE-CONTROL.
     00006              SELECT SAM-FILE ASSIGN TO "INPUT"                                                  00015
     00007              SELECT ISAM-FILE ASSIGN TO "OUTPUT"                                                00021
     00008                     ORGANIZATION IS INDEXED
     00009                     RECORD KEY ISAM-KEY                                                         00024
     00010                     FILE STATUS IS SAM-FILE-STATUS.                                             00030
     00011          DATA DIVISION.
     00012          FILE SECTION.
     00013          COPY SAM-FILE REPLACING ==S$LG==  BY ==SAM-REC-LENGTH==
     00014                                  ==S$INH== BY ==LENGTH==.
 C 1 00015          FD  SAM-DATEI RECORD IS VARYING IN SIZE FROM 1 TO 255                 00000BB8      255 R00006 R00036 R00047
                                                                                                             R00051
 R 1 00016                                  DEPENDING ON SAM-REC-LENGTH.                                      00033
 C 1 00017          01  SAM-REC.                                                          00000BB8      255
 R 1 00018            05 CHARS              PIC X OCCURS 1 TO 255 DEPENDING ON LENGTH      00000BB8        1  00031
     00019          COPY ISAM-FILE REPLACING ==I$LG==  BY ==ISAM-REC-LENGTH==
     00020                                   ==I$INH== BY ==LENGTH==.
 C 1 00021          FD  ISAM-FILE RECORD IS VARYING IN SIZE FROM 9 TO 263                 00000CC0      263 R00007 M00036 R00047
 R 1 00022                                  DEPENDING ON ISAM-REC-LENGTH.                                      00032
 C 1 00023          01  ISAM-REC.                                                         00000CC0      263 R00042
 C 1 00024            05 ISAM-KEY           PIC 9(8).                                      00000CC0        8 R00009 M00038 R00041
 C 1 00025            05 REC-CONTENT.                                                      00000CC8      255 M00051
 R 1 00026              10                  PIC X OCCURS 1 TO 255 DEPENDING ON LENGTH.     00000CC8        1  00031
     00027          WORKING-STORAGE SECTION.
     00028          01                        PIC X VALUE "N".                             00000DC8        1 r00039 m00052
     00029            88 FILE-END             VALUE "E".                                                      R00039 R00052
     00030          01  ISAM-FILE-STATUS      PIC XX   EXTERNAL.                           EXTERNAL        2 A00010
     00031          01  LENGTH                PIC 9(3) BINARY.                             00000DD0        2 R00018 R00026 M00050
     00032          01  ISAM-REC-LENGTH       PIC 9(3) BINARY.                             00000DD8        2 A00022 M00040
     00033          01  SAM-REC-LENGTH        PIC 9(3) BINARY.                             00000DE0        2 A00016 R00040
     00034          PROCEDURE DIVISION.
     00035          EXECUTION.                                                            00001038
     00036              OPEN INPUT SAM-FILE, OUTPUT ISAM-FILE                                                  00015  00021
     00037              PERFORM READ SAM                                                                       00049
     00038              PERFORM VARYING ISAM-KEY FROM 100 BY 100                                               00024
     00039                      UNTIL FILE-END                                                                 00029
     00040                COMPUTE ISAM-FILE-LENGTH = SAM-REC-LENGTH                                            00032  00033
     00041                                         + LENGTH OF ISAM-KEY                                        00024
     00042                WRITE ISAM-REC                                                                       00023
     00043                      INVALID KEY CALL "IOERROR"
     00044                END-WRITE
     00045                PERFORM READ-SAM                                                                     00049
     00046              END-PERFORM
     00047              CLOSE SAM-FILE, ISAM-FILE.                                                             00015  00021
     00048              STOP RUN
    >>>>>  71168 >>>>> 1    PERIOD MISSING BEFORE PARAGRAPH, SECTION OR END OF PROGRAM. PERIOD ASSUMED.
     00049          READ-SAM.                                                             00001278          R00037 R00045
     00050              MOVE 255 TO LENGTH                                                                     00031
     00051              READ SAM-FILE INTO REC-CONTENT                                                         00015  00025
     00052                    AT END SET FILE-END TO TRUE                                                      00029
     00053              END-READ.
```

A library listing is output as the second part of the source listing. The sources from which the COBOL program processed in this compilation run was created can be found in this library listing. A line is created for each COPY statement, containing the following information:

(10) Sequence number of the program line containing the COPY statement

(11) Link name from the COPY statement

(12) Library type

(13) Element (member) name

(14) Date

(15) Version number under which the library element is entered in the library. The date and version number are not always present.

(16) File name under which the library is entered in the file system

```
COBOL2000  V01.5A00  COPYING                          LIBRARY LISTING                09:58:34 2009-02-12 PAGE 0003

  (10)    (11)     (12)      (13)                        (14)        (15)                      (16)

 SOURCE LIBRARY- (LIB-) ELEMENT-NAME                    USER       VERSION                  FILE-NAME
 SEQ-NO NAME     ORG                                    DATE

 SOURCE         PLAM   COPYING.COB                   2004-07-29 ~                      :2OSC:$CAC21.Manualexample
 00013  COBLIB   PLAM   SAM-FILE                     2004-07-29 ~                      :2OSC:$CAC21.Manualexample
 00019  COBLIB   PLAM   ISAM-FILE                    2004-07-29 ~                      :2OSC:$CAC21.Manualexample
```

Special features of the source listing for free format

Significant differences between fixed and free format which are visible for the user affect the listing.

In the listing the free format line is broken to form partial lines, each of which is up to 80 characters long. This partial line length is derived from the old line length (sequence area + indicator area + program text area + comment area). Free format means that program text is involved over the entire partial line length.

```
...
...
     00079 *>       !         !         !         !         !         !         !         !
     00080
     00081          add 1 to a. add 1 to b. add 1 to c. add 1 to d. add 1 to e. add 1 to f.
     +      add 1 to g. add 1 to h. add 1 to i. add 1 to j. add 1 to k. add 1 to l. add 1 t
     +     o m. add 1 to n. add 1 to o. add 1 to p. add 1 to q. add 1 to r. add 1 to s. add
     +      1 to t.
     00082
     00083 *>       !         !         !         !         !         !         !         !
     00084
     00085
     +
     +                      add 1 to u. add 1 to v. add 1 to w. add 1 to x. add 1 to y. add
     +      1 to z.
     00086
     00087 *>       !         !         !         !         !         !         !         !
     00088
     ...
...
```

Lines which are not longer than 80 characters a listed as previously, making full use of the available space. A line which is 81 to 160 characters long is divided into a partial line with 80 characters and a smaller partial line containing 1 to 80 characters. The latter partial line is not assigned a preceding line number for identification purposes, but only a single “+” character.

By the same token, lines containing 161 to 240 characters are divided into three lines. Lines which are 241 characters long or more are divided into four partial lines. In this case the fourth partial line is at most 8 characters long as the maximum line length is limited to 248 characters.

In this form of the listing the partial lines are directly beneath each other.

If the compressed listing (see the section “LISTING option”) is activated, the same listing extract looks as follows:

```
...
...
     00079 *>       !         !         !         !         !         !         !         !
     00080
     00081          add 1 to a. add 1 to b. add 1 to c. add 1 to d. add 1 to e. add 1 to f.    00014  00015  00016
                                                                                              00017  00018  00019
     +      add 1 to g. add 1 to h. add 1 to i. add 1 to j. add 1 to k. add 1 to l. add 1 t    00020  00021  00022
                                                                                              00023  00024  00025
     +     o m. add 1 to n. add 1 to o. add 1 to p. add 1 to q. add 1 to r. add 1 to s. add    00026  00027  00028
                                                                                              00029  00030  00031
                                                                                              00032
     +      1 to t.                                                                          00033
     00082
     00083 *>       !         !         !         !         !         !         !         !
     00084
     00085
     +
     +                     add 1 to u. add 1 to v. add 1 to w. add 1 to x. add 1 to y. add    00034  00035  00036
                                                                                              00037  00038
     +      1 to z.                                                                          00039
     00086
     00087 *>       !         !         !         !         !         !         !         !
     00088
     ...
...
```

As the XREF outputs occasionally force the insertion of additional empty lines, it can only be seen from the “+” character at the beginning of the line whether such an inserted line has been generated owing to XREF entries or whether a genuine partial line is involved which by chance only contains blanks (see example line 00085).

The assignment of the XREF entries to the corresponding partial lines always relates to the first character of a file name.

> The line break of a free format COBOL line does not always result in a listing which is easy to read. It thus makes sense to achieve a more straightforward listing through a suitable grouping of the program text elements, for example by aligning the program text elements to specified tabulator positions (20, 40 or 80 characters). In many cases it is advisable to restrict the line length to 80 characters.

17.3.4 Format control statements TITLE, EJECT, SKIP

The COBOL2000 compiler supports the format control statements TITLE, EJECT and SKIP. These statements can be used in the compiler unit to control the format of the source listing. Format control statements are governed by the following rules:

- They must not be concluded with a period.
- They must be contained exclusive in a line from column 12.
- They themselves do not appear in the source listing.

TITLE statement

Function

This statement is used to print the specified title instead of the standard title (SOURCE LISTING) in the header lines of the source listing that follow. In addition, a page feed is performed unless a new page was to be started anyway.

Format

```
TITLE literal
```

Rule

literal must be a non-numeric literal of up to 53 characters.

EJECT statement

Function

This statement causes the following text of the source listing to begin at the top of the next page. This statement has no effect if a new page was to be started anyway.

Format

```
EJECT
```

SKIP statement

Function

The SKIP statement is used to shift the following text by up to three lines. The statement has no effect if blank lines would be the first to appear at the top of the next page.

Format

```
{SKIP1}
{SKIP2}
{SKIP3}
```

Example 17-1

Format control statements

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.
DATA DIVISION.
    TITLE "WORKING-STORAGE SECTION" ————————————(1)
WORKING-STORAGE SECTION.
```

```
01 ALPHA1 PIC 99 VALUE 1.
01 BETA1 PIC 99 VALUE 2.
01 GAMMA1 PIC 99.
    TITLE "PROCEDURE DIVISION"————————————————(2)
PROCEDURE DIVISION.
    EJECT ————————————————————————————————————(3)
BEGIN SECTION.
MULT.
    MULTIPLY ALPHA1 BY BETA1 GIVING GAMMA1.
    MULTIPLY BETA1 BY GAMMA1 GIVING ALPHA1.
    MULTIPLY GAMMA1 BY ALPHA1 GIVING BETA1.
    SKIP3 ————————————————————————————————————(4)
 END SECTION.
 STOPP.
     STOP RUN.
```

Effect:

(1) The header line of the next page of the source listing will read: “WORKING-STORAGE SECTION”

(2) The header line of the next page(s) of the source listing will read: “PROCEDURE DIVISION”.

(3) The following text to be printed (BEGIN SECTION...) will begin on a new page.

(4) The following text to be printed (END SECTION) will be preceded by three blank lines.

17.3.5 Diagnostic (error message) listing

The diagnostic listing generated by COBOL2000 provides information about all syntactical and semantic errors detected during the compilation.

The header line is followed by a subheading line which divides the listed diagnostic message lines into the following fields:

(1) SOURCE SEQ NO — indicates the sequence number of the source line in which the error occurred.

(2) MSG INDEX — indicates the message identifier.

(3) SEVERITY CODE — indicates the error class (see table 37 in section "Messages of the COBOL2000 system").

(4) ERROR MESSAGE — contains an explanatory text and, if applicable, the corrective action taken by COBOL2000 or a default value assumed by COBOL2000.

The diagnostic listing is concluded by summary information on the total number of all detected errors and the total number of errors in the various severity classes.

```
ACOBOL2000  V01.5A00  COPYING                        DIAGNOSTIC LISTING          09:58:34 2009-02-12  PAGE 0007

   (1)      (2)      (3)          (4)

   SOURCE  MSG   SEVERITY
   SEQ-NO  INDEX     CODE   ERROR MESSAGE

    00048  71168       1    PERIOD MISSING BEFORE PARAGRAPH, SECTION OR END OF PROGRAM. PERIOD ASSUMED.
    TOTAL 00001 STATEMENTS IN THIS DIAGNOSTIC LISTING.
          00001 IN SEVERITY CODE 1
```

17.3.6 Locator map listing

(1) Specification of program division, section and program name.

(2) File name, file sequence number and address of file control block of all files used in the program.

(3) SOURCE SEQ NO
indicates the sequence number of the source line containing the definition.

(4) MODULE REL ADDR
indicates the relative starting address of a data definition within the module.

(5) GROUP REL ADDR
identifies the relative starting address of a data definition within a 01-level entry (hexa-decimal).

(6) POSITION IN GROUP DEC
is the number of the first byte of a data definition within a 01-level entry (decimal, counting from 1).

(7) LEV NO
contains the level number of the definition. A “G” preceding the level number identifies an item of data as “global”.

(8) contains the data-name defined by the user.

(9) LENGTH IN BYTES
indicates, in decimal (DEC) and hexadecimal (HEX) notation, the length of the area to which the data name is assigned.

(10) FORMAT
indicates the symbolic name of the data class.

(11) REFERENCED BY STATEMENTS
lists all the source line numbers, in ascending order, in which there are statements that reference this data definition. If there are more cross-references than fit on a line, continuation lines are generated (see the LINE-SIZE parameter in the section “LISTING option”).

The type of reference appears in front of the line number:

M modify

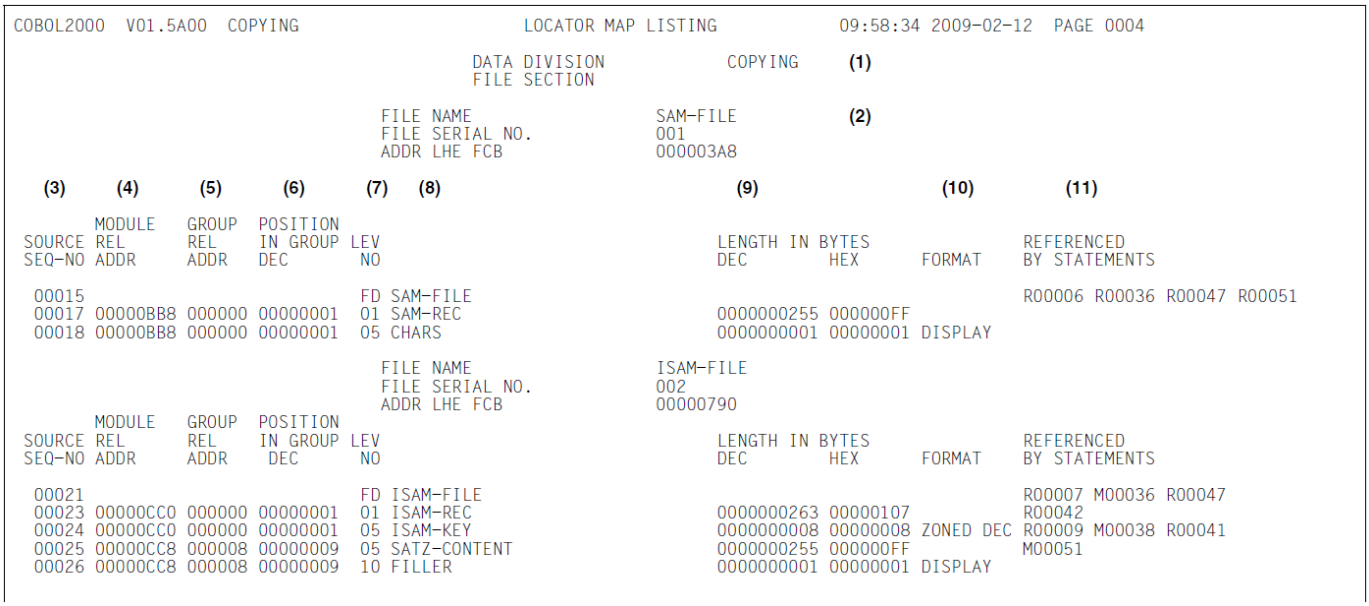
R read

A address

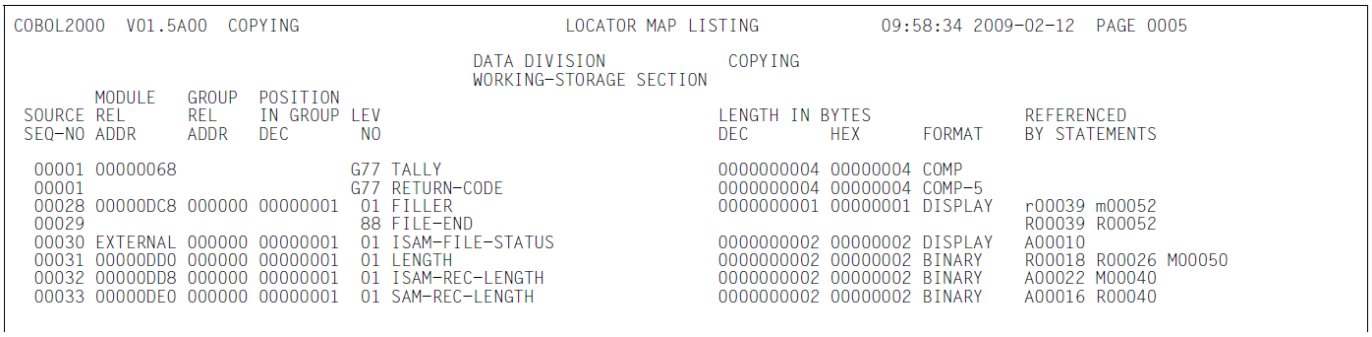
The corresponding lowercase letters indicate implicit accesses (this affects corresponding subordinate fields with MOVE CORRESPONDING, for example, or the data item referred to by a condition name).

(12) LVL
indicates the nesting level of the program, starting at 000 for the outside program.

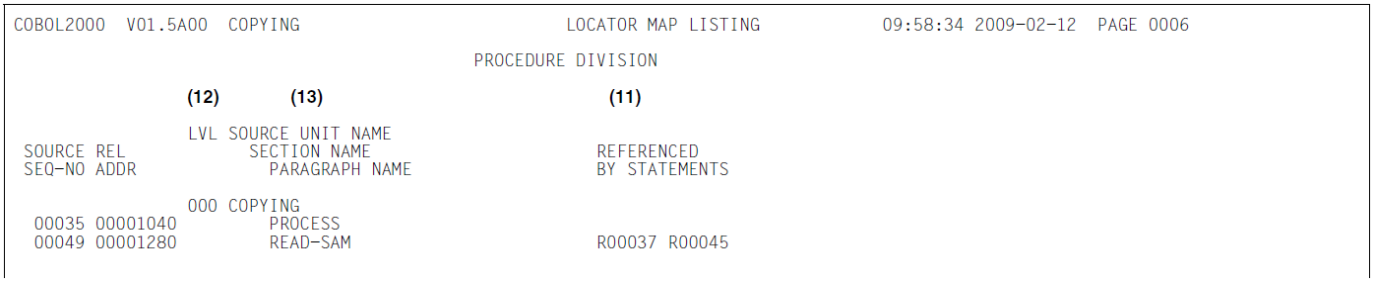
(13) SOURCE UNIT NAME / SECTION NAME / PARAGRAPH NAME
indicates the program name and the section and paragraph names occurring in this program.



```
COBOL2000  V01.5A00  COPYING                    LOCATOR MAP LISTING           09:58:34 2009-02-12  PAGE 0004

                                        DATA DIVISION            COPYING     (1)
                                        FILE SECTION

                                FILE NAME                 SAM-FILE           (2)
                                FILE SERIAL NO.           001
                                ADDR LHE FCB              000003A8

  (3)     (4)      (5)      (6)      (7)  (8)                          (9)                  (10)         (11)

        MODULE   GROUP  POSITION
 SOURCE REL      REL    IN GROUP LEV                                LENGTH IN BYTES              REFERENCED
 SEQ-NO ADDR     ADDR   DEC       NO                                DEC        HEX      FORMAT    BY STATEMENTS

  00015                          FD SAM-FILE                                                      R00006 R00036 R00047 R00051
  00017 00000BB8 000000 00000001  01 SAM-REC                        0000000255 000000FF
  00018 00000BB8 000000 00000001  05 CHARS                          0000000001 00000001 DISPLAY

                                FILE NAME                 ISAM-FILE
                                FILE SERIAL NO.           002
                                ADDR LHE FCB              00000790
        MODULE   GROUP  POSITION
 SOURCE REL      REL    IN GROUP LEV                                LENGTH IN BYTES              REFERENCED
 SEQ-NO ADDR     ADDR    DEC      NO                                DEC        HEX      FORMAT    BY STATEMENTS

  00021                          FD ISAM-FILE                                                     R00007 M00036 R00047
  00023 00000CC0 000000 00000001  01 ISAM-REC                       0000000263 00000107           R00042
  00024 00000CC0 000000 00000001  05 ISAM-KEY                       0000000008 00000008 ZONED DEC R00009 M00038 R00041
  00025 00000CC8 000008 00000009  05 SATZ-CONTENT                   0000000255 000000FF           M00051
  00026 00000CC8 000008 00000009  10 FILLER                         0000000001 00000001 DISPLAY
```



```
COBOL2000  V01.5A00  COPYING                    LOCATOR MAP LISTING           09:58:34 2009-02-12  PAGE 0005

                                        DATA DIVISION            COPYING
                                        WORKING-STORAGE SECTION
        MODULE   GROUP  POSITION
 SOURCE REL      REL    IN GROUP LEV                                LENGTH IN BYTES              REFERENCED
 SEQ-NO ADDR     ADDR   DEC       NO                                DEC        HEX      FORMAT    BY STATEMENTS

  00001 00000068                G77 TALLY                           0000000004 00000004 COMP
  00001                         G77 RETURN-CODE                     0000000004 00000004 COMP-5
  00028 00000DC8 000000 00000001  01 FILLER                         0000000001 00000001 DISPLAY   r00039 m00052
  00029                          88 FILE-END                                                      R00039 R00052
  00030 EXTERNAL 000000 00000001  01 ISAM-FILE-STATUS               0000000002 00000002 DISPLAY   A00010
  00031 00000DD0 000000 00000001  01 LENGTH                         0000000002 00000002 BINARY    R00018 R00026 M00050
  00032 00000DD8 000000 00000001  01 ISAM-REC-LENGTH                0000000002 00000002 BINARY    A00022 M00040
  00033 00000DE0 000000 00000001  01 SAM-REC-LENGTH                 0000000002 00000002 BINARY    A00016 R00040
```



```
COBOL2000  V01.5A00  COPYING                    LOCATOR MAP LISTING           09:58:34 2009-02-12  PAGE 0006

                                        PROCEDURE DIVISION

                (12)       (13)                          (11)

                LVL SOURCE UNIT NAME
 SOURCE REL           SECTION NAME                     REFERENCED
 SEQ-NO ADDR            PARAGRAPH NAME                 BY STATEMENTS

                000 COPYING
  00035 00001040        PROCESS
  00049 00001280        READ-SAM                       R00037 R00045
```

18 Related publications

The manuals are available as online manuals, see http://manuals.ts.fujitsu.com, or in printed form which must be paid and ordered separately at http://manualshop.ts.fujitsu.com.

[1] **COBOL2000** (BS2000/OSD)
COBOL Compiler
Reference Manual

[2] **CRTE (BS2000/OSD)**
Common Runtime Environment
User Guide

[3] **BS2000/OSD-BC**
Commands
User Guide

[4] **BS2000/OSD-BC**
Introductory Guide to DMS
User Guide

[5] **SDF** (BS2000/OSD)
SDF Dialog Interface
User Guide

[6] **SORT** (BS2000/OSD)
User Guide

[7] **JV** (BS2000/OSD)
Job Variables
User Guide

[8] **AID** (BS2000)
Advanced Interactive Debugger
Debugging of COBOL Programs
User Guide

[9] BS2000
TSOSLNK
User Guide

[10] **BLSSERV**
Dynamic Binder Loader / Starter in BS2000/OSD
User Guide

[11] **LMS**(BS2000)
SDF Format
User Guide

[12] **BS2000/OSD-BC**
System Installation
User Guide

[13] **UDS/SQL** (BS2000/OSD)
Design and Definition
User Guide

[14] **UDS/SQL** (BS2000/OSD)
Creation and Restructuring
User Guide

[15] **UDS/SQL** (BS2000/OSD)
Application Programming
User Guide

[16] **ESQL-COBOL** (BS2000/OSD)
ESQL-COBOL for SESAM/SQL-Server
User Guide

[17] **SESAM/SQL-Server** (BS2000/OSD)
SQL Reference Manual Part 1: SQL Statements
User Guide

[18] **SQL for UDS/SQL**
Language Reference Manual

[19] **EDT** (BS2000/OSD)
Statements
User Guide

[20] **AID** (BS2000)
Advanced Interactive Debugger
Core Manual
User Guide

[21] **AID** (BS2000/OSD)
Debugging on Machine Code Level
User Guide

[22] **BINDER**
Binder in BS2000/OSD
User Guide

[23] **openUTM** (TRANSDATA, BS2000)
Planning and Design
User Guide

[24] **openUTM** (BS2000/OSD, UNIX, Windows)
Programming Applications with KDCS for COBOL, C and C++
User Guide

[25] **openUTM** (BS2000/OSD)
Generating and Handling Applications
User Guide

[26] **openUTM** (BS2000/OSD, UNIX, Windows)
Administering Applications
User Guide

[27] **XML for openUTM**
Datamarshalling with XML

[28] **SDF-P** (BS2000/OSD)
Programming in the Command Language
User Guide

[29] **POSIX (BS2000/OSD)**
Commands
User Guide

[30] **POSIX** (BS2000/OSD)
POSIX Basics for Users and System Administrators
User Guide

[31] **C/C++** (BS2000/OSD)
POSIX Commands of the C/C++ Compiler
User Guide

[32] **BS2000/OSD**
Softbooks English
CD-ROM

[33] **XHCS**
(BS2000/OSD)
8-Bit Code and Unicode support in BS2000/OSD
User Guide

[34] **IMON** (BS2000/OSD)
Installation Monitor
User Guide