FOR1 (BS2000) V2.2A

Fortran Compiler

Edition January 1992

Comments… Suggestions… Corrections…

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

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

Certified documentation according to DIN EN ISO 9001:2000

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

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

Copyright and Trademarks

Contents

1 Introduction

The FOR1 compiler translates FORTRAN source programs into object modules.

The range of language elements of FOR1 covers the ANS FORTRAN 77 standard (ANSI X3.9-1978) as well as FORTRAN IV and its extensions. FOR1 adds substantially to the application capabilities of standard FORTRAN. This compiler features efficient optimization as well as a high degree of operating and debugging convenience.

The large number of setting options allows FOR1 to be tailored to a wide variety of applications. It is possible to control the location of the source program, the process of compilation, the structure of the object modules generated, and the output of the listings. This certainly does not mean, however, that the user has to make a large number of specifications for each compiler run: the compiler operates with default values if the user does not make explicit entries.

The FOR1 compiler can be controlled in two different ways:
- by compiler options entered after calling the compiler
- through operands of an SDF command

SDF (System Dialog Facility) is the new BS2000 dialog interface. With SDF it is possible, for example, to enter the commands through menus. Information on the form and meaning of the permissible entries appears - if required - directly on the screen.
One SDF command is available for the compilation of FOR1 source programs and one for the dynamic linkage, loading and starting of compiled programs.

The FOR1 compiler subjects the source programs to numerous syntax and and semantic checks. When errors occur during compilation, the compiler issues messages giving information on the location and cause of the error and corrective action for the compiler. For errors occurring at run time, the runtime system issues informative messages.

Various debugging aids are available to the user for testing the programs. Debugging aids can also be incorporated in the source program text by means of debugging statements or subprograms. By specifying debugging options it is possible to activate additional tests at program run time. Along with these debugging aids integrated in FOR1, you can work with the BS2000 interactive debugging aid AID at program run time (see "AID - Debugging of FORTRAN Programs" [3]).

In order to enhance the efficiency of the executable programs, the compiler performs optimization measures, if required. The user has the choice here between optimization levels of differing intensity.

The various possible settings for FOR1 can be combined with one another almost without restriction. This means for example that optimization can even take place during the debugging phase. Object modules generated with different compiler settings are runtime compatible; they can normally be linked and executed without any problem.

1.1 Target group

This manual is intended for users of FORTRAN in BS2000. The reader should have a knowledge of the FORTRAN programming language and be familiar with basic usage of the BS2000 operating system.

The range of language elements of FOR1 is described in the "FOR1 Reference Manual" [21].

Operating system components are explained briefly at the appropriate points.

1.2 Summary of contents

The following topics are discussed in this user guide:

- Supply of FORTRAN source programs in BS2000
- Language processing by the FOR1 compiler
- Maintenance of object modules
- Linkage to executable programs, and loading
- Execution of FOR1 programs
- Generation of FORTRAN source programs in interactive mode
- File usage
- Reduction of program runtimes (optimization)
- Debugging aids
- Efficient programming

This manual is primarily intended as a reference work. A detailed table of contents and a detailed index make it easier for the reader to find the appropriate information.

In addition, all the compilation options and runtime variants that are available are shown in summary tables.

SDF metasyntax	Section 1.3.2
SDF form for controlling compilation, with references to the appropriate compiler options	Tables 2-2 through 2-14
SDF form for controlling execution, with references to the appropriate runtime options	Section 6.2
Compiler options with references to the appropriate SDF operands	Section 2.3.3
Abbreviations for compiler options and option values	Appendix A.1
Compilation operands for the input and output of PLAM library elements	Table 1-1
Summary of compiler control statements	Section 2.4
Formats of interactive commands	Table 3-1
Formats of debugging statements	Section 7.4.1
Structure of debugging aid subprograms	Section 7.5.1
Mapping of FORTRAN records to DMS	Section 8.4.2

Users who are not familiar with the FOR1 compiler are advised to read chapter 2 first and then refer to the descriptions of the relevant compiler options or SDF operands.

Users who are already familiar with the FOR1 options and now want to know how to control the compiler by means of SDF operands are advised to read sections 2.2.1 and 2.2.2. Table 2-15 in section 2.3.3 shows which SDF operands correspond to the individual compiler options. The summary Tables 2-2 through 2-14 and Table 6-1 also indicate correspondence between SDF operands and compiler/runtime options.

In addition, the descriptions of the individual compiler and runtime options are preceded in each case by a separate section on the format of the corresponding SDF operands:

Operands of the START-FOR1-COMPILER command:

SOURCE	Section 3.2.2
INCLUDE-LIBRARY	Section 3.5.2
DIALOG	Section 3.6.1
SOURCE-PROPERTIES	Section 4.1.1
COMPILER-ACTION	Section 4.2.1
MODULE-LIBRARY	Section 4.3.1
LISTING	Section 4.6.1
COMPILER-TERMINATION	Section 4.8.1
MONJV	Section 4.9
LANGUAGE	Section 4.10.1
COMPILER	Section 4.11
TEST-SUPPORT	Section 7.1
OPTIMIZATION	Section 9.2.1
FPOOL-LIBRARY	Section 12.1.1

Operands of the SDF command START-FOR1-PROGRAM:

FROM-FILE	Section 5.1
CPU-LIMIT, TESTOPT, MONJV, OBJECT-CONTINUATION, RUNTIME-OPTIONS	Section 6.1

1.3 Metasyntax

User inputs are shown in bold print in the representations of dialog examples and execution listings.

The metalanguage conventions used for representing statements and options in this manual are described in the following two sections.

1.3.1 Metasyntax for representing compiler and runtime options

OBJECT	Uppercase letters designate keywords and must be input in this form.
name	Lowercase letters designate variables for which the user substitutes actual values in the entries.
`YES` $\underline{\texttt{NO}}$	Underlining a value indicates that it is a default value inserted by the FOR1 compiler or operating system if no user specification is made.
$\left\{\begin{matrix}\texttt{YES}\\ \underline{\texttt{NO}}\end{matrix}\right\}$	Braces enclose several alternatives, one of which must be selected by the user. The alternatives are stacked one above the other. If one of the stacked entries is a default value, no user specification is needed if the default value is the one desired.
`{YES|NO}`	Alternatives are also indicated by a vertical stroke between two adjacent specifications, one of which has to be selected by the user.
`[ ]`	Brackets enclose optional entries which may be made or may also be omitted.
`( )`	Parentheses form part of the operand and must be entered.
␣	The symbol for a blank (space) is used if at least one blank is needed for syntactical reasons.

`[,...]`

Three dots indicate that the preceding metalanguage unit may be repeated several times in succession.

`:=` A syntax variable to the left of the assignment character is defined by the specification to the right of the assignment character.

Special characters

These must be used unchanged.

1.3.2 SDF syntax description

This syntax description is valid for SDF Version 1.4A. The syntax of the SDF command language is explained in the following 3 tables.

Table 1: SDF metasyntax

Certain characters and representations are used in the command formats; their meaning is explained in Table 1.

Table 2: Data types

Variable operand values are represented in SDF by data types. Each data type represents a specific value set. The number of data types is limited to those described in Table 2.

The description of the data types is valid for the entire set of SDF commands/statements. Therefore only deviations (if any) from the attributes described here are explained in the relevant operand descriptions.

Table 3: Suffixes for data types

Data-type suffixes define additional rules for data-type input. They can be used to limit or extend the value set

The description of the data-type suffixes is valid for the entire set of commands/statements. Therefore only deviations (if any) from the attributes described here are explained in the relevant operand descriptions.

Table 1: SDF metasyntax

Representation	Meaning	Examples

UPPERCASE LETTERS	Uppercase letters denote keywords. Some keywords begin with *.	TESTOPT = NONE
		
		VERSION = *STD

=	The equal sign connects an operand name with the associated operand values.	SOURCE-FORMAT = FIXED
< >	Angle brackets denote variables whose range of values is described by data types and suffixes (see Tables 2 and 3).	LIBRARY = <full-filename 1..54>
Underscoring	Underscoring denotes the default value of an operand.	SUMMARY = YES / NO
/	A slash serves to separate alternative operand values.	TESTOPT = NONE / AID
(...)	Parentheses denote operand values which initiate a structure.	LAYOUT = PARAMETER(...)

Indentation	Indentation indicates that the operand is dependent on a higher-ranking operand.	SOURCE = NO / YES (...)
		
		YES(...)
		| INSERT-ERROR-WEIGHT =

|	A vertical bar identifies related operands within a structure. Its length marks the beginning and end of a structure. A structure may contain further structures. The number of vertical bars preceding an operand corresponds to the depth of the structure.	*LIBRARY-ELEMENT(...) | LIBRARY = | ,ELEMENT = | | VERSION =
,	A comma precedes further operands at the same structure level.	,SOURCE = NO ,DIAGNOSTICS = NO

list-poss(n)	"list-poss" signifies that the operand values following it may be entered as a list. If a value is specified for (n), it means that the list may contain no more than that number of elements. A list of two or more elements must be enclosed in parentheses.	list-poss: <integer 0..99>

Table 2: Data types

Data type	Character set	Special rules
alphanum-name	A...Z 0...9 $,#,@	Must begin with letter or digit.
c-string	EBCDIC characters	Must be enclosed in single quotes; may be preceded with the letter C; any single quotes occurring within the c-string must be entered twice.
full-filename	A...Z 0...9 $,#,@ hyphen period	Input format: $\text{:cat:\$user.}\left\{\begin{array}{l}\text{file}\\ \text{file(no)}\\ \text{group}\\ \text{group}\left\{\begin{array}{l}\text{(*abs)}\\ \text{(+rel)}\\ \text{(-rel)}\end{array}\right\}\end{array}\right\}$: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 and 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") $\text{group}\left\{\begin{array}{l}\text{(*abs)}\\\text{(+rel)}\\\text{(-rel)}\end{array}\right\}$ name of a file generation (character set: as for "file") (*abs) absolute generation number (1-9999) * 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.
name	A...Z 0...9 $,#,@	Must not comprise only 0...9.

Table 3: Suffixes for data types

Suffix	Meaning
x..y	a) with data type "integer": interval specification x minimum value permitted for "integer". x is an (optionally signed) integer y maximum value permitted for "integer". y is an (optionally signed) integer b) with the other data types: length specification x minimum length for the operand value; x is a whole number. y maximum length for the operand value; y is a whole number. x=y the length of the operand value must be precisely x

1.4 Changes since the last version of the manual (FOR1 V2.1A)

The manual has been extensively reorganized.
The ISP command language has been replaced throughout by the SDF command language.

The main factual additions and changes are shown in the following table with references to the relevant sections.
Corrections to contents and text throughout the manual are not separately indicated.

Subject	New	Changed	Omitted	Section
ILCS program communication interface	X			11.1 - 11.7
SDF control		X		
- LINKAGE operand	X			2.2.3, 4.2.1
- FORTRAN90-CHECK operand	X			2.2.3, 4.1.1
- EXTENDED-SYSTEM operand			X	
- LISTING form		X		2.2.3
- Support for the symbolic version identifier of the LMS	X			2.2.3, 3.2.2 3.6.1, 4.6.1
COMOPT control		X		
- LINKAGE option	X			4.2.2.6
- FORTRAN90-CHECK option	X			4.1.2.8
- EXTENDED-SYSTEM option			X	
- Support for the symbolic version designation of the LMS	X			3.2.3, 3.6.2 4.6.2.2
Mathematical routines		X		10.3
FPOOL function GETDATE	X			12.2.10
Support for the NK-SAM file format	X			8.2.4
Year number entry in compiler lists		X		4.7, A.6
Dynamic link loading		X		5.5
PARMOD parameter		X		A.9.1
FOR1MODLIB			X	
Description of debugging with IDA			X	
SIA (Scientific Instruction Assist)			X	
Listing of library modules			X	
Listing of error messages			X	

1.5 General requirements concerning compilation, linking and program execution

The FOR1 compiler is suitable for machines having a main memory of 2 Mbytes or greater. Depending on the size of the user programs, additional main memory space may be advisable.

The following components are required in order to generate an executable FOR1 program:

- the FOR1 compiler
- the FOR1MODLIBS runtime system
- to enable the user to preload the compiler (see 1.8):
  the ENTER procedure SYSENT.FOR1.022.LOAD1
- error text file for I/O errors:
  the INCLUDE element IFNIOS in the FOR1 macro library FOR1MACLIB (see A5)
- to enable use of the central FPOOLs (see 12.2):
  the object module library FOR1.FPOOLLIB and the associated file for the interface specifications FOR1.FPOOL

The procedure SYSPRC.FOR1.022.SHARE is available for the generation of shareable modules (see 5.8.1).

In this manual it is assumed that the above components have been entered under the TSOS user ID (e.g. by calling the compiler using the command: `/START-PROGRAM FROM-FILE=$FOR1`).

1.6 System environment of the FOR1 compiler

Fig. 1-1 shows the FOR1 compiler in the BS2000 environment.

This figure is not any longer available for the online pdf.

Fig. 1-1: System environment of the FOR1 compiler

FOR1 and its load modules can be used as application programs under the BS2000 operating system (program class 2 = pageable, processor state TU, memory classes 6, 5, 4).

1.7 Management of programs in program libraries

The FOR1 compiler can access program libraries (PLAM libraries). Program libraries are PAM files processed using PLAM (Program Library Access Method). Access to PLAM library elements permits uniform and efficient management of different element types. The following types (for example) can be stored as elements of a PLAM library:

Type S	Source programs, %INCLUDE items
Type M	Macros
Type R	Object modules
Type C	Load modules
Type L	Link and load modules (LLMs)
Type P	Compiler listings

FOR1 supports type S, R, P and M PLAM library elements.

In PLAM libraries any type of element can be stored in a *single* library. It is possible for a number of elements to have the same name. These can be differentiated by type or version designation.

The maintenance of data in PLAM libraries has the following advantages:

– savings of up to 30% of memory space can be achieved by storing different element types together and by employing additional compression methods

– access times are reduced compared with using conventional libraries

– EAM memory is freed (object modules are stored directly in the form of PLAM library elements)

Compilation operands for input/output of PLAM library elements

Input and output of various PLAM library elements is controlled by means of compiler options or SDF operands. The following types can be processed as PLAM library elements by FOR1:

Type	PLAM element type	Control by compiler option	See section	SDF operands
Compiler option	S	OPTIONS	3.3.2	-
Source programs	S	SOURCE	3.2.3	SOURCE
Change lines	S	UPD	3.4	-
INCLUDE source program sections	S	INCLUDE-LIBRARY	3.5.3	INCLUDE-LIBRARY
Dialog save	S	DIALOG-SAVE	3.6.4	DIALOG, SAVE-FILE
Object modules	R	MODULE-LIBRARY	4.3.2	MODULE-LIBRARY
Shareable object modules separate from the nonshareable object modules	R	SHARE-LIBRARY	4.2.2.2 5.8	COMPILER-ACTION SHAREABLE-CODE OUTPUT-LIBRARY
Listings	P	LIST-OUTPUT	4.6.2.4	LISTING OUTPUT

Table 1-1: Compiler options and SDF operands for input/output of PLAM library elements

The linkage editors DBL, TSOSLNK and BINDER can also process PLAM library modules; TSOSLNK and BINDER can store the generated modules in PLAM libraries (see chapter 5).

1.8 Preloading the compiler

FOR1 is shareable without having to take special action. In other words once a user has loaded an overlay of the compiler in virtual memory, this overlay can be shared by all other tasks.

The overlays of the compiler remain in virtual memory as long as they are used by at least one task. An overlay is loaded as soon as any one user requires it.

Loading times are eliminated completely if the FOR1 overlays are kept in virtual memory through preloading (batch jobs). The following message appears on the screen at compile time if the compiler has not been preloaded:

```
FOR1: COMPILER NOT PRELOADED (BAD LOAD PERFORMANCE)
```

In the summary listing, the message
"(COMPILER NOT PRELOADED)" appears in the compile time section.

If the compiler has been preloaded, marked savings of CPU time and switch-on time per program unit will be obtained as a result of the elimination of loading times.

The shareable compiler must be preloaded by the system administrator. A system with a user address space of more than two megabytes will require a preload job. If no more than two megabytes are available, a second preload job will be started automatically.

It is advisable to start the preload jobs when the system is started, using the procedure

```
/CALL-PROC NAME=SYSPRC.FOR1.022.SYSLOD
```

Preloading can also be started by the user, using the enter file

```
/ENTER-JOB FROM-FILE=$SYSENT.FOR1.022.LOAD1,RESOURCES=PAR(CPU-LIMIT=50)
```

This procedure causes the program SYSPRG.FOR1.022.LOAD to be started, which assigns common memory pools for the compiler overlays. If the user address space is too small (<2MB), SYSPRG.FOR1.022.LOAD will be interrupted and procedure SYSENT.FOR1.022.LOAD2 will be started. The procedure SYSENT.FOR1.022.LOAD2 in turn calls SYSPRG.FOR1.022.LOAD and creates the remaining common memory pools. The interrupted program SYSPRG.FOR1.022.LOAD is continued, which has now the task of holding the overlays in the common memory pools if the compiler no longer requires them (hence the name "holder task").

If the compiler is called after the preload jobs have been started, the individual compiler overlays will be loaded into the assigned common memory pools. Not until all addressed overlays have been loaded into the common memory pools will the message "FOR1: COMPILER NOT PRELOADED(...)" cease to be issued. If, for example, compilation has been performed with COMOPT OPTMIZE=1, this message will no longer be issued after preloading. If, however, a compilation is performed with COMOPT

OPTIMIZE=3, the message is issued again, as a compiler overlay that has not yet been loaded is referenced.

Preload jobs run in a "VPASS-100" loop, i.e. they are mostly in the VPASS queue and their load on the system is minimal. These preload jobs are terminated by SHUTDOWN, or prematurely (by the system administrator) by SEND-MESSAGE or CANCEL-JOB.

Example: Preloading the compiler

```
/ENTER-JOB FROM-FILE=SYSENT.FOR1.022.LOAD1,                                    (1)
 RESOURCES=PAR(CPU-LIMIT=50)

/START-PROG FROM-FILE=$FOR1                                                    (2)

 % BLS0500 PROGRAM 'FOR1', VERSION '2.2A00' OF '91-06-05' LOADED.
 % BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG. 1991 ...
  FOR1: V2.2A00 READY, GIVE COMPILER OPTION
 *COMOPT SOURCE=QUELLE.TEST,END
  FOR1: COMPILER NOT PRELOADED (BAD LOAD PERFORMANCE)
  FOR1: NO ERRORS DURING COMPILATION OF P. U. TEST
  END OF  F O R 1  COMPILATION; CPU TIME USED:   4.649 SEC
/DEL-SYS-FILE FILE-NAME=OMF
/START-PROG FROM-FILE=$FOR1                                                    (3)
 %  BLS0500 PROGRAM 'FOR1', VERSION '2.2A00' OF '91-06-05' LOADED.
 %  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG. 1991 ...
  FOR1:    V2.2A00 READY, GIVE COMPILER OPTION
 *COMOPT SOURCE=QUELLE.TEST, END
   FOR1: NO ERRORS DURING COMPILATION OF P. U. TEST
   END OF  F O R 1  COMPILATION: CPU TIME USED:    4.333 SEC
```

Explanation of example:

(1) Calling the ENTER procedure SYSENT.FOR1.022.LOAD1 causes the preload job to be started. The ENTER procedure must run under the user-own ID, in this case.

(2) The compiler is preloaded when the program is compiled for the first time. The message "COMPILER NOT PRELOADED" is still issued during this compilation session.

(3) The compiler has now been preloaded. Message "COMPILER NOT PRELOADED" will no longer be issued. The preloaded compiler requires less CPU time for the compilation. The saving in CPU time is especially noticeable for FORTRAN programs with many subprograms. As a result of preloading, only the first program unit requires loading time; no loading times are required for subsequent program units.

1.9 Runtime system

In the same way as the FOR1 compiler, the FOR1 runtime system is also shareable.

1.9.1 Structure

The FOR1 runtime system comprises individual modules that are linked together to form the main module IF@RTS1. This main module is located together with several language linkage and debugging aid modules in the library FOR1MODLIBS.

During linking, the required runtime modules are not linked in completely; only linkage modules are linked in. These linkage modules enable the actual runtime system (main module IF@RTS1) to be dynamically loaded during program execution. This considerably reduces the memory requirement for stored FOR1 programs.

The FOR1 runtime system is responsible for the following basic tasks:

- initializing the runtime communication area (RTCA) and terminating programs
- executing I/O operations
- supplying predefined functions (intrinsic functions) and subprograms (debug subprograms)
- error handling at execution time

The predefined functions include the standard modules for mathematical functions as well as routines for string handling. In addition to these functions, a number of ready-made subprograms are provided, mainly for testing purposes.

The I/O section of the runtime system implements input/output statements on three functional levels:

1. FORTRAN level
   Controlling the various types of input/output (formatted, unformatted, NAMELIST-directed and list-directed).

2. Conversion level
   Conversion of data between the internal and external representations

3. System connection level
   Calling the input/output functions of the operating system (access operations, positioning).

The names of the FOR1 library modules are given when runtime errors occur and when the calling hierarchy is output.

1.9.2 Loading using /ADD-SHARED-PROGRAM and /LOAD-PROGRAM

The system administrator loads the runtime system as a shareable system into class-4 memory using the commands

```
/ADD-SHARED-PROG ENTRY-NAME=IF@RTS1, LIB-NAME=$TSOS.FOR1MODLIBS
/LOAD-PROG *MODULE(LIB=$TSOS.FOR1MODLIBS, ELEM=IF@RTS1)
```

The main memory requirement for the load module is irrespective of the number and size of the required runtime modules. The class-4 memory remains occupied until SHUTDOWN.

If the IF@RTS1 module has not been loaded as a shareable module, almost the entire runtime system will be dynamically loaded at runtime into the user address space, thus resulting in a maximum memory requirement. This is the reason why the runtime system FOR1MODLIBS should always be loaded as a shareable system into class-4 memory.

Only one version of the runtime system may reside in class-4 memory.

1.9.3 Loading via DSSM

Generating the subsystem catalog

The shareable FOR1 runtime system can be loaded into class-4, class-5 or class-6 memory by the system administrator via DSSM, on condition that the FOR1 runtime system was declared when the subsystem catalog was generated.

Loading the shareable runtime system via DSSM has the following advantages:

- loading into class-5 or class-6 memory relieves class-4 memory;
- more than one version of the shareable runtime system can be used within one application by loading an appropriate new subsystem catalog.

The subsystem catalog is generated with UGEN (see "System Installation" manual [38]). The steps required to generate the subsystem catalog with UGEN are described briefly below:

1. Call the UGEN utility routine and select the UGEN branch for generating a sub-system catalog (SSMCAT):

   ```
   /CALL-PROC NAME=$userid.UGEN
    GEN SSC
   ```

2. Define the file for SSMCAT:

   ```
   DSMCAT DSSM-catalog-name,CAT=NEW
   ```

3. Declare the individual subsystems:

   - by explicitly specifying their names or
   - via subsystem input files (ASSIGN-SYSDTA command)

   The FOR1 runtime system declarations are defined in the subsystem input files:

   - $TSOS.SYSSSD.FOR1.022.CL4 for class 4
   - $TSOS.SYSSSD.FOR1.022.CL5 for class 5
   - $TSOS.SYSSSD.FOR1.022.CL6 for class 6

   The command

   ```
   /ASSIGN-SYSDTA TO-FILE=$TSOS.SYSSSD.FOR1.022.{CL4|CL5|CL6}
   ```

   causes the appropriate subsystem input file of the FOR1 runtime system to be used in order to generate the subsystem catalog.

4. Terminate the UGEN session using the END statement:

   ```
   END
   ```

Activation and deactivation of the FOR1 runtime subsystem

The FOR1LZS subsystem must be activated explicitly by the system administrator by means of the DSSM command CREATE-SUBSYSTEM:

```
/CRE-SUBSYS SUBSYS=FOR1LZS, VERSION='0N.NN00'
```

where "N.NN" is the name of the version, e.g. 2.2A.

The DELETE-SUBSYSTEM command is used to deactivate the subsystem:

```
/DEL-SUBSYS SUBSYS=FOR1LZS
```

The DSSM command SHOW-SUBSYSTEM-STATUS gives information about a sub-system:

```
/SHOW-SUBSYS-STA SUBSYS=FOR1LZS
```

Special considerations concerning expansion of the subsystem catalog

- When the subsystem catalog is expanded dynamically to include the shareable FOR1 runtime system, the user must bear in mind that the subsystem catalog with which the operating system was started can accommodate up to 20 new sub-systems with a total of up to 100 entry points.

– Note that when the subsystem catalog is dynamically expanded, the order of the existing subsystems does not change and no subsystem will be deleted.

  The subsystem name of the shareable runtime system for class-4, class-5 and class-6 memory is ’FOR1LZS, VERSION 0N.NN00’, where "N.NN" is the name of the version, e.g. 2.2A.
  Because of this name and version match, more than one subsystem declaration cannot be used simultaneously for generating a subsystem catalog. Should this nevertheless be necessary, the version numbers in the relevant declaration files will have to be changed.

2 Controlling the FOR1 compiler

FOR1 compiles FORTRAN source programs into object modules. More than one program unit can be compiled in one compilation run.

With the aid of compilation operands the user can define the conditions for the compilation for each compilation run. Compilation operands control the input of the source program, the internal execution of the compilation, the output of the object modules and the generation and output of log listings.

The compilation operands can be specified in the following ways:

- with the SDF command START-FOR1-COMPILER (see section 2.2);
- with COMOPT statements (see section 2.3);
- with the PARAMETER command, with some restrictions (see appendix A.5.2).
- with compile time statements in the source program, certain supplementary operands (see section 2.4)

Each user can select the suitable variant for his application from the large number of possible entries for compilation operands. All compilation operands have preset default values that take effect if no corresponding entry is made. Thus, in the case of a straightforward compilation run (see sections 2.2 and 2.3) for example, it is necessary only to specify the source program file or input the source program.

Multiple entry of compilation operands

If a compilation operand is entered more than once, the value entered last applies. If a PARAMETER command is valid for a task, the specification of a compiler option or of an SDF operand overwrites the entry in the corresponding PARAMETER command.

Validity

Compiler options and SDF operands are valid only until the end of the compiler session for which they are specified.
The PARAMETER command is valid either until a new PARAMETER command is given or, in procedures, until a STEP command or end-of-task is encountered.

2.1 Starting FOR1

The FOR1 compiler can be started in two ways:

- If the compilation operands are to be entered as SDF operands, the user starts the compiler by specifying the desired operands in the `/START-FOR1-COMPILER` command.
  The compiler commences the compilation immediately the command is interpreted.
- If the compilation operands are to be entered as compiler options, the user starts the compiler with the `/START-PROGRAM $FOR1` command.
  After FOR1 has been called, the version number of the compiler is displayed at the terminal as follows:

  ```
  % BLS0500 PROGRAM FOR1, VERSION '2.2A00' OF '91-06-05' LOADED.
  % BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG. 1991 ...
  FOR1: V2.2A00 READY, GIVE COMPILER OPTION
  ```

  In interactive mode FOR1 then displays an asterisk on each line and waits for entry of compiler options. Once the compiler option END has been entered, the compilation process commences.

Program monitoring by job variables

When the compiler is called, a predefined program monitoring job variable can be specified. This enables the user to inquire about program termination of the compilation session (see section 4.9).

Preloading the compiler

If FOR1 is not preloaded, the following message appears during the compilation:

```
FOR1: COMPILER NOT PRELOADED (BAD LOAD PERFORMANCE)
```

Preloading is advisable if the compiler is called fairly often. When the compiler is preloaded, loading time is saved for each user who calls FOR1. The compiler can be preloaded by the system administrator or by the user himself (see section 1.8).

Messages from the compiler

When compilation has been successful, the compiler issues the following message for each program unit compiled:

```
FOR1: NO ERRORS DURING COMPILATION OF P.U. name
```

name is the name of the compiled program unit.

At the end of the compilation the compiler issues a message indicating the CPU time used (in seconds):

```
END OF  F O R 1  COMPILATION; CPU TIME USED: n.nnn SEC
```

The user can specify whether the compiler messages should be issued in German or in English. The SDF operand LANGUAGE and the compiler options DIALOG and LANGUAGE are available for this purpose. The messages are issued in English by default.

2.2 Control through SDF commands

2.2.1 Entering SDF commands

SDF (**S**ystem **D**ialog **F**acility) encompasses the full functional scope of the previous BS2000 commands (in ISP format). The capabilities of the SDF command language include the following:

- The user can enter SDF commands via command menus and operand forms enabling him to select between three levels of guidance. In this "guided" interactive mode even a user who is not very familiar with SDF is capable of issuing a BS2000 command since SDF displays on the screen the operands to be selected, accompanied by brief explanations and permissible values.
- The user can also enter operands without the above prompts. He may abbreviate commands, have a correction dialog displayed in case of invalid entries, or temporarily switch from unguided dialog to guided dialog.

For an introduction to SDF and many examples, see the manual "Introductory Guide to the SDF Dialog Interface" [16].
The user commands in the SDF command language are dealt with in the "User Commands (SDF Format)" manual [12].

The user can also have information about SDF displayed on the screen by entering the command `/HELP-SDF?`. Operand forms are then displayed, from which he may select the further information he wishes to obtain (e.g. concerning types of dialog guidance, abbreviation rules, function keys and statements for menu control).

The language in which the explanations of the SDF menus are output is specified by the system administrator. The language of the SDF messages can be specified by the user with the MODIFY-MSG-ATTRIBUTES command.

Two SDF commands are available for the compilation and execution of FOR1 programs, respectively:

- START-FOR1-COMPILER is the SDF command provided for the compilation of a FOR1 source program. With a few exceptions, the operands of this command permit selection of all variants that can be defined by compiler options.
- START-FOR1-PROGRAM is the SDF command for dynamic linking, loading and starting a compiled FOR1 program. The operands of this command permit selection of the most important functions, which can be specified in the dynamic binder loader (DBL) call, in the static loader (ELDE) call, as well as by means of runtime options.

Guided dialog

Guided dialog permits SDF commands to be issued with the aid of a menu. A guided dialog starts with a summary of all application areas displayed on the screen. After selecting a specific application area (e.g. PROGRAMMING-SUPPORT), the user receives a summary of all commands belonging to the selected area. When a command has been selected (e.g. START-FOR1-COMPILER), an operand form is output, which may be followed by other operand forms or subforms associated with a specific operand.

Using guided dialog the user may choose between minimum, medium or maximum user guidance. These levels differ in the scope of information that SDF displays. Guided dialog is especially recommended for users who are not completely familiar with a command. Dialog guidance and information output directly to the screen about the meaning and syntax of commands and operands as well as about permissible and preset operand values obviate the need for having to consult the manual all too frequently. When an invalid SDF command is input, an error message will be issued and a correction dialog initiated, making reentry of the entire command unnecessary.
In guided dialog, any desired command can be entered in the NEXT line of a screen, regardless of the current application area.

Unguided dialog

SDF commands can not only be entered in guided dialog with the aid of menus, but also in concise form analogous to the former BS2000 command format. In this unguided dialog mode, the user may choose between two modes of command input, depending on whether only error messages (expert mode) or error messages plus correction dialog (NO mode) are requested on invalid input. The SDF command language permits abbreviations to be made so that SDF commands can be considerably shortened.

Temporary guided dialog

Switching from unguided dialog to guided dialog can be effected at any time by entering a question mark so as to allow the user to have missing information concerning commands or operands displayed on the screen whenever he wishes. For example, after entering

```
%CMD:  START-FOR1-COMPILER? TEST
```

you will receive the operand form for guided dialog (GUIDANCE = MINIMUM). In the header of the form, "SOURCE=TEST" has already been registered under "OPERANDS". By overwriting preset operand values with a question mark, further information about specific operands may be requested.

Setting the dialog mode

The dialog mode is defined by the GUIDANCE operand in the MODIFY-SDF-OPTIONS command. The following dialog modes can be selected:

```
GUIDANCE = {UNCHANGED | MAXIMUM | MEDIUM | MINIMUM | EXPERT | NO}
```

`UNCHANGED`	The previous definition still applies.
`MAXIMUM`	Guided dialog: maximum guidance, i.e. all operand values with additions, help texts for commands and operands.
`MEDIUM`	Guided dialog: all operand values without additions, help texts for commands only.
`MINIMUM`	Guided dialog: minimum guidance, i.e. default values of operands only, no additions, no help texts.
`EXPERT`	Unguided dialog: expert mode, i.e. system displays "/" to request command input; no syntax error dialog; detailed error messages; blocked command input.
`NO`	Unguided dialog: system displays "% CMD:" to request command input; syntax error dialog (correction of invalid entries without reentry of the entire command), blocked command input (several commands separated by logical end-of-line characters can be sent off concurrently).

Command input with keyword operands or positional operands

Operand values can be specified as keyword operands or as positional operands. In the case of positional operands, a comma must be entered for each omitted operand.

Command with keyword operands:

```
START-FOR1-COMPILER SOURCE=DAT1,INCLUDE-LIBRARY=*NONE, FPOOL-LIBRARY=*NONE,
DIALOG=YES(LANGUAGE=ENGLISH, DIALOG-INTERRUPT=ERRORS-ONLY,SAVE-FILE=
*STD-NAME (INCLUDE-EXPANSION=YES), LOG-CHANGED-LINES=YES)
```

Command with positional operands:

```
/START-FOR1-COMPILER DAT1,,,YES(,ERRORS-ONLY,*STD-NAME(YES),YES)
```

If, in the operand sequence of a command, an operand is entered in the form "operand=operand value", all subsequent operands of the same level must also be specified as keyword operands.

Since the sequence of the operands cannot be guaranteed in the long term, it is advisable to use only keyword operands in procedures.

Abbreviation rules

Command names, operand names and constant operand values may be abbreviated as follows:

– characters may be omitted from right to left, e.g. `START-FOR1-C` instead of `START-FOR1-COMPILER`, `SO=` instead of `SOURCE=`, `*LIBRARY-EL` instead of `*LIBRARY-ELEMENT`.

– characters within substrings may also be omitted from right to left, e.g.: `S-F-C` instead of `START-FOR1-COMPILER`, `S-FO-P` instead of `START-FOR1-PROGRAM`, `*L-E` instead of `*LIBRARY-ELEMENT`.

– the abbreviation has to be unique; for example, upon entry of `START-FOR1` it is reported that this is ambiguous with regard to `START-FOR1-COMPILER` and `START-FOR1-PROGRAM`.

Information about the abbreviation rules can be obtained using the /HELP-SDF command.

2.2.2 Example of simple compilation and program run with SDF commands

A simple compilation run is understood here to mean a compilation which, with the exception of a few specified operands, is mainly controlled by preset operand values. A simple program run includes linking, loading and execution of the compiled program by means of the dynamic binder loader (DBL).

```
/START-FOR1-COMPILER SOURCE=QUELLE.TEST                                        (1)
 %  BLS0500 PROGRAM 'FOR1', VERSION '2.2A00' OF '91-06-05' LOADED.
 %  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG. 1991 ...
 FOR1:    V2.2A00 READY,GIVE COMPILER OPTION
 FOR1: COMPILER NOT PRELOADED (BAD LOAD PERFORMANCE)
 FOR1: NO ERRORS DURING COMPILATION OF P.U. TEST
 END OF  F O R 1  COMPILATION; CPU TIME USED: 0.210 SEC.                       (2)
/SET-TASKLIB FOR1MODLIBS                                                       (3)
/START-FOR1-PROGRAM                                                            (4)
 %  BLS0001 DBL VERSION 070 RUNNING
 %  BLS0517 MODUL 'TEST' LOADED
 BS2000  F O R 1  : FORTRAN PROGRAM "TEST "
 STARTED ON 1991-07-16 AT 14:17:39
 BS2000  F O R 1  : FORTRAN PROGRAM "TEST " ENDED PROPERLY AT 14:17:40
 CPU - TIME USED  :       0.0031 SECONDS
 ELAPSED TIME     :       0.8700 SECONDS
```

Explanation of example:

(1) The SDF command START-FOR1-COMPILER starts FOR1.
In the SOURCE operand, the cataloged file QUELLE.TEST is defined as the input source for the source program.
By default the source listing, diagnostic listing, map listing, options listing and

summary listing are output to SYSLST (LISTING=STD).
By default the object module is placed in the temporary EAM area (OMF). Object modules in the temporary EAM area which originate from previous compilations are automatically deleted (MODULE-LIBRARY=*OMF(DELETE-OLD-CONTENTS=YES).

(2) The compiler responds with the version number and date. The compiler reports that FOR1 is not preloaded. No errors were found during the compilation run. The CPU time used for the compilation is output.

(3) If the modules of the runtime system are not located in the system's TASKLIB, the user-own module library must be assigned as TASKLIB before the dynamic binder loader is called with START-FOR1-PROGRAM. The user-own module library here is called FOR1MODLIBS.

(4) The SDF command START-FOR1-PROGRAM causes the object module TEST (name of the program in the cataloged file QUELLE.TEST) to be linked, loaded and started.
Since no FROM-FILE operand is specified here, the object module TEST is taken from the temporary EAM area for the current task (default).
The full form of this command would be:
START-FOR1-PROGRAM FROM-FILE=*MODULE(LIBRARY=*OMF, ELEMENT=*ALL).

Abbreviated form:

By applying the abbreviation rules the SDF commands in the above example can be entered in a considerably reduced form.

```
/S-F- QUELLE.TEST
%  CMD0187 ABBREVIATION OF OPERATION NAME 'S-F-' AMBIGUOUS WITH REGARD     (2a)
 TO 'SET-FILE-ATTRIBUTES, SHOW-FILE-LINK, START-FOR1-COMPILER,...
/S-F-C QUELLE.TEST                                                        (2b)
/SE-T FOR1MODLIBS                                                          (3)
/S-FO-P                                                                    (4)
```

(2a) When the abbreviation is not unique, an error message is issued in expert mode (a correction dialog will be displayed additionally in other dialog modes).

(2b) The SOURCE operand is specified as a positional operand.
Command and operand names as well as constant operand values can be abbreviated by truncating characters from right to left: START-FOR1-COMPILER is abbreviated to S-F-C.

(3) SET-TASKLIB is abbreviated to SE-T.

(4) START-FOR1-PROGRAM is abbreviated to S-FO-P.
The default for the FROM-FILE operand is *MODULE(LIBRARY=*OMF, ELEMENT=*ALL), so that the command can be further abbreviated to S-FO-P.

2.2.3 Summary: SDF command START-FOR1-COMPILER and corresponding compiler options

The following tables contain a complete list of all the operands of the START-FOR1-COMPILER command. The operands are shown with the corresponding compiler options. The metalanguage conventions defined in section 1.3 are used in the tables.

The following compiler options have no corresponding SDF operands:
OPTIONS, UPD, CODE, SUPPLIEDBOUND, UNIT, PAD

The default values for some SDF operands differ from the defaults for the corresponding compiler options (see lists in Tables 2-2 through 2-14).

Summary: The main operands of the SDF command START-FOR1-COMPILER

START-FOR1-COMPILER
SOURCE = ...
,INCLUDE-LIBRARY = ...
,FPOOL-LIBRARY = ...
,DIALOG = ...
,SOURCE-PROPERTIES = ...
,COMPILER-ACTION = ...
,MODULE-LIBRARY = ...
,LISTING = ...
,TEST-SUPPORT = ...
,OPTIMIZATION = ...
,COMPILER-TERMINATION = ...
,MONJV = ...
,LANGUAGE = ...

Table 2-1: Main operands of the SDF command START-FOR1-COMPILER

SOURCE form: Input source for source program

SDF form	First subform	Second subform	Corresp. COMOPTs
SOURCE = *SYSDTA			SOURCE = *
= <full-filename 1..54>			= file
= *LIBRARY-ELEMENT(...)	LIBRARY = <full-filename 1..54> ELEMENT = <full-filename 1..41>(...)	VERSION = *HIGHEST-EXISTING = <alphanum-name 1..24>	*LIBRARY-ELEMENT (LIBRARY = plamlib, ELEMENT = name (VERSION = *HIGHEST EXISTING)) = version))

Table 2-2: SDF command START-FOR1-COMPILER, SOURCE form

INCLUDE-LIBRARY form: Access to libraries with %INCLUDE items

SDF form	Corresponding COMOPTs
INCLUDE-LIBRARY = *NONE	INCLUDE[-LIBRARY] = *NO
= list-poss: <full-filename 1..54>	= filename or = (filename1[,filename2][,..])

Table 2-3: SDF command START-FOR1-COMPILER, INCLUDE-LIBRARY form

FPOOL-LIBRARY form: Controlling FPOOL processing

SDF form	Corresponding COMOPTs
FPOOL-LIBRARY = *NONE	NOFPOOL

= list-poss:
= <full-filename 1..54>
	FPOOL = fpoolname or
	FPOOL = ([fpoolname[,fpoolname][,..]])

Table 2-4: SDF command START-FOR1-COMPILER, FPOOL-LIBRARY form

DIALOG form: Interactive analysis

SDF form	First subform	Second subform	Corresp. COMOPTs
DIALOG = NO			NODIALOG
= YES(...)			DIALOG = { param (param[,param] (param[,param [,param]]) }
	DIALOG-INTERRUPT = AFTER-ANY-PROG-UNIT		param := E[DIT] = ALL
	= ERRORS-ONLY		= NO
	SAVE-FILE = *NONE		DIALOG-SAVE No DIALOG-SAVE option
	= *STD-NAME(...)	INCLUDE-EXPANSION = NO	= (*STD-FILE ,INCLUDE-EXPANSION = NO)
		= YES	= YES)
	= <full-filename 1..54>(...)	INCLUDE-EXPANSION = NO	= ([FILE=]file ,INCLUDE-EXPANSIONS = NO)
		= YES	= YES)
	= *LIBRARY-ELEMENT (...)	INCLUDE-EXPANSION = NO	,INCLUDE-EXPANSIONS = NO)
		= YES	= YES)
		LIBRARY = <full-filename 1..54>	= ([FILE=] *LIBRARY-ELEMENT (LIBRARY = plamlib,
		ELEMENT = <full-filename 1..41> (...)	ELEMENT = name
		VERSION = <alphanum-name 1..24>	(VERSION = version))
		= *UPPER-LIMIT	= *UPPER-LIMIT))
	LOG-CHANGED-LINES = NO		CHANGE in LIST- or LISTFILE option not specified

	= YES		LIST = (CHANGE) or
			LISTFILE = listfilename
			(CHANGE) if COMOPT DIALOG is also specified

Table 2-5: SDF command START-FOR1-COMPILER, DIALOG form

As the prefix for dialog commands the @ character is preset for SDF form DIALOG, the % character for COMOPT DIALOG.

SOURCE-PROPERTIES form: Defining and checking the characteristics of the source program on compilation

SDF form	First subform	Corresponding COMOPTs
SOURCE-PROPERTIES = STD		Subform defaults
= PARAMETER(...)	COMPILEABLE-COMMENTS = *NONE	CCOM no CCOM option
	= <c-string 1..60>	= 'comment-marks'
	LINE-END-COMMENTS = *NONE	LINEEND no LINEEND option
	= <c-string 1..10>	= 'endmarks'
	LANGUAGE-STANDARD = FOR1	STANDARD-CHECK = NO
	= ANS77	= ANS77
	IMPLICIT-DECLARATION = YES	IMPLICIT
	= NO	NOIMPLICIT
	EXPONENT-UNDERFLOW = IGNORED	NOEXPUNDERFLOW
	= ERROR	EXPUNDERFLOW
	SOURCE-FORMAT = FIXED	SOURCE-FORMAT = FIXED
	= FREE	= FREE
	SAVE-CONSTANT = *STD	SAVE-CONSTANT = YES for OPT=NO,0,1,2 = NO for OPT=3 or 4
	= NO	= NO
	= YES	= YES
	FORTRAN90-CHECK = NO	FORTRAN90-CHECK = NO
	= YES	= YES

Table 2-6: SDF command START-FOR1-COMPILER, SOURCE-PROPERTIES form

COMPILER-ACTION form: Defining the characteristics of the generated code

SDF form	First subform	Second subform	Corresp. COMOPTs
COMPILER-ACTION = SYNTAX-CHECK			NOGEN
= MODULE-GENERATION (...)			Subform defaults
	SHAREABLE-CODE = NO		OBJECT no SHARE operand value
	= YES(...)		= SHARE
		OUTPUT-LIBRARY = *MODULE-LIBRARY	SHARE-LIBRARY = *MODULE-LIBRARY
		= <full-filename 1..54>	= plamlib
	MINIMAL-PRECISION = REAL-4(...)		REAL
		EXTERNAL-DATA = NO	= (4)
		= YES	= 4
	= REAL-8(...)	EXTERNAL-DATA = NO	= (8)
		= YES	= 8
	= REAL-16(...)	EXTERNAL-DATA = NO	= (16)
		= YES	= 16
	CONSTANT-PRECISION = AS-NEEDED		NOTRUNCONST
	= REAL-4		TRUNCONST
	CANCEL-CONDITION		
	= NONE		GEN/NOGEN=FAILURE
	= ERROR		NOGEN = ERROR
	= SEVERE-ERROR		NOGEN = SEVERE
	LINKAGE = STD		LINKAGE = STD
	= FOR1-SPECIFIC		= FOR1-SPECIFIC

Table 2-7: SDF command START-FOR1-COMPILER, COMPILER-ACTION form

MODULE-LIBRARY form: Destination of generated object modules

SDF form	First subform	Corresp. COMOPTs
MODULE-LIBRARY = <full-filename 1..54>		MODULE-LIBRARY =plamlib
= *OMF(...)		=*OMF
	DELETE-OLD-CONTENTS =YES	
	=NO	

Table 2-8: SDF command START-FOR1-COMPILER, MODULE-LIBRARY form

LISTING form: Generation of compiler listings

SDF form	First subform	Second subform	Corresp. COMOPTs
LISTING = NO			NOLIST
= STD			Subform defaults
= PARAMETER (...)	OPTIONS = YES		LIST = (OPTIONS)
	= NO		
	SOURCE = NO		
	SOURCE = YES(...)	INSERT-ERROR-WEIGHT = NOTE	LIST = (SOURCE), MSGLEVEL = NOTE
		= WARNING	= WARNING
		= ERROR	= ERROR
	DIAGNOSTICS = NO		
	= YES(...)	MINIMAL-WEIGHT = NOTE	LIST = (DIAG), MSGLEVEL = NOTE
		= WARNING	= WARNING
		= ERROR	= ERROR
	DATA-ALLOCATION-MAP = YES		LIST = (MAP)
	= NO		
	CROSS-REFERENCE = NO		
	= YES		LIST = (XREF)

	EXTERNAL-DICTIONARY = NO		
	= YES		LIST = (ESD)
	ASSEMBLER-CODE = NO		
	= YES		LIST = (OBJECT)
	SUMMARY = YES		LIST = (SUMMARY)
	= NO		
	OPTIMIZED-SOURCE = NO		
	= YES		LIST = (DECOMP)
	SORTING = BY-PROG-UNIT		NOCOLLECT
	= BY-LIST-TYPE		COLLECT = (LIST, LISTFILE)
	LAYOUT = STD		LINECNT=64,EJECT
	= PARAMETER(...)	LINES-PER-PAGE = 64	LINECNT = 64
		= <integer 20..255>	= number
		PAGE-EJECT-STMT = ACCEPTED	= EJECT
		= IGNORED	= NOEJECT
		TEXT-SEPARATOR = '|'	TEXT-SEPARATOR = '|'
		= '!'	= '!'
	OUTPUT = *SYSLST		LIST-OUTPUT = *SYSLST
	= <full-filename 1..54>		= listfilename
	= *STD-FILE		= *STD-FILE
	= *LIBRARY-ELEMENT(...)	LIBRARY = <full-filename 1..54> ELEMENT-PREFIX = *NONE = <alphanum-name 1..38> (...)	= *LIBRARY-ELEMENT (LIBRARY = plamlib, ELEMENT-PREFIX = *NONE = prefix
		VERSION = *UPPER-LIMIT = <alphanum-name 1..24>	(VERSION = version)) = UPPER-LIMIT))

Table 2-9: SDF command START-FOR1-COMPILER, LISTING form

TEST-SUPPORT form: Controlling test facilities

SDF form	First subform	Second subform	Corresp. COMOPTs
TEST-SUPPORT = STD			
= NO			
= PARAMETER (...)	STATEMENT-TABLE = YES		TESTOPT = (STNR)
	= NO		
	TOOL-SUPPORT = NO		SYMTEST = MAP
	= AID		SYMTEST = ALL, OPTIMIZE = NO
	CHECK-CODE = NO		TESTOPT = (STNR)
	= ALL		TESTOPT = ALL
	= YES(...)	PROCEDURE-ARGUMENTS = NO	missing TESTOPT = (ARG)
		= YES	TESTOPT = (ARG)
		ARRAY-BOUNDS = NO	missing TESTOPT = (BOUNDS)
		= YES	TESTOPT = (BOUNDS)
		ARRAY-SUBSCRIPTS = NO	missing TESTOPT = (SUBSCR)
		= YES	TESTOPT = (SUBSCR)
		SUBSTRING-BOUNDS = NO	missing TESTOPT = (STRING)
		= YES	TESTOPT = (STRING)
		BRANCH-STMTS = NO	missing TESTOPT = (CNTRL)
		= YES	TESTOPT = (CNTRL)
		VARIABLE-ASSIGNMENT = NO	missing TESTOPT = (UNDEF)
		= YES	TESTOPT = (UNDEF)
		USER-DEBUG-STMTS = NO	missing TESTOPT = (DEBUG)
		= YES	TESTOPT = (DEBUG)

Table 2-10: SDF command START-FOR1-COMPILER, TEST-SUPPORT form

OPTIMIZATION form: Controlling optimization

SDF command	First subform	Second subform	Corresp. COMOPTs
OPTIMIZATION = NO			OPTIMIZE = NO
= LOW			= 0
= MEDIUM(...)	CONDITIONAL-LOOPS = IGNORED		= 1
	= RISK-OPTIMIZED		= 2
	OPTIMIZE-PROCEDURES = NO		PROCEDURE-OPTIMIZATION = NO, STD
	= YES		= YES
	= SPECIAL		= SPECIAL-ATTEMPTS
= HIGH(...)	CONDITIONAL-LOOPS = IGNORED		OPTIMIZE = 3
	= RISK-OPTIMIZED		= 4
	OPTIMIZE-PROCEDURES = NO		PROCEDURE-OPTIMIZATION = NO
	= YES		= YES, STD
	= SPECIAL		= SPECIAL-ATTEMPTS
	OPTIMIZATION-HINTS = STD		OPTIMIZE = ({3|4}, FUNCTION-SIDEEFFECT = YES PARAMETER-SIDEEFFECT = NO, REORDER = NO)
	= PARAMETER(...)	REORDER-EXPRESSIONS	REORDER
		= YES	= YES
		= NO	= NO
		FUNCTION-SIDEEFFECTS = YES	FUNCTION-SIDEEFFECT = YES
		= NO	= NO
		ARGUMENT-SIDEEFFECTS = NO	PARAMETER-SIDEEFFECT = NO
		= YES	= YES

Table 2-11: SDF command START-FOR1-COMPILER, OPTIMIZATION form

COMPILER-TERMINATION form: Compiler termination conditions

SDF form	First subform	Corresponding COMOPTs
COMPILER-TERMINATION = <u>STD</u>		
= PARAMETER(...)	CPU-LIMIT = <u>NONE</u>	
	= <integer 1..32767>	Limiting the maximum compilation time in the START-PROGRAM command
	MAX-ERROR-WEIGHT = <u>NONE</u>	ERRKILL = <u>FAILURE</u>
	= ERROR	= ERROR
	= SEVERE-ERROR	= SEVERE
	MAX-ERROR-NUMBER = <u>100</u>	MAXERR = <u>100</u>
	= <integer 1..2147483639>	= n

Table 2-12: SDF command START-FOR1-COMPILER, COMPILER-TERMINATION form

MONJV form: Monitoring the compilation by job variables

SDF form	Corresponding COMOPTs
MONJV = <u>*NONE</u>	<u>No</u> definition of a job variable
= <full-filename 1..54>	Definition of a job variable with MONJV = jvname

Table 2-13: SDF command START-FOR1-COMPILER, MONJV form

LANGUAGE form: Specifying the message language

SDF form	Corresponding COMOPTs
LANGUAGE = <u>ENGLISH</u>	LANGUAGE=<u>ENGLISH</u>
= DEUTSCH	LANGUAGE=GERMAN

Table 2-14: SDF command START-FOR1-COMPILER, LANGUAGE form

The last operand of the START-FOR1-COMPILER command is called COMPILER (see section 4.11). This operand is not visible in guided dialog and can only be entered in NO or expert mode.

2.3 Control through compiler options

2.3.1 Entering compiler options

The user specifies the compiler options in one or more COMOPT statements. COMOPT statements can be entered directly on the screen in interactive mode, they can be written to a file or may be contained in the source program file. COMOPT statements can be entered in the following ways:

- The COMOPT statements are read from SYSDTA:

  SYSDTA is the primary assignment:
  In interactive mode the user can enter the COMOPT statements on the screen after starting the compiler. The compiler requests the compiler options explicitly by displaying an asterisk (*) in column 1.
  In batch mode the COMOPT statements are read from the procedure file.
- SYSDTA was assigned to a cataloged file or to a library element:
  The COMOPT statements are read from the file or from the library element. If the assigned file is the source program file, the COMOPT statements must precede the source program.
- SYSDTA was assigned to SYSCMD (compilation procedures):
  The COMOPT statements are read from the procedure file.

The assignment of SYSDTA can be changed with the ASSIGN-SYSDTA command (see section 3.2.1).

Compiler option OPTIONS:

The user specifies the input source for the COMOPT statement(s) in the compiler option OPTIONS (see section 3.3.2).

Section 2.3.3 contains a summary of all compiler options.

Format of a COMOPT statement

```
[*]COMOPT␣option [,option]...
```

- [*]COMOPT is permissible only at the beginning of a COMOPT statement.
- A compiler option *option* comprises a keyword, followed where applicable by an equal sign and one or more option values.

  Example: LIST=(SOURCE, XREF,SUMMARY)
- The names of options and option values can be abbreviated. Starting from the right, as many letters may be omitted as still leave the abbreviated name unique. Additional abbreviated forms also exist; these are summarized in section 2.3.3 and in appendix A.1.
- If any errors occur when a COMOPT line is processed, the options of that line that have already been interpreted remain in effect.
- COMOPT statements apply for precisely one compilation.
- If the compiler options are entered by means of punched cards, only columns 1-72 may be used.

Terminating the COMOPT input

The COMOPT input can be terminated in the following ways:

- `[[*]COMOPT] END`
  as the last of the COMOPT statement lines
- `END`
  as the last keyword in the sequence of specified compiler options
- any desired FORTRAN source program line

Example:

```
*COMOPT LIST=(SOURCE,XREF,SUMMARY)
*COMOPT LISTFILE=(LIST)
*COMOPT END         or *END
```

or:

```
[*]COMOPT␣LIST=(SOURCE,XREF,SUMMARY),LISTFILE=(LIST),END
```

NO prefix

Most compiler options consist of a keyword and a list of option values. The keyword may be preceded by the prefix NO. If it is, the compiler option concerned does not apply to the option values that have been entered; instead it applies to the complementary set of option values that are possible.

Example:

The LIST option value specification is used for controlling the output of listings.

```
*COMOPT NOLIST=(MAP,XREF,ESD)
```

In this case all but the three specified listings will be output.

This principle also covers blank option value lists.

`*COMOPT LIST = ()`	No listings are output.
`*COMOPT NOLIST=()`	All listings are output.

If no option value list is present, the NO prefix switches off the option; without the NO prefix, preset option values are effective.

`*COMOPT NOLIST`	No listings are output.
`*COMOPT LIST`	The standard listings are output.

Explicit request for option values

For certain compiler options, the compiler can be made to explicitly request option values from the user.

Example:

```
*COMOPT SOURCE=/
```

Compilation is interrupted so that the user may assign SYSDTA to the source program file by means of an ASSIGN-SYSDTA command.

2.3.2 Example of simple compilation and program run with compiler options

A simple compilation run is understood here to mean a compilation which, with the exception of a few specified operands, is mainly controlled by preset option values. A simple program run includes linking, loading and execution of the compiled program by means of the dynamic binder loader (DBL).

```
/DEL-SYS-FILE OMF                                                            ( 1)
/START-PROGRAM $FOR1                                                         ( 2)
 %  BLS0500 PROGRAM 'FOR1', VERSION '2.2A00' OF '91-06-05'LOADED.            ( 3)
 %  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG. 1991 ...
  FOR1:    V2.2A00 READY, GIVE COMPILER OPTION
*COMOPT SOURCE=QUELLE.TEST                                                   ( 4)
  FOR1: GIVE COMPILER OPTION
*COMOPT LIST
  FOR1: GIVE COMPILER OPTION                                                 ( 5)
*COMOPT END
  FOR1: COMPILER NOT PRELOADED (BAD LOAD PERFORMANCE)
  FOR1: NO ERRORS DURING COMPILATION OF P. U. TEST                           ( 6)
  END OF F O R 1  COMPILATION; CPU TIME USED:   0.179 SEC
/SET-TASKLIB FOR1MODLIBS                                                     ( 7)
/START-FOR1-PROGRAM                                                          ( 8)
 %  BLS0001 DBL VERSION 070 RUNNING
 %  BLS0517 MODULE 'TEST' LOADED
 BS2000  F O R 1 : FORTRAN PROGRAM "TEST "
 STARTED ON 1991-07-16 AT 14:20:09
 BS2000  F O R 1 : FORTRAN PROGRAM "TEST " ENDED PROPERLY AT 14:20:11      ( 9)
 CPU - TIME USED :       0.0305 SECONDS
 ELAPSED TIME    :       1.9980 SECONDS
```

Explanation of example:

(1) Erases any object modules in the temporary EAM area that were created as a result of previous compilations. This ensures that the dynamic binder loader uses the object module from the current compilation when it is next called.

(2) FOR1 is called with the START-PROGRAM $FOR1 command.

(3) The compiler responds with version number and date. It displays an asterisk and awaits the entry of compiler options.

(4) The entry COMOPT SOURCE=QUELLE.TEST causes the source program to be read in from the cataloged file QUELLE.TEST.
If the compiler option LIST is specified, the preset option values of the LIST option apply. In this case the following listings are output to SYSLST (see section 4.5):
- source listing
- diagnostic listing
- map listing
- summary listing
- options listing

(5) Displaying an asterisk, the compiler awaits further options until [COMOPT] END is entered. [COMOPT] END terminates the entry of compiler options. The compiler compiles the source program that has been read in.

(6) The compiler reports that FOR1 has not been preloaded. No errors were found during compilation. The CPU time used for compilation is displayed.

(7) If the modules of the runtime system are not included in the TASKLIB of the system, the user-own module library must be assigned as TASKLIB before the binder loader DBL is called.
The user-own module library here is called FOR1MODLIBS. If the FOR1 runtime modules are present in the TASKLIB of the system, then this statement may be omitted.

(8) Unless specified otherwise, generated object modules are output to the temporary EAM area (*OMF). The START-FOR1-PROGRAM command calls the dynamic binder loader which links, loads and starts the object modules. The temporary EAM area exists only for the duration of the task, i.e. it is deleted at the end of the task. If the object modules are to be made permanently available, there are two ways of doing this:
- The object modules are output directly to a PLAM library specified with the MODULE-LIBRARY option (see section 4.3).
- The object modules are entered in an object module library by the library program LMS (see section 4.5).

(9) After the program has executed an end message appears, displaying the CPU time and total time used.

2.3.3 Summary: Compiler options and corresponding SDF operands

All the compiler options are summarized in the following table.
The function of the prefix NO is explained in section 2.3.1.

[*]COMOPT...	**Code**	**Meaning**	**a) Description in section** **b) Corresponding SDF operand form** **c) Corresponding SDF operand subform**
CCOM='comment-marks'	CC	Identifier for the compilation of comment lines (max. 60 characters)	a) 4.1.2.1 b) SOURCE-PROPERTIES c) COMPILEABLE-COMMENTS

CODE= { ([SOURCE= { EBCDIC | ISO | BCD }]
[,UPD= { EBCDIC | ISO | BCD }])
{ EBCDIC | ISO | BCD } }
	CO	Code of source program or change lines	a) 4.1.2.9

{ COLLECT=({ LIST | LISTFILE[,LIST] }) NOCOLLECT }	COL	Collects listings of the same type for several different program units	a) 4.6.2.3 b) LISTING c) SORTING
[NO[COMPATIBLE=]] { BGFOR | BS3FOR }	COMPAT, COM, BGF, BS3	Avoiding incompatibilities between FOR1 compiler and Siemens BGFOR or Telefunken BS3 FORTRAN compilers	a) 4.2.2.8
{ DIALOG [= { param (param[,param]) (param[,param,[param]]) }] NODIALOG } param:= { ! | ? | @ | % | # | $ D | E E[DIT]={ALL | FIRST | NO} }	D ND	DIALOG activates Interactive Analysis	a) 3.6.2 b) DIALOG LANGUAGE c) DIALOG-INTERRUPT Default of dialog prefix: @
DIALOG-[SAVE]= ([{ *STD [-FILE] [FILE=] { file | plamspecification } }] [,[INCLUDE-EXPANSIONS=] { NO | YES }]) plamspecification:= [LIBRARY-ELEMENT] ([LIBRARY=]plamlib, [ELEMENT=]name[([VERSION=] { *UPPER-LIMIT | version })])	DIALOG-	Outputs results to a file	a) 3.6.4 b) DIALOG c) SAVE-FILE

continued ▷

continued

[*]COMOPT...	**Code**	**Meaning**	**a) Description in section** **b) Corresponding SDF operand form** **c) Corresponding SDF operand subform**
[NO]EJECT	EJ	Form feed	a) 4.6.2.7 b) LISTING c) LAYOUT, PAGE-EJECT-STMT
END		Terminator for compiler options	a) 2.3.1

ERRKILL= {E[RROR]
S[EVERE]
F[AILURE]}
	EK	Degree of error at which compilation is to be terminated	a) 4.8.2
			b) COMPILER-TERMINATION
			c) MAX-ERROR-WEIGHT

[NO]EXPAND	EX	Controlling the output of the text inserted by the %INCLUDE statement	a) 4.6.2.8 b) LISTING c) TYPE, SOURCE, INCLUDE-EXPANSION
[NO]EXPUNDERFLOW	EU	Setting the appropriate program mask	a) 4.1.2.5 b) SOURCE-PROPERTIES c) EXPONENT-UNDERFLOW

FORTRAN90-CHECK= {YES
NO}
	F90	Checks the source program for language extensions incompatible with Fortran90	a) 4.1.2.8
			b) SOURCE-PROPERTIES
			c) FORTRAN90-CHECK

{FPOOL [={(fpoolname[,fpoolname]...) fpoolname}] NOFPOOL}	F	Controlling FPOOL processing	a) 12.1.2 b) FPOOL-LIBRARY
{GEN NOGEN [={E[RROR] S[EVERE] F[AILURE]}]}	G	Controlling generation of the object program	a) 4.2.2.5 b) COMPILER-ACTION c) CANCEL-CONDITION
[NO]IMPLICIT	I	Implicit type associations forbidden by NOIMPLICIT	a) 4.1.2.4 b) SOURCE-PROPERTIES c) IMPLICIT-DECLARATION
INCLUDE [-LIBRARY]= {*NO (filename1[,filename2][,...])}	INC	The INCLUDE-LIBRARY option defines hierarchical access to libraries	a) 3.5.3 b) INCLUDE-LIBRARY
LANGUAGE={ENGLISH|GERMAN}	LA	Language in which FOR1 messages are output from the moment the options are read in	a) 4.10.2 b) LANGUAGE

LINKAGE= {STD
FOR1-SPECIFIC}
	LNK	Generation of standard linkage objects	a) 4.2.2.6
			b) COMPILER-ACTIONS
			c) LINKAGE

LINECNT= {64
number}
	LC	Number of lines per page for compiler listings	a) 4.6.2.6
			b) LISTING
			c) LAYOUT, LINES-PER-PAGE

LINEEND='end-marks'	LE	Identifier for end-of-line	a) 4.1.2.2 b) SOURCE-PROPERTIES c) LINE-END-COMMENTS

continued ▷

continued

[*]COMOPT...	**Code**	**Meaning**	**a) Description in section** **b) Corresponding SDF operand form** **c) Corresponding SDF operand subform**
[NO]LIST [=([listentry][,...])] listentry:={ALL|MIN|NONE| OPTIONS| SOURCE|DIAG|ESD| MAP XREF|ATR|OBJECT|DECOMP| SUMMARY|CHANGE}	L	Selection of listings to be output on SYSLST	a) 4.6.2.2 b) LISTING c) TYPE CHANGE: b) DIALOG c) LOG-CHANGED-LINES
[NO]LISTFILE [=[listfilename] [({listentry | LIST} [,...])]]	LF	Selection of listings to be output to a cataloged file; see also LIST	a) 4.6.2.5 b) LISTING c) OUTPUT, TYPE
LIST-OUT[PUT]= {listfilename *STD[-FILE] *SYSLST [*LIBRARY-ELEMENT] ([LIBRARY=]plamlib [,[ELEMENT[-PREFIX]=]{prefix | *NONE} [([VERSION=]{version | *UPPER-LIMIT})]]])}	LIST-	Controlling the listing output to PLAM library or cataloged file	a) 4.6.2.4 b) LISTING c) OUTPUT
MAXERR= {100 | number}	ME	Number of messages at which compilation is terminated	a) 4.8.2 b) COMPILER-TERMINATION c) MAX-ERROR-NUMBER
MODULE[-LIBRARY]= {*OMF | plamlib}	MOD	Controlling the output of object modules to EAM file or PLAM library	a) 4.3.2 b) MODULE-LIBRARY

MSGLEVEL= {N[OTE]|W[ARNING] | E[RROR]
(SOURCE={N|W|E}
[,DIAG={N|W|E}])
(DIAG={N|W|E}
[,SOURCE={N|W|E}])}
	MSG	Degree of error at which messages are to be output	a) 4.6.2.1
			b) LISTING
			c) SOURCE, DIAGNOSTICS
			INSERT-ERROR-WEIGHT
			MINIMAL-WEIGHT

{OBJECT [= {SHARE | *}] NOOBJECT}	OBJ, O	Controlling the generation of object modules and defining the output medium; generating a shareable code portion	a) 4.2.2.1 b) COMPILER-ACTION c) SHAREABLE-CODE

continued ▷

continued

[*]COMOPT...	**Code**	**Meaning**	**a) Description in section** **b) Corresponding SDF operand form** **c) Corresponding SDF operand subform**
OPT[IMIZE]= { NO 0 1 2 3 (3,parameter[,...]) 4 (4,parameter[,...]) } parameter:= { FUNC[TION-SIDEEFFECT]= { YES NO } PARAM[ETER-SIDEEFECT]= { YES NO } REORDER= { YES NO } }	OPT	Optimization control	a) 9.2.2 b) OPTIMIZATION c) CONDITIONAL-LOOPS OPTIMIZATION-HINTS
OPTIO[NS]= { * file lib(name) plamspecification / + } plamspecification: see SOURCE option	OPTIO	File containing further compiler options	a) 3.3.2
OUTPUT[= { filename (filename[,expand]) ([WS=]filename [,OS=filename] [,expand]) ([WS=](filename [,expand]) [,OS=(filename [,expand])]) }]	OU	Output to file for Interactive Analysis: original program or workfile	a) 3.6.4 b) DIALOG c) SAVE-FILE
[NO]PAD	P	Input records are padded with blanks	a) 4.2.2.10 b) Default is PAD. If LANGUAGE-STANDARD= ANS77, NOPAD is set.
PROCEDURE[-OPTIMIZATION]= { STD NO YES SPECIAL[-ATTEMPTS] }	PR	Controlling optimization when calling procedures	a) 9.2.3 b) OPTIMIZATION c) OPTIMIZE-PROCEDURES
REAL= { 4 (4) 8 (8) 16 (16) }	R	Defining the minimum length for REAL and COMPLEX entities	a) 4.2.2.3 b) COMPILER-ACTION c) MINIMAL-PRECISION

continued ▷

continued

[*]COMOPT...	**Code**	**Meaning**	**a) Description in section** **b) Corresponding SDF operand form** **c) Corresponding SDF operand subform**

SAVE-CONSTANT={NO
YES}
	SA	Transferring a copy, if constant as actual argument	a) 4.1.2.7
			b) SOURCE-PROPERTIES
			c) SAVE-CONSTANT

SHARE-LIB[RARY]= {*MODULE-LIBRARY
plamlib}
	SH	Storing the shareable object modules in a separate PLAM library	a) 4.2.2.2
			b) COMPILER-ACTION
			c) SHAREABLE-CODE

SOURCE[={* file lib(name) plamspecification / + (PRIMARY)}] plamspecification:= [*LIBRARY-ELEMENT]([LIBRARY=]plamlib, [ELEMENT=]name[([VERSION=] {*HIGHEST-EXISTING version})])	SRC *LIB	Location of source program	a) 3.2.3 b) SOURCE c) *LIBRARY-ELEMENT
SOURCE-FORMAT={FIXED|FREE}	SOURCE-SF	Compiling an unformatted source program	a) 4.1.2.6 b) SOURCE-PROPERTIES c) SOURCE-FORMAT

ST[AN]D[ARD-CHECK]= {ANS77
NO}
	STD	Checking a FOR1 source program for deviations from ANS FORTRAN 77	a) 4.1.2.3
			b) SOURCE-PROPERTIES
			c) LANGUAGE-STANDARD

[NO]SUPPLIEDBOUND	SUP, SB	Interpretation of dimension entry 1 as * in subprograms	a) 4.2.2.9 b) Default is NOSUPPLIEDBOUND.
SYMTEST={NO|MAP|ALL}	SYM	Generation of LSD information for symbolic debugging with AID	a) 7.6.1 b) TEST-SUPPORT c) TOOL-SUPPORT
[NO]TESTOPT [=([testparameter][,...])] testparameter:={ALL|STNR|ARG| BOUNDS|SUBSCR| STRING|CNTRL|UNDEF| DEBUG}	TO	Selection of debug functions	a) 7.3 b) TEST-SUPPORT c) CHECK-CODE

TEXT-SEPARATOR= {'|'
'!'}
	TEX	Vertical lines in lists represented as "|" or "!"	a) 4.6.2.9
			b) LISTING
			c) LAYOUT

[NO]TRUNCONST	TC	Truncation of REAL constants without exponent	a) 4.2.2.4 b) COMPILER-ACTION c) CONSTANT-PRECISION

continued ▷

continued

[*]COMOPT...	**Code**	**Meaning**	**a) Description in section** **b) Corresponding SDF operand form** **c) Corresponding SDF operand subform**
`UNIT= ({READ|WRITE|PRINT|PUNCH}=nn [,{READ|WRITE|PRINT|PUNCH}=nn]..)`	U	Assigning file numbers to input/output statements	a) 4.2.2.7
`UPD [={*|file|lib(name)|/|+}]`	UPD	Location of update file	a) 3.4

Table 2-15: Summary: Compiler options and corresponding SDF operands

2.4 Summary: Compile time statements in the source program

Compile time statements control output of the source listing, input of the source text and processing of FPOOL subprograms.

Compile time statements can also be located in a FORTRAN source program, in addition to the FORTRAN language statements. These compile time statements are treated like FORTRAN statements.

A prefixed percent sign identifies compile time statements (with the exception of *DELETE). The compiler treats these statements just like FORTRAN language statements, assigning them a statement number, to which any error message might also refer.

Compile time statements can be placed anywhere within the FORTRAN source program. A compile time statement is valid only within the program unit in which it is specified.

Compile time statements, like FORTRAN language statements, may also have continuation lines. Continuation lines are marked by means of an entry in column 6.

The individual compile time statements are described in various sections of this manual, depending on the subject with which they are associated.

Statements for controlling the source listing

Format	Meaning	Section number
%EJECT	Form feed	4.6.3.2
%EXPAND $\begin{Bmatrix}\text{ON}\\ \text{OFF}\end{Bmatrix}$	Controlling the output of texts inserted by means of the %INCLUDE statement	4.6.3.1
%SPACE n	Insertion of blank lines in source listing	4.6.3.3
%TITLE [’text’]	Output of specified text in header line of source listing	4.6.3.4

Statement for inserting texts in the source program

Format	Meaning	Section number
$\text{\%INCLUDE} \begin{Bmatrix}\text{name} \\ \text{lib(name)}\end{Bmatrix}$ $[\text{,CODE } [= \begin{Bmatrix}\underline{\text{EBCDIC}} \\ \text{ISO} \\ \text{BCD}\end{Bmatrix}]\ ]$ [,parameter-list]	Insertion of source program lines in the FORTRAN source program	3.5.1

Statement for temporary modification of source program lines

Format	Meaning	Section number
*DELETE i1 [,i2]	Temporary deletion of source program lines	3.4

Statements for processing FPOOL subprograms

Format	Meaning	Section number
%FPOOL fpoolname [(fpoolfct[,...])]	Specification of file with interface descriptions of all listed functions	12.1.3
%NOFPOOL (fpoolfct [,fpoolfct,...])	All functions are excluded from FPOOL processing	12.1.3

3 Source program input

The following topics are discussed in this chapter:

Section 3.1, "Creating the source program", describes the possibilities for creating a source program which FOR1 is to read. A source program may be contained in a file or be entered directly. Interactive analysis is available as a tool for direct input.

Section 3.2, "Defining the input location of the source program", describes the possible ways of informing the compiler from where the source program is to be read. The source program can be assigned to the system file SYSDTA or specified in the compilation operands (by way of the SOURCE compiler option or the SDF command START-FOR1-COMPILER).

Section 3.3, "Defining the input location of the compiler options", describes where the compiler can find compiler options that may be present. Compiler options can be made available via the system file SYSDTA or with the aid of the OPTIONS compiler option.

Section 3.4, "Temporary updating of a source program: UPD compiler option", describes how the user can change the character string read from the source program file by specifying change lines for a special compilation process.

Section 3.5, "Inserting source program lines", describes how source program lines can be inserted during the compilation run. The compile time statement %INCLUDE, the SDF operand INCLUDE-LIBRARY and the INCLUDE-LIBRARY compiler option are available.

Section 3.6, "Entering the source program with Interactive Analysis", describes how the user can enter, interactively analyze and correct a source program by means of the DIALOG compiler option or the SDF operand DIALOG.

3.1 Creating the source program

A source program may be contained in a source program file or be entered directly. Interactive Analysis is available as a tool for direct input.

3.1.1 Creating a source program file

Normally a FORTRAN source program is created using an editor, and stored in a file. The FOR1 compiler reads the source program from this file.
The source program remains accessible in the file and is available for changes and further compilation runs after the first compilation. FOR1 can process source programs from SAM and ISAM files.

A source program file can exist in different forms:

- as a cataloged file in the form of a SAM or ISAM file
- as an element of a library
  Programs are stored in compressed form in libraries.
  The input and output of PLAM library elements of type S can be controlled by compiler options or SDF operands.
- as a group file (GAM file; GAM = Group Access Method)
  The term group file refers to a set of ISAM file records whose keys begin with a particular string of characters, i.e. the name of that group file (see section 3.2.3, example 3).

Fig. 3-1 shows the various options for generating a source program file:

In batch mode, a file for source programs stored on floppy disk or punched cards can be generated with the aid of the DATA command (see "User Commands (ISP Format)" manual [11]).

In interactive mode, a source program file can be generated as follows:
- using the file editor EDT (see "EDT Statements" manual" [20]).
  Though the EDT is dialog-oriented, it can also be used in batch mode. Source program files can be stored in a PLAM library using the EDT.
- using Interactive Analysis of FOR1 by saving the directly entered source program (see section 3.6).

This figure is not any longer available for the online pdf.

Fig. 3-1: Creating a source program file

Copying source programs from magnetic disk or magnetic tape

If the source program is already available on magnetic disk or tape and is to be transferred to a file, BS2000 offers the following facilities:

- The COPY-FILE command for copying files
- File conversion programs
  These are primarily important when it comes to data structures which deviate from the standard case (e.g. record lengths greater than 256 characters).
- The software product ARCHIVE
  ARCHIVE stores files on magnetic tape or disks and permits reconstruction of files with the aid of these backup copies (see "ARCHIVE" manual [4]).
- The software product PERCON
  PERCON transfers data from one data medium to another or to several media of different types. Data can be edited simultaneously and the file attributes modified (see "PERCON" manual [36]).

Storing a source program file

Source program files can be stored in PLAM libraries with the aid of the library management system LMS or with the aid of file editor EDT (as of Version 16.1A).

Modifying a source program file

Source program files may be changed in the following ways:
- EDT (see "EDT Statements" manual [20])
- Interactive Analysis (see section 3.6; for an example of modifications within the framework of Interactive Analysis see section 3.6.8).

3.1.2 Direct input of the source program

A source program can be input directly into the compiler, i.e. without intermediate storage in a cataloged file.

To permit this, FOR1 must be called with the command START-PROGRAM $FOR1 or with the command START-FOR1-COMPILER with no SOURCE operand specified. The source program can then be input. Input of the source program must concluded with the characters "/*" in columns 1 and 2.

Example:

The source program is read in from the terminal via SYSDTA. Source program input is terminated by "/*".

```
/START-PROG $FOR1       bzw.  /START-FOR1-COMPILER
% BLS0500 PROGRAM 'FOR1' VERSION '2.2A00' OF '91-06-05' LOADED
% BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG. 1991 ...
FOR1: V2.2A00 READY, GIVE COMPILER OPTION
*      PROGRAM TEST
   .
   .
   .
*      END
*/*
```

In the case of direct input, the source program is only available for one compiler run. Error correction requires that the entire source program be reentered.

Interactive Analysis is advisable for direct input (see section 3.6) since the FOR1 compiler stops when format errors such as keying errors occur. Built-in file editing functions permit immediate correction of these errors, and compilation can continue. A further advantage offered by Interactive Analysis is that the corrected source program can be saved in an external file.

3.2 Defining the input location of the source program

A source program can be input directly to the compiler or may be contained in a file.

The user must inform the compiler of the input location of the source program, i.e. from where the compiler is to read the source program. The following possibilities exist:

– The compiler is to read the source program from the system file SYSDTA. To permit this, before FOR1 is called SYSDTA is assigned to the source program file by means of the ASSIGN-SYSDTA command or, in the case of direct input, it is assigned to the data display terminal. Compiler options contained in the source program file must precede the source program text.

– The user specifies the input location of the source program in the SDF operand SOURCE or in the SOURCE compiler option.

3.2.1 ASSIGN-SYSDTA command

Operating System BS2000 uses standardized system files for performing inputs and outputs. SYSDTA is system file for inputs.

BS2000 provides a standard assignment, the "primary assignment". The primary assignment is the data display terminal in interactive mode and the spoolin file in batch mode. The primary assignment can be changed.

If FOR1 is to read source programs or compiler options via SYSDTA, the user must ensure that SYSDTA is correctly assigned before calling FOR1. The assignment of SYSDTA is changed with the ASSIGN-SYSDTA command.

```
ASSIGN-SYSDTA

 TO-FILE = <full-filename 1..54> / *LIBRARY-ELEMENT(...) / *PRIMARY /
           *SYSCMD / *DISKETTE(...)

   *LIBRARY-ELEMENT(...)
     |   LIBRARY = <full-filename 1..51>
     |
     |  ,ELEMENT = <full-filename 1..38>(...)
     |
     |    <full-filename 1..38>(...)
     |     |  VERSION = *STD / <text 0..10>
     |     | ,TYPE = STD / D / M

   *DISKETTE(...)
     |   UNIT = *ANY / <alphanum-name 2..2>
     |
     |  ,FILE-NAME = <name 1..8>
     |
     |  ,VOLUME = list-poss(10): <alphanum-name 1..6>
```

TO-FILE =
Input source to which SYSDTA is to be assigned.

TO-FILE = <full-filename 1..54>
Name of the file to which SYSDTA is to be assigned. The file must possess the following attributes:
– it must already have been cataloged
– it must have variable-length records
– SAM or ISAM access method
– start of ISAM key: byte 5
– length of ISAM key: 8 bytes

TO-FILE = *LIBRARY-ELEMENT(...)

LIBRARY = <full-filename 1..51>
Name of an LMS library

ELEMENT = <full-filename 1..38> (...)
Name of an element in the specified library. Hyphens are also permitted, but not as the last character. The total length of the library name - without catalog ID and user ID - plus element name must not exceed 39 characters.

VERSION = *STD / <text 0..10>
Addition of the version to the element name

VERSION = *STD
Latest version

TYPE = STD / D / M
Type of the element
The default value is element type S (source programs)

D	Element type D (text data)
M	Element type M (macro)

TO-FILE = *PRIMARY
Resets SYSDTA to the primary assignment.

TO-FILE = *SYSCMD
Combines SYSDTA and SYSCMD, i.e. the system reads both commands and data via SYSCMD.

TO-FILE = *DISKETTE (...)

UNIT = *ANY / <alphanum-name 2..2>
Mnemonic device name of a floppy disk drive.
SYSDTA is assigned to the specified floppy disk drive.

FILE-NAME = <name 1..8>
Name of the floppy disk file

VOLUME = list-poss(10): <alphanum-name 1..6>
Volume serial number of the floppy disk. Up to 10 volume serial numbers are allowed.

A detailed description of the ASSIGN-SYSDTA command is given in the "User Commands (SDF Format)" manual [12].

After compilation, it is advisable to reassign SYSDTA. The primary assignment is restored by entering ASSIGN-SYSDTA TO-FILE=*PRIMARY.

Restriction:

It is not possible to read source program and change lines from SYSDTA simultaneously.

Examples

Example 1: Reading from a file

Reading in the source program from file QUELL.MAT:

```
/ASS-SYSDTA TO-FILE=QUELL.MAT
/START-PROG $FOR1
/ASS-SYSDTA TO-FILE=*PRIMARY
```

FOR1 cannot find any COMOPT line in QUELL.MATT containing the SOURCE compiler option and therefore assumes SYSDTA to be the location of the source program (default).

Example 2: Reading from a library

Reading a source program from the LMS library LMS.BIBL; ELE is the element name.

```
/ASS-SYSDTA TO-FILE=*LIB-ELEM(LIB=LMS.BIBL,ELEM=ELE)
/START-PROG $FOR1
/ASS-SYSDTA TO-FILE=*PRIMARY
```

Example 3: Two groups of compiler options in a file

Two groups of compiler options are contained in the file OPT. Each group contains the location of the source program and is concluded by END.

Contents of OPT:

```
COMOPT SOURCE=SRC.PROG1, ... ,END
COMOPT SOURCE=SRC.PROG2, ... ,END
```

FOR1 is to read the compiler options from SYSDTA; the input location of the source program is specified in the SOURCE compiler option.

Compilation:

```
/ASS-SYSDTA OPT
/START-PROG $FOR1
/START-PROG $FOR1
/ASS-SYSDTA *PRIMARY
```

The first compilation uses the first set of options, the second compilation uses the second set. SYSDTA must not be reassigned between these compilations. Positioning is thus retained.

3.2.2 SDF operand SOURCE

The SDF operand SOURCE can be used to inform the compiler of the input location of a source program file.

```
START-FOR1-COMPILER

 SOURCE = *SYSDTA / <full-filename 1..54> / *LIBRARY-ELEMENT(...)

   *LIBRARY-ELEMENT(...)
     |
     |  LIBRARY = <full-filename 1..54>
     | ,ELEMENT = <full-filename 1..41>(...)
     |     | VERSION = *HIGHEST-EXISTING / <alphanum-name 1..24>
```

The SDF operands and the corresponding compiler options are shown in Table 2-2.

3.2.3 SOURCE compiler option

The SOURCE compiler option can be used to inform the compiler of the input location of the source program.

The following default applies:
If the SOURCE compiler option is missing or if it is specified without an option value, the source program is read from SYSDTA.

```
             ⎡             ⎡*                ⎤ ⎤
             ⎢             ⎢file             ⎥ ⎥
             ⎢             ⎢lib(name)        ⎥ ⎥
             ⎢SOURCE   [=  ⎨                 ⎬]⎥
[*]COMOPT    ⎨             ⎢plamspecification⎥ ⎬
             ⎢             ⎢/                ⎥ ⎥
             ⎢             ⎣+                ⎦ ⎥
             ⎢                                 ⎥
             ⎣SOURCE    =(PRIMARY)             ⎦
```

*	The source program is read from SYSDTA. The default is SOURCE=*. If the source program is input directly, the entry must be concluded with the characters "/*" in columns 1 and 2 (see section 3.1.2).
file	Name of a cataloged file in which the source program is located. Maximum length including catalog ID and user ID: 54 characters.

lib(name)

lib
Name of a PLAM library or GAM file, from which the source program is read. *lib* is first interpreted as a PLAM library and then as a GAM file. Including catalog ID and user ID, *lib* may be up to 54 characters in length.

name
Name of a PLAM library element or group key of a GAM file.
- PLAM library element: *name* may be up to 8 characters long.
- Group key: *name* may be up to 12 alphanumeric characters long and must be shorter than the group key.

plamspecification
Specification of a PLAM library element

```
[*LIBRARY-ELEMENT] ([LIBRARY=]plamlib,
                                               ⎧*HIGHEST-EXISTING⎫
                           [ELEMENT=]name[([VERSION=]⎨                 ⎬)])
                                               ⎩version          ⎭
```

plamlib
Name of a PLAM library. Maximum length including catalog ID and user ID: 54 characters.

name
Name of a PLAM library element. The name may consist of up to 64 characters.

version
Version designation of the PLAM library element. A version designation may be up to 24 characters in length.

<u>*HIGHEST-EXISTING</u>
If more than one version of an element exists, the compiler reads the highest existing version by default.

/ When the the compiler has read all the compiler options, an interrupt occurs returning control to system mode. The compiler issues the following message:

```
FOR1: ASSIGN SYSDTA TO READ SOURCE
```

SYSDTA can now be reassigned with the ASSIGN-SYSDTA command. After the user has entered the RESUME-PROGRAM command, the compiler reads the source program from SYSDTA and commences the compilation.

+ When the compiler has read all the compiler options, it issues the following message:

```
FOR1: GIVE SOURCE FILE SPECIFICATION - OR ?
```

Now the user can enter the location of the source program via the primary assignment of SYSDTA, in interactive mode at the terminal.

The following locations can be specified:

```
⎧*                ⎫
⎪(SYSCMD)         ⎪
⎪(PRIMARY)        ⎪
⎨(CARD)           ⎬
⎪file             ⎪
⎪lib(name)        ⎪
⎩plamspecification⎭
```

*
The source program is read from SYSDTA. No reassignment of SYSDTA takes place. If a location was specified with the ASSIGN-SYSDTA command, the source program is read from there.

(SYSCMD)
The system file SYSDTA is combined with SYSCMD. The source program is read from SYSCMD.

(PRIMARY)
The system file SYSDTA is returned to its primary assignment. The source program is read from the primary assignment of SYSDTA, i.e. from the terminal.

(CARD)
The system file SYSDTA is assigned a card reader. The source program is read from the card reader.

file
Name of the file from which the source program is to be read (for description, see above).

lib(name)
Specification of the library element or section of the GAM file from which the source program is to be read (for description, see above).

plamspecification
Specification of the PLAM library element from which the source program is to be read (for description, see above).

SOURCE=(PRIMARY)
The source program is input from the terminal in WRITE/READ mode, irrespective of the SYSDTA assignment. The specification is only effective in conjunction with COMOPT DIALOG (see section 3.6) and is necessary if a source program is to be written in interactive mode during execution of a procedure.

Note

Only in the case of group files (GAM files, see example 3) may source programs and change lines be contained in the same file.

Examples:

Example 1: Reading from cataloged file

Reading the source program from the cataloged file QUELL.MAT

```
/START-PROG $FOR1
*COMOPT SOURCE=QUELL.MAT
*END
```

Example 2: Reading from library

Reading the source program from the library PROGLIB. Element name: PROGA

```
/START-PROG $FOR1
*COMOPT SOURCE=PROGLIB(PROGA)
*END
```

Example 3: Reading from group file (GAM file)

The group file QUELL.MAT contains three source programs:

```
GGD     (record key 19010000-19900000)
INV     (record key 20010000-20570000)
MAT1    (record key 21010000-21500000)
```

Group files are always ISAM type files. The record keys can be generated in EDT using the SET statement (see "EDT Statements" manual [20]).

The program INV consists precisely of those lines whose keys begin with the string 20; this program represents a group file.

Program INV can be compiled using the following statements:

```
/START-PROG $FOR1
*COMOPT SOURCE=QUELL.MAT(20)
*END
```

From the group file QUELL.MAT, the compiler only takes into account those records whose keys begin with 20. Leading zeros must be specified.

Example 4: Reading a PLAM library element

With the aid of LMS, a FORTRAN source program is stored as a PLAM library element of type S, after which FOR1 is called. The PLAM library element ELEM1, version 001, is specified in the SOURCE option.

```
/START-PROG $LMS
$LIB BIBL, BOTH, ANY
$ADDS QUELLE.TEST>ELEM1/001
$END

/START-PROG $FOR1
*COMOPT SOURCE = (BIBL,ELEM1(001)), END
```

Example 5: Compiler options and source program in different files

The options for compiling a program are contained in the file OPT. The location of the source program is not to be specified until option processing time. The source program is in the file QUELL.MAT.

1. Request input location of source program with `COMOPT SOURCE=+`

   Contents of file OPT:

   ```
   COMOPT SOURCE=+
   COMOPT LIST=(ALL)
   END
   ```

   Compilation:

   ```
   /ASS-SYSDTA TO-FILE=OPT
   /START-PROG $FOR1
   FOR1: GIVE SOURCE FILE SPECIFICATION-OR?
   QUELL.MAT
   ```

In this case the user can request information about the input possibilities by entering a question mark after the input prompt.

2. Request input location of source program with `COMOPT SOURCE=/`

   Contents of file OPT:

   ```
   COMOPT SOURCE=/
   COMOPT LIST=(ALL)
   END
   ```

   Compilation:

   ```
   /ASS-SYSDTA TO-FILE=OPT
   /START-PROG $FOR1
   FOR1: ASSIGN SYSDTA TO READ SOURCE
   /ASS-SYSDTA TO-FILE=QUELL.MAT
   /RESUME-PROG
   ```

3.3 Defining the input location of the compiler options

The user specifies the compiler options in one or more COMOPT statements. Compiler options can be entered in the following ways:

- The user can input the COMOPT statement(s) directly by calling the compiler without previously reassigning the system file SYSDTA by means of the ASSIGN-SYSDTA command. The compiler explicitly requests the compiler options by displaying an asterisk (*) in column 1.
- The user can write the COMOPT statement(s) to a file and input them via this file. This file can be the source program file or a user-own file. This file is assigned to the system file SYSDTA by the ASSIGN-SYSDTA command before the compiler is called. The compiler reads from the system file SYSDTA. Compiler options that are located in a source program file must precede the source program text.
- The user can specify the input location of the COMOPT statement(s) in the OPTIONS compiler option (see section 3.3.2). There is no SDF operand corresponding to the OPTIONS compiler option.

3.3.1 ASSIGN-SYSDTA command

The ASSIGN-SYSDTA command is described in section 3.2.1, "ASSIGN-SYSDTA command".

3.3.2 OPTIONS compiler option

The OPTIONS compiler option informs the compiiler of the input location of the compiler options. There is no SDF operand corresponding to the OPTIONS compiler option.

The following default applies:
If the OPTIONS compiler option is missing or if it is specified without an option value, the compiler options are read from SYSDTA.

```
[*]COMOPT    | OPTIONS   [= ⎡*                ⎤
             |              ⎢file             ⎥
             |              ⎢lib(name)        ⎥
             |              ⎨                 ⎬]
             |              ⎢plamspecification⎥
             |              ⎢/                ⎥
             |              ⎣+                ⎦
```

<u>*</u>	The compiler options are read from SYSDTA.
file	Name of a cataloged file in which the compiler options are located. Maximum length including catalog ID and user ID: 54 characters.

lib(name)

lib
Name of a PLAM library or GAM file, from which the compiler options are read. *lib* is first interpreted as a PLAM library and then as a GAM file. Including catalog ID and user ID, *lib* may be up to 54 characters in length.

name
Name of a PLAM library element or group key of a GAM file.
- PLAM library element: *name* may be up to 8 characters long.
- Group key: *name* may be up to 12 alphanumeric characters long and must be shorter than the group key.

plamspecification

Specification of a PLAM library element

```
[*LIBRARY-ELEMENT] ([LIBRARY=]plamlib,
                                               ⎧*HIGHEST-EXISTING⎫
                            [ELEMENT=]name[([VERSION=]⎨                 ⎬)])
                                               ⎩version          ⎭
```

plamlib
Name of a PLAM library. Maximum length including catalog ID and user ID: 54 characters.

name
Name of a PLAM library element. The name may consist of up to 64 characters.

version
Version designation of the PLAM library element. A version designation may be up to 24 characters in length.

<u>*HIGHEST-EXISTING</u>
If more than one version of an element exists, the compiler reads the highest existing version by default.

/
When the the compiler has read all the compiler options, an interrupt occurs returning control to system mode. The compiler issues the following message:

```
FOR1: ASSIGN SYSDTA TO READ OPTION
```

SYSDTA can now be reassigned with the ASSIGN-SYSDTA command. After the user has entered the RESUME-PROGRAM command, the compiler reads the compiler options from SYSDTA and commences the compilation.

+ When the compiler has read all the compiler options, it issues the following message:

```
FOR1: GIVE OPTION FILE SPECIFICATION - OR ?
```

Now the user can enter the location of the compiler options via the primary assignment of SYSDTA, in interactive mode at the terminal.

The following locations can be specified:

```
⎧*                ⎫
⎪(SYSCMD)         ⎪
⎪(PRIMARY)        ⎪
⎨(CARD)           ⎬
⎪file             ⎪
⎪lib(name)        ⎪
⎩plamspecification⎭
```

*
The compiler options are read from SYSDTA.No reassignment of SYSDTA takes place. If a location was specified with the ASSIGN-SYSDTA command, the compiler options are read from there.

(SYSCMD)
The system file SYSDTA is combined with SYSCMD. The compiler options are read from SYSCMD.

(PRIMARY)
The system file SYSDTA is returned to its primary assignment. The compiler options are read from the primary assignment of SYSDTA, i.e. from the terminal.

(CARD)
The system file SYSDTA is assigned a card reader. The compiler options are read from the card reader.

file
Name of the file from which the compiler options are to be read (for description, see above).

lib(name)
Specification of the PLAM library element or section of the GAM file from which the compiler options are to be read (for description, see above).

plamspecification
Specification of the PLAM library element from which the compiler options are to be read (for description, see above).

Examples:

Example 1: Compiler options from cataloged file

Reading the compiler options from the cataloged file OPT. The source program is located in the cataloged file QUELLE.

Contents of OPT:

```
COMOPT SOURCE=QUELLE
COMOPT LIST=(ALL)
END
```

Compilation:

```
/START-PROG $FOR1
*COMOPT OPTIONS=OPT
*END
```

Example 2: Compiler options from library

Reading the compiler options from the library PROGLIB. Element name: OPT. See example 1 for contents of OPT.

```
/START-PROG $FOR1
*COMOPT OPTIONS=PROGLIB(OPT)
*END
```

Example 3: Requesting compiler options during the compilation

The options for compiling a program are contained in the file OPT. They are read from SYSDTA. Further compiler options are contained in the file OPT1, but the location of these compiler options is not to be specified until the options are processed. The source program is contained in the file QUELLE.

Contents of file OPT1:

```
COMOPT LIST=(ALL)
END
```

1. Request input location of compiler options with `COMOPT OPTIONS=+`

   Contents of file OPT:

   ```
   COMOPT SOURCE=QUELLE
   COMOPT OPTIONS=+
   END
   ```

   Compilation:

   ```
   /ASS-SYSDTA TO-FILE=OPT
   /START-PROG $FOR1
   FOR1:GIVE OPTION FILE SPECIFICATION - OR ?
   OPT1
   ```

   In this case the user can request information about the input possibilities by entering a question mark after the input prompt.

2. Request input location of compiler options with `COMOPT OPTIONS=/`

   Contents of file OPT:

   ```
   COMOPT SOURCE=QUELLE
   COMOPT OPTIONS=/
   END
   ```

   Compilation:

   ```
   /ASS-SYSDTA TO-FILE=OPT
   /START-PROG $FOR1
   FOR1: ASSIGN SYSDTA TO READ OPTION
   /ASS-SYSDTA TO-FILE=OPT1
   /RESUME-PROG
   ```

3.4 Temporary changing of a source program: UPD compiler option

The text of a source program can be modified with change lines for the period of compilation. The changes are documented in the source program listing but have no effect on the source program file.
The input location of the change lines is specified in the UPD compiler option. There is no SDF operand corresponding to the UPD compiler option.

Requirement

Temporary changing of a source program requires an identification for the source program and change lines in columns 73 through 80. The identification must consist of eight EBCDIC characters. The source program and change line identifiers must be present in ascending sequence. The positional value of each individual character results from the position of the character in the EBCDIC character set (A has a lower positional value than 1). The change lines are included in the source program according to the identifiers, or replace those lines which have the same identifiers. If several change lines are to be inserted consecutively, only the first such change line need be identified accordingly; subsequent unidentified lines are included immediately thereafter.

*Temporary deletion of source program lines: *DELETE statement*

If specific lines in the source program are to be skipped, the following statement can be specified in the change lines:

```
*DELETE i1[,i2]
```

i1, i2 eight-character identifiers, i1 ≤ i2

All lines in the specified range (i1 through i12) are skipped and no longer printed in the source program listing.

This statement must begin in column 1. Column 8 must contain a blank. The identifier i1 must begin in column 9. If identifier i2 is specified, the comma must be in column 17 and the identifier i2 must begin in column 18. There may be no blanks in the range [i1,i2].

In the source program listing, the *DELETE statement appears as a comment line.

*Example: Change lines with *DELETE statement*

Source program file Column 73-80

```
      PROGRAM UPDATE                                                    AAAA1000
      B=8.                                                              AAAA2000
      D=10.                                                             AAAA3000
      CONTINUE                                                          AAAA4000
      A=B+2.                                                            AAAA5000
      C=D/A                                                             AAAA6000
      CONTINUE                                                          AAAA7000
      A=B/(C*D)                                                         AAAA7010
      WRITE(2,*) C                                                      AAAA7020
      WRITE(2,*) D                                                      AAAA7030
      WRITE(2,*) A                                                      AAAA7040
      END                                                               AAAA8000
```

Change lines

```
      WRITE(2,10) A                                                     AAAA6030
   10 FORMAT(11X,'NENNER = ',F8.2)                                      AAAA6060
*DELETE AAAA7020,AAAA7030                                               AAAA6080
      WRITE(2,*) A,B,C,D                                                AAAA7040
```

This results in the following source listing

```
      PROGRAM UPDATE                                                    AAAA1000
      B=8.                                                              AAAA2000
      D=10.                                                             AAAA3000
      CONTINUE                                                          AAAA4000
      A=B+2.                                                            AAAA5000
      C=D/A                                                             AAAA6000
      WRITE(2,10) A                                                     AAAA6030
   10 FORMAT(11X,'NENNER = ',F8.2)                                      AAAA6060
      CONTINUE                                                          AAAA7000
      A=B/(C*D)                                                         AAAA7010
*DELETE AAAA7020,AAAA7030                                               AAAA6080
      WRITE(2,*) A,B,C,D                                                AAAA7040
      END                                                               AAAA8000
```

UPD compiler option

The UPD compiler option is used to inform the compiler from where it is to read change lines. There is no SDF operand corresponding to the UPD compiler option.

The following default applies:

- If the UPD compiler option is missing, it is assumed that no change lines are present.
- If the UPD compiler option is specified without an operand value, the change lines are read from the file having the LINK name FUPDLINK. In this case the update file must be assigned by means of the following command before FOR1 is called: /SET-FILE-LINK LINK-NAME=FUPDLINK,FILE-NAME=filename

```
                          ⎡*        ⎤
                          ⎢file     ⎥
[*]COMOPT      UPD [=    ⎨lib(name)⎬]
                          ⎢/        ⎥
                          ⎣+        ⎦
```

*	The change lines are read from SYSDTA.
file	Name of a cataloged file in which the change lines are located. Maximum length including catalog ID and user ID: 54 characters.

lib(name)

lib
Name of a LMS library in OSM format or GAM file, from which the change lines are read. *lib* is first interpreted as a library and then as a GAM file. Including catalog ID and user ID, *lib* may be up to 54 characters in length.

name
Name of a library element or group key of a GAM file.

- PLAM library element: *name* may be up to 8 characters long.
- Group key: *name* may be up to 12 alphanumeric characters long and must be shorter than the group key.

/ When the the compiler has read all the compiler options, an interrupt occurs returning control to system mode. The compiler issues the following message:

```
FOR1: ASSIGN SYSDTA TO READ SOURCE
```

SYSDTA can now be reassigned with the ASSIGN-SYSDTA command. After the user has entered the RESUME-PROGRAM command, the compiler reads the source program from SYSDTA and commences the compilation.

+ When the compiler has read all the compiler options, it issues the following message:

```
FOR1: GIVE UPDATE FILE SPECIFICATION-OR?
```

Now the user can enter the location of the change lines via the primary assignment of SYSDTA, in interactive mode at the terminal.

The following locations can be specified:

```
⎧*            ⎫
⎪(SYSCMD)     ⎪
⎨(PRIMARY)    ⎬
⎪(CARD)       ⎪
⎪file         ⎪
⎩lib(name)    ⎭
```

*
The change lines are read from SYSDTA. No reassignment of SYSDTA takes place. If a location was specified with the ASSIGN-SYSDTA command, the change lines are read from there.

(SYSCMD)
The system file SYSDTA is combined with SYSCMD. The change lines are read from SYSCMD.

(PRIMARY)
The system file SYSDTA is returned to its primary assignment. The change lines are read from the primary assignment of SYSDTA, i.e. from the terminal.

(CARD)
The system file SYSDTA is assigned a card reader. The change lines are read from the card reader.

file
Name of the file from which the change lines are to be read (for description, see above).

lib(name)
Specification of the library element or section of the GAM file from which the change lines are to be read (for description, see above).

Restrictions:

- COMOPT UPD and %INCLUDE must not be used at the same time.
- COMOPT UPD and COMOPT DIALOG should not be used at the same time.
- Source program and change lines may only reside in the same file if the file is a group file.
- If SOURCE-FORMAT=FREE compiler option is specified in conjunction with UPD compiler option, an error message will be issued. The compiler option specified last applies.

Examples:

Example 1: Source program from file, change lines from file

The source program is read from the file QUELL.MAT. Change lines are located in the file UP having the specified LINK name FUPDLINK.

```
/SET-FILE-LINK LINK-NAME=FUPDLINK,FILE-NAME=UP
/ASS-SYSDTA *SYSCMD
/START-PROG $FOR1
*COMOPT UPD
*COMOPT SOURCE=/
*END
FOR1: ASSIGN SYSDTA TO READ SOURCE
/ASS-SYSDTA TO-FILE=QUELL.MAT
/RESUME-PROG
```

Example 2: Source program from file, change lines from library

The source program is read from the file QUELL.MAT. Change lines are located in the file ELEMUP in the library LIB.

```
/ASS-SYSDTA *SYSCMD
/START-PROG $FOR1
*COMOPT SOURCE=QUELL.MAT
*COMOPT UPD=+
*END
FOR1: GIVE UPDATE FILE SPECIFICATION-OR?
LIB(ELEMUP)
```

3.5 Inserting source program lines

3.5.1 %INCLUDE statement

The compile time statement %INCLUDE is used to insert source program lines into a source program. The %INCLUDE statement is written to the source program text like a FORTRAN statement. During compilation, the source program text specified by the %INCLUDE statement is inserted at the place occupied by the %INCLUDE statement. The %INCLUDE statement is only valid in the program unit in which it is specified. The source program text to be inserted by the %INCLUDE statement can be modified before it is inserted into the source program.

Insertion of text portions can take place over a number of different levels, i.e. %INCLUDE statements can be included in a text which has been inserted. Output of the inserted source program text can be controlled by means of the EXPAND compiler option (see section 4.6.2.8) or the %EXPAND statement (see section 4.6.3.3).

```
            ┌name      ┐        ┌EBCDIC┐
%INCLUDE    {          }[,CODE={ ISO    }][,'az1'='nz1'][,'az2'='nz2'][...]
            └lib(name) ┘        └BCD   ┘
```

name
: Name of a S-type PLAM library element containing the source program text to be inserted. If *name* is specified, a check is first made to determine whether a PLAM library element with this name exists. Access takes place according to the hierarchy of PLAM libraries which has been specified in the SDF operand INCLUDE-LIBRARY or in the INCLUDE compiler option.

  or

  File name of a cataloged file containing the source program text to be inserted.

lib(name)
: *lib* is the name of a PLAM library. The library element *name* of this library contains the source program text to be inserted.

  or

  lib is the name of a group file (GAM file). The group key *name* designates the source program text to be inserted (see section 3.2.3, example 4).

  lib is first assumed to be a PLAM library, then a GAM file. Any hierarchy for searching through the PLAM libraries as defined by the SDF operand INCLUDE-LIBRARY or the INCLUDE compiler option is not taken into account when *bibl* is specified.

CODE By default it is assumed that the source program is in EBCDI code. Conversion to ISO or BCD code is possible.

’os1’=’ns1’ [,’os2’=’ns2’] [...]
os : old string
ns : new string

The source program text to be inserted is updated in accordance with the specifications in this list before it is inserted in the source program. The source program text is searched in parallel for *old string1* and *old string2*. *old string1* is replaced by *new string1*, *old string2* is replaced by *new string2*. Blanks are evaluated during searching and replacing. If the source program text is lengthened as a result of replacement, continuation records are generated as required.

In the options listing, only names specified explicitly in the INCLUDE compiler option are listed (and not those specified in the %INCLUDE statement).

Notes

- The PROGRAM, SUBROUTINE, FUNCTION, type FUNCTION, BLOCK DATA, END statements are ignored if they occur in a text inserted by means of %INCLUDE. For this reason it is not possible to insert one or more program units by using %INCLUDE.
- The %INCLUDE keyword may not contain any blanks.
- Length of *old string*: $1 \leq$ length ≤ 72.
- Length of *new string*: $0 \leq$ length ≤ 32767.
- If *old string* extends beyond a record boundary, *old string* will not be recognized within the text.
- Nesting depth of the %INCLUDE statements: ≤ 32767.
- If an %INCLUDE statement extends over several lines, no intervening comment or blank lines may occur.

Example:

In program A, source program lines located in file B are inserted by means of the %INCLUDE statement. INCLUDE item B contains INCLUDE item C, whose source program lines are changed before insertion. INCLUDE item C contains INCLUDE item D.

Program A:

```
      PROGRAM A
C       THE INCLUDE FILE B
C       IS INSERTED
      %INCLUDE B
      END
```

File B:

```
C       TYPE DECLARATION IN
C       FILE B
      REAL M,N
C       INCLUDE FILE C
C       IS INSERTED
      %INCLUDE C,'N=3.'='N=5.'
```

File C:

```
C       CALCULATION IN
C       INCLUDE FILE C
      M=2.
      N=3.
      X=M*N
C       INCLUDE FILE D
C       IS INSERTED
      %INCLUDE D
```

File D:

```
C       RESULT OUTPUT IN
C       FILE D
      WRITE*,'RESULT=',X
```

In the source listing, the source program lines of program A and all source program lines inserted by means of the %INCLUDE statement are printed. Source program line 'N=3.' in INCLUDE item C is changed to 'N=5.' The nesting depth of the source program sections inserted using the %INCLUDE is displayed in column I of the source program listing (see section 4.7.2).

```
****  SOURCE  LISTING  ****   SIEMENS-NIXDORF FORTRAN COMPILER    FOR1 V2.2A00DATE = ...
                                                PROGRAM UNIT: A
DO/IF SEG  STMT  I/H LINE     SOURCE-TEXT
--------------------------------------------------------------------------------------
       1/1     1         1 |    PROGRAM A
                         2 |C   THE INCLUDE FILE B
                         3 |C   IS INSERTED
         1     2         4 |    %INCLUDE B
                  1/0    1 |C   TYPE DECLARATION IN
                  1/0    2 |C   FILE B
         1     3  1/0    3 |    REAL M,N
                  1/0    4 |C   INCLUDE FILE C
                  1/0    5 |C   IS INSERTED
         1     4  1/0    6 |    %INCLUDE C,'N=3.'='N=5.'
                  2/0    1 |C   CALCULATION IN
                  2/0    2 |C   INCLUDE FILE C
         1     5  2/0    3 |    M=2.
         1     6  2/0    4 |    N=5.
         1     7  2/0    5 |    X=M*N
                  2/0    6 |C   INCLUDE FILE D
                  2/0    7 |C   IS INSERTED
         1     8  2/0    8 |    %INCLUDE D
                  3/0    1 |C   RESULT OUTPUT IN
                  3/0    2 |C   FILE D
         1     9  3/0    3 |    WRITE *,'RESULT=',X
         1    10         5 |    END
```

3.5.2 SDF operand INCLUDE-LIBRARY

The SDF operand INCLUDE-LIBRARY defines hierarchical access to a number of libraries. The libraries are searched in the specified sequence for the element specified in the %INCLUDE statement.

```
START-FOR1-COMPILER
------------------------------------------------------------------------

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

The SDF operands and corresponding compiler options are shown in table 2-3.

3.5.3 INCLUDE-LIBRARY compiler option

The INCLUDE-LIBRARY compiler option defines hierarchical access to a number of libraries. The libraries are searched in the specified sequence for the element specified in the %INCLUDE statement.

```
                                   ⎧*NO                               ⎫
[*] COMOPT   | INCLUDE [-LIBRARY] = ⎨filename                          ⎬
                                   ⎩(filename1[,filename2][,...])     ⎭
```

*NO It is assumed that no search hierarchy exists. In this case the BS2000 catalog is searched.

filename1[,filename2][,...]
Names of PLAM libraries or GAM files containing %INCLUDE items. The sequence of library names defines the search hierarchy. Up to 10 library names may be specified. As the last hierarchical level, the BS2000 catalog is searched. If a library element was specified in the %INCLUDE statement using *lib(name)*, then no libraries are searched in accordance with the hierarchy defined by the INCLUDE option.

3.6 Entering the source program with Interactive Analysis

Interactive Analysis allows correctly formatted programs to be produced in a line-by-line dialog on unformatted screens. This operating mode is hereinafter referred to as "interactive", whereas "batch" designates the mode in which the user is not allowed to intervene in the compiler analysis for the duration of the compiler run.

In interactive mode, an error detected by the compiler can be immediately recovered, using the corrected statement to continue the compiler run. In this section the term "update" refers to changes made to a source program during a compiler interrupt.

In comparison to batch mode, the advantages of interactive mode are as follows:

- there is no need to wait for the end of compilation,
- evaluation of compiler listings may be dispensed with,
- there is no need for source program error recovery by means of an editor,
- no recompilation of the whole updated source program (or program unit); correct program sections are compiled only once,
- at the end of compilation, the corrected source program, if requested, is made available in an ISAM file (DIALOG-SAVE,OUTPUT option).

The original file remains unchanged throughout Interactive Analysis. When corrections are necessary, the compiler always uses a copy of the original file, which is hereinafter referred to as the work file. Following compilation, the original file or the work file may be deleted or recataloged at the user's discretion.

The execution of Interactive Analysis is controlled by means of dialog (interactive) commands, referred to simply as "commands" in the following.

Note

Interactive Analysis reports an error whenever the statement sequence in the program deviates from the sequence required by the standard (e.g. use of a variable prior to its declaration).

3.6.1 Controlling Interactive Analysis: SDF operand DIALOG

The SDF operand DIALOG is used for controlling Interactive Analysis.

```
START-FOR1-COMPILER

,DIALOG = NO / YES(...)

   YES(...)
   |
   |  ,DIALOG-INTERRUPT = AFTER-ANY-PROG-UNIT / ERRORS-ONLY
   |
   |  ,SAVE-FILE = *NONE / *STD-NAME(...) / <full-filename 1..54>(...)/
   |               *LIBRARY-ELEMENT(...)
   |
   |     *STD-NAME(...)
   |      |   INCLUDE-EXPANSION = NO / YES
   |
   |     <full-filename 1..54>(...)
   |      |   INCLUDE-EXPANSION = NO / YES
   |
   |     *LIBRARY-ELEMENT(...)
   |      |   INCLUDE-EXPANSION = NO / YES
   |      |   LIBRARY = <full-filename 1..41>
   |      |  ,ELEMENT = <full-filename 1..54> (...)
   |      |     |  VERSION = *UPPER-LIMIT / <alphanum-name 1..24>
   |
   |  ,LOG-CHANGED-LINES = NO / YES
```

The SDF operands and corresponding compiler options are shown in table 2-5.

3.6.2 Starting Interactive Analysis: DIALOG compiler option

In order to start Interactive Analysis, the compiler is called as usual by `/START-PROGRAM $FOR1` (or using any other given name).
Interactive mode is then entered by specifying the compiler option `*COMOPT DIALOG ...`

```
            ┌                 ┌param                    ┐ ┐
            │ DIALOG       [=<│(param[,param])          │]│
[*]COMOPT  <│                 └(param[,param[,param]])  ┘ │>
            │                                             │
            └ NODIALOG                                    ┘
```

(NODIALOG is underlined as the default.)

param:= { command prefix | dialog language | editor mode }

command prefix:= { ! | ? | @ | % | # | $ } (default: %, underlined)

dialog language:= { D | E } (default: E, underlined)

Message text output
- D in German
- E in English

editor mode:= E[DIT] = { ALL | FIRST | NO } (default: ALL, underlined)

```
This operand presents the following options
for a transfer of control to the user at the
beginning of a new program unit
ALL       before every new program unit
FIRST     only before the first program unit
NO        only in the event of an error
```

Interactive Analysis is interrupted according to the EDIT operand and the following message appears:

```
'FOR1: NEW PROGRAM UNIT - GIVE COMMAND OR @HELP'
```

The user can input commands, for example, for file processing or input BS2000 commands with %SYSTEM.

Notes

- Presetting for COMOPT DIALOG: %, E, EDIT=ALL
- Presetting for SDF operand DIALOG: @, E, EDIT=ALL
- COMOPT DIALOG cannot be specified in conjunction with the COMOPT UPD compiler option.

3.6.3 Outline

Source program input

There are different ways of entering a source program, controlled by the SOURCE option:
- from a file
- directly at the terminal

The user can specify direct input of the source program at the terminal by means of

- COMOPT SOURCE=*, omitting the SOURCE option or specifying the SOURCE option without value if the primary assignment of SYSDTA is the terminal;
- COMOPT SOURCE=(PRIMARY) if the options are read from a file assigned to SYSDTA.

The compiler supplies the line number. Then the user must enter the entire statement, noting the following:

- A semicolon as the first character in an input line is replaced by 6 blanks following its entry. This puts any text which follows the semicolon at column 7 following entry.
- As a consequence of entering continuation lines only, the current program unit is compiled once more (extension of a FORTRAN statement).
- Entering a line identifier (columns 73-80) is not permitted (use %SET or %INSERT command).
- The statement length is restricted by the size of the screen.
- A number of FORTRAN lines or dialog commands may be entered at the same time, separated by the end-of-line symbol for the terminal concerned (see MODIFY-TERMINAL-OPTIONS; "User Commands (SDF Format)" [12]).
- FORTRAN statements with a length of more than 72 characters are converted by the FOR1 compiler to the correct FORTRAN format, including continuation lines.
- "/*" in columns 1 and 2 identifies the end of an entry.
- Following interactive compilation, the source program, if requested, is available to the user in the form of a file (WS); the file name must be specified in the OUTPUT option and/or the %SAVE or %WRITE command.
- For further information see section 3.6.6, "Updating the work file".

Entering dialog commands

Dialog commands may be entered
- at any time from the terminal during direct program input
- during program input from a file, depending on the EDIT operand: either before compilation after the entire source program has been read into the work file (EDIT=FIRST) or before compilation of every individual program unit (EDIT=ALL).
- in "update" mode.

Execution of Interactive Analysis

The compiler reads the source program into a work file, verifying one statement after the other.
If a syntax error is found, the statement concerned is shown on the screen, identifying the position where the error occurred, and displaying a message text. Now the compiler is in "update" mode, in which the user can enter FORTRAN statements or commands (for Interactive Analysis). At this point the options are as follows:

- Work file update (error correction)
- No update; continuation of Interactive Analysis with the next statement or restart of analysis from the beginning of the current program unit (ignoring any previous corrections)
- Cancelation of interactive mode and exit to batch mode
- Termination of the compiler run and/or output of listings

Following a correction, Interactive Analysis resumes with the same statement, which is checked again for valid syntax.

Output of listings or files

The following listings or files can be selected for output:
- Compiler listing (as in batch mode)
- Change listing: list of modified, inserted or deleted lines with reference to the original program, also including commands entered for Interactive Analysis
- Original program file (useful in the case of direct input from the terminal): With direct input from the terminal, the initial input is identical to the original program
- Work file (INCLUDE expansions optional)

Output is controlled by the LIST, LIST-OUTPUT, LISTFILE, OUTPUT, DIALOG-SAVE options or by the SDF operand LISTING.

3.6.4 Controlling Interactive Analysis output: DIALOG-SAVE and OUTPUT compiler options

DIALOG-SAVE compiler option

```
[*]COMOPT  | DIALOG-SAVE = ([ ⎧*STD[-FILE]                       ⎫]
           |                 ⎨          ⎧file             ⎫      ⎬
           |                 ⎪[FILE=]   ⎨                 ⎬      ⎪
           |                 ⎩          ⎩plamspecification⎭      ⎭
           |
           |                                          ⎧NO ⎫
           |                 [,[INCLUDE-EXPANSIONS=]  ⎨   ⎬])
           |                                          ⎩YES⎭
```

*STD[-FILE]

When *STD-FILE is specified, the result of the interaction is written to a file with the name

```
FOR1.SAV.WS.prog[.tsn[.time]]
```

(see OUTPUT option).

file
Name of a cataloged file to which the work file is written. Maximum length including catalog ID and user ID: 54 characters.

plamspecification
Specification of a PLAM library element to which the work file is written:

```
[*LIBRARY-ELEMENT] ([LIBRARY=]plamlib,

                                           ⎧*UPPER-LIMIT⎫
                   [ELEMENT=]name[([VERSION=]⎨            ⎬)])
                                           ⎩version     ⎭
```

plamlib
Name of a PLAM library. Maximum length including catalog ID and user ID: 54 characters.

name
Name of a PLAM library element. The name may consist of up to 64 characters.

version
Version designation of the PLAM library element. A version designation can be up to 24 characters long.

*UPPER-LIMIT
The work file is entered with the highest possible version designation.

INCLUDE-EXPANSIONS

=YES	The source program text of the INCLUDE items is included in the output, the associated INCLUDE statements appear in the form of comments.
=<u>NO</u>	Default value: the INCLUDE items are not expanded on output.

Restriction:

The DIALOG-SAVE option must not be specified in conjunction with the OUTPUT option.

OUTPUT compiler option

```
                         ┌ filename                                          ┐
                         │ (filename [,expand])                              │
[*]COMOPT    OUTPUT [=  {  ( WS= filename [,OS=filename] [,expand])          }]
                         │ ([WS=] (filename [,expand])                       │
                         └        [,OS= (filename [,expand])])               ┘
```

filename

For the first two specifications, refers to the work file.

If WS filename and OS filename are identical, the updated workfile overwrites the original file
- if requested, in interactive mode (with prompting)
- otherwise in batch mode

If no filename is specified, the compiler generates standard names using the following format:

FOR1.SAV.WS.prog[.tsn[.time]] for the work file; linkname: EDWSLINK

prog	Name of the program unit
tsn	Task sequence number, 4-digit
time	Compiler start time in the form hhmmss

"tsn", "time" are appended stepwise as required to produce unique file names.

An OS-file is created only when the OS-operand is specified. The linkname for the OS-file is EDOSLINK. The assignment of a file to this linkname has precedence over the filename-specification in the OS-operand.

expand := [NO]EXPAND (INCLUDE expansions)

Default: NOEXPAND
If EXPAND is used, the INCLUDE items are transferred expanded to the work file and the associated %INCLUDE statements are shown as comment lines.

In the third specification, "expand" refers to the work file and to the original program.

WS	Work file (work source)
OS	Original program (original source)

The original program file and work file are not created until all program units have been compiled. In the meantime they can be saved with the %SAVE or %WRITE command.

If an output file with the same name already exists,
- it may be overwritten following a prompt in interactive mode;
- it will be overwritten in batch mode.

The line numbers generated by the compiler are not used as indices; instead, new indices are generated for output to ISAM files (default key length: 8 bytes).

Restriction:

The OUTPUT option must not be specified in conjunction with the DIALOG-SAVE option.

3.6.5 Error display

During Interactive Analysis, the compiler reads and analyzes the individual statements of a source program one after the other; it is therefore necessary to observe the order specified by the FORTRAN 77 standard. All declarative statements must be placed at the beginning of the program. If an erroneous statement occurs, compilation is interrupted following an analysis of the entire statement.

The following lines are displayed on the screen by the compiler:

Line 1	erroneous statement, including line number.
Line 2	error number and marking of the error position.
Line 3	message text.

To be more precise, the display shows the first line of the erroneous statement as well as another line for the case that an error position is marked there. If it is a statement with continuation lines, the user may opt to display the entire statement using the %PRINT command. It is also possible for more than one error to be displayed.

If one screen is not sufficient to contain all the lines, the user may optionally (prompted by the system) display another screen, thereby overwriting the previous one.

There are two ways the user can react to an error display:

- by entering a FORTRAN statement,
- by entering a command (for Interactive Analysis).

Syntax analysis does not detect all errors, for which reason the user is given control at the following times:

- after format analysis, to correct undefined statement labels,
- after semantic analysis, to correct semantic errors,
- after object module output to the temporary EAM area to correct errors encountered during the interpretation of DATA statements (before the creation of compiler listings).

In each of these three cases involving correction of an error, the entire program unit has to be recompiled, but none of the preceding program units.

In the third of the above cases, at the end of compilation, the module exists more than once for the program unit concerned, which could lead to bind/load errors. If the unit is the first program unit, this problem can be avoided by deleting the erroneous module prior to recompilation (using the command %SYS DEL-SYS-FILE OMF). Otherwise the user may eliminate the erroneous modules by transferring them from the temporary EAM area to a library in which they are overwritten by the succeeding, corrected modules.

3.6.6 Updating the work file

The update state begins when a syntax error is displayed by Interactive Analysis. Compilation is interrupted as long as an update is in progress. At this point the user can correct the source program in the work file created automatically by the compiler, by recovering errors, moving lines, and adding or deleting FORTRAN statements.

The user specifies whether the compiler is to remain in the update state or whether compilation is to be resumed:

The update state is **ended** by:
- %CONTINUE
- %BATCH
- %STOP
- %RESTART
- mere correction of lines of the illegal current FORTRAN statements
- pressing the DÜ (or ENTER) key (equivalent to %CONTINUE)
- entering "/*" in columns 1 and 2 (= end of data input from terminal.

The update state is **continued** by:
- any other command
- entry of a FORTRAN statement.

Changing the source program:
In the update state, the user can change the source program (in the work file) as follows; the relevant command is given in parentheses:
- replace lines (%SET)
- delete lines (%DELETE)
- insert lines (%INSERT)
- copy lines (%COPY)
- move lines (%MOVE)

Other functions include:
- work file inspection and paging (%PRINT).
- error file inspection (%PRINT ERROR) and paging; all pending errors are listed in that file.
- save work file (%SAVE or % WRITE).
- inspection of HELP file (%HELP); it describes the Interactive Analysis.
- renumber work file (%RENUMBER).
- change processor state (%BATCH; %STOP).
- execute BS2000 commands (%SYSTEM).
- ignore any corrections made to the current program unit (%RESTART).

Interactive Analysis protects compiled program units against updating. Therefore these program units cannot be referenced by these commands (exceptions: %SAVE or %WRITE).

3.6.6.1 Line numbering

Lines are numbered consecutively in the order they are read in. The line number consists of up to 8 decimal digits, with a decimal point inserted before the last four digits for better readability. As lines are read in, they are numbered, starting from 1.0000 and continuing by the current increment (default: 1.0000) for each subsequent line.

The line with the lowest line number can be referenced by %, the line with the highest line number by $.
If lines from an INCLUDE item are involved, they are assigned a prefix identifying the lines as INCLUDE item lines. INCLUDE item lines are numbered consecutively beginning with 1. Line numbering within an INCLUDE element also starts with 1.0000 and continues with the increment of 1.0000.

Example:	I3.0012.0000
Interpretation:	Line 12 of the 3rd INCLUDE item

If standard numbering is not sufficient, the user must enter an appropriate %RENUMBER command at the request of the compiler.

Entering line numbers:

When entering line numbers, the following abbreviations may be used, uniqueness permitting:
- Leading zeros before the decimal point may be omitted.
- Trailing zeros behind the decimal point may be omitted. If the decimal point was omitted, the system will add ".0000" to the line number.

Example:
Permissible entries for line numbers are: %2; %02.; %2.01; %2.0

Output of line numbers:

On line number output, leading zeros before the decimal point are suppressed, except the first place preceding the decimal point. Following the decimal point, four places are always output.

Current line number:

The current line number is the last one which appears in an error message.

Line range:

The line range [ln1,ln2] includes all currently used lines with the number "k", where ln1 ≤ k ≤ ln2.
A list of line ranges, i.e. several line ranges separated by commas, is also allowed instead of a line range in a command.
"ln1" or "ln2" may also be a construct of the form %+ln1 or $-ln1 or $+ln2. If the range is blank an error message is issued.

Increment:

The new line number is formed by adding the current increment to the previous line number. If the new line is already in use, the user is asked by the system whether that line may be overwritten (%INSERT, %COPY, %MOVE).

By default, the compiler sets the current increment to 1.0000.

If a command allows the entry of a line number, then either the specified increment is used as the current one, or the increment depends on the number of decimal digits supplied in the line number.

Example:
Increment 1 is assumed for k=2; 0.1 for k=2.4 and also for k=2.0; increment 0.01 for k=2.40 etc.

3.6.6.2 Entering statements and commands

Corrections may also be made to lines whose number is not the current line number, either through a command (which allows line range entries) or by overwriting output lines (as in EDT). In this case the user either employs the FORTRAN statement shown in the error display or statements displayed by means of the %PRINT command.

Anticipatory/retrograde update

If the Update relates to a line whose line number is less than or equal to the current line number, it is "retrograde update", otherwise it is "anticipatory update". Retrograde update, which is only allowed within the current program unit, has the effect that the entire program unit up to the current line number is recompiled. Previously compiled program units can not be updated any more.

Chaining of statements

A number of statements may be written on the same line, provided that they are separated by the end-of-line symbol (as in EDT). A line length exceeding 72 characters is allowed. In addition, statements chained in this manner may extend over more than one line, but not over more than one screen. The compiler will resolve such chains into single statements with line numbers of their own, which do not, however, show in the work file. The end of such chains of statements may also be a command terminating the update. Lines separated by end-of-line are assigned line numbers. Lines merely created by line break are not assigned numbers of their own in the work file. There the input line is kept at full length.

Positioning to column 7

If a semicolon is entered as the first character of an input line, it is replaced by 6 blanks following entry. As a result, the text following the semicolon is positioned at column 7 after input.

Line length

For an update, the user must enter the entire line (the whole statement). A line length exceeding 72 characters is allowed. A break then occurs in column 72, with the compiler generating the necessary continuation lines. Line length is limited by the size of the screen. If the statement to be corrected has an identifier in columns 73-80, that identifier is also entered in the continuation lines, unless the command requires a different identifier. At the beginning of the line, however, the FORTRAN input conventions must be observed. Comments using the LINEEND option are only allowed when the input line is not longer than 72 characters.

3.6.6.3 Line break

For a terminal entry consisting of more than 72 characters, a break is generated automatically by the compiler after column 72. If a statement is presented, every continuation line which is necessary begins with an ampersand (&) in column 6. Necessary continuation lines to a comment line all begin with a C in column 1, four blanks, and a & character in column 6. CCOM lines are taken into account.

This break is only visible in the source program listing and in the source program output file. The line is shown unbroken on the terminal.

Example:

```
                        Column 6  ...        Column 72
                            |                     |
                            v                     v

Input:                        A=B+  ...            C**2+4

Storage in                    A=B+  ...            C**
work file:                  &2+4

Input:                C   LONG    ...       COMMENT LINE

Storage in            C   LONG    ...       COMMEN
work file:            C    &TLINE

Input (COMOPT
LINEEND declared):            A=B   ...       ;";" LINEEND

Storage in                    A=B   ...       ;";" L
work file:                  &INEEND
```

This causes an error, as the continuation line is not treated as comment.

```
Input:                CC      A=B+  ...            C**2+4

Storage in            CC      A=B+  ...            C**
work file:            CC    &2+4
```

3.6.6.4 Analysis of an update

The compiler resumes Interactive Analysis with the desired line (more precisely, with the first statement on that line) governed by line number, which need not be the current line number. In Interactive Analysis, the update is processed in the order of corrections and will be interrupted if an error occurs.

Line identifier in columns 73-80

When specified in the appropriate commands, a line identifier is inserted in the associated line starting with column 73, also in compiler-generated continuation lines. Apostrophes in the line identifier must be doubled. Fewer than eight characters are entered left-justified starting with column 73, padding the line through column 80 with blanks. If the line identifier contains no character ('' specified) or more than eight characters, an error message is issued.

3.6.6.5 Errors in an INCLUDE item

Such errors must be eliminated outside of Interactive Analysis.
In interactive mode, the user can only correct the work file; corrections to an INCLUDE expansion which occurs in several places must be carried out separately. Modifications to an INCLUDE statement have no effect on the INCLUDE item itself.

3.6.7 Commands for controlling Interactive Analysis

Entries referred to as "commands" are used to control Interactive Analysis. In the following description they are also called "interactive" or "dialog" commands, to distinguish them from BS2000 commands.

3.6.7.1 Rules for the entry of commands

All dialog commands except the "paging commands" are entered in command mode. Paging commands are entered in paging mode, identified by "*+-0" on the terminal. %PRINT is the command for exiting from command mode to paging mode.

Commands for interactive compilation should only be entered from the terminal. The original program should only contain comment lines, FORTRAN statements or compile time statements.

Dialog commands included in the original program, although they are executed in place, lead to errors in batch mode because they are not known there (OPTION NODIALOG). Also they are not transferred to the output file.

Erroneous commands are rejected with an error message indicating the cause of the error.

Commands overwrite the existing lines only if the user responds to a corresponding prompt in the affirmative (%INSERT, %COPY, %MOVE).

Pressing the DÜ (or ENTER) key alone has the same effect as
- %CONTINUE in command mode,
- * in paging mode (see %PRINT command).

Command prefix

Commands begin with a prefix, which the user can define in the DIALOG option. Any of the characters !, ?, #, $, %, @ is permitted. "%" is the default if Interactive Analysis is controlled through the DIALOG option; "@" is the default if Interactive Analysis is controlled through the SDF operand DIALOG.

During a compiler run, the command prefix can be changed by means of the following input:

```
current command prefix: new command prefix
```

Example:
Command prefix "#" is to be substituted for the default "%"; enter %:#.

When entering a source program from a file, a command prefix in column 6 is interpreted as the continuation character of a FORTRAN line.
When input is from the terminal, doubling the prefix causes a line to be interpreted as a data line.

Abbreviations

By omitting residual letters, command names may be truncated from the right, provided they are still uniquely identifiable. In addition there are some special abbreviations (see section 3.6.7.2, Table 3-1).

Blanks

Blanks may be omitted when entering a command, with the exception of a command followed by an operand value beginning with a letter, a blank is needed to separate the two, for example %PRINT␣ERRORS. The user is, however, allowed to insert any number of blanks.

3.6.7.2 Summary of all dialog commands

The following is a summary of all dialog commands, indicating the shortest command names possible.

Command	Abbrev. form	Meaning
`%BATCH [ {L|LF|WS|CP|OS|PU|ALL} [,...]]`	B	Cancelation of Interactive Analysis, transfer of control to batch mode
`%CONTINUE`	CON	Resumption of an interrupted compiler run
`%COPY ln1[-ln2] [,ln3[-ln4]][,...] {, |TO} ln5[-ln6][,ln7[-ln8]][,...] [(s)] [,]['id'[(incr)]]`	COP	Copying a range of lines
`%DELETE ln1[-ln2] [,ln3[-ln4]]...`	D	Deleting a range of lines
`%HELP`	H	Brief description of all dialog commands
`%INSERT ln1[(s)] [,]['id'[(incr)]] [:string]`	IN	Insertion of a range of lines
%LOWER { ON | <u>OFF</u> }	L L ON L OF	Lowercase letters allowed

continued ⊳

continued

Command	Abbrev. form	Meaning
%MOVE ln1[-ln2] [,ln3[-ln4]][,...] {, |TO} ln5[-ln6][,ln7[-ln8]][,...] [(s)] [,]['id'[(incr)]]	M	Moving a range of lines
%PRINT [{ln1[-ln2] [,ln3[-ln4]]... [,[NO]EXPAND] E[RRORS]}]	P	Prints a range of lines or the error file; paging possible afterwards
+ +n - -n -- ++ * 0 !		Paging commands, used after a %PRINT command Chaining the paging commands
%:{@|#|?|!|%|$}		Changing the command prefix
%RENUMBER [ln1 [(s)]]	R	Renumbering
%RESTART	RES	Restart with the current program unit of the original program
%SAVE ['filename' [, [NO]EXPAND]]	SA	Saving the work file (ISAM)
%[SET] ln1 [(s)] [,] ['id'[(incr)]] [:string]	SE	Changing a line
%STOP [{L|LF|WS|CP|OS|PU|ALL} [,...]]	ST	Stopping a compiler run
%SYSTEM ['BS2000 command']	SY	Execution of BS2000 commands
%WRITE ['filename'[,[NO]EXPAND]]	W	Saving the work file (SAM)

Table 3-1: Summary of all dialog commands

3.6.7.3 BATCH (continuation in batch mode)

```
%BATCH [{L | LF | WS | CP | OS | PU | ALL} [, ...]]
```

L	Listing
LF	Listfile
WS	Work file (work source)
CP	Change listing (Change protocol)
OS	Original file (Original source)
PU	Batch mode for the current program unit
ALL	L,LF,WS,CP,OS

Interactive Analysis is canceled and the specified listings are printed after the compiler run terminates in batch mode. If "PU" is entered, batch mode is activated for the current program unit only.

Listing and listfile are printed to the default extents unless they were requested before by the compiler option.

L and/or LF are relevant for specification of the change listing:
- If L and/or LF is specified, the change listing is included in these listings
- If neither L nor LF is specified, the change listing is output to SYSLST

3.6.7.4 CONTINUE (resumption of compiler run)

```
%CONTINUE
```

This command is used to continue a compiler run interrupted by an error condition.

Note that, after a retrograde update, the compiler resumes at the beginning of the current program unit; this is indicated by a corresponding message. Otherwise the compiler resumes with the next statement, recompiling a corrected statement since Interactive Analysis comprises more than just syntax checking (example: structure of DO loops).

3.6.7.5 COPY (copying a range of lines)

```
%COPY ln1 [-ln2] [,ln3[-ln4]][,...]{, | TO} ln5[-ln6][,ln7[-ln8]][,...]
          [(s)] [,] ['id'[(incr)]]
```

ln1,...	Line number
s	Step
id	Line identifier
incr	Line identifier increment

The copying range defined by line numbers ln1 and ln2 is copied to the location specified by line number ln5. The new line numbers are derived from ln5 and s. When "s" is omitted, the current step width is assumed as the default to determine the new line number. If "ln1" only is specified, or if ln1 and ln2 are identical, line ln1 is copied.

A line identifier, if specified, is entered left-justified starting with column 73, including compiler-generated continuation lines.
Apostrophes within a line ID must be doubled. An error message appears for a line ID which has more than 8 characters or is empty (i.e. either '␣' or "). If an increment is supplied, it is added (from the right) to the numeric value of the line ID for every line generated by COPY.

The COPY command is not executed (an error message will appear) when the maximum line number is encountered or the copying range is empty.

Before existing lines are overwritten, the user is prompted for confirmation. Depending on the reply, the command is executed or rejected.

3.6.7.6 DELETE (deletion of a range of lines)

```
%DELETE [ln1 [-ln2][,ln3[-ln4]]...]
```

ln1,...	Line number

The range [ln1-ln2]... is deleted.

If the line number specification is omitted, the %DELETE command will delete the whole range of lines. To prevent inadvertent deletion of the entire range of lines, the command must be confirmed before it is executed.

3.6.7.7 HELP (brief description of all dialog commands)

```
%HELP
```

A brief description of all commands is output on the terminal.

3.6.7.8 INSERT (insertion of a range of lines)

```
%INSERT ln1 [(s)] [,] ['id'[(incr)]] [:string]
```

ln1	Line number
s	Step
id	Line identifier (columns 73-80)
incr	Line identifier increment
string	String (FORTRAN line)

From "ln1" and "s", the compiler determines the line numbers to be generated, expecting sequential input. The line number is supplied by the compiler.

If the step is omitted, the current step is assumed as the default.

A line identifier, if specified, is entered left-justified starting at column 73, including any continuation lines generated by the compiler. Apostrophes within the line ID must be doubled. If the line has more than 8 characters or is empty (i.e. either '␣' or ''), an error message is issued. A specified increment is added (from the right) to the numerical value of the line ID for every line generated by means of COPY.
Unlike the %SET command, %INSERT never overwrites lines.

3.6.7.9 LOWER (lower/upper case)

```
%LOWER {OFF|ON}
```

%LOWER OFF	The compiler converts any lowercase letters entered on the screen to upper case.
%LOWER ON	The compiler distinguishes between lowercase and uppercase letters entered on the screen.

3.6.7.10 MOVE (moving a range of lines)

```
%MOVE ln1 [-ln2] [,ln3[-ln4]][,...]{, | TO} ln5[-ln6][,ln7[-ln8]][,...]
        [(s)] [,] ['id'[(incr)]]
```

ln1,...	Line number
s	Step
id	Line identifier
incr	Line identifier increment

The MOVE range [ln1,ln2] is moved to the location specified by line number ln5, deriving the new line numbers from "ln5" and "s". If s is not specified, the current step is used to determine the new line number. If only line ln1 is specified, or if ln1 and ln2 are identical, line ln1 is moved.

If specified, a line identifier is entered left-justified starting with column 73, including any continuation lines generated by the compiler. Apostrophes within the line ID must be doubled. An error message is issued for any line ID which has more than 8 characters or is empty (i.e. either '␣' or ''). If an increment is specified, it is added (from the right) to the numeric value of the line ID for every line generated by means of COPY.

The MOVE command remains unexecuted in the following situations:
- when the maximum line number is encountered
- when the range to be moved is empty

Before existing lines are modified by this command, the user is prompted for confirmation. Whether or not the command is executed depends on his reply.

3.6.7.11 PRINT (prints a range of lines or the error file; paging)

```
               ⎧ ln1 [-ln2] [,ln3[-ln4]]... [,expand] ⎫
%PRINT   [⎨                                             ⎬]
               ⎩ E[RRORS]                               ⎭
```

ln1,...	Line number, specifically % for the beginning, $ for end.

expand:= [NO] EXPAND

EXPAND	INCLUDE items are expanded (default value).
NOEXPAND	INCLUDE items are not expanded.

ERRORS	Error list is printed.

This command prints the line range [ln1,ln2] of the work file, or, if ERRORS is entered, the beginning of the error file. This process involves a transfer of control from command mode to paging mode.

Paging through the line range specified by %PRINT

The following commands are supported in paging mode:

```
+    Advance one screen
+n   Advance n lines (n is an integer)
-    Return one screen
-n   Return n lines (n is an integer)
--   Position to the beginning of the specified range (%)
++   Position to the end of the specified range ($)
*    Advance one screen; at the end of the specified range of lines,
     change to command mode; the above commands, however, retain the
     paging mode
0    End of paging mode, change to command mode.
```

Paging commands can be chained by using exclamation marks.

Example:

%PRINT %	Shows the line with the lowest number.
%P %-$	Shows the entire work file.
++!-	Shows the end of the specified line range.

3.6.7.12 RENUMBER

```
%RENUMBER [ ln1 [(s)] ]
```

ln1	Line number
s	Step

Specification of "s" defines the new current step.

The lines of the work file are renumbered with the current step, starting with ln1.

If "ln1" and "s" are omitted, renumbering starts from 1.0000, setting the current step to 1.0000.

3.6.7.13 RESTART

```
%RESTART
```

The compiler run is restarted at the beginning of the current program unit, canceling all modifications made to that program unit (resetting to the original program). The options remain unchanged.

If the original program was entered directly from the terminal, it need not be reentered. The compiler recompiles/reanalyzes all previous FORTRAN lines, unless they were generated by command (%INSERT, %MOVE,...).

3.6.7.14 SAVE or WRITE (saving the work file)

```
{%SAVE|%WRITE}['filename'[, [NO]EXPAND ]]
```

SAVE	Creates an ISAM file.
WRITE	Creates a SAM file.
filename	File name in accordance with BS2000 conventions. If this name is omitted, the name is taken from the OUTPUT (WS) option or the default is assumed.
EXPAND	INCLUDE items are expanded, displaying the %INCLUDE statements as comment lines.
<u>NOEXPAND</u>	INCLUDE items are not expanded (default).

The current work file is created under the specified name and is thus saved.
If a BS2000 file with the specified name already exists, the user is prompted for permission to overwrite it.

The original program cannot be overwritten by the %SAVE or %WRITE command if it has been read from SYSDTA, or if it is a element of an INCLUDE library or a group file (GAM file).

3.6.7.15 SET (modification of a line)

```
%[SET] ln1 [(s)] [,] ['id'[(incr)]] [:string]
```

ln1	Line number
s	Step
id	Line identifier
incr	Line identifier increment
string	String (FORTRAN line)

This command can be used to modify the contents of an existing line ln1. The current step can be changed at the same time. The line is written with the specified string (starting from column 1).

An existing line identifier is retained and, if necessary, entered in continuation lines, unless a line ID was specified in the command.

A line identifier, if specified, is entered left-justified starting with column 73, including continuation lines generated by the compiler. Apostrophes within the line ID must be doubled. If the line ID has more than 8 characters or if it is empty (i.e. either '␣' or ''), an error message is issued. An increment, if supplied, is added (from the right) to the numeric value of the line ID for every line generated by COPY.

Notes

- If there is no line with the number ln1, a new line is generated.
- The compiler generates continuation lines as necessary.
- Specification of "s" defines the current step.
- "SET" may be omitted when this command is entered.

3.6.7.16 STOP (termination of the compiler run)

```
%STOP  [{L | LF | WS | CP | OS | PU | ALL} [, ...]]
```

L	Listing
LF	Listfile
WS	Work file (work source)
CP	Change list (change protocol)
OS	Original file (original source)
PU	– Cancelation of compilation of the current program unit – No listing output – Resumption with the next program unit.
ALL	L,LF,WS,CP,OS

The compiler run is canceled and the specified listings are printed, with the exception of "PU".

In the case of program entry from the terminal where the current program unit is not yet complete, the compiler terminates abnormally. The following message appears:

```
STOP REQUESTED DURING COMPILATION OF P.U. ...
```

Status indicator for any job variable that may be set: $A2001

Listing and listfile are printed to their default extent, unless they were requested beforehand in the LIST or LISTFILE option.

L and/or LF are relevant to the specification of the change listing. If L and/or F have been specified, the change listing is included in those listings. If neither L nor F has been specified, the change listing is printed output, to SYSLST.

3.6.7.17 SYSTEM (execution of BS2000 commands)

```
%SYSTEM   ['[/]BS2000 command']]
```

The BS2000 command enclosed in apostrophes is executed.

If no BS2000 command has been entered, control is transferred to the system (BREAK); return from this mode with /RESUME-PROGRAM.

3.6.8 Example of Interactive Analysis

The following three-part example outlines how to use Interactive Analysis.

1. Creation of a new program
2. Debug run of this program
3. Extension of this program

Sections A.6.2 through A.6.10 in the Appendix show listings relating to this program.

```
/DEL-SYS-FILE OMF
/START-PROG $FOR1
 %  BLS0500 PROGRAM 'FOR1', VERSION '2.2A00' OF '91-06-05' LOADED.
 %  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG. 1991 ...
  FOR1:    V2.2A00 READY, GIVE COMPILER OPTION
*COMOPT D=(D,@),OUT=WS.X,LF=LF.X(SRC,D,OP,XR),END                        (01)
 @   1.    :      PROGRAM DIALOG                                         (02)
 @   2.    :      INTEGER I1,I2
 @   3.    :      REEL A(10),B
 @   3.0000:         REEL A(10),B
            ........ 1                                                   (03)
 1 ERROR     MISSPELLED STATEMENT KEYWORD, REAL ASSUMED
 @   3.    :      REAL A(10),B                                           (04)
 @   4.    :****
 @   5.    :      DO 10 I=1.5<      B(I)=A(I)+I                          (05)
 @   7.    :10    A(I)=I
 @   8.    :      WRITE (2,11) A
 @   9.    :11    FORMAT (' **',5F5.2)
 @  10.    :      CALL SUBA(A(1))
 @  11.    :      END
 @   7.0000:10       A(I)=I                                              (06)
   WARNING   UNREFERENCED LABEL #10
 @   7.    :@P                                                           (07)
 @   1.0000:         PROGRAM DIALOG
 @   2.0000:         INTEGER I1,I2
 @   3.0000:         REAL A(10),B
 @   4.0000:****
 @   5.0000:         DO 10 I=1.5
 @   6.0000:            B(I)=A(I)+I
 @   7.0000:10       A(I)=I
 @   8.0000:         WRITE (2,11) A
 @   9.0000:11       FORMAT (' **',5F5.2)
 @  10.0000:         CALL SUB(A(1))
 @  11.0000:         END
 @   7.    :                                                             (08)
 @   5.0000:         DO 10 I=1,5                                         (09)
 @   5.0001:
 @CON                                                                    (10)
  FOR1: RECOMPILATION OF ACTUAL P.U. INITIATED
 @  11.0000:          END
   WARNING   STATEMENT FUNCTION B UNUSED                                 (11)
 @  11.    :@I3.5                                                       (12)
 @I3.5
   !
  FOR1:  ERROR, WRONG INCLUDE NUMBER
 @  11.    :@IN3.5                                                       (13)
 @   3.5   :      DIMENSION B(5)
 @   3.6   :@R                                                           (14)
 @  13.    :@P1-2
 @   1.0000:         PROGRAM DIALOG
 @   2.0000:         INTEGER I1,I2
 @  13.    :@D2                                                          (15)
  FOR1: 1 LINE(S) DELETED
 @  13.    :@CON
```

```
  FOR1: RECOMPILATION OF ACTUAL P.U. INITIATED
  FOR1: LIST FILE GENERATED = LF.X
  FOR1: NO ERRORS DURING COMPILATION OF P.U. DIALOG
 @  13.    :      SUBROUTINE SUBA(X)                                      (16)
 @  14.    :      Y=X*X<      END
  FOR1: NO ERRORS DURING COMPILATION OF P.U. SUBA
 @  16.    :/*                                                            (17)
  FOR1: WS FILE CREATED = WS.X
  END OF  F O R 1  COMPILATION;  CPU TIME USED: 1.142 SEC.

/START-PROG FROM-FILE=*MODULE(*OMF)                                       (18)
 %  BLS0001 ### DBL VERSION 070 RUNNING
 %  BLS0517 MODULE 'DIALOG' LOADED
 BS2000  F O R 1 : FORTRAN PROGRAM "DIALOG"
 STARTED ON 1991-09-03 AT 14:23:53
 ** 1.00 2.00 3.00 4.00 5.00
 ** 0.00 0.00 0.00 0.00 0.00
 BS2000  F O R 1 : FORTRAN PROGRAM "DIALOG  " ENDED PROPERLY AT 14:23:55
 CPU - TIME USED :       0.0205 SECONDS
 ELAPSED TIME    :       1.7630 SECONDS



/DEL-SYS-FILE OMF
/START-PROG $FOR1
 %  BLS0500 PROGRAM 'FOR1', VERSION '2.2A00' OF '91-06-05' LOADED
 %  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG. 1991 ...
  FOR1:    V2.2A00 READY, GIVE COMPILER OPTION
*COMOPT SRC=WS.X,D=(D,@,E=FIRST),OUT=WS.X1,END                            (19)
  FOR1: NEW PROGRAM UNIT - GIVE COMMAND OR @HELP @P
 @   1.0000:         PROGRAM DIALOG
 @   2.0000:         REAL A(10),B
 @   3.0000:         DIMENSION B(5)
 @   4.0000:****
 @   5.0000:         DO 10 I=1,5
 @   6.0000:         B(I)=A(I)+I
 @   7.0000:10       A(I)=I
 @   8.0000:         WRITE (2,11) A
 @   9.0000:11       FORMAT (' **',5F5.2)
 @  10.0000:         CALL SUB(A(1))
 @  11.0000:         END
 @  12.0000:         SUBROUTINE SUB(X)
 @  13.0000:         Y=X*X
 @  14.0000:         END
 @  15.    :@D7                                                           (20)
  FOR1: 1 LINE(S) DELETED
 @  15.    :@CON
 @  11.0000:          END
   SEVERE    UNTERMINATED DO-BLOCK CLOSED BY LABELED CONTINUE
 @  11.    :@RES                                                          (21)
  FOR1: RECOMPILATION OF ACTUAL P.U. INITIATED USING ORIGINAL SOURCE
  FOR1: NEW PROGRAM UNIT - GIVE COMMAND OR @HELP @M8-9TO13.9              (22)
  FOR1: MOVE-LIST TARGET RANGE OVERLAPS EXISTING LINES - OVERWRITE? (Y/N) N
 N
  FOR1: 0 LINE(S) MOVED
 @  15.    :@M8-9T13.5
  FOR1: 2 LINE(S) MOVED
 @  13.7   :@P13-14
 @  13.0000:          Y=X*X
 @  13.5000:          WRITE (2,11) A
 @  13.6000:11        FORMAT (' **',5F5.2)
 @  14.0000:          END
 @  13.7   :@CON
  FOR1: NO ERRORS DURING COMPILATION OF P.U. DIALOG
  FOR1: WS FILE CREATED = WS.X1
  FOR1: NO ERRORS DURING COMPILATION OF P.U. SUB
  END OF  F O R 1  COMPILATION;  CPU TIME USED: 0.613 SEC.                (23)
```

Explanation of example:

(01) Start of Interactive Analysis; options: DIALOG, OUTPUT, LISTFILE; English error messages are requested; command prefix @.

(02) Request for entry. The following program is entered directly from the terminal.

(03) An error is marked in line 3. The message text is displayed.

(04) Error correction: reentry.

(05) Two chained FORTRAN statements. The compiler resolves them into two separate statements; see current line numbers 5 and 7.

(06) This is a result of the error in statement line 5.

(07) PRINT command.

(08) The current line number remains 7.

(09) Direct overwriting of line 5 in the output area; implicit step width 0.0001 as shown by the next line number.

(10) CONTINUE command.

(11) B is interpreted as a statement function name because no dimension is declared.

(12) The INSERT command cannot be abbreviated as I.

(13) A dimension is declared.

(14) RENUMBER command.

(15) DELETE command.

(16) Program entry can be continued.

(17) End of entry. The program is stored in the file WS.X.

(18) Program execution.

(19) Program extension; input file WS.X; work file output to WS.X1. Control is only required before the first program unit (EDIT operand).

(20) Deletion of line 7 causes an unrecoverable error.

(21) RESTART command; resumption with the state of the file WS.X.

(22) MOVE command with 0.1 as implicit step. Since existing lines would be overwritten, the command is modified and reentered.

(23) The CONTINUE command terminates this run of Interactive Analysis because EDIT=FIRST was set - unlike the first Interactive Analysis run. See also (01).

4 Source program compilation

4.1 Specifying and checking the attributes of the source program

4.1.1 SDF operand SOURCE-PROPERTIES

```
START-FOR1-COMPILER

,SOURCE-PROPERTIES = STD / PARAMETER(...)

   PARAMETER(...)

      |  COMPILEABLE-COMMENTS = *NONE / <c-string 1..60>
      | ,LINE-END-COMMENTS = *NONE / <c-string 1..10>
      | ,LANGUAGE-STANDARD = FOR1 / ANS77
      | ,IMPLICIT-DECLARATION = YES / NO
      | ,EXPONENT-UNDERFLOW = IGNORED / ERROR
      | ,SOURCE-FORMAT = FIXED / FREE
      | ,SAVE-CONSTANT = *STD / YES / NO
      | ,FORTRAN90-CHECK = YES / NO
```

The SDF operands and corresponding compiler options are shown in table 2-6.

4.1.2 Specifying and checking the attributes of the source program by compiler options

4.1.2.1 CCOM option

`[*]COMOPT`	`CCOM = ’comment-marks’`

All comment lines which have one of these specified characters in column 2 are treated and compiled as ordinary FORTRAN statements after substitution on columns 1 and 2 by two blanks. Comment marks may consist of up to 60 printable characters of the EBCDIC set, allowing any characters except blanks.
The option COMOPT CCOM = ’ ’ can be used to delete the defined comment marks.

If compiler option SOURCE-FORMAT=FREE is specified in conjunction with the CCOM option, an error message will be issued. The option specified last applies.

4.1.2.2 LINEEND option

`[*]COMOPT`	`LINEEND = ’end-marks’`

Each of the characters specified in "end-marks" assumes the function of an end mark if encountered in any of the columns 7-72 outside of character and Hollerith constants. Anything following such a character in the program line is treated as comments by the compiler.
Up to 10 characters may be defined as end marks in the LINEEND option. The characters specified in *end-marks* must be within the EBCDIC character set but outside of the FORTRAN character set. "%" must not be used either, since it is required for FOR1 compile time statements and debug statements.

Blanks in the LINEEND option are ignored and therefore cannot be used as end marks. The exclamation mark, question mark or vertical line is permitted, for example.

COMOPT LINEEND = ’ ’ can be used to delete the specified end marks.

If compiler option SOURCE-FORMAT=FREE is specified in conjunction with the LINEEND option, an error message will be issued. The option specified last applies.

4.1.2.3 STANDARD-CHECK option

Specification of the STANDARD-CHECK=ANS77 option causes the FORTRAN source program to be inspected for deviations from the ANS FORTRAN 77 standard.

`[*]COMOPT`	`ST[AN]D[ARD-CHECK] =` $\left\{\begin{matrix}\texttt{ANS77}\\ \underline{\texttt{NO}}\end{matrix}\right\}$

Deviations of the FOR1 language from the ANS-FORTRAN-77 language standard are output in the form of warning messages. Normally deviations from this standard are not reported.

To distinguish them from other error messages, these deviation messages are given the suffix ANS FORTRAN 77 DEVIATION following the error type and error number; cf. for example:

```
FA206  ANS FORTRAN 77 DEVIATION: MORE THAN 19
       CONTINUATION LINES NOT ALLOWED
```

If the compiler option MSGLEVEL=ERROR is activated, no messages about deviations are output.

Restrictions:

If the variable I from statement ASSIGN n TO I appears in statements other than a FORMAT or GOTO statement, the standard checker does not report this as a deviation from the ANS FORTRAN 77 standard.

If compiler option SOURCE-FORMAT=FREE is specified in conjunction with the STANDARD-CHECK=ANS77 option, an error message will be issued. The option specified last applies.

Example: STANDARD-CHECK option

The program STDTEST is compiled using COMOPT STD=ANS77. Specification of COMOPT DIALOG=D or COMOPT LANGUAGE=GERMAN provides German messages on deviations from the ANS77 standard. These deviations are displayed in the SOURCE listing and in the DIAGNOSTIC listing.

```
  ****  SOURCE  LISTING  ****      SIEMENS-NIXDORF FORTRAN COMPILER     FOR1 V2.2A00  DATE = ...  TIME = 11:49:53    PAGE    1
                                                     PROGRAM UNIT: STDTEST
 DO/IF SEG   STMT  I   LINE        SOURCE-TEXT                                                  COL73-80 RECORD-ID.
-------------------------------------------------------------------------------------------------------------------------------
*        1      1        1 |       PROGRAM STDTEST                                            |         00010000          *
***** WARNING (FA203) *******             -1-                                                *ANS FORTRAN 77 DEVIATION******
                         2 |C                                                                 |         00010100
         1      2        3 |       INTEGER I,J                                                |         00020000
*        1      3        4 |       REAL  A(3,3) , B(3) ,E(3) /3*0./                           |         00030000          *
***** WARNING (FA249) *******                               -1-                              *ANS FORTRAN 77 DEVIATION******
*        1      4        5 |       NAMELIST /NAM/ A,B                                         |         00040000          *
***** WARNING (FA230) *******                                                                *ANS FORTRAN 77 DEVIATION******
***** WARNING (SA088) *******                                                                *ANS FORTRAN77 DEVIATION*******
*        1      5        6 |       WRITE *, 'ENTER VALUE FOR ARRAY A(3,3) AND VECTOR B(3)',   |         00050000          *
*        1               7 | 1              'IN NAMELIST-FORMAT /NAM/'                        |         00060000          *
***** WARNING (FA259) *******                                                                *ANS FORTRAN 77 DEVIATION******
*        1      6        8 |       READ (1,NAM)                                               |         00070000          *
                         9 |C                                                                 |         00070100
***** WARNING (SA089) *******                                                                *ANS FORTRAN77 DEVIATION*******
         2      7       10 |       DO 100, I=1,3                                              |         00080000
   1     2      8       11 |       DO 100, J=1,3                                              |         00090000
   2     3      9       12 |          E(I) = A(I,J) * B(J) + E(I)                             |         00100000
   2     5     10       13 | 100   CONTINUE                                                   |         00110000
                        14 |C                                                                 |         00120000
         6     11       15 |       WRITE (2,'(1X,3E20.4E2)') E                                |         00130000
         6     12       16 |       STOP                                                       |         00140000
         6     13       17 |       END                                                        |         00150000
```

```
  *** DIAGNOSTIC  LISTING ***      SIEMENS-NIXDORF FORTRAN COMPILER     FOR1 V2.2A00  DATE = ...  TIME = 11:49:53    PAGE    2
                                                     PROGRAM UNIT: STDTEST
 DO/IF SEG   STMT  I   LINE        SOURCE-TEXT                                                  COL73-80 RECORD-ID.
-------------------------------------------------------------------------------------------------------------------------------
*        1      1        1 |       PROGRAM STDTEST                                            |         00010000          *
                              ............ 1 .......................................................
   1 WARNING (FA203)  ANS FORTRAN 77 DEVIATION: IDENTIFIER TOO LONG
*        1      3        4 |       REAL  A(3,3) , B(3) ,E(3) /3*0./                           |         00030000          *
                              ............................. 1 ....................................
   1 WARNING (FA249)  ANS FORTRAN 77 DEVIATION: INITIALISATION IN TYPE STATEMENTS NOT ALLOWED
*        1      4        5 |       NAMELIST /NAM/ A,B                                         |         00040000          *
     WARNING (FA230)  ANS FORTRAN 77 DEVIATION: NAMELIST INSTRUCTION NOT STANDARD
     WARNING (SA088)  ANS FORTRAN77 DEVIATION: NAM IS NML SPECIFIER
*        1      5        6 |       WRITE *, 'ENTER VALUE FOR ARRAY A(3,3) AND VEKTOR B(3)',   |         00050000          *
*        1               7 |     1         'IN NAMELIST FORMAT /NAM/'                         |         00060000          *
     WARNING (FA259)  ANS FORTRAN 77 DEVIATION: WRITE WITHOUT CONTROL PARAM-LIST IS NOT ALLOWED
*        1      6        8 |       READ (1,NAM)                                               |         00070000          *

     WARNING (SA089)  ANS FORTRAN77 DEVIATION: NAM IS NML SPECIFIER
```

4.1.2.4 IMPLICIT option

`[*]COMOPT`	`[NO]IMPLICIT`

The NOIMPLICIT option prevents implicit type assignment for FORTRAN variables. An error message is generated in the event of IMPLICIT statements and undefined variables.

This does not affect the preset type of intrinsic functions.

4.1.2.5 EXPUNDERFLOW option

`[*]COMOPT`	`[NO]EXPUNDERFLOW`

This option sets the appropriate program mask for underflow.
This program mask is set once and only once. This option is therefore only effective in conjunction with a main program and therefore for all subprograms.

With NOEXPUNDERFLOW (default), the program is not interrupted when underflow occurs. The corresponding item is set to 0.

EXPUNDERFLOW causes the program to be interrupted and an error message to be issued when underflow occurs.

4.1.2.6 SOURCE-FORMAT option

The SOURCE-FORMAT option is used to define whether FOR1 is to compile a FORTRAN source program with program lines in columnar format (FIXED), or a free-form source program (FREE).

`[*]COMOPT`	$\texttt{SOURCE-FORMAT} = \left\{\begin{array}{l}\underline{\texttt{FIXED}}\\ \texttt{FREE}\end{array}\right\}$

<u>FIXED</u>	The source program format complies with the rules given for columns in the FOR1 Reference Manual [21].
FREE	The form of the source program is free.

If the compiler options STANDARD-CHECK=ANS77, LINEEND, UPD, CCOM and DIALOG are specified on the one hand and SOURCE-FORMAT=FREE on the other, an OPTION error message will be issued. The option specified last applies.

4.1.2.7 SAVE-CONSTANT option

[*]COMOPT	SAVE-CONSTANT=$\begin{Bmatrix}\text{YES}\\ \text{NO}\end{Bmatrix}$

The SAVE-CONSTANT option can be used to control how constants are transferred to subprograms.

Default:	YES	when OPTIMIZE = NO, 0, 1, 2 is specified
	NO	when OPTIMIZE = 3, 4 is specified

YES If an actual argument is a constant, the address of a copy of this constant is passed to the subprogram, preventing overwriting of the constants in the subprogram.

NO No copy is created, rather the address of the constants is transferred. As a result, the constants can be modified by the subprogram.

Example:

```
    PROGRAM CON
    PARAMETER (CONST=7.5)
    DO 10 I=1,5
    CALL SUB1 (CONST)
 10 WRITE *, ' MAIN PROGRAM: CONST = ',CONST
    END
    SUBROUTINE SUB1 (X)
    REAL*4 X
    X=X+2.5
    WRITE *, ' SUBPROGRAM: X     = ',X
    RETURN
    END
```

When SAVE-CONSTANT=YES is specified, the copy of the CONST constant is transferred to the subprogram. The constant is protected against overwriting in the subprogram, since only the copy is overwritten when the value is returned to the main program. Each time the loop is passed, the following is displayed:

```
SUBPROGRAM:   X     = 0.10000000E+02
MAIN PROGRAM: CONST = 0.75000000E+01
```

On the other hand when SAVE-CONSTANT=NO is specified, the value modified in the subprogram is returned to the CONST constant in the main program. The following list is output:

```
SUBPROGRAM:    X     = 0.10000000E+02
MAIN PROGRAM:  CONST = 0.10000000E+02
SUBPROGRAM:    X     = 0.12500000E+02
MAIN PROGRAM:  CONST = 0.12500000E+02
```

```
SUBPROGRAM:     X     = 0.15000000E+02
MAIN PROGRAM:   CONST = 0.15000000E+02
SUBPROGRAM:     X     = 0.17500000E+02
MAIN PROGRAM:   CONST = 0.17500000E+02
SUBPROGRAM:     X     = 0.20000000E+02
MAIN PROGRAM:   CONST = 0.20000000E+02
```

4.1.2.8 FORTRAN90-CHECK option

[*]COMOPT	FORTRAN90-CHECK = {$\underline{\text{YES}}$	NO}

Specifying the FORTRAN90-CHECK option causes a FORTRAN source program to be checked for FOR1 extensions not supported by the Fortran90 compiler. Should such extensions occur in the source program, warning messages are output. In order to distinguish these from the other error messages, these deviation messages have the suffix FORTRAN90 DEVIATION following the error type and error number, cf. for example:

```
FA301    FORTRAN90 DEVIATION: NESTED BOOLEAN IF-STATEMENTS
```

Since the deviation messages are of the WARNING error level, output is suppressed if the MSGLEVEL=ERROR option is activated.

A description of the FOR1 extensions that are no longer supported by the Fortran90 compiler is given in section 10.2 which also lists the texts of the corresponding deviation messages.

4.1.2.9 CODE option

[*]COMOPT	$\text{CODE = }\left\{\begin{array}{l}\text{([SOURCE=}\left\{\begin{array}{l}\underline{\text{EBCDIC}}\\ \text{ISO}\\ \text{BCD}\end{array}\right\}\text{][,UPD=}\left\{\begin{array}{l}\underline{\text{EBCDIC}}\\ \text{ISO}\\ \text{BCD}\end{array}\right\}\text{])}\\ \left\{\begin{array}{l}\underline{\text{EBCDIC}}\\ \text{ISO}\\ \text{BCD}\end{array}\right\}\end{array}\right\}$

This option specifies the code in which the source program and change lines of the UPDATE file are presented. If no keyword SOURCE or UPDATE is specified, the code applies both to the source program and to the change lines.

The compiler option CODE has no equivalent SDF operand.

4.2 Specifying the attributes of the generated code

4.2.1 SDF operand COMPILER-ACTION

```
START-FOR1-COMPILER

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

   MODULE-GENERATION(...)
   |
   |    SHAREABLE-CODE = NO / YES(...)
   |
   |      YES(...)
   |      |   OUTPUT-LIBRARY = *MODULE-LIBRARY / <full-filename 1..54>
   |
   |   ,MINIMAL-PRECISION = REAL-4(...) / REAL-8(...) / REAL-16(...)
   |
   |      REAL-4(...)
   |      |   EXTERNAL-DATA = NO / YES
   |
   |      REAL-8(...)
   |      |   EXTERNAL-DATA = NO / YES
   |
   |      REAL-16(...)
   |      |   EXTERNAL-DATA = NO / YES
   |
   |   ,CONSTANT-PRECISION = AS-NEEDED / REAL-4
   |
   |   ,CANCEL-CONDITION = NONE / ERROR / SEVERE-ERROR
   |
   |   ,LINKAGE = STD / FOR1-SPECIFIC
```

The SDF operands and the corresponding compiler options are shown in table 2-7.

4.2.2 Specifying the attributes of the generated code by compiler options

4.2.2.1 OBJECT option

Using the OBJECT option it is possible to control whether

- object modules are to be generated at all;
- if object modules are generated, whether these are stored in the EAM area;
- shareable object modules are generated.

```
                ┌NOOBJECT                  ┐
                │                          │
[*]COMOPT       {              ┌SHARE┐     }
                │OBJECT  [= ( {       })]  │
                └              └*    ┘     ┘
```

NOOBJECT
No object modules are generated.

OBJECT=(SHARE)
Shareable object modules are generated.
(See section 5.8).

OBJECT=(*)
By default object modules are generated and entered in the the temporary EAM file for the current task. The object modules are added to the modules already contained in the EAM file. If OBJECT=(*) is *explicitly* specified *and* MODULE-LIBRARY is specified, then the sequence of input is decisive: The more recently specified storage location overwrites the earlier specification. In such cases the compiler issues the following warning message:

```
MA 45  OBJECT OUTPUT CONFLICT
```

Note

When COMOPT NOOBJECT is specified, no object modules are generated. In this case therefore the following lists are not generated and can thus not be output:

- ESD listing
- map listing
- XREF listing
- attribute listing
- object listing

4.2.2.2 SHARE-LIBRARY option

If shareable object modules have been generated by means of the OBJECT=(SHARE) option, they can be stored in a separate PLAM library using the SHARE-LIBRARY option.

[*]COMOPT	SHARE-LIB[RARY] = $\left\{\begin{array}{l}\underline{\texttt{*MODULE-LIBRARY}}\\ \texttt{plamlib}\end{array}\right\}$

*MODULE-LIBRARY
Shareable and nonshareable object modules are stored in the same way. They are output either to the PLAM library specified by the MODULE-LIBRARY option or, by default, to the temporary EAM file.

plamlib
Name of a PLAM library in which the shareable object modules are stored. The nonshareable object modules are stored in accordance with the specifications in the MODULE-LIBRARY option. The names of the shareable object modules are determined in accordance with the same rules as for the MODULE-LIBRARY option. Maximum length including catalog ID and user ID: 54 characters.

4.2.2.3 REAL option

When compiling a program unit by means of the REAL option it is possible to increase the precision of the REAL and COMPLEX types of floating-point entries, without changing the source program. This may be necessary in the event of overflow or underflow, forced convergence of iterative processes in numerically extreme conditions, when transferring FORTRAN programs to other systems, and in other instances.

Unparenthesized option

[*]COMOPT	REAL = $\left\{\begin{array}{l}\underline{4}\\ 8\\ 16\end{array}\right\}$

Where REAL=8, all REAL variables, constants, functions and arrays with a length of 4 are converted to entities with a length of 8; all COMPLEX data with a length of 8 are converted to entities with a length of 16. REAL*16 and COMPLEX*32 type entities remain unchanged.
Where REAL=16, all REAL entities have a length of 16, while all COMPLEX entities have a length of 32.

Restrictions

- Some intrinsic functions (e.g. AMAXO) do not exist in all REAL lengths. If this is the case the compiler issues a message.
- COMMON and EQUIVALENCE will change the order of the elements. The compiler cannot check for correctness.
- If program units compiled with different REAL options are linked, errors may occur due to the different lengths of arguments and functions. Non-equal arguments in the called subprogram can be detected by specifying TESTOPT=(ARG,...).

Parenthesized option

`[*]COMOPT`	`REAL =` $\left\{\begin{array}{c}\texttt{(\underline{4})}\\ \texttt{(8)}\\ \texttt{(16)}\end{array}\right\}$

In the case of REAL=(8) and REAL=(16) options, as with REAL=8 and REAL=16, the lengths of constants, variables and arrays are increased.

The following remain unchanged, as compared to the unparenthesized option:

- Variables and arrays if they are in COMMON.
- Variables and arrays if they are actual or dummy arguments of subprograms (SUBROUTINEs) or external functions.
- External function calls, definitions and function input.
- Intrinsic functions if they are stated as actual arguments in a subprogram call.

Special considerations

- If intrinsic functions are increased in length, the arguments may possibly remain too short. If this is the case type conversion is performed at runtime and without any warning.
- Variables or arrays which only become actual arguments of user functions when the dummy arguments in functions are replaced will not remain short, they are lengthened (see example 1).
- REAL and COMPLEX constants used as actual arguments will always be lengthened. If they occur as actual arguments, REAL and COMPLEX variables will not be lengthened. Compound variable REAL and COMPLEX expressions used as actual arguments will however only remain short as long as they do not contain any constants of the REAL or COMPLEX data types; otherwise they will be lengthened (see examples 3 and 4). If this is not taken into consideration during program linkage, an errored run can be expected if TESTOPT=(ARG,...) is not specified, or an error message if TESTOPT=(ARG,...) is specified.

Example 1:

```
REAL*4 F,A,B,X
F(X)=1.+A(X)                       A(X) is an external function.
B =1.
B =F(B) → (inserted) B=1.+A(B)     With the REAL=(8) option, B is given
                                   the type REAL*8 and will not remain
                                   REAL*4, since the compiler does not
                                   recognize the undesired length until it
                                   is too late. However A stays at length 4.
```

Example 2:

```
REAL*4 F,A,B,X
F(X)=X+A(B)
B =1.
B =F(B) → (inserted) B=B+A(B)      B keeps the length REAL*4 with the
                                   option REAL=(8).
```

Example 3:

```
COMPLEX*8 C,D                      Option REAL=(8)
CALL FU(C,D+1)                     Both arguments are of type COMPLEX*8.
CALL FU(C,D+(1,0))                 The first argument is of type COMPLEX*8,
                                   the second of type COMPLEX*16.
```

Example 4:

```
COMPLEX*8 C,D                      Option REAL = (8)
CALL FUN(C,D,D+C)                  All three arguments are of the
                                   COMPLEX*8 type.
```

4.2.2.4 TRUNCONST option

[*]COMOPT	[NO]<u>TRUNCONST</u>

The NOTRUNCONST option is used to internally represent REAL constants without a REAL exponent consisting of more than 7 valid decimal digits as REAL*8 constants and those with more than 16 valid decimal digits as REAL*16 constants.

Up to 32 valid decimal digits may be employed. These rules also apply for COMPLEX constants.

4.2.2.5 GEN option

```
                ┌ GEN                            ┐
                │                                │
[*]COMOPT       {            ┌E[RROR]   ┐       }
                │ NOGEN  [=  {S[EVERE]   }]      │
                └            └F[AILURE] ┘       ┘
```

<u>GEN</u> The compiler generates object modules.

NOGEN **Without** operand value specification: the compiler merely checks for syntax and semantics; no object modules are generated.
The following listings can neither be generated nor output: ESD listing, MAP listing, XREF listing, attribute listing and object listing.

With operand value specification: no object modules are generated if an error of the degree of the GEN option or an error of greater severity occurs.

Note

Here the prefix NO does **not** denote the complementary set of parameters.

4.2.2.6 LINKAGE option

The LINKAGE option serves to specify the conventions to be used as the basis for configuring the interfaces of the generated modules.

`[*]COMOPT`	`LINKAGE = {` <u>`STD`</u>	`FOR1_SPECIFIC }`

LINKAGE = <u>STD</u>
Standardized ILCS interfaces are generated (Inter Language Communication Services = standard linkage). For further details on the standard linkage concept, see chapter 11.

LINKAGE = FOR1-SPECIFIC
Conventional FOR1-specific interfaces are generated (as with FOR1 versions ≤ 2.1).

Compilation with the (default) value LINKAGE = STD is possible only if the following are also specified:
– PROCEDURE-OPTIMIZATION = NO
– <u>NOCOMPATIBLE</u> [={BGFOR | BS3FOR}]

If incompatible option values are specified, the sequence of input determines which values are actually used:

If an option is input that is incompatible with a previously entered option, then the value of the previously entered option will be modified accordingly. Thus, for example, by entering PROCEDURE-OPTIMIZATION=YES or COMPATIBLE=BGFOR, a previously set LINKAGE=STD option will be switched to LINKAGE=FOR1-SPECIFIC.

The compiler issues warning messages in such cases:

– `MA43 LINKAGE=FOR1-SPECIFIC EXPECTED`
  is output if, prior to entry of the option which is incompatible with LINKAGE=STD, the LINKAGE option was *not explicitly* specified, e.g. in the case of:
  ```
  *COMOPT SOURCE=TEST,COMPATIBLE=BGFOR,END
  ```

– `MA20 ILLEGAL OPTION COMBINATION`
  is output if, in addition to the option which is incompatible with LINKAGE=STD, LINKAGE=STD is specified *explicitly*, e.g. in the case of:
  ```
  *COMOPT SOURCE=TEST,LINKAGE=STD,COMPATIBLE=BGFOR,END
  ```

It may also happen that *both* messages are output, i.e. when LINKAGE=STD is explicitly specified *after* the corresponding incompatible option, e.g. in the case of:

```
*COMOPT SOURCE=TEST,COMPATIBLE=BGFOR,LINKAGE=STD,END
```

Since the compiler evaluates the specified compiler options in linear fashion, on evaluation of this COMOPT statement COMPATIBLE=BGFOR first causes the preset LINKAGE=STD, which is still in effect at this time, to be changed to LINKAGE=FOR1-SPECIFIC and MA43 to be output. However, since LINKAGE=STD is subsequently specified explicitly, LINKAGE=FOR1-SPECIFIC is changed back to LINKAGE=STD, NOCOMPATIBLE is set and MA20 output.

Note

The option setting `PROCEDURE-OPTIMIZATION=STD` is the subject of special handling. If the LINKAGE=STD option is set explicitly, the compiler interprets `PROCEDURE-OPTIMIZATION=STD` as `PROCEDURE-OPTIMIZATION=NO`. This fact is indicated by the following message:

```
MA42 PROC-OPT=NO BECAUSE LINK=STD
```

4.2.2.7 UNIT option

```
                          ⎡READ ⎤          ⎡READ ⎤
                          ⎢WRITE⎥          ⎢WRITE⎥
[*]COMOPT       UNIT=(⎨     ⎬ = nn [,⎨     ⎬ = nn]...)
                          ⎢PRINT⎥          ⎢PRINT⎥
                          ⎣PUNCH⎦          ⎣PUNCH⎦
```

nn	Integer value, 0 ≤ nn ≤ 99

This option defines assignments between input/output statements and file numbers. If no file number is specified in the input/output statements of the FORTRAN source program, the file numbers specified in the UNIT option are assumed.

There is no SDF operand corresponding to this compiler option.

4.2.2.8 COMPATIBLE option

```
                                     ⎡BGFOR  ⎤
[*]COMOPT       [NO[COMPATIBLE=]]   ⎨        ⎬
                                     ⎣BS3FOR ⎦
```

The COMPATIBLE option serves to avoid incompatibilities between the FOR1 compiler and Siemens BGFOR or TR440-BS3-FORTRAN compilers.

COMPATIBLE = BGFOR:

1. OPTION NOTRUNCONST is activated.
2. OPTION PAD is activated.
3. Each DO loop is executed at least once (non-reject loop).
4. BLANK='ZERO' is preset for all files.

COMPATIBLE = BS3FOR:

Each DO loop is executed at least once (non-reject loop).

There is no SDF operand corresponding to this compiler option.

Notes

Here the prefix NO does **not** denote the complementary set of parameters.

COMOPT COMPATIBLE and COMOPT LINKAGE=STD are incompatible (see 4.2.2.6).

4.2.2.9 SUPPLIEDBOUND option

`[*]COMOPT`	`[NO]SUPPLIEDBOUND`

The SUPPLIEDBOUND option causes the dimensional entry "1" of a one-dimensional array to be interpreted as "*" in subprograms. This does not apply to arrays which have been declared as COMMON areas.

There is no SDF operand corresponding to this compiler option.

4.2.2.10 PAD option

`[*]COMOPT`	`[NO]PAD`

When COMOPT PAD is specified, input records which are too short to satisfy the condition specified in the program are filled with blanks. The COMPATIBLE=BGFOR option implicitly activates the PAD option, unless NOPAD has been specified explicitly.

There is no SDF operand corresponding to this compiler option. When the SDF command START-FOR1-COMPILER is used, PAD is the default. If LANGUAGE-STANDARD=ANS77, NOPAD is set.

Example:

```
.
.
.
CHARACTER RECORD*100
READ (10,FMT='(A100)') RECORD
.
.
.
```

If the input record length is shorter than 100 bytes, an I/O message is issued, as long as COMOPT PAD has not been specified:

```
IO3A: RECORD POINTER OUTSIDE I/O BUFFER
```

4.3 Determining the output location of the generated object module

4.3.1 SDF operand MODULE-LIBRARY

```
START-FOR1-COMPILER

,MODULE-LIBRARY = *OMF(...) / <full-filename 1..54>

   *OMF(...)

      | DELETE-OLD-CONTENTS = YES / NO
```

The SDF operands and the corresponding compiler options are shown in table 2-8.

4.3.2 MODULE-LIBRARY compiler option

The object modules generated during a compiler run can be output either to the temporary EAM file of the current task or to a PLAM library. Object module output can be controlled by means of the MODULE-LIBRARY option.

`[*]COMOPT`	`MODULE[-LIBRARY] =` $\left\{\begin{array}{l}\underline{\texttt{*OMF}}\\ \texttt{plamlib}\end{array}\right\}$

*OMF Object modules are output to the temporary EAM file as the default.

plamlib Name of a PLAM library to which the generated object modules are to be output. If the PLAM library does not yet exist, it will be created. *plamlib* may be up to 54 characters in length (length of the fully qualified file name).

Each object module contains, as a type R library element, a name formed from the names of the program unit:

- The element name is the name of the program unit if this name is not longer than 7 characters.
- The element name consists of the first 4 and last 3 characters of the program unit name if the latter contains more than 7 characters.
- The element name is $PU#nnn, beginning with nnn=000, if a main program does not include a PROGRAM statement.
- For unnamed BLOCK DATA subprograms, elements are generated with the names of the COMMON blocks.

If a PLAM library element with an identical name already exists, it is overwritten, without a message being issued.

4.4 Structure and nomenclature of object modules

Object modules are either stored directly in a PLAM library or written to the EAM area.

Object modules are composed of 80-byte records. The records can be subdivided into five different types:

- ESD (external symbol dictionary) records: The ESD records provide information about the external names that are defined or referred to in the module as well as about the COMMON areas appearing in the module. The ESD records must be placed at the beginning of a module.
- TXT (text) records:
  The TXT records contain the object program text to be loaded, i.e. the memory images of the instruction sequences, constants and initial values of data.
- RLD (relocation dictionary) records:
  The RLD records provide information about the address constants in the module. These address constants must then be modified by the linkage editor or the dynamic binder loader according to the location of the module.
- LSD (list for symbolic debugging) records: The LSD records provide information concerning the symbolic debugging in object modules. This information is necessary for using the Advanced Interactive Debugger (AID) of BS2000. LSD records are generated only if the SYMTEST=ALL option or the SDF operand TOOL-SUPPORT=AID is specified.
- END record:
  The END record is always generated at the end of an object module.

The format of the records is described in detail in the "System Standards" manual [39].

FOR1 compilation generates one or more control sections (CSECTs) for each program unit. CSECTs are the smallest entities that can be shifted during linkage.

The sections generated by FOR1 are as follows:

- Code and constant section
  For each program unit except BLOCK DATA there is exactly one code and constant section. These contain the instruction sequences generated as well as all user-defined and compiler-generated constants.
  - For compilation *without* COMOPT=(SHARE), *one* object module with the name of the program unit (prog) is generated, which also contains the code and constant section (in addition to the data section and, where applicable, COMMON sections).
  - For compilation *with* COMOPT=(SHARE), a separate object module is generated for the code and constant section. The name of this module is the name of the program unit, padded with the character "@" to a length of 8 (prog[@]...).

  BLOCK DATA program units have no code and constant section.
- Data section
  For each program unit there is exactly one data section. It contains all variable data established by the user or the compiler, and their initial values.
  - For compilation *without* COMOPT=(SHARE), *one* object module with the name of the program unit is generated, which also contains the data section (in addition to the code section, constant section and, where applicable, COMMON sections).
  - For compilation *with* COMOPT=(SHARE), a separate object module is generated for the data section (and, where applicable, COMMON sections), which bears the name of the program unit.
- COMMON section
  For each COMMON block in a program unit or BLOCK DATA program unit, FOR1 generates one COMMON section containing the name of the common block. This section contains all the data of the COMMON block and initial values.
  - For compilation *without* COMOPT=(SHARE), *one* object module is generated, which also contains the COMMON sections.
  - For compilation *with* COMOPT=(SHARE), the COMMON sections are in the same module as the data section.
  - For unnamed BLOCK-DATA program units, a separate module is generated for each COMMON section.

4.5 Object module maintenance

The FOR1 compiler can output object modules directly to PLAM libraries. The name of the PLAM library is defined in the MODULE-LIBRARY option (see 4.3).

If the MODULE-LIBRARY option is not specified, then the compiler outputs the generated object modules to the temporary object module file (OMF) in the EAM area. This file exists only for the duration of the task. However, it is also possible to transfer the object modules subsequently from the EAM area to PLAM libraries (see example).

LMS

The library management program LMS supports the creation and processing of program libraries (PLAM libraries). Source programs, INCLUDE files, UPD files, object modules, load modules, LLMs and listings, for example, can be managed in program libraries.

For further information concerning the range of functions see appendix A.10.4. A detailed description can be found in the "LMS" manual [25].

Example:

```
/ASSIGN-SYSDTA TO-FILE=QUELLE.TEST
/START-PROG $FOR1
/ASSIGN-SYSDTA TO-FILE=*PRIMARY                         (1)
/START-PROG $LMS                                        (2)
$LIB BIBL,USAGE=OUT,STATE=NEW                           (3)
$ADDR *OMF>ELEM1                                        (4)
$END                                                    (5)
```

Explanation of example:

(1) Before LMS is called, SYSDTA is returned to the primary assignment.
(2) LMS is called.
(3) The LIB statement is used to assign the name of the new output library.
(4) The ADD statement is used to add the module from the EAM area as an element ELEM1 of type R to the assigned output library.
(5) The END statement concludes the LMS run.

Library element ELEM1 of library BIBL can be further processed using the static loader TSOSLNK, the binder BINDER or the dynamic binder loader DBL (see chapter 5).

4.6 Generating compiler listings

During compilation the compiler (if required) generates a number of listings containing information about the structure of the source program and object program, information about compiler errors and information about the process of compilation. By entering options or SDF operands, the user has the ability to control the output of these listings. He may decide which of the listings are to be output and where these listings are to be written. The contents and printer format of the individual listings are described in section 4.7; examples of compiler listing are given in Appendix A.6.

4.6.1 Controlling messages and listings: SDF operand LISTING

```
START-FOR1-COMPILER
-------------------------------------------------------------------------
,LISTING = STD / NO / PARAMETER(...)

   PARAMETER(...)
     |
     |   OPTIONS = YES / NO
     |
     |  ,SOURCE = NO / YES(...)
     |
     |     | YES(...)
     |     |    |  INSERT-ERROR-WEIGHT = NOTE / WARNING / ERROR
     |
     |  ,DIAGNOSTICS = NO / YES(...)
     |
     |     YES(...)
     |       |  MINIMAL-WEIGHT = NOTE / WARNING / ERROR
     |
     |  ,DATA-ALLOCATION-MAP = YES / NO
     |
     |  ,CROSS-REFERENCE = NO / YES
     |
     |  ,EXTERNAL-DICTIONARY = NO / YES
     |
     |  ,ASSEMBLER-CODE = NO / YES
     |
     |  ,SUMMARY = YES / NO
     |
     |  ,OPTIMIZED-SOURCE = NO / YES
```

Fortsetzung>

Fortsetzung

```
    ,SORTING = BY-PROG-UNIT / BY-LIST-TYPE

    ,LAYOUT = STD / PARAMETER(...)

       PARAMETER(...)
           LINES-PER-PAGE = 64 / <integer 20..255>
          ,PAGE-EJECT-STMT = ACCEPTED / IGNORED
          ,TEXT-SEPARATOR = '|' / '!'

    ,OUTPUT = *SYSLST / <full-filename 1..54> / *STD-FILE /
              *LIBRARY-ELEMENT(...)

       *LIBRARY-ELEMENT(...)
           LIBRARY = <full-filename 1..54>
          ,ELEMENT-PREFIX = *NONE / <alphanum-name 1..38> (...)
               VERSION = *UPPER-LIMIT / <alphanum-name 1..24>
```

The SDF operands and the corresponding compiler options are shown in table 2-9.

4.6.2 Controlling messages and listings by compiler options

4.6.2.1 MSGLEVEL option (diagnostic messages and error degree)

`[*]COMOPT`	`MSGLEVEL = { N[OTE]│W[ARNING]│E[RROR] }`
	`{ (SOURCE={N│W│E}[,DIAG={N│W│E}]) }`
	`{ (DIAG={N│W│E}[,SOURCE={N│W│E}]) }`

This option controls the output of diagnostic messages in the source listing or diagnostic listing as a function of error message levels. Five different degrees of error (severity classes) may occur:

- N [OTE] Notes provide information, for example information on optimization capabilities, and are not the result of errors in the FORTRAN source program in respect of the FOR1 language. Since notes do not relate to errors, the user need not make any changes to the program, since the compiled program will still be executed properly. Notes are only output in the source or dialog listing if the error degree NOTE has been specified in the MSGLEVEL option.
- W [ARNING] Warnings point to places in the source program which are correct from a language point of view, but which generally point to logical errors. For example, warnings are issued for data items which are specified but never used, for continuous loops which may occur, for statements which can never reach execution, etc. When executing programs for which compile time warnings appear, errors may then occur or inadvertent responses may be the case. The user should therefore check to make sure that the indicated places in his or her source program are what was intended, or otherwise determine whether errors have been made.
- E [RROR] This error category delineates errors in the source program relating to the FOR1 language. The indicated errors can, however, be corrected by the compiler, so that statements in which an error is encountered may nevertheless be compiled. The corrective measures taken by the compiler are displayed in the diagnostic text (in their complete form). Examples of errors include missing parentheses in arithmetic expressions, excessively long data item names, etc.
  The corrections performed by the compiler may alter the intended meaning of the statement. For this reason elimination of the causes of errors is best performed by the user.

- S [EVERE] Severe errors are reported if the compiler recognizes errors but is unable to correct them. The compiler automatically takes action, such as assigning default values, so that it is able to continue. In the least favorable case, an invalid FORTRAN statement will be completely replaced by a CONTINUE statement.
- F [AILURE] Failures cause the compilation process to be terminated at once. Failures may be due to compiler errors as well as system errors (e.g. addressing errors).

All diagnostic messages of a severity class identical to or more severe than that specified in the MSGLEVEL option are output.

MSGLEVEL	Possible output message levels
N	N, W, E, S, F
W	W, E, S, F,
E	E, S, F

4.6.2.2 LIST option (selection of listings)

By using the LIST option the user specifies which of the possible listings are to be output to system file SYSLST. The LIST option is also used to select the listings which are output to the file defined in the LIST-OUTPUT option.

`[*]COMOPT`	`[NO]LIST   [=([listing-entry] [,...])]`

```
listing-entry:=  {ALL|MIN|OPTIONS|SOURCE|DIAG|ESD|MAP|XREF|
                 ATR|OBJECT|DECOMP|SUMMARY|CHANGE|NONE}
```

Output:

ALL	All listings except the decompiler listing
MIN	Options listing, diagnostic listing and summary listing
SOURCE	Source listing
DIAG	Diagnostic listing
ESD	ESD listing
MAP	Map listing
XREF	Cross-reference listing
ATR	Attribute listing
OBJECT	Object listing
DECOMP	Decompiler listing
SUMMARY	Summary listing
OPTIONS	Options listing
CHANGE	Listing of changes (see "Interactive Analysis", section 3.6)
NONE	No listings

When more than one LIST option is specified, the last one applies. The order in which the operands are specified bears no relation to the order in which the listings are output. The relations between PARAMETER command and LIST option/SDF operand LISTING should be borne in mind, as appropriate. The role of the NO prefix is discussed in section 2.3.1. The decompiler listing is generated only when OPTIMIZATION = 3 | 4 and output only when the operand value DECOMP is specified explicitly. All listings are therefore output only if LIST=(ALL,DECOMP) is specified.

The following cases can be differentiated:

- COMOPT LIST with operand values specified:
  The desired listings are output.
- COMOPT LIST with no operands specified:
  The default operands apply. The SOURCE, DIAG, MAP, SUMMARY and OPTIONS listings are output (standard listings).

- COMOPT LIST=(NONE):
  No output of listings.
- COMOPT NOLIST with operands specified:
  The listings specified by the operands are suppressed; all other are output.
- COMOPT NOLIST without operand specification:
  The LIST option is deactivated; no output of listings.

With the NOGEN and NOOBJECT compiler options the following listings are not generated since they are dependent on the generation of the object code and cannot therefore be output:

- ESD listing
- map listing
- XREF listing
- attribute listing
- object listing

Examples:

No LIST option
  No listing is output to SYSLST.

COMOPT LIST
  Listings corresponding to the defaults are output to SYSLST.

COMOPT LIST = (SOURCE,DIAG)
  Source program and diagnostic listing are output to SYSLST.

COMOPT NOLIST=(ATR,OBJECT)
  All listings except the attribute and object are output to SYSLST.

COMOPT NOLIST or COMOPT LIST = ( )
  No listings whatsoever are output to SYSLST.

COMOPT NOLIST = ( )
  All listings are output to SYSLST.

4.6.2.3 COLLECT option (arrangement of listings)

```
           ⎧             ⎧L[IST]                       ⎫ ⎫
           ⎪COLLECT=(    ⎨                             ⎬)⎪
[*]COMOPT  ⎨             ⎩L[IST]F[ILE] [,L[IST]]       ⎭ ⎬
           ⎪                                            ⎪
           ⎩NOCOLLECT                                   ⎭
```

LIST The listings selected with the aid of the LIST option are collected for all program units compiled during a compilation run and subsequently output, arranged by type.

LISTFILE
The listings selected by means of the LISTFILE option are collected for all program units that are compiled during a compilation run, arranged by type.

<u>NOCOLLECT</u>
The listings are output, arranged by program unit.

The COLLECT option is effective only if the listings are output directly via SYSLST, or are stored in an ISAM file.

If output is to an ISAM file, the listings which have been output are sorted with the aid of the ISAM key. If the COLLECT option is set, the following identification for the type of listing is entered in the first position of the key:

SOURCE	0	XREF	5
DIAG	1	ATR	5
CHANGE	2	OBJECT	6
ESD	3	DECOMP	7
MAP	4	SUMMARY	8
		OPTIONS	9

Attribute listings and cross-reference listings (XREF) have the same identification. The information from the attribute listing is contained in its entirety in the cross-reference listing. If both listings are requested, only the cross-reference listing will therefore be output.

4.6.2.4 LIST-OUTPUT option (output location of listings)

The LIST-OUTPUT option can be used to define where listings are output. The type and composition of the listings is selected using the LIST option. The LIST-OUTPUT option may not be used at the same time as the LISTFILE option.

If a file with the link name SAVLINK was assigned prior to compilation, the listings are output to this file, not to the one specified by the LIST-OUTPUT option.

```
                                   ⎧listfilename                           ⎫
                                   ⎪                                       ⎪
                                   ⎪*STD[-FILE]                            ⎪
                                   ⎪                                       ⎪
                                   ⎪*SYSLST                                ⎪
[*]COMOPT     LIST-OUT[PUT] =      ⎪                                       ⎪
                                   ⎪[*LIBRARY-ELEMENT]([LIBRARY=]plamlib   ⎪
                                   ⎨                                       ⎬
                                   ⎪                       ⎧prefix⎫        ⎪
                                   ⎪   [,[ELEMENT[-PREFIX]=]⎨      ⎬       ⎪
                                   ⎪                       ⎩*NONE ⎭        ⎪
                                   ⎪                                       ⎪
                                   ⎪                  ⎧version     ⎫       ⎪
                                   ⎪      [([VERSION=]⎨            ⎬)]]])  ⎪
                                   ⎩                  ⎩*UPPER-LIMIT⎭       ⎭
```

listfilename

Name of a cataloged ISAM file to which the listings are to be output. The ISAM file must have variable-length records with a key length of 8 bytes. The file *listfilename* is assigned the linkname SAVLINK, which is canceled after the completion of the compiler run.

*STD[-FILE]

When *STD-FILE is specified, the listings are output to a file with the name `SAVLST.FOR1.prog[.tsn[.time]].`
The file name is structured as follows:

prog	Name of the first program unit
tsn	four-digit task sequence number
time	Starting time of the compiler run in the form: hhmmss

Qualification of the file name using *tsn* and *time* takes place only if this is required for the catalog entry to be unambiguous. The linkname SAVLINK, which is canceled after the compiler run has been completed, is assigned to the output file.

*SYSLST

Listings are output to system file SYSLST (default value).

*LIBRARY-ELEMENT

Listings are output to a PLAM library element (of type P).

plamlib Name of the PLAM library. Maximum length including catalog ID and user ID: 54 characters.

[ELEMENT-PREFIX=]

Partial qualification of element name for emphasizing a selection of library elements.

prefix String defined by means of partial qualification of the element name. *prefix* may be up to 38 characters in length.

*NONE No partial qualification takes place (default value).

[VERSION=]

Version designation for the library element

version String denoting the version; up to 24 characters in length.

*UPPER-LIMIT

The generated listings are entered with the highest possible version designation.

The names given to PLAM library elements are dependent on the way the listings are sorted as selected by the COLLECT option:

- If the listings are arranged by program units (default value), all listings from one compiler run are stored in **one** library element having the name *prog*:

  prog When COMOPT OBJECT=(NOSHARE) is specified:
  Name of the first program unit or $PU#000 when the name is omitted;
  When COMOPT OBJECT=(SHARE) is specified:
  Name of the first program unit, padded with the character "@" to achieve a length of 8, or $PU#000@ when the name is omitted.

- If the listings are arranged by type (COLLECT=(LIST) specified), one library element having the name *prog.type* is created for each desired listing type:

  prog.type

  prog is formed as described above, *type* specifies the type of the listings contained in the library element.

4.6.2.5 LISTFILE option (output to cataloged file)

The LISTFILE option can be used to control output of log lists to an ISAM file. The listings to be output can be selected individually.
The LISTFILE option may not be used at the same time as the LIST-OUTPUT option.

```
[*]COMOPT    [NO]LISTFILE [=[listfilename] [( ⎧listing-entry⎫ [,...])]]
                                             ⎩LIST         ⎭
```

listfilename

Name of a cataloged ISAM file to which the listings are to be output. The ISAM file must have variable-length records with a key length of 8 bytes. If *listfilename* is omitted or the specified file does not meet the requirements, a separate file with the standard name

`SAVLST.FOR1.prog[.tsn[.time]]` is created, where:

prog	Name of the first program unit
tsn	Four-digit task sequence number
time	Starting time of the compilation process in the form hhmmss

In this case, "*STD-FILE" appears in the option list instead of the standard name.

Qualification of the file name with *tsn* and *time* takes place only if this is required for unambiguity of the catalog entry.

listing-entry:= {ALL|MIN|<u>OPTIONS</u>|<u>SOURCE</u>|<u>DIAG</u>|ESD|<u>MAP</u>|XREF|ATR|OBJECT|DECOMP|<u>SUMMARY</u>|CHANGE|NONE}

The meaning of the operands is the same as in the LIST option. Specification of COMOPT LISTFILE=*listfilename* without operands results in output of OPTIONS, SOURCE, MAP and SUMMARY listings, and, as required, DIAG, to the file *listfilename*. Specification of COMOPT LISTFILE without operands causes these listings to be output to a file generated by the compiler and with a standard file name.

If the user has specified COMOPT COLLECT=(LF), the listings are arranged according to type, on output.

LIST Those listings which are output to SYSLST are also written to the specified file listfilename.

The generated file can be printed using the following command, for example:

```
/PRINT-FILE listfilename, FILE-PART=PAR(FROM=9), LAYOUT-CONTR=
 PAR(CONTR-CHAR=EBCDIC)
```

4.6.2.6 LINECNT option (lines per page)

`[*]COMOPT`	`LINECNT =` $\left\{\begin{array}{l}\underline{64}\\ \text{number}\end{array}\right\}$

number Integer value

This option controls the number of lines to be output per page.
The specified number must be greater than 19 and less than 256 (19 < number < 256).

4.6.2.7 EJECT option (form feed)

`[*]COMOPT`	`[NO]`<u>`EJECT`</u>

This option controls the way form feed is carried out.

EJECT Each form feed is actually carried out.

NOEJECT
Instead of form feeds between different listings, three separator lines are generated each time. Form feeds due to %EJECT are ignored.

4.6.2.8 EXPAND option (list of insertions)

`[*]COMOPT`	`[NO]`<u>`EXPAND`</u>

This option controls the output of text included in the source program by means of the %INCLUDE statement (EXPAND mode).

EXPAND
The %EXPAND statements supplied in the source program are interpreted, the EXPAND mode being activated at the same time as a result.

NOEXPAND
The %EXPAND statements supplied in the source program cease to be effective; no %INCLUDE insertions are listed.

4.6.2.9 TEXT-SEPARATOR option (representation of vertical lines)

`[*]COMOPT`	`TEXT-SEPARATOR =` $\left\{\begin{array}{c}\underline{'\,|\,'} \\ '\,!\,'\end{array}\right\}$

The option defines how vertical lines are represented in compiler listings.

’|’	Vertical lines in compiler listings are represented by the character "|". In the German character set, "|" corresponds to the "ö" character.
’!’	Vertical lines in compiler listings are represented by the character "!".

4.6.3 Controlling the source listing with compile time statements

The print image of the source listing is controlled with the following compile time statements, which are written to the source program. The effectiveness of these %EJECT and %EXPAND statements depends on the COMOPT EJECT and COMOPT EXPAND options.

Compile time statements are handled like all other FORTRAN statements. They are only valid for the program unit in which they are specified.

With respect to FORTRAN statements, compile time statements differ as follows:
- they are terminated by comment or blank lines
- their keyword may not contain blanks.

4.6.3.1 %EXPAND statement

```
          ⎧ON ⎫
%EXPAND   ⎨   ⎬
          ⎩OFF⎭
```

%EXPAND ON
: The EXPAND mode is activated, i.e. source program parts inserted by means of the %INCLUDE statement are included in the source listing generated by the compiler (source text).

%EXPAND OFF
: Switches off the EXPAND mode.

An %EXPAND statement acts on %INCLUDE statements on the same nesting level. The %EXPAND mode is passed on to deeper nesting levels if no new %EXPAND statement follows.

Example:

```
PROGRAM A

%EXPAND OFF        File B         File C          File D
%INCLUDE B
%INCLUDE D
END             [ B1 = B2       [ C1 = C2       [ %EXPAND OFF
                [ %EXPAND ON    [ %INCLUDE D    [ D1 = D2
                [ %INCLUDE C
```

```
Result:          Compiled         Source listing          I
                 program

                 PROGRAM A         PROGRAM A
                 %EXPAND OFF       %EXPAND OFF
                 %INCLUDE B        %INCLUDE B
                 B1 = B2
                 %EXPAND ON
                 %INCLUDE C
                 C1 = C2           C1 = C2                2
                 %INCLUDE D        %INCLUDE D             2
                 %EXPAND OFF       %EXPAND OFF            3
                 D1 = D2           D1 = D2                3
                 %INCLUDE D        %INCLUDE D
                 %EXPAND OFF
                 D1 = D2
                 END               END
I:INCLUDE level
```

4.6.3.2 %EJECT statement

```
%EJECT
```

Under the EJECT option this statement produces a form feed in the source program listing.

4.6.3.3 %SPACE statement

```
%SPACE n
```

n	Integer value: $0 \le n \le 255$

This statement causes n blank lines to be inserted in the source listing. However only the number of blank lines which still fit into the same page will be inserted. Blank lines are printed in the source listing only if CONTROL-CHARACTER=EBCDIC is specified in the PRINT-FILE command.

4.6.3.4 %TITLE statement

```
%TITLE ['text']
```

This statement has the same effect as %EJECT. In addition, the specified text is printed in the header line of each subsequent page in the source listing of the same program unit. The maximum length of the text is 55 characters. If an apostrophe is to be contained within the text, two apostrophes must be written.

If a %TITLE statement is contained within the first *n* lines of a source program ((*n*: 64 or a number of lines defined in the LINECNT option), *text* is printed on the first page of the source listing and no form feed is carried out.

4.7 Description of compiler listings

This section defines the contents and print image of the listings generated by LIST and LISTFILE. The specified listings refer to the same compilation process.

Listing header

Two header lines are written at the beginning of each page for all listings:

- The first line specifies the type of listing, a title, the date and time when compilation started, as well as a consecutive page number. For the source listing, the predefined standard title "SIEMENS-NIXDORF FORTRAN COMPILER FOR1 Vn.nn" may be replaced by a user-supplied title specified in the %TITLE statement of the FORTRAN source program.
- The second line indicates the name of the program unit to which the listing refers. The date is output in ISO format (yyyy-mm-dd).

4.7.1 Options listing

Output of this listing is effected by supplying the operand value OPTIONS in the LIST or LISTFILE option.

By default the options listing is output to SYSLST. The options listing consists of three parts:

- Task environment
- List of specified options (options file)
- List of options in effect

See Appendix A.6.10 for an example of an options listing.

Only the operand values defined by COMOPTs appear in the options listing, even if other operand values are in effect as the result of % statements (%INCLUDE, %FPOOL).

"Task environment" displays, besides other information, the task sequence number (TSN) of the task.
This 4-digit number is assigned to a task at the LOGON command and remains effective until the LOGOFF command. The TSN is part of the name generated for a file by the compiler if there is no user-specified file name.

"Options file" also includes messages about invalid options.

In the list of "options in effect", those options for which the user supplied no entries or only invalid entries, and which therefore are governed by the predefined default values, are marked with D (default). A "P" denotes options which have been activated by the PARAM command (see A.4).

The options listing appears at the end of the complete listing.

4.7.2 Source listing

Output of the source listing is achieved by supplying the operand value SOURCE in the LIST or LISTFILE option.

Standard output of this listing is to SYSLST. Sections A.6.1 and A.6.2 show source listings).

Entries in the columns of the source listing have the meaning shown below. If a given line has no entry in a particular column, the last preceding entry in that column applies.

Column	**Meaning**
DO/IF	Nesting depth of the DO loops (except implicit DO) and of BLOCK IF
SEG	Number of the highest segment reached in the FORTRAN statement. A segment refers to the greatest possible sequence of statements that all have the same entry point. In program execution, all statements of a given segment are processed sequentially. The segments of a compiled program are numbered consecutively, beginning with 1. An I/O statement with implicit DO may comprise several segments. As parts of statements may be shifted in the course of loop optimization, it is possible for a statement to move to another segment or to be distributed among two or more segments (see section 9.5.5). In this case, the source listing does not indicate any of the segment numbers moved. The segment numbers specified serve to interpret the results of the debug statement %COUNT (see 7.4.7). If the compiler option OPT={3|4} or SOURCE-FORMAT=FREE is specified, no segment numbers are displayed.
STMT	Consecutive number of the FORTRAN statement counted for each program unit, starting with 1 (no source program line).

I/H	Nesting depth / hierarchical level I Nesting depth of the text parts inserted by means of the %INCLUDE statement. H Hierarchical level of the PLAM-INCLUDE item. H refers to the order of PLAM libraries as specified in the INCLUDE option. Thus, for example, 1/3 designates the PLAM library specified in the third position of the INCLUDE option. The INCLUDE item has a nesting depth of 1.
LINE	Consecutive line number, counted for the entire compilation, starting with 1. For text parts inserted by the %INCLUDE statement, line numbering restarts separately from 1. Thus the line numbers are usually the same as the serial numbers in the corresponding library elements, which makes changes and corrections easier. Exceptions to this are source programs that are modified temporarily (following the first modified line) and elements inserted by the %INCLUDE statement if text substitutions cause continuation lines to be generated (from the first continuation line generated onwards).
RECORD-ID.	Key of the ISAM records if the source program was read from an ISAM file.

Message in the event of an error

During compilation, the compiler may generate diagnostic messages, which appear on the source listing (messages about invalid options are issued on the options listing). Diagnostic messages, while immediately following the lines they are associated with, are designed so as not to disturb the listing image of the actual source program text. The statements concerned are marked by an asterisk in both margins of the page. If a diagnostic message relates to a particular column position in the statement, that column will be marked in the diagnostic message. This marking is also a numbering of the diagnostic messages for a statement. Diagnostic messages that do not relate to a particular statement column position are written at the end of the messages for that statement.

In the left margin the message level is displayed and an error number issued.

In the right margin a short diagnostic text is output, which usually suffices for correction.

The action taken by the compiler as a result of the diagnostics is documented in the full message text in the diagnostic listing.

The MSGLEVEL option may be used to determine the message level (severity class) from which messages are to appear in the source listing.

4.7.3 Diagnostic listing

Diagnostic listing output is achieved by supplying the parameter value DIAG in the LIST or LISTFILE option.

Standard diagnostic listing output is to SYSLST. The diagnostic listing displays the messages about invalid options as well as all messages generated during source program compilation.

Appendix A.6.1 shows a diagnostic listing.

The statements to which the messages relate are shown in the diagnostic listing in the same form as in the source listing. In addition to the diagnostics, the corrective action taken by the compiler is listed in the diagnostic listing.

If a message relates to a specific column in a source program line, then the column position will be marked by a number in the line beneath. If several messages relate to the same column of a source program line, then this column position will be marked only once, with the number corresponding to the first of these messages.

4.7.4 Listing of external names (ESD LISTING)

Output of this listing is achieved by supplying the operand value ESD in the LIST or LISTFILE option.

Standard output of this listing is to SYSLST. If COMOPT NOGEN or COMOPT NOOBJECT is set, no ESD listing can be produced since it is dependent on the generation of the object code.

This listing provides information about the external (linkable) names of the program unit. The names establish the connection between the program unit and other program units, runtime routines and external procedures. The ESD information which is output corresponds to the ESD records which are generated in the course of compilation.

Appendix A.6.4 shows an ESD listing.

The columns of the ESD listing have the following meanings:

Column	Meaning
IDENTIFIER	External name. This name may either be defined by the user, e.g. in an EXTERNAL statement; or the name may be generated by the compiler, e.g. the name of the data section for the program unit; or it may be predefined, e.g. the entry point in a predefined function. If a user-defined name is more than 7 characters in length, it is truncated to 7 characters for the appropriate ESD record, by using only the first 4 and the last 3 characters. The listing will show the abridged name. It is the responsibility of the user to make sure that this truncation will not create duplicate names. Within the program unit, however, the complete name applies.
ESID	Number of the external name. The external names are numbered consecutively for each module, starting with 1.
TYPE	Type of the external name. There are four types of external names: SD Section definitions. These relate to the names of the control sections (CSECTS) of the program units. The nomenclature for the generated control sections is described in section 4.4. CM Names of COMMON blocks external to BLOCK DATA program units (COMMON block requests). In the case of an unnamed COMMON block, 8 blank characters are assumed as a name. VC V constants, external names that point to code sections and code entry points of other programs. Used for calling subprograms. LD Entry points (label definition); i.e. external names defined in the program entry (ENTRY statement). At link-edit time, an attempt is made to associate each request of an external definition (V constant, COMMON block request) with an existing definition of an external name (section definition, entry point).

DISPL	Displacement from the beginning of the module, for external name definition. This displacement is specified in bytes in hexadecimal notation. When external definitions are requested, the displacement shown from the beginning of the module is that of the positions in which the address of the requested COMMON block or subprogram is stored (address constant).
LENGTH	Length of a section or a COMMON block, hexadecimal in bytes. No length entry appears for V-constants and entry points.

4.7.5 Map listing

Output of the map listing is achieved by specifying the operand value MAP in the LIST or LISTFILE option. Standard map listing output is to SYSLST. If COMOPT NOGEN or COMOPT NOOBJECT is set, no map listing can be produced since it is dependent on the generation of the object code.

Information about the arrangement of the data items used in the program unit is output to the map listing under the headings SYMBOL, TYPE and ADR. The map listing is subdivided according to the sections which are generated for the program unit (CODE + CONSTANTS SECTION, LOCAL DATA SECTION, COMMON DATA SECTION). Appendix A.6.5 shows a map listing.

For the code and constants section, it provides information on the occurrence of statement labels. For statement labels that occur in assigned or computed GOTOs as well as for invoked functions and subprograms, the listing also describes the addresses in which the addresses of these items are stored. For the data section, the listing contains information about the data in that program unit as well as, for arrays and CHARACTER variables, the associated descriptors.

For the COMMON section, the listing only contains information about the data and about the descriptors of dynamic arrays in the COMMON area. The information about the associated descriptors is provided in the data section.

There are two forms of map listing output:
- sorted in alphabetical order
- sorted according to ascending displacement of items from the beginning of the module.

The map listing columns have the following meanings:

Column	Meaning
SYMBOL	Name of the described item Descriptors are marked by an appended .D; addresses, by an appended .A.
TYPE	Attributes of the items described.
ADR	Displacement of the items described, from the beginning of the module, hexadecimal in bytes.

4.7.6 Cross-reference listing

Output of this listing is achieved by specifying the operand value XREF in the LIST or LISTFILE option. It is not output by default. If COMOPT NOGEN or COMOPT NOOBJECT is set, no cross-reference listing can be produced since it is dependent on the generation of the object code.

Appendix A.6.6 shows a cross-reference listing.

The XREF listing provides information about attributes and references for all symbolic names and statement labels which occur in the program unit. Under IDENTIFIER, names are entered in succession:
– symbolic names, sorted in alphabetical order
– statement labels, sorted in ascending order.

Further entries are provided for each name under the following columns:

Column	**Meaning**
DISPL	Displacement of the items described from the beginning of the module, hexadecimal in bytes.
DESCR	Displacement of the descriptors and addresses from the beginning of the module, hexadecimal in bytes.
SPEC	Statement number where the name was declared. "*" means that the name was not declared explicitly.
ATTRIBUTES AND REFERENCES	1. Attributes of the name 2. Listing of statement numbers where the name occurs. Some numbers are preceded by marks as follows: – "/" indicates that the name is declared there; there are three types of declarations (type statement, DIMENSION statement, COMMON statement). – "=" indicates that the value of the variable may change during program execution (by assignment, actual argument, I/O listing, etc.). Accordingly, multiple marks preceding the same statement number indicate multiple value assignments within one statement. If the same statement label has both marked and unmarked entries, then the unmarked ones are shown first.

4.7.7 Attribute listing

Output of this listing is achieved by specifying the operand value ATR in the LIST or LISTFILE option. It is not output by default. If COMOPT NOGEN or COMOPT NOOBJECT is set, no attribute listing can be produced since it is dependent on the generation of the object code.

The information from the attribute listing is contained in its entirety in the cross-reference listing. Therefore, when both listings are requested, only the XREF listing is output.

4.7.8 Object listing

This listing is not output by default.

Output of this listing is achieved by supplying the operand value OBJECT in the LIST or LISTFILE option.

If the NOGEN or NOOBJECT option is in effect, no object program text is generated and, consequently, no object listing can be output.

In the object listing, the compiler-generated object program text for the code and constant section is shown in the form of an assembly listing. Since a BLOCKDATA program unit does not contain a code and constant section, there is no object listing output for a BLOCKDATA program unit.

Appendix A.6.7 shows an object listing.

The layout of the object listing is the same as that of the listings generated by the assembler.

The columns of the object listing have the following meanings:

Column	Meaning
FLG	Flag of the instruction sequences generated for an invalid FORTRAN statement. In this case an asterisk appears on the comment line, indicating the beginning of an instruction sequence for a FORTRAN statement.
DISPL	Displacement from the beginning of the module, hexadecimal in bytes.
OPERATION	Object program text in hexadecimal form.

ADDR1,ADDR2	Addresses used in the instructions. These addresses are the contents of the base registers added to the displacement. They indicate the displacement from the beginning of the module and thus correspond to the entries in the DISPL column.
STMNT	Instruction numbering.
ASSEMBLY CODE	Generated assembly code. The numbers that occur here are decimal.
SYMB.ADDR1, SYMB.ADDR2	Names used in the FORTRAN source program. For constants, the edited value is output. The form of this output depends on the type of constant. Since the output for a given instruction does not exceed the line boundary, constants of greater length may be truncated. The entities generated by the compiler are in the following form:

```
%Tnnnnnnnn     Temporary auxiliary variable
               nnnnnnnn     Hexadecimal digits

%Vnn           Compiler variable
               nn           Decimal digits

%Lnnnnn        Internal statement label
               nnnnn        Decimal digits

%Inn           Iteration counter for DO loops
               nn           Decimal digits
```

The statement labels of the FORTRAN source program also appear in the assembly code; they are marked by the preceding character #. In the event of transfer of control to another module, the module name is shown with the load instruction of the appropriate address.

In addition, comment lines are included in the assembly code. These comment lines are placed at the beginning of segments and FORTRAN statements, as well as at the beginning of what is referred to as code "slices" (SLICE no.), i.e. code areas with a fixed position of the code base register. Comment lines at the beginning of a statement indicate the type of statement and the corresponding FORTRAN statement from the FORTRAN source program. For statements shifted in the course of optimization, the type of statement is given as MOVED STMT.

4.7.9 Decompiler listing

Output of this listing is only achieved by explicitly specifying the DECOMP operand in the LIST or LISTFILE option. When specifying (ALL) in the LIST or LISTFILE option, the decompiler listing is not displayed. A prerequisite for output of the decompiler listing ist that OPT=3 or 4 has been specified.

The optimization results in changes to the code (see chapter 9), so that reference to the source program can no longer readily be made when debugging with interactive debugging aids. As the result of optimization, the order of statements may be changed, for example; a statement can be split up into several statements or can be completely omitted. Clear-defined tracing of execution or setting of test points on the basis of the source program is therefore no longer possible.

Updates as a result of optimization (OPTIMIZE=3 or 4) can be displayed in the decompiler listing, since the decompiler listing is created from information which contains the code generation of the optimization. The decompiler listing provides a high-level description of the object code, which facilitates debugging of an optimized program (which actually cannot be further debugged). When variables are loaded to registers or whether or not register contents are saved, and when they will be saved, can, however, only be traced with certainty with the aid of the object listing.

Compared to the original source program, the decompiler listing offers a number of special features:

1. The decompiler listing contains not only statements from the original source program, but also variables and auxiliary variables which the compiler stores internally. To distinguish them from FORTRAN source

   program variables, the names of the internal variables begin with a "%" character:

```
%Tnnnnnnnn    Temporary auxiliary variable
              nnnnnnnn    Hexadecimal digits

%Vnn          Compiler variable
              nn          Decimal digits

%Inn          Iteration counter for DO loops
              nn          Decimal digits
```

   These internal variables are listed under the same names in the object listing.

   In addition to internal variables, internal statement labels also occur in the decompiler listings:

```
Lnnnnn        internal statement label
              nnnnn       decimal number, up to 5 digits in length
```

   In the object listing, an internal statement label is displayed in the form %Lnnnnn.

   No LSD information is generated for the internal variables and auxiliary variables.

2. All arrays with the exception of dynamic arrays are declared as one-dimensional arrays in the decompiler listing. The upper and lower bounds of the array are defined relative to the imagined address of A(0,0,...,0) (see section 9.3.4) and displayed in the decompiler listing.
3. For all arrays, addressing of an array element takes place as if the array was a one-dimensional array. In the decompiler listing, addressing is not represented via the subscript of an array element, rather via addressing in bytes, divided by length of the data type.

   In the case of constant subscripts and constant array element sizes, the result of this division is displayed in the decompiler listing. (The division is not performed in object code because only byte addressing is used on object code level.)
4. During optimization, common subexpressions (array multipliers) are created as the result of subscript expansion (see 9.3.4). In the case of arrays with variable bounds as dummy arguments and in the case of dynamic arrays, the array multipliers are not yet known at compilation time. The array multipliers, which cannot be calculated until runtime, are displayed in the decompiler listing in the following form:

```
%array02, %array03, ... , %arrayn
           array multipliers of the array named "array"

           array       Name of a dynamic array or of a formal array
                       with variable bound

           n           Dimension of the array named "array"
```

5. EQUIVALENCE and DATA statements as well as BLOCKDATA subprograms are not decompiled, i.e. the decompiler listing does not contain these statements.
6. Input/output statements are not decompiled into READ or WRITE statements; instead the corresponding runtime routine calls are displayed.
7. Machine-dependent optimizations, i.e. register allocations (maintenance of frequently used variables in registers) are not displayed in the decompiler listing.

Examples

Example 1: Declaration of arrays

```
  ****  SOURCE  LISTING  ****      SIEMENS-NIXDORF FORTRAN COMPILER ...
                                                    PROGRAM UNIT: SAMPLE
 DO/IF SEG   STMT  I/H LINE        SOURCE-TEXT
--------------------------------------------------------------------------
*      1/1      1         1 |      PROGRAM SAMPLE
         1      2         2 |      INTEGER * 4 DYNARRAY(:,:)
         1                3 |     *           ,ARRAY(-1:1,-2:2)
         1      3         4 |      END

  *** DECOMPILER  LISTING ***      SIEMENS-NIXDORF FORTRAN COMPILER ...
                                                    PROGRAM UNIT: SAMPLE
    STMT | DECOMPILED TEXT
--------------------------------------------------------------------------
       1 |      SAMPLE PROGRAM
       1 |      ABNORMAL
       1 |      INTEGER  *  4 DYNARRAY (:,:)
         | *    *** DECLARATION OF ARRAY MULTIPLIERS ***
       1 |      INTEGER  * 4 %DYNARRAY02
         | *    *** END OF ARRAY MULTIPLIER DECLARATION ***
       1 |      INTEGER  *  4 ARRAY (-7:7)
         | ***** STATEMENT 3 (END) ****************
       3 |      END
```

For the dynamic array DYNARRAY(:,:), the array multiplier %DYNARRAY02 is created. ARRAY(-1:1,-2:2) is displayed in the decompiler listing as a one-dimensional array.

Example 2: Arrays in subprograms

```
 ****  SOURCE  LISTING  ****      SIEMENS-NIXDORF FORTRAN COMPILER ...
                                                    PROGRAM UNIT: SUBARR
DO/IF SEG   STMT  I/H LINE        SOURCE-TEXT
---------------------------------------------------------------------------
      1/1      1         1 |      SUBROUTINE SUBARR ( ARR1, ARR2, ARR3 )
        1      2         2 |      INTEGER * 4 ARR1(-5:5,M,N)
        1                3 |     *           ,ARR3(0:10,0:N)
        1      3         4 |      INTEGER * 2 ARR2(0:10,0:*)
        1      4         5 |      COMMON      M,N
        1      5         6 |      ARR1(1,1,1) = 5
        1      6         7 |      ARR2(I,J) = 5
        1      7         8 |      END

 *** DECOMPILER  LISTING ***      SIEMENS-NIXDORF FORTRAN COMPILER ...
                                                    PROGRAM UNIT: SUBARR
   STMT | DECOMPILED TEXT
---------------------------------------------------------------------------
      1 |       SUBROUTINE SUBARR ( /ARR1 /,/ARR2 /,/ARR3 /)
      1 |       ABNORMAL
      1 |       INTEGER  *  4 M
      1 |       INTEGER  *  4 N
      1 |       INTEGER  *  4 I
      1 |       INTEGER  *  4 J
      1 |       COMMON /   / M, N
        | *     *** DECLARATION OF AUXILIARY VARIABLES ***
      1 |       INTEGER * 4 %T00010000
      1 |       INTEGER * 4 %T00010044
      1 |       INTEGER * 4 %T000100CC
      1 |       INTEGER * 4 %T00010110
      1 |       INTEGER * 4 %T00010154
        | ***** STATEMENT 2 (DIMENSION) **********
      2 |       INTEGER * 4 ARR1(-5+
        |      .                 (11)+
        |      .                 (11*(M )) :
        |      .                 5+(11)*(M)+
        |      .                 (11*(M ))*(N))
        | *     *** DECLARATION OF ARRAY MULTIPLIERS ***
      2 |       INTEGER * 4 %ARR103
      2 |       %ARR103 = 11*(M )
        | *     *** END OF ARRAY MULTIPLIER DECLARATION ***
        | ***** STATEMENT 2 (DIMENSION) **********
      2 |       INTEGER * 4 ARR3(0+0 :
        |      .                 10+(11)*(N))
        | ***** STATEMENT 3 (DIMENSION) **********
      3 |       INTEGER * 2 ARR2(*)
        | ***** STATEMENT 1 (ENTRY) **************
        | ***** STATEMENT 5 (ASSIGNMENT) *********
      5 |       %T00010000=11+%ARR103
      5 |       %T00010044=%T00010000*4
      5 |       ARR1((4+%T00010044)/4)=5
        | ***** STATEMENT 6 (ASSIGNMENT) *********
      6 |       %T000100CC=J*11
      6 |       %T00010110=I+%T000100CC
      6 |       %T00010154=%T00010110*2
      6 |       ARR2(%T00010154/2)=5
        | ***** STATEMENT 7 (END) ****************
      7 |       END
```

ARR1(-5:5,M,N) is a formal array with variable bounds. ARR1 is declared as a one-dimensional array, whereby the lower and upper bounds are displayed as follows (see also section 9.3.4):

$A_s=A_0 + l(s_1*m_0 + ... + s_n*m_{n-1})$

The bounds are represented relative to A_0:

```
ARR1( u1*m0                             ARR1( -5*1
     +u2*m1                                    +1*11
     +u3*m2 ) :          corresponds to        +1* (11*M)) :
    ( o1*m0                                  (  5*1
     +o2*m1                                    +M*11
     +o3*m2 )                                  +N*(11*M))
```

Expression 11*M appears here repeatedly and is used as array multiplier %ARR103.

For ARR3, the bounds are calculated as follows:

$(u_1*m_0 + u_2*m_1 : o_1*m_0 + o_2*m_1) = (0 + 0 : (10*1) + (N*11))$

Addressing of array element ARR1 is accomplished with the aid of temporary auxiliary variables and with the aid of array multiplier %ARR103 = 11*M. The address of array element ARR1(1,1,1) relative to the imagined address A0 is:

$l(s_1*m_0 + s_2*m_1 + s_3*m_2) = 4*(1*1 + 1*11 + 1*11*M)$

%T00010000 = 11 + (11*M) results in ARR1(4 + %T00010044) = 5.
Subscript list value expression "4+%T00010044" refers to addressing in bytes, so that this value must be divided by array element size 4, to obtain the subscript list value: ARR1((4+%T00010044)/4) = 5. ARR2(I*1+J*11) is determined following the same procedure as for ARR2(I,J). The auxiliary variable is multiplied by 2, according to the length of the INTEGER 2 variable and then divided again to obtain the subscript list value.

Example 3: CHARACTER variables

```
  ****  SOURCE  LISTING  ****      SIEMENS-NIXDORF FORTRAN COMPILER ...
                                                     PROGRAM UNIT: CHAR
 DO/IF SEG   STMT  I/H LINE        SOURCE-TEXT
-----------------------------------------------------------------------
       1/1      1         1 |      SUBROUTINE CHAR ( CHAR1, CHAR2, CHAR3 )
         1      2         2 |      CHARACTER * (*)   CHAR1
         1      3         3 |      CHARACTER * (*,V) CHAR2
         1      4         4 |      CHARACTER * (N)   CHAR3
         1      5         5 |      COMMON      N, M
         1                6 |
         1      6         7 |      CHAR1(:)   = CHAR2(:M)
         1      7         8 |      CHAR3(M:) = CHAR1(M-1:M+3)
         1      8         9 |      END

  *** DECOMPILER  LISTING ***      SIEMENS-NIXDORF FORTRAN COMPILER ...
                                                     PROGRAM UNIT: CHAR
    STMT | DECOMPILED TEXT
-----------------------------------------------------------------------
       1 |      SUBROUTINE CHAR ( /CHAR1 /,/CHAR2 /,/CHAR3 /)
       1 |      ABNORMAL
       1 |      CHARACTER* (*) CHAR1
       1 |      CHARACTER*( * ,V) CHAR2
       1 |      INTEGER  *  4 N
       1 |      INTEGER  *  4 M
       1 |      COMMON /   / N, M
         | *     *** DECLARATION OF AUXILIARY VARIABLES ***
       1 |      INTEGER * 4 %T00010000
       1 |      INTEGER * 4 %T00010044
       1 |      INTEGER * 4 %T00010088
       1 |      INTEGER * 4 %T000100CC
       1 |      INTEGER * 4 %T00010110
       1 |      INTEGER * 4 %T00010154
       1 |      INTEGER * 4 %T00010198
       1 |      INTEGER * 4 %T000101DC
         | ***** STATEMENT 4 (CHAR) ***************
       4 |      CHARACTER*(N) CHAR3
         | ***** STATEMENT 1 (ENTRY) **************
         | ***** STATEMENT 6 (ASSIGNMENT) *********
       6 |      CHAR1=CHAR2(:M)
         | ***** STATEMENT 7 (ASSIGNMENT) *********
       7 |      %T00010110=M-1
       7 |      %T00010154=M+3
       7 |      CHAR3(M:)=CHAR1(%T00010110:%T00010154)
         | ***** STATEMENT 8 (END) ****************
       8 |      END
```

The temporary variables listed here are required for subchain processing. The initial and final positions of the subchain are replaced by temporary auxiliary variables.

Example 4: Output statements

```
 ****  SOURCE  LISTING  ****      SIEMENS-NIXDORF FORTRAN COMPILER    FOR1 V2.2A00
                                                    PROGRAM UNIT: IO
DO/IF SEG   STMT  I/H LINE        SOURCE-TEXT
--------------------------------------------------------------------------
      1/1      1         1 |      SUBROUTINE IO ( IAR )
        1      2         2 |      INTEGER * 4 IAR(10),
        1                3 |     *            INCREMENT
        1      3         4 |      N = 2
        1      4         5 |      WRITE(10,FMT=99) N
        3      5         6 |      WRITE(2,FMT='(I4)') (IAR(J),J=1,10)
        3      6         7 |      RETURN
        3      7         8 |99    FORMAT(I4)
        3      8         9 |      END

 *** DECOMPILER  LISTING ***      SIEMENS-NIXDORF FORTRAN COMPILER ...
                                                    PROGRAM UNIT: IO
   STMT | DECOMPILED TEXT
--------------------------------------------------------------------------
      1 |       SUBROUTINE IO ( /IAR /)
      1 |       ABNORMAL
      1 |       INTEGER  *  4 IAR (1:10)
      1 |       INTEGER  *  4 INCREMENT
      1 |       INTEGER  *  4 N
      1 |       INTEGER  *  4 J
      1 |       INTEGER  *  4 %I1
        | *     EXTERNAL PROC. REFERENCE IF@XICA
        | *     EXTERNAL PROC. REFERENCE IF@XFCO
        | *     EXTERNAL PROC. REFERENCE IF@XTCA
        | *     *** DECLARATION OF AUXILIARY VARIABLES ***
      1 |       INTEGER * 4 %T00010154
      1 |       INTEGER * 4 %T00010220
        | ***** STATEMENT 1 (ENTRY) **************
        | ***** STATEMENT 3 (ASSIGNMENT) *********
      3 |       N=2
        | *     ******** BEGIN OF I/O - STATEMENT *******
        | ***** STATEMENT 4 (WRITE) **************
      4 |       CALL IF@XICA('FORMATTED,SEQUENTIAL_FILE ,WRITE/ENCODE',
        |      .             UNIT      = 10,
        |      .             FMT       = '(I4)')
      4 |       CALL IF@XFCO(N)
      4 |       CALL IF@XTCA
        | *     ******** END OF I/O - STATEMENT *******
        | *     ******** BEGIN OF I/O - STATEMENT *******
        | ***** STATEMENT 5 (WRITE) **************
      5 |       CALL IF@XICA('FORMATTED,SEQUENTIAL_FILE ,WRITE/ENCODE',
        |      .             UNIT      = 2,
        |      .             FMT       = '(I4)')
        | *     ******** END OF I/O - STATEMENT *******
        | ***** STATEMENT 5 (MOVED STMT) *********
      5 |       %T00010220=4
      5 |       %I1=10
      5 | L12   CALL IF@XFCO(IAR((%T00010220)/4))
        | ***** STATEMENT 5 (INCR STMT) **********
      5 |       %T00010220=%T00010220+4
      5 |       %I1 = %I1- 1
      5 |       IF (%I1 .NE. 0) GOTO L12
      5 | L13   CALL IF@XTCA
        | ***** STATEMENT 6 (RETURN) *************
      6 |       RETURN
      6 |       END
```

Input/output statements are represented by the corresponding runtime calls. In addition, the respective parameters of the runtime routines (type, unit, format, etc.) are listed.

Example 5: DO loop

```
  ****  SOURCE  LISTING  ****      SIEMENS-NIXDORF FORTRAN COMPILER ...
                                                    PROGRAM UNIT: LOOP
 DO/IF SEG   STMT  I/H LINE        SOURCE-TEXT
----------------------------------------------------------------------
*      1/1      1         1 |      SUBROUTINE LOOP (N)
         1      2         2 |      INTEGER * 4 IAR(10),
         1                3 |     .            INCREMENT
         1      3         4 |      DO 10,I=1,N,INCREMENT
   1     3      4         5 |10       IAR(I) = 5
         3      5         6 |      END

  *** DECOMPILER  LISTING ***      SIEMENS-NIXDORF FORTRAN COMPILER ...
                                                    PROGRAM UNIT: LOOP
    STMT | DECOMPILED TEXT
---------------------------------------------------------------------
       1 |      SUBROUTINE LOOP ( N )
       1 |      ABNORMAL
       1 |      INTEGER  *  4 N
       1 |      INTEGER  *  4 IAR (1:10)
       1 |      INTEGER  *  4 INCREMENT
       1 |      INTEGER  *  4 I
       1 |      INTEGER  *  4 %I1
       1 |      INTEGER  *  4 %V1
       1 |      INTEGER  *  4 %V2
         | *    *** DECLARATION OF AUXILIARY VARIABLES ***
       1 |      INTEGER * 4 %T00010000
       1 |      INTEGER * 4 %T00010044

       1 |      INTEGER * 4 %T000100CC
       1 |      INTEGER * 4 %T00010110
       1 |      INTEGER * 4 %T00010154
         | ***** STATEMENT 1 (ENTRY) **************
         | ***** STATEMENT 3 (DO) *****************
       3 |      %T00010000=N-1
       3 |      %T00010044=%T00010000+INCREMENT
       3 |      %I1=%T00010044/INCREMENT                         (1)
       3 |      IF (%I1 .LE. 0) GO TO L13
         | ***** STATEMENT 4 (MOVED STMT) *********
       4 | L14  %T00010110=4                                     (2)
       4 |      %T00010154=INCREMENT*4
         | ***** STATEMENT 4 (ASSIGNMENT) *********
       4 | L3   CONTINUE                                         (7)
       4 | 10   IAR(%T00010110/4)=5                              (3)
         | ***** STATEMENT 4 (INCR STMT) **********
       4 |      %T00010110=%T00010110+%T00010154                 (4)
         | ***** STATEMENT 4 (DOEND) **************
       4 |      %I1 = %I1- 1                                     (5)
       4 |      IF (%I1 .NE. 0) GOTO L3                          (6)
       4 | L15  CONTINUE
         | ***** STATEMENT 5 (END) ****************
       5 | L13  END
```

In the decompiler listing, loop initialization, loop control and loop continuation of an optimized DO loop are evident (cf. chapter 9):

(1) The number of loop passes is calculated as
(final value - begin value + increment) / increment.
This value is determined with the aid of temporary auxiliary variables and iteration counter %I1.

(2) The initial value and increment of the loop are assigned prior to the loop range and expressed in the form of bytes.

(3) In the loop range, the DO variable %T00010110 is again divided by the size, in order to obtain the subscript list value.

(4) The DO variable is increased by the distance between the array elements.

(5) The iteration counter is reduced by 1.

(6) The iteration counter is checked for 0.

(7) If %I1 is not equal to zero, the loop range is passed again.

Example 6: Optimizations in the loop range

```
 ****  SOURCE  LISTING  ****      SIEMENS-NIXDORF FORTRAN COMPILER ...
                                                   PROGRAM UNIT: OPT
DO/IF SEG   STMT  I/H LINE        SOURCE-TEXT
------------------------------------------------------------------------
      1/1      1         1 |      PROGRAM OPT
        1      2         2 |      DO 1 I=1,5
  1     1      3         3 |      L=7
  1     2      4         4 |      M=M+N*L
  1     2      5         5 |      K=I*3+L*4
  1     4      6         6 |    1 N=N*7+K
        4      7         7 |      END
 *** DECOMPILER  LISTING ***      SIEMENS-NIXDORF FORTRAN COMPILER ...
                                                   PROGRAM UNIT: OPT
   STMT | DECOMPILED TEXT
------------------------------------------------------------------------
      1 |       PROGRAM OPT
      1 |       ABNORMAL
      1 |       INTEGER  *  4 I
      1 |       INTEGER  *  4 L
      1 |       INTEGER  *  4 M
      1 |       INTEGER  *  4 N
      1 |       INTEGER  *  4 K
      1 |       INTEGER  *  4 %I1
        | *     *** DECLARATION OF AUXILIARY VARIABLES ***
      1 |       INTEGER * 4 %T00010000
      1 |       INTEGER * 4 %T00010044
      1 |       INTEGER * 4 %T00010198
        | ***** STATEMENT 2 (DO) *****************
        | ***** STATEMENT 3 (MOVED STMT) *********
      3 |       L=7                                             (1)
        | ***** STATEMENT 5 (MOVED STMT) *********
        | ***** STATEMENT 5 (MOVED STMT) *********
      5 |       %T00010198=31                                   (4)
      5 |       %I1=5
        | ***** STATEMENT 3 (ASSIGNMENT) *********
        | ***** STATEMENT 4 (ASSIGNMENT) *********
      4 | L3    %T00010000=N*7                                  (2)
      4 |       M=M+%T00010000                                  (3)
        | ***** STATEMENT 5 (ASSIGNMENT) *********
      5 |       K=%T00010198                                    (6)
        | ***** STATEMENT 6 (ASSIGNMENT) *********
      6 | 1     N=%T00010000+K                                  (3)
        | ***** STATEMENT 5 (INCR STMT) **********
      5 |       %T00010198=%T00010198+3                         (5)
        | ***** STATEMENT 6 (DOEND) **************
      6 |       %I1 = %I1- 1
      6 |       IF (%I1 .NE. 0) GOTO L3
        | ***** STATEMENT 7 (END) ****************
      7 | L5    END
```

In this example (cf. 9.4.1) with OPT=3, continuing optimizations in the loop range are peformed, which are then displayed in the decompiler listing:

(1) Assignment L=7 is placed in front of the loop range.
(2) L is replaced by the constant 7.
(3) As a result, (N*7) is recognized as a common subexpression of statements 1 and 4 and is replaced by %T00010110.
(4) The begin value of K when I=1 is %T00010198=1*3+28=31.
(5) Multiplication with DO variable I in statement 5 is replaced by an addition in which 3 is added each time as the increment. As a result, continuation of the DO variables is superfluous.
(6) Calculation of the value of K is simplified to assignment of temporary auxiliary value %T00010198.

4.7.10 Summary listing

This listing can be output by entering SUMMARY in the LIST or LISTFILE option.

By default this listing is output to SYSLST.

Appendix A.6.8 shows a summary listing.

If several program units are compiled in one run, the system will then (at the end of compilation and in addition to the summary listing for each program unit) generate an overall summary listing containing the statistics for the total compilation, such as:

- Listing of all modules generated
- Accumulated errors in a given error class
- Total CPU time and ELAPSED time, etc.
- The message "(COMPILER NOT PRELOADED)", if such was the case.

The layout of the overall summary listing is the same as that of the summary listing.

Appendix A.6.9 shows an overall summary listing.

4.7.11 Change listing

The change listing includes all command entries as well as source lines which have been replaced and inserted.

This listing can be output by supplying CHANGE in the LIST or LISTFILE option.

CHANGE is only effective together with COMOPT DIALOG.

Appendix A.6.3 shows a listing of changes. It was generated in the example given in section 3.6.8.

4.8 Termination of compilation

4.8.1 SDF operand COMPILER-TERMINATION

```
START-FOR1-COMPILER
```

```
,COMPILER-TERMINATION = STD / PARAMETER(...)

   PARAMETER(...)
     |   CPU-LIMIT = NONE / <integer 1..32767>
     |
     |  ,MAX-ERROR-WEIGHT = NONE / ERROR / SEVERE-ERROR
     |
     |  ,MAX-ERROR-NUMBER = 100 / <integer 1..2147483639>
```

The SDF operands and the corresponding compiler options are shown in table 2-12.

4.8.2 Termination of compilation: ERRKILL and MAXERR compiler options

ERRKILL option

[*]COMOPT	ERRKILL = { E [RROR] | S [EVERE] | F [AILURE] }

The ERRKILL option specifies that compilation of a program is to be abnormally terminated if an error occurs whose message level is the same as that of the ERKILL option, or more severe.

ERRKILL	Termination at error level
E	E,S,F
S	S,F
F	F

MAXERR option

[*]COMOPT	MAXERR = { 100 | n }

n Integer value $\leq 2^{31}-1$

Compilation terminates abnormally once the number of errors (ERRORs) specified in the option has been reached.

4.9 Monitoring of compilation by job variables: SDF operand MONJV

```
START-FOR1-COMPILER
```

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

The SDF operands and the corresponding compiler options are shown in table 2-13.

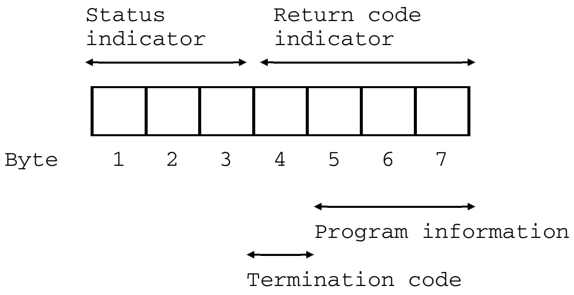
Meaning of the job variable indicators

With the aid of software product JV (Job Variables), jobs and programs run under BS2000 can be monitored and controlled (see "BS2000 Job Variables" manual [24]).
The user defines a monitoring job variable, which is specified as an operand in a LOGON, ENTER-JOB or START-PROGRAM command. The operating system takes this job variable and enters in it information on the current status of a program ("status indicator") and further information defined at program level ("return code indicator"). After the program has terminated, the user can interrogate this information; this same information can also be used to control further jobs and programs.

A job variable can monitor both a FOR1 compiler run and the execution of a FORTRAN program (see section 6.5.3). Status and return code indicators of the monitoring job variables are supplied with TERM macro parameters by the FOR1 or FORTRAN object program.

The status indicator of the job variables is set as a function of the parameter in the MODE operand, the return code indicator as a function of the parameter in the URETCD operand (see "Executive Macros" manual [26]).

Job variables for program monitoring are structured as follows:



The status indicator is set left-justified in the first three bytes of the job variable.

\$T␣	Program terminated normally.
\$A␣	Program aborted. This indicator is likewise set by the system upon program abortion.
\$R␣	Once a program has started, the status indicator is set to "\$R".

The return code indicator is entered in bytes 4 to 7 of the job variable. The first byte of the return code indicator contains the termination code. When the URETCD parameter in the TERM macro is omitted (see "Executive Macros" manual [26]) or when the program is aborted by the system, 4 blanks are entered in the return code indicator.

After a FOR1 compiler run, the termination code may contain the following information:

0	The compiler completed the compilation run normally. No warnings were issued and no errors detected.
1	The compiler completed the compilation run normally, however warnings or errors (ERRORS, SEVERE ERRORS) were reported while the source program was being compiled. The results of the compilation are usable, but subject to restriction.
2	The compiler run was free of error, however compilation was prematurely terminated due to the specifications in the MAXERR or ERRKILL options. The results of compilation are usable, but subject to restriction.
3	The compiler terminated the compilation run due to a defined compiler error. Results of the compilation run are unavailable or unusable.

Bytes 5 to 7 of the return code contain the program information. After a compiler run the program information may include the following:

000	The compiler has completed the compilation run normally. While the source program was being compiled, no error messages (NOTE, WARNING, ERROR, SEVERE or FAILURE) were output.

001 The compiler completed the compilation run normally, however notes have been reported.

002 The compiler completed the compilation run normally, however warnings were reported.

003 The compiler completed the compiler run normally, but errors have been reported, for which a correction has been made.

004 The compiler completed the compiler run normally. Severe errors which cannot be remedied have occurred.

006 Due to severe errors, the compiler terminated the compiler run, but in a controlled manner. Object programs are unusable or not available.

The following table shows the relationship between status indicator, termination code and program information:

Status indicator	Termination code	Program information	Remarks
`$T␣`	0	000	No errors, WARNINGS, NOTES during compilation.
`$T␣`	0	001	NOTES were reported during the compilation process.
`$T␣`	1	002	WARNINGS reported during compilation.
`$T␣`	1	003	ERRORS reported during compilation.
`$T␣`	1	004	SEVERE ERRORS reported during compilation.
`$A␣`	2	003	Compilation prematurely terminated due to ERRKILL=E or because the MAXERR limit was reached.
`$A␣`	2	004	Compilation prematurely terminated due to ERRKILL=S or because the MAXERR limit was reached.
`$A␣`	3	006	Compilation prematurely terminated due to compiler error (FATAL ERRORS).
`$A␣`	- (undefined)	—— (undefined)	Compilation terminated in an undefined manner due to severe compiler error (DUMP).

Table 4-1: Status indicator, termination code and program information for job variables

Example:

A FOR1 run is monitored with the aid of job variable JOBVAR1. After compilation, the linkage editor should only be called if the compiler has not output any messages or notes whatsoever.

```
/BEGIN-PROC LOG=C, PAR=YES(PROC-PAR=(&PROGRAM), ESC-CHAR=C'&')
/REMARK COMPILING AND LINKING A FORTRAN PROGRAM
/DEL-SYS-FILE OMF
/ASSIGN-SYSDTA TO-FILE=*SYSCMD
/CRE-JV JOBVAR1                                                   (01)
/START-PROG $FOR1, MONJV=JOBVAR1                                  (02)
*COMOPT SOURCE=QUELLE.&PROGRAM
*COMOPT END
/SET-JOB-STEP
/SHOW-JV JV-NAME(JOBVAR1)                                         (03)
/SKIP-COMM TO-LABEL=ENDE, IF=JV(CONDITION=(JOBVAR1,5,3)>'001')    (04)
/START-PROG $TSOSLNK
PROG LADE.&PROGRAM
INCLUDE *
RESOLVE ,$RZ.FOR1MODLIBS
END
/.ENDE DEL-JV JV-NAME(JOBVAR1)                                    (05)
/END-PROC
```

Explanation of example:

(01) The job variable JOBVAR1 is entered in the catalog.

(02) With the START-PROGRAM command, the job variable JOBVAR1 is assigned as a program-monitoring job-variable to the program to be called.

(03) The value of the job variable is output to SYSOUT by means of the SHOW-JV command.

(04) The SKIP-COMMANDS command is used to test whether the program information (bytes 5 through 7) of the job variables contains a value greater than 001. If this is the case, the compiler has issued warnings or error messages. If this is the condition, a branch is made to the statement using the end mark ".ENDE".

When ERRORS is reported, the runtime log will appear as follows:

```
.
.
.
(IN)     /SHOW-JV JV-NAME(JOBVAR1)
(OUT)    %$T 1003
(  )
(IN)     /SKIP-COMM TO-LABEL=ENDE, IF=JV(CONDITION=(JOBVAR1,5,3)>'001')
(OUT)    %  CJC0010 SKIP COMMANDS: CONDITION = TRUE
(IN)     /.ENDE DEL-JV JV-NAME(JOBVAR1)
(IN)     /END-PROC
.
.
.
```

If no error messages or merely NOTES (program information ’000’ or ’001’) have been displayed, the linkage editor is called. The runtime log will then appear as follows:

```
.
.
.
(IN)     /SHOW-JV JV-NAME(JOBVAR1)
(IN)     /GETJV (JOBVAR1,1),CHAR
(OUT)    %$T 0000
(  )
(IN)     /SKIP-COMM TO-LABEL=ENDE, IF=JV(CONDITION=(JOBVAR1,5,3)>'001')
(OUT)    %  CJC0011 SKIP COMMANDS: CONDITION = FALSE
(IN)     /START-PROG $TSOSLNK
.
.
.
```

(05) The DELETE-JV command is used to delete the job variable entry from the catalog.

4.10 Specifying the message language

4.10.1 SDF operand LANGUAGE

`START-FOR1-COMPILER`
`,LANGUAGE = `<u>`ENGLISH`</u>` / DEUTSCH`

The SDF operands and the corresponding compiler options are shown in table 2-14.

4.10.2 LANGUAGE compiler option

`[*]COMOPT`	`LANGUAGE={`<u>`ENGLISH`</u>`|GERMAN}`

<u>ENGLISH</u>
When the compiler options have been read in, FOR1 messages will be output in English.

GERMAN
When the compiler options have been read in, FOR1 messages will be output in German.

4.11 SDF operand COMPILER

The last operand of the SDF command START-FOR1-COMPILER is

```
COMPILER = $FOR1 / <full-filename 1..54>
```

It permits the file name of the compiler to be specified if this name should deviate from the preset designation.

The COMPILER operand is not visible in guided dialog and can only be specified in NO or EXPERT mode.

5 Linking, loading and starting

A FORTRAN source program is compiled into an object module (= module to be linked, or bound) by the FOR1 compiler. Object modules are held available in a PLAM library, an object module library or the temporary EAM file (*OMF).

Object modules themselves consist of machine code but are unable to execute in this form at this stage since the machine code is not yet complete. Each object module contains references to external addresses (external references), i.e. to further modules which must supplement it in order to allow execution.

The additionally required modules are runtime system modules (see section 1.9) and possibly further object modules such as separately compiled source programs or subprograms in other languages, for example.

The main functions of the linkage editor comprise calling the object modules required for the load module from the various sources (files, libraries) and linking them together. The process of **linking** (or binding) itself consists in the linkage editor supplementing each object module with those addresses that relate to areas outside the object module.

The end result of linkage is a load module (module for loading) or a link and load module ((LLM). These still have to be loaded into main memory and started. Link and load modules are not described in the present manual; they are described in detail in the "Binder-Loader-Starter" manual [13].

The following possible means of linking, loading and starting are available:

Static linkage editor TSOSLNK

TSOSLNK links one or more object modules to produce a load module and stores this load module in a cataloged file or in a PLAM library (as a type C element).
Before modules generated by TSOSLNK can be executed, they must be loaded into main memory using the loader ELDE.

Binder 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 link and load module (LLM). BINDER stores link and load modules in PLAM libraries (element type L). BINDER is available from operating system version V10; a detailed description may be found in the "Binder-Loader-Starter" manual [13].

Dynamic binder loader DBL

DBL links modules (object modules, link and load modules) in a single process to produce a temporarily loadable unit, loads this unit into main memory and starts it. The linked program is no longer available once it has been executed.
Dynamic linking (binding) and loading is advantageous principally in the debugging phase.

SDF command START-FOR1-PROGRAM

The operands of this command control the main functions of the dynamic binder loader DBL and of the static loader ELDE. Object modules and load modules can be processed.

5.1 Linking, loading, starting: SDF command START-FOR1-PROGRAM, FROM-FILE operand

The FROM-FILE operand of the SDF command START-FOR1-PROGRAM offers the following facilities:

An object module generated by FOR1 can be linked, loaded and started using DBL (*MODULE(...)/<full-filename 1..54>).

A load module generated by TSOSLNK can be laoded and started using ELDE (PHASE(...)).

```
START-FOR1-PROGRAM
--------------------------------------------------------------------------

 FROM-FILE = <full-filename 1..54> / *MODULE(...) / *PHASE(...)

   *MODULE(...)
   |
   |  LIBRARY = *OMF / *STD / <full-filename 1..54>
   |
   | ,ELEMENT = *ALL / <full-filename 1..32>
   |
   | ,PROGRAM-MODE = 24 / ANY

   *PHASE(...)
   |
   |  LIBRARY = <full-filename 1..54>
   |
   | ,ELEMENT = <full-filename 1..41>(...)
   |     |  VERSION = *HIGHEST-EXISTING / <alphanum-name 1..24>
```

Table 5-1: SDF operand FROM-FILE: Linking and loading

The SDF operands and corresponding commands are summarized below.

5.2 Summary: SDF operand FROM-FILE and corresponding DBL and ELDE control

The following table compares the SDF operands of the START-FOR1-PROGRAM command with the corresponding operands in the call for ELDE and DBL.

SDF operands are given in accordance with the metasyntax of section 1.3.2

SDF form	First subform	Second subform	Corresponds in ELDE to	Corresponds in DBL to
FROM-FILE = <full-filename 1..54>			Cataloged file containing the load module	
= *MODULE(...)	LIBRARY = <u>*OMF</u>			Temporary EAM object file
	= *STD			Object module library Search hierarchy of the DBL if the object module library is not specified.
	= <full-filename 1..54>			Name of the object module library
	ELEMENT = <u>*ALL</u>			Object module All object modules of the specified library
	= <full-filename 1..32>			Name of the object module
	PROGRAM-MODE = <u>24</u>			PROG-MOD = <u>24</u>
	= ANY			= ANY
= *PHASE(...)	LIBRARY = <full-filename 1..54>		Name of the load module library	
	ELEMENT = <full-filename 1..41>(...)		Name of the load module	
		VERSION = <alphanum..name 1..24>	Version designation of library element	
		= <u>*HIGHEST-EXISTING</u>	Highest version	

Table 5-2: SDF form FROM-FILE (specifications concerning object/load module)

5.3 Static linkage (linkage editor TSOSLNK)

The static linkage editor TSOSLNK links:

- one or more object modules to form a load module and stores this load module in either a cataloged file or a PLAM library (as a type C element). The PROGRAM statement of TSOSLNK is used for this purpose.
  The load module is loaded and started by the static loader ELDE.
- several object modules to form a single prelinked module (main module) and stores this in the temporary EAM area (*OMF) or in a PLAM library (as a type R element). The MODULE statement is used for this purpose (see "TSOSLNK" manual [41]).
  The prelinked module generated is used as input either for TSOSLNK or the dynamic binder loader DBL.

Control statements for linkage editor TSOSLNK

Only a limited number of control statements for TSOSLNK are given here. A complete description is contained in the "TSOSLNK" manual [41].

```
/START-PROG $TSOSLNK                                                     (1)
                     ⎧FILENAM=file                               ⎫
 PROGRAM program [,  ⎨                                           ⎬]      (2)
                     ⎩LIB[RARY]=lib[,ELEM[ENT]=element[(version)]]⎭

                 [,SYMTEST={ALL|N|MAP}]                                  (3)

                 [,MAP={Y|N}]                                            (4)

                 [,LOADPT={address|*XS}]                                 (5)

          ⎧module[(version)][,lib]        ⎫
          ⎪(module[(version)],...)[,lib]  ⎪
          ⎪,lib                           ⎪
 INCLUDE  ⎨                               ⎬                              (6)
          ⎪module,*                       ⎪
          ⎪(module,...),*                 ⎪
          ⎩*                              ⎭

           ⎧module       ⎫
 RESOLVE  [⎨             ⎬],lib                                          (7)
           ⎩(module,...) ⎭

 END                                                                     (8)
```

Explanation:

(1) Linkage editor TSOSLNK is called.

(2) The PROGRAM statement defines the name of the load module and where the load module is to be stored.

program	The name that the load module is to receive must be specified here. If no other operand (*FILENAM* or *LIB*) is specified, the cataloged file is given this name.
FILENAM=file	*file* selects a name which the cataloged file in which the load module is stored, is to receive. The maximum length including catalog ID and user ID is 54 characters.
LIB=lib [,ELEM=element]	The load module is stored in the PLAM library with the name *lib* as a type C element under the name *element*. If only the *LIB* operand is specified, *program* is assumed as the element name. If the library *lib* does not yet exist, it will be created.
version	Version designation of the PLAM library element *element*

(3) SYMTEST=ALL
SYMTEST=ALL permits symbolic addresses to be referenced for debugging using the Advanced Interactive Debugger (AID). FOR1 programs can be debugged symbolically using an AID version ≥ 1.0C. To allow symbolic debugging, the compiler option SYMTEST=ALL must be specified at compile time.

SYMTEST=N
The program cannot be symbolically debugged.

SYMTEST=MAP
The linkage editor will generate an object structure listing which is also written to the load module. This information permits minimum debugging. Full-scale symbolic debugging is only possible when AID loads the symbolic information dynamically.

(4) MAP=Y causes an overview of programs to be output to SYSLST which contains information on the size, length and addresses of the object modules that have been input.

(5) LOADPT=address specifies a virtual address (hexadecimal: X'...') to which the loader is to load a program. If this operand is omitted, the virtual address X'000000' will be assumed.
LOADPT=*XS specifies the program's load address in the address space above 16 megabytes. All control sections (CSECTs) must have the attribute RMODE=ANY. The static loader ELDE will then start the program in the 31-bit addressing mode.

(6) The INCLUDE statement defines the object modules that TSOSLNK is to link.

Object modules can reside:

- in the temporary EAM area (*OMF) if they were generated by FOR1 during the current task.
- in an object module library created by the LMS library management program.
- in a PLAM library generated by the LMS library management program or by the MODULE-LIBRARY compiler option.

module	Name of the object module that is to be read in from the temporary EAM area (*OMF) or from the object module library or PLAM library *lib*. If more than one object module is specified (maximum 20), the list must be enclosed in parentheses.
version	Version designation of the object module *module*. The version designation applies only to PLAM libraries. If the *version* specification is omitted, the object module having the highest version designation will be linked in.
lib	Name of the object module library or PLAM library from which the object modules are to be read in. If the *module* specification is omitted, the linkage editor will read in all the object modules from the library. If the *lib* specification is omitted, the linkage editor will search for the object module *module* in the library TASKLIB.
*	Temporary EAM area (*OMF) for the current task. If the *module* specification is omitted, the linkage editor reads in all object modules contained in the temporary EAM area.

(7) The RESOLVE statement serves to inform the linkage editor of the libraries which are to be searched using the Autolink procedure for previously unresolved external references.

TSOSLNK Autolink procedure:

If TSOSLNK finds external references in an object module which cannot be resolved by the modules that were specified in INCLUDE statements, then it uses the following Autolink procedure:

- TSOSLNK first searches to determine whether a library was specified in conjunction with the external reference in a RESOLVE statement.
- If TSOSLNK cannot resolve the external reference in the first step, then it will search all libraries specified in RESOLVE statements. Here the last RESOLVE statement is taken into consideration first; the penultimate, second etc. Libraries that are not to be searched can be excluded from the search by means of EXCLUDE statements.
- If TSOSLNK is also unable to resolve the external reference in the second stage, it will search the TASKLIB library unless this has been prevented by the NCAL statement or a corresponding EXCLUDE statement. If there is no library called TASKLIB under the user ID of the current task, TSOSLNK uses the system library $TSOS.TASKLIB.

If unresolved external references still exist after the Autolink procedure, TSOSLNK outputs their names in a list to SYSOUT and SYSLST.

If the FOR1 runtime system is not incorporated into the TASKLIB library of the user or the system TASKLIB, then it must be specified in a RESOLVE statement:

```
RESOLVE ,$userid.FOR1MODLIBS
```

If the function pool FPOOL is used, the object module library $userid.FOR1.FPOOLLIB must be specified in a RESOLVE statement:

```
RESOLVE ,$userid.FOR1.FPOOLLIB
```

(8) The entries for TSOSLNK must be concluded with the END statement.

Example:

Compilation:

```
/DEL-SYS-FILE OMF
/START-PROG $FOR1
*COMOPT SOURCE=QUELL.MAT
*COMOPT MODULE-LIBRARY=PLAM.LIB                          (1)
*COMOPT FPOOL=fpool
*END
```

Linkage with TSOSLNK:

```
/START-PROG $TSOSLNK
*PROGRAM PROGAB,FILENAM=OBJ.MAT                          (2)
*INCLUDE (MODA,MODB),PLAM.LIB                            (3)
*RESOLVE (FUNKT1,FUNKT2),FUNKLIB                         (4)
*RESOLVE ,FOR1MODLIBS                                    (5)
*RESOLVE ,$TSOS.FOR1.FPOOLLIB                            (6)
*END
/
.
.
.
```

Explanation of example:

(1) The object modules MODA and MODB are generated during compilation and written to the PLAM library PLAM.LIB.

(2) The load module is to receive the name PROGAB and is to be stored in the cataloged file OBJ.MAT.

(3) Statement for linking the object modules MODA and MODB from the library PLAM.LIB.

(4) The external references FUNKT1 and FUNKT2 are resolved by object modules from the library FUNKLIB. If a BLOCKDATA module (for initializing a named COMMON block) is to be linked in from FUNKLIB, this must be effected with an INCLUDE statement.

(5) The external references to modules of the FOR1 runtime system are resolved. Only adapter modules are linked in. The actual runtime system is dynamically loaded at runtime. If the FOR1 runtime system is contained in the system TASKLIB or in the TASKLIB of the user ID, statement (5) can be omitted.

(6) The external references to modules from the FPOOL are resolved.

Linking BLOCK DATA subprograms

When linking is performed using TSOSLNK, BLOCK DATA subprograms must also be linked in. If the BLOCK DATA subprograms reside in the temporary EAM file, they will be linked in by the "INCLUDE *" statement.
If the BLOCK DATA subprograms do not reside in the temporary EAM file, they must be explicitly specified in the INCLUDE statement.

The name of the object module to be specified in the INCLUDE statement in the case of a named BLOCK DATA program unit is the name of this very program unit. When the BLOCK DATA program unit is unnamed, the names of all COMMON blocks of this program unit must be specified in INCLUDE statements.

Example: Linking an unnamed BLOCK DATA subprogram

Source program in the cataloged file QUELL.TEST:

```
PROGRAM TEST
COMMON /A/I,/B/R
WRITE *,I,R
END
BLOCK DATA
COMMON /A/I,/B/R
DATA I,R/1,2.2/
END
```

Compilation:

```
/START-PROG $FOR1
*COMOPT SOURCE=QUELL.TEST
*COMOPT MODULE-LIBRARY=PLAM.LIB
*END
```

Linkage:

```
/START-PROG $TSOSLNK
*PROGRAM TEST,FILENAM=L.TEST
*INCLUDE TEST,PLAM.LIB
*INCLUDE A,PLAM.LIB
*INCLUDE B,PLAM.LIB
*RESOLVE ,$TSOS.FOR1MODLIBS
*END
```

Initializing named COMMON blocks

A named COMMON block can be initialized not only in a BLOCK DATA subprogram, but in any program unit. If the same COMMON block is initialized in several program units at the same time, the linkage editor (TSOSLNK; DBL) will issue the message "DUPLICATE CSECT". If this linkage editor message is ignored, unwanted reinitializations may be effected in overlay systems.

If a COMMON block is initialized in all program units, the linkage editor will be unable to recognize it as a COMMON block. The corresponding information will be lost.

5.4 Static loading (loader ELDE)

To enable a load module generated by TSOSLNK to be executed it must be loaded into main memory. The loader ELDE is available in BS2000 for this purpose.
ELDE is called when a START-PROGRAM or LOAD-PROGRAM command is entered that refers to a cataloged file or an element of a PLAM library (element type C):

– The START-PROGRAM command instructs ELDE to load the load module into memory and start it.

– The LOAD-PROGRAM command instructs ELDE to load the load module into memory without starting it. In this way it is possible to enter further commands prior to program execution, for debugging purposes for example. The program can then be started with the RESUME-PROGRAM command.

Calling ELDE

Only a limited number of specifications for the START-PROGRAM and LOAD-PROGRAM commands are given here. A complete description of both commands is contained in the "TSOSLNK" manual [41].

```
START-PROGRAM or LOAD-PROGRAM
--------------------------------------------------------------

 FROM-FILE = <full-filename 1..54> / *PHASE(...)

   *PHASE(...)
     |
     |   LIBRARY = <full-filename 1..54>
     |
     |  ,ELEMENT = <full-filename 1..41>
     |
     |  ,VERSION = *STD / <text 1..24>

,TEST-OPTIONS = NONE / AID
,MONJV = *NONE / <full-filename 1..54>
```

FROM-FILE =
Specifies the input source.

FROM-FILE = <full-filename 1..54>
The input source is the cataloged file containing the load module generated by TSOSLNK.

FROM-FILE = *PHASE(...)
The input source is the PLAM library containing the load module generated by TSOSLNK as a type C element.

LIBRARY = <full-filename 1..54>
Name of the PLAM library in which the load module is stored.

ELEMENT = <full-filename 1..41>
Name of the library element in which the load module is stored. The library element must be a type C element.

VERSION =
Specifies the version designation of the element.

VERSION = *STD
The element with the highest version designation is used.

VERSION = <text 1..24>
Explicit specification of the element version.

TEST-OPTIONS =
Specifies whether symbolic addresses may be used in the source program during debugging with AID. Programs using symbolic addresses can only be debugged if LSD information has been generated for the programs during compilation (SYMTEST=ALL compiler option).

TEST-OPTIONS = NONE
The LSD information is not copied into the load module. Symbolic debugging with AID is still possible by using a %SYMLIB statement to specify to AID a library containing the LSD information.

TEST-OPTIONS = AID
Permits the use of symbolic addresses in the source program during debugging with AID.

MONJV = *NONE / <full-filename 1..54>
Name of a job variable which is to monitor the program. If *NONE is specified, the program is not monitored with a job variable.
On execution, the program stores a code in the return code indicator of this job variable, which provides information on possible execution errors, The individual codes and their meanings are summarized in a table in section 4.9.
This operand is only available to users having the software product "Job Variables" (see section A.10.5).

Example:

The load module PROGAB generated in the example given in section 5.3 can be loaded and executed by the following command:

```
/START-PROGRAM FROM-FILE=OBJ.MAT
```

OBJ.MAT is the cataloged file to which the PROGAB load module is written.

5.5 Dynamic link loading (binder loader DBL)

The dynamic binder loader DBL temporarily links modules (object modules, link and load modules) to produce a loadable unit, loads them into memory and executes them in a single process. The generated load unit is automatically deleted after the program has executed.
The mode of operation of DBL is described in detail in the "Binder-Loader-Starter" manual [13].

DBL has two run modes. The mode is selected using the RUN-MODE operand of the START-PROGRAM and LOAD-PROGRAM commands.

RUN-MODE=<u>STD</u>
In this mode, DBL is fully compatible with DLL. DLL is only supplied up to and including BS2000 V9.5.
RUN-MODE=STD is the default.

RUN-MODE=ADVANCED
In this mode, DBL can also process link and load modules (LLMs), and supports the new functions of BS2000 V10 and up. This mode is not described here. A detailed description may be found in the "Binder-Loader-Starter" manual [13].

Calling DBL

DBL is called by entering a START-PROGRAM or LOAD-PROGRAM command referring to the temporary EAM area, an element of an object module library or an element of a PLAM library (element type R):

- The START-PROGRAM command instructs DBL to link, load and immediately start the object module.
- The LOAD-PROGRAM command instructs DBL to link and load the object module. After link loading, further commands (e.g. debugging commands) can be entered. The user then starts the program with the RESUME-PROGRAM command.

Only a limited number of specifications for the START-PROGRAM and LOAD-PROGRAM commands are given here. A complete description of both commands is contained in the "Binder-Loader-Starter" manual [13].

```
START-PROGRAM or LOAD-PROGRAM

FROM-FILE = *MODULE(...)

  *MODULE(...)
     |
     | LIBRARY = *STD / *OMF / <full-filename 1..54>
     |
     |,ELEMENT = *ALL / <full-filename 1..8>
     |
     |,PROGRAM-MODE = 24 / ANY
     |
     |,TEST-OPTIONS = NONE / AID
     |
     |, MONJV = *NONE / <full-filename 1..54>
```

FROM-FILE = *MODULE (...)
The dynamic binder loader DBL is called.

LIBRARY =
Specifies the input source from which object modules are fetched. The input source for object modules can be the temporary EAM area (*OMF), an object module library or a PLAM library (type R elements).

LIBRARY = *STD
The input source is the library assigned with the SET-TASKLIB command. If the object module is not found there, then the library called TASKLIB for the current task and subsequently the system TASKLIB ($TSOS.TASKLIB) are searched.

LIBRARY = *OMF
The input source is the temporary EAM area.

LIBRARY = <full-filename 1..8>
Name of an object module library or PLAM library that is used as the input source.

ELEMENT =
Specifies the modules to be fetched from the specified library.

ELEMENT = *ALL
Permissible only for object modules from the temporary EAM area. All object modules are fetched from the EAM area.

ELEMENT = <full-filename 1..32>
Name of the object module
full-filename may be:
– name of an object module
– name of a control section (CSECT name)
– name of an entry point (ENTRY name)
– name of a COMMON block

Elements from PLAM libraries must be of element type R.

PROGRAM-MODE =
Determines the part of the address space (above or below 16 Mbytes) in which the program is to be loaded.

PROGAM-MODE = 24
The module is loaded below 16 Mbytes. The program is executed in 24-bit addressing mode. External references are resolved only by CSECTs or ENTRYs which lie below 16 Mbytes.
Loading a program with 31-bit addressing mode (AMODE=31) is aborted with an error message.

PROG-MODE = ANY
The module can be loaded above or below 16 Mbytes. The load address is defined by evaluating the RMODE and AMODE attributes:
If RMODE=24, the load address will lie below 16 Mbytes.
If RMODE=ANY and AMODE=ANY, the load address will be above 16 Mbytes.

TEST-OPTIONS =
Specifies whether symbolic addresses may be used in the source program during debugging with AID.
Programs using symbolic addresses can only be debugged if LSD information has been generated for the programs during compilation (SYMTEST=ALL compiler option).

TEST-OPTIONS = NONE
The LSD information is not copied into the load module. Symbolic debugging with AID is still possible by using a %SYMLIB statement to specify to AID a library containing the LSD information.

TEST-OPTIONS = AID
Symbolic addresses can be referenced during debugging with AID.

MONJV = *NONE / <full-filename 1..54>
Name of a job variable which is to monitor the program. If *NONE is specified, the program is not monitored with a job variable.
On execution, the program stores a code in the return code indicator of this job variable, which provides information on possible execution errors, The individual codes and their meanings are summarized in a table in section 4.9.
This operand is only available to users having the software product "Job Variables" (see section A.10.5).

Example:

Compilation:

```
/DEL-SYS-FILE OMF
/START-PROG $FOR1
*COMOPT SOURCE=QUELL.MAT
*COMOPT OBJECT=(*)                                      (1)
*END
```

Linkage, loading, starting:

```
/SET-TASKLIB FOR1MODLIBS                                (2)
/START-PROG FROM-FILE=*MODULE(LIB=*OMF)                 (3)
```

Explanation of example:

(1) The object modules generated are stored in the temporary EAM area (*OMF).

(2) Assigns the FOR1 runtime system. This is only necessary if the FOR1 runtime system is not contained in the TASKLIB of the current task or in the system TASKLIB.

(3) All object modules from the temporary EAM area are to be linked, loaded and started using DBL (ELEMENT=*ALL is the default).

5.6 Memory allocation of started programs

The memory area of a started program comprises the generated load module and the memory area set up at the commencement of execution. The program is executed under the control of the FOR1 runtime system.
Fig. 5-1 shows the memory allocation for object programs loaded without explicit specification of a load address.

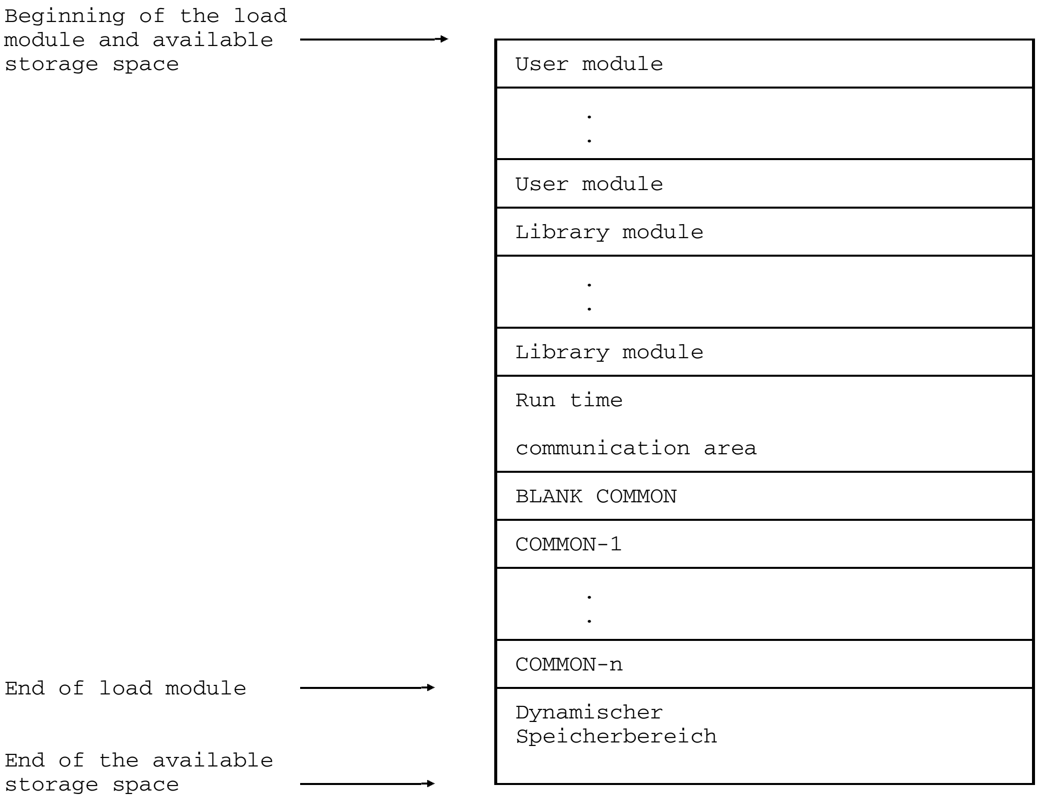

Beginning of the load module and available storage space
End of load module
End of the available storage space
User module
.
.
User module
Library module
.
.
Library module
Run time communication area
BLANK COMMON
COMMON-1
.
.
COMMON-n
Dynamischer Speicherbereich

Fig. 5-1: Memory allocation for a started program

User modules

These are modules created from object modules by resolving external references.

Library modules

These are modules that the linkage editor adds (from the FOR1 runtime library or other module libraries) at the time the external references are resolved.

Run time communication area (RTCA)

The RTCA contains information used to control the execution of the started program. The RTCA is set up by the program initialization routine after the user and library modules and is about 4 Kbytes long. The RTCA information includes the following:

- Table of input/output units
  This table provides a reference for each file number to the associated input/output routines as well as indicating the type of the relevant file.
- Hash table for access to the file descriptors.
  File descriptors are established for each file in the dynamic memory area by the FOR1 runtime system.
- Address of the "current" file descriptor, i.e. the file currently being accessed.
- Address of the "current" input/output routine
- Address for the error exit when input/output operations are executed
- Information on the current task, such as date, time of day, CPU time used

COMMON-n, BLANK COMMON

Common storage area allocated by the linkage editor for the COMMON blocks.

Dynamic memory area

The dynamic memory area is used for the following purposes:

Input/output buffer areas

Transfer to or from a file is not made separately for each record but the complete contents of a buffer area are transferred at a time.

Conversion buffers

Most input/output operations involve conversions between the internal and external representation of data. The conversion routines use dynamically allocated conversion buffers for larger volumes of data.

File descriptors

The properties of a file in the FORTRAN program are described in a file descriptor. This enables the validity of an input/output operation to be verified and the information required for the INQUIRE statement to be made available. Also entered is the most recent input/output operation carried out on this file, providing a capability for checking the validity of the succession of input/output operations. Each access to the file causes the associated file descriptor to be modified.

File control block (FCB)

The file control block is the dominant communication area for all input/output operations. It describes the characteristics of a file from the viewpoint of the Data Management System. One file control block exists for each file, except for EAM files and system files.

FORMAT descriptors for variable FORMAT

The results of interpreting variable formats are stored in the form of FORMAT descriptors so that the interpretation, once accomplished, may be used again, if required.

5.7 Binder BINDER

BINDER links modules (object modules, link and load modules) to form a logically and physically structured loadable unit. This unit is known as a link and load module (LLM). BINDER stores link and load modules in PLAM libraries (element type L).
BINDER is available as of BS2000 V10. BINDER is described in detail in the "Binder-Loader-Starter" manual [13].

5.8 Shareable programs

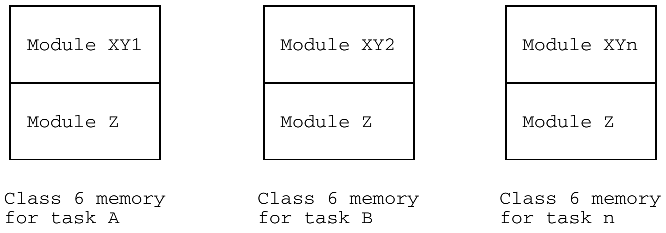
In large programs it may be advantageous for particular program sections to be declared shareable if they are accessed by several users (tasks) concurrently.

Shareable program sections have the following advantages:
- storage space savings (the shareable module is held in working storage only once)
- time savings through reduced paging.

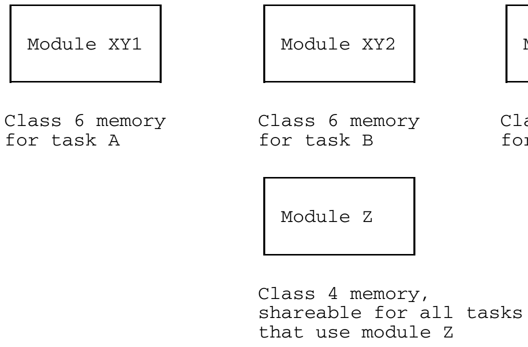
Program execution with shareable and nonshareable programs

The following diagrams illustrate program execution with shareable and nonshareable programs:

Memory allocation for nonshareable programs (module Z is loaded into class 6 memory three times):


Module XY1
Module Z
Class 6 memory for task A
Module XY2
Module Z
Class 6 memory for task B
Module XYn
Module Z
Class 6 memory for task n

Memory allocation for shareable programs (module Z is loaded once into class 4 memory):


Module XY1
Class 6 memory for task A
Module XY2
Class 6 memory for task B
Module XYn
Class 6 memory for task n
Module Z
Class 4 memory, shareable for all tasks that use module Z

Generating shareable programs

Shareable programs are generated as follows:

– The source program is compiled using the OBJECT=(SHARE) compiler option.
  This compiler option separates the shareable portion of the source program (code section) from the nonshareable portion (data section).
  FOR1 generates a shareable object module and a nonshareable object module.

  The shareable code section must not contain any address reference to the nonshareable data section. The link between code section and data section is therefore established in such a way that a small code section is placed in the data section. This small code section will contain the registers giving the addresses of the code and data sections. Subsequently, a branch is taken to the code section proper (see Fig. 5-2).

– Shareable object modules and nonshareable object modules are stored in PLAM libraries.

– Shareable and nonshareable object modules can be further processed either with or without the SHARE procedure SYSPRC.FOR1.022.SHARE. These two methods are described in sections 5.8.1 and 5.8.2.

  The system administrator declares the shareable portion as shareable by means of the ADD-SHARED-PROGRAM command. The shareable portion is loaded into class 4 memory as soon as the first user requests it. It then remains available there until all tasks have been terminated by SHUTDOWN.

  The nonshareable portion is loaded into class 6 for each task.

Restrictions:

– The OBJECT=(SHARE) and SYMTEST=ALL compiler options are mutually exclusive. If both are specified, the latter specification applies and an error message is issued.

– Intervention by means of the symbolic debugger AID is not possible since shareable program sections are in class 4 memory.

– In the case of language combinations, shareable and nonshareable programs must not be mixed.

```
***************************************
**        D A T A    A R E A        **
***************************************
**    NON-SHARE ENTRY CODE PART      **
***************************************
*
MUPS     CSECT
         USING *,15
         STM   0,12,20(13)
         LR    9,13                   (1) Load address of data section
         LA    13,24(0,15)            (2) Load address of code section
         L     11,20(0,15)
         MVI   0(9),236
         BCR   15,11                  (3) Branch to code section
         DROP  15
         DC    V(MUPS@@@@)
***************************************
**         NON-SHARE DATA PART       **
***************************************
*
         DS    0D
         USING *,13
         DS    312C
         ORG   MUPS+96
         DC    A(I@@@RTCA)             RUNTIME COMMUNICATION AREA
         ORG   MUPS+104
         DC    X'03'
         DC    'MUPS   '
              .
              .
              .
```

```
***************************************
**        C O D E    A R E A        **
***************************************
*
MUPS@@@@ EQU   *
*
***** STATEMENT 1 (ENTRY) *************
*       INTEGER FUNCTION MUPS(J)
*                                                **** SEGMENT 1 ****
* CODE SLICE BEGIN *                   (SLICE 1)
         USING *,15
         USING *,11
         L     12,16(0,11)
         BC    15,20(0,11)
         DC    AL1(03)
         DC    CL7'MUPS   '
         DC    A(MUPS####)
         USING MUPS####,12
         USING MUPS,13
         LR    15,11
         STM   14,15,12(9)
         LR    14,11
         ST    13,8(0,9)
         CLI   0(13),0
         BC    8,76(0,11)
         STM   12,0,104(9)
              .
              .
              .
```

Fig. 5-2: Generated code with COMOPT OBJECT=(SHARE). Extracts from the OBJECT listing

5.8.1 Shareable programs using procedure SYSPRC.FOR1.022.SHARE

5.8.1.1 Procedure

1. Compilation using OBJECT=(SHARE) compiler option

   If shareable programs are to be generated, this action must be initiated during the compilation by specifying the following compiler option:

   ```
   COMOPT OBJECT=(SHARE)
   ```

   This compiler option separates the shareable portion of the source program (code section) from the nonshareable portion (data section).

   FOR1 generates a shareable object module and a nonshareable object module.

   The name of the shareable object module corresponds to the name of the program unit and is padded out to eight characters on the right with @.

   The nonshareable object module is given the name of the program unit.

2. Storage of shareable and nonshareable object modules in PLAM libraries. The following methods are open to the user.

   – The user uses the SHARE-LIBRARY and MODULE-LIBRARY compiler options for the compilation.

     Shareable object modules are stored in accordance with the specifications in the SHARE-LIBRARY compiler option (see section 4.2.2.2). If the SHARE-LIBRARY compiler option is omitted, the shareable object modules will be stored in accordance with the specifications in the MODULE-LIBRARY compiler option.

     Nonshareable object modules are stored in accordance with the specifications in the MODULE-LIBRARY compiler option (see section 4.3.2).

   – After the compilation, the shareable and nonshareable object modules are placed in PLAM libraries by the library management program LMS.

3. Using the SHARE procedure

   Before the SHARE procedure SYSPRC.FOR1.022.SHARE is called the user must write to a file the names of those shareable object modules that are to be combined to form a single shareable module (see example). The names are specified without @.

   The user then calls the SHARE procedure.

   The SHARE procedure generates two adapter modules: one adapter module for the shareable module and one adapter module for the nonshareable module. These adapter modules establish the link between shareable module and nonshareable module.

   The SHARE procedure links together the shareable adapter module and the shareable object module to form a shareable module and stores the nonshareable adapter module in the specified library.

   The SHARE procedure can be called in both batch and interactive tasks. In a batch task, the parameters must be specified on calling the SHARE procedure. In an interactive task, the parameters can be specified either on calling the procedure or they are requested through prompting on the screen during procedure execution.

   The call and parameters for the SHARE procedure are described in section 5.8.1.2.

4. Declaring module as shareable and loading into class 4 memory

   The shareable module generated by the SHARE procedure remains to be declared shareable by the system administrator (ADD-SHARED-PROGRAM command) and loaded into class 4 memory. This is described in the "BS2000 System Administration" manual [40].

5. Linking the nonshareable modules using TSOSLNK

   The user still has to link the nonshareable object module, the nonshareable adapter module and where appropriate further object modules to form a load module, using the linkage editor TSOSLNK.

6. Starting the program

   The user starts the program using the loader ELDE by calling the nonshareable load module:

   ```
   /START-PROGRAM FROM-FILE=    ...       or
   /LOAD-PROGRAM FROM-FILE=    ...
   ```

5.8.1.2 Parameters for procedure SYSPRC.FOR1.022.SHARE

```
/CALL-PROCEDURE SYSPRC.FOR1.022.SHARE, [SHRNAMES=name[,X[,breg]]

                             ,LIBN=libn,LIBS=libs, LIBT=libt

                             ,ADAPTS=adapts,ADAPTN=adaptn,SHRMOD=shrmod

                             [,HELP={YES|NO}][,ER={YES|NO}][,XS = {YES|NO}]

                             [,MACLIB = {$TSOS.MACROLIB|libname}]]
```

name	Name of the file containing the names of the shareable modules. The name is padded right-justified with the character "@" to a length of 8 characters. X and breg not specified: The base register is 11.
X	Base register is 15.
breg	Base register is used as a branch register to the shareable modules.
libn	Name of the nonshareable module library. This module library contains the nonshareable modules and the nonshareable adapter modules generated by the SHARE procedure.
libs	Name of the shareable module library. This module library contains the shareable modules and the shareable adapter modules generated by the SHARE procedure.
libt	Type of the module libraries (LMS or LMR) in which the modules are stored.
adapts	Name that the shareable adapter module is to receive. This module is required for linking the nonshareable program sections with the shareable sections. It is linked ahead of the shareable module by the SHARE procedure.
adaptn	Name that the nonshareable adapter module is to receive. This module is required for linking the nonshareable program sections with the shareable sections.
shrmod	Name that the shareable module is to receive. *shrmod* contains the shareable object module and the shareable adapter module.

$\texttt{HELP=}\left\{\begin{array}{l}\underline{\texttt{YES}}\\ \texttt{NO}\end{array}\right\}$

Default is YES. After the SHARE procedure has started, a brief explanation is given on the meaning of the individual parameters.

$\texttt{ER=}\left\{\begin{array}{l}\underline{\texttt{YES}}\\ \texttt{NO}\end{array}\right\}$

Default is YES. The generated auxiliary files T.T.T.T.SHR, T.T.T.T.LNK and T.T.T.T.LST.SHR.&SHRMOD are deleted.

$\texttt{XS=}\left\{\begin{array}{l}\underline{\texttt{NO}}\\ \texttt{YES}\end{array}\right\}$

=<u>NO</u>: The program is loaded below 16 Mbytes.
=YES: The program is loaded above 16 Mbytes.

$\texttt{MACLIB=}\left\{\begin{array}{l}\underline{\texttt{\$TSOS.MACROLIB}}\\ \texttt{libname}\end{array}\right\}$

The MACLIB operand assigns the system macro library for the assembly of adapter modules. Permissible operand values are $TSOS.MACROLIB (default) or another library name *libname*. If XS=YES has been set, a macro library with modules from a BS2000 version ≥ 9.0 must be specified.

Portability

The names of the shareable module and the associated module library can be changed for portability reasons, although they were defined with the aid of the SHARE procedure when they were generated.

This can be accomplished by changing the appropriate names in the nonshareable adapter, where the names are stored at the relative address 0 in the form:

```
DC CL8'MODS'
DC CL54'MODBIBS'
```

The change can then be made with the aid of LMS during transfer of object modules, or with the aid of DPAGE during transfer of load modules.

The addresses of the individual object modules can be found in the linkage editor listing.

5.8.1.3 Example

The main program PROG is contained in the file S.PROG and is to be compiled as nonshareable.

The subprograms SUB1 and SUB2 are contained in the file S.SUB and are to be compiled as shareable.

```
 PROGRAM  PROG
 INTEGER A,B
 CALL SUB1(A,B)
 IF (A.EQ.100 .AND. B.EQ.20)  WRITE (2,*) A,B,'CALL AND RETURN OK'
 END


 SUBROUTINE SUB1(X,Y)
 INTEGER X,Y
 X=10
 Y=10.50
 CALL SUB2(X,Y)
 IF (X.EQ.100 .AND. Y.EQ.20) WRITE (2,*) X,Y,'CALL AND RETURN FROM
-SUB2 OK'
 RETURN
 END


 SUBROUTINE SUB2(X,Y)
 INTEGER X,Y
 X=X*X
 Y=Y+Y
 WRITE (2,*) 'HERE SUBROUTINE SUB2'
 RETURN
 END
```

1. Compilation:

Compilation of main program PROG:

```
 /START-PROG $FOR1
 *COMOPT SOURCE=S.PROG,LIST=(SRC,D,OP,XR)
 *END
```

Compilation of subprograms SUB1 and SUB2:

COMOPT OBJECT=(SHARE) separates the shareable code section and the nonshareable data section.

```
/START-PROG $FOR1
*COMOPT SOURCE=S.SUB,LIST=(SRC,D,OP,XR)
*COMOPT OBJECT=(SHARE)
*END
```

2. Storing the shareable and nonshareable object modules:

The nonshareable object modules are stored in the library LIB.NOSHARE. The shareable object modules are stored in the library LIB.SHARE.

Main program:

```
/START-PROG $FOR1
*COMOPT SOURCE=S.PROG,MODULE-LIBRARY=LIB.NOSHARE,END
```

Subprograms:

```
/START-PROG $FOR1
*COMOPT SOURCE=S.SUB,OBJECT=(SHARE),SHARE-LIBRARY=LIB.SHARE
*COMOPT MODULE-LIBRARY=LIB.NOSHARE,END
```

3. SHARE procedure:

Using the EDT, the names of the shareable modules are written to a file having the name SHRNAM before the SHARE procedure is called.

```
/START-PROG $EDT
 SUB1
 SUB2
 @W'SHRNAM'
 @H
```

SHARE procedure call SYSPRC.FOR1.022.SHARE:

```
/CALL-PROC SYSPRC.FOR1.022.SHARE, SHRNAMES=SHRNAM,LIBN=LIB.NOSHARE,
 LIBS=LIB.SHARE,LIBT=LMS,ADAPTS=ADAPTS,ADAPTN=ADAPTN,
 SHRMOD=SHRMOD,HELP=NO,ER=NO
```

Two adapter modules are generated for linking the shareable and nonshareable program sections. The adapter module called ADAPTS is linked ahead of the shareable object module. The shareable module produced is given the name SHRMOD. The adapter module called ADAPTN for the nonshareable object module is stored in the library LIB.NOSHARE.

4. Declaration of shareability and loading of the shareable module called SHRMOD are tasks performed by the system administrator (see "BS2000 System Administration" manual [40]).

5. Linking the nonshareable modules using TSOSLNK:

```
/START-PROG $TSOSLNK
 PROG PROG,FILENAM=L.PROG              (1)
 INCLUDE PROG,LIB.NOSHARE              (2)
 INCLUDE ADAPTN,LIB.NOSHARE
 RESOLVE ,LIB.NOSHARE                  (3)
 RESOLVE ,FOR1MODLIBS
 END
```

(1) The load module is to receive the name PROG and be stored in the cataloged file L.PROG.

(2) The nonshareable object module PROG and the nonshareable adapter module called ADAPTN are to be linked in. Both are contained in the library LIB.NOSHARE.

(3) External references are to be resolved by the libraries LIB.NOSHARE and FOR1MODLIBS.

6. Starting the program by starting the nonshareable load module:

```
/SET-TASKLIB FOR1MODLIBS
/START-PROGRAM FROM-FILE=L.PROG
```

5.8.2 Shareable programs without procedure SYSPRC.FOR1.022.SHARE

5.8.2.1 Procedure

1. Compilation using OBJECT=(SHARE) compiler option (see section 5.8.1.1).
2. Storage of shareable and nonshareable object modules in PLAM libraries (see section 5.8.1.1).
3. Declaring as shareable and loading shareable object module into class 4 memory

   The system administrator declares the shareable object module as shareable using the ADD-SHARED-PROGRAM command and loads it into class 4 memory (see "BS2000 System Administration" manual [40]).
4. Starting the program

   The user starts and loads the program using the dynamic binder loader DBL by calling the nonshareable object module.

   ```
   /START-PROGRAM FROM-FILE=*MODULE    ...       or
   /LOAD-PROGRAM FROM-FILE=*MODULE ...
   ```

5.8.2.2 Example

The main program PRNOSHR is contained in the file SOURCE.PRNOSHR and is to be compiled as nonshareable.

The subprograms SUBSHR1, SUBSHR2 and SUBSHR3 are contained in the file SOURCE.SUB123 and are to be compiled as shareable.

```
PROGRAM PRNOSHR
WRITE (2,*) 'START PRNOSHR'
CALL SUBSHR1 ()
WRITE (2,*) 'END PRNOSHR'
END

SUBROUTINE SUBSHR1 ()
WRITE (2,*) '   START SUBSHR1'
CALL SUBSHR2 ()
WRITE (2,*) '   END   SUBSHR1'
END

SUBROUTINE SUBSHR2 ()
WRITE (2,*) '      START SUBSHR2'
CALL SUBSHR3 ()
WRITE (2,*) '      END   SUBSHR2'
END

SUBROUTINE SUBSHR3 ()
WRITE (2,*) '         START SUBSHR3'
WRITE (2,*) '         END   SUBSHR3'
END
```

1. Compilation and storing the object modules in PLAM libraries

   Subprograms SUBSHR1, SUBSHR2 and SUBSHR3:

   COMOPT OBJECT=(SHARE) separates the shareable code section and the nonshareable data section.
   The shareable object module is stored in the library SHR.LIB.
   The nonshareable object module is stored in the library MOD.LIB.

   ```
   /START-PROG $FOR1
   *COMOPT SOURCE=SOURCE.SUB123
   *COMOPT OBJECT=(SHARE),MODULE-LIBRARY=MOD.LIB,SHARE-LIBRARY=SHR.LIB
   *COMOPT END
   ```

   Main program PRNOSHR:

   The main program PRNOSHR is compiled and the generated object module is stored in the library MOD.LIB:

   ```
   /START-PROG $FOR1
   *COMOPT SOURCE=SOURCE.PRNOSHR,MODULE-LIBRARY=MOD.LIB
   *COMOPT END
   ```

2. Declaration of shareability and loading of the shareable module into class 4 memory are tasks performed by the system administrator (see "BS2000 System Administration" manual [40]).

3. Assigning the TASKLIB and starting the program, using DBL:

   ```
   /SET-TASKLIB LIBRARY=FOR1MODLIBS
   /START-PROG FROM-FILE=*MODULE(LIBRARY=MOD.LIB,ELEMENT=PRNOSHR)
   ```

   During program execution, the following messages are output to SYSOUT:

   ```
   START PRNOSHR
      START SUBSHR1
         START SUBSHR2
            START SUBSHR3
            END   SUBSHR3
         END   SUBSHR2
      END  SUBSHR1
   END PRNOSHR
   ```

6 Program execution

6.1 Controlling program execution: Operands of the SDF command START-FOR1-PROGRAM

```
START-FOR1-PROGRAM
----------------------------------------------------------------

 FROM-FILE = ...

,CPU-LIMIT = JOB-REST / <integer 1..32767>

,TESTOPT = NONE / AID

,MONJV = *NONE / <full-filename 1..54>

,OBJECT-CONTINUATION = NO / YES  1)

,RUNTIME-OPTIONS = NO / YES(...)

   YES(...)
   |
   |   LINE-OVERPRINT = LASER / YES / NO
   |
   |  ,SYSDTA-UNIT = (1,5,97) / list-poss: <integer 0..99>
   |
   |  ,SYSOPT-UNIT = (7,98) / list-poss: <integer 0..99>
   |
   |  ,SYSLST-UNIT = (6,99) / list-poss: <integer 0..99>
   |
   |  ,SYSIPT-UNIT = 8 / list-poss: <integer 0..99>
   |
   |  ,SYSOUT-UNIT = 2 / list-poss: <integer 0..99>
   |
   |  ,FOR1-COUNT-UNIT = 6 / list-poss: <integer 0..99>
   |
   |  ,START = NOT-XS / XS
   |
   |  ,EXPONENT-UNDERFLOW = UNCHANGED / YES / NO
```

1) The operand OBJECT-CONTINUATION=YES (continuation of program execution in the event of an error) can only be used in batch mode.

6.2 Summary: SDF operand RUNTIME-OPTIONS and corresponding runtime options

SDF form	First subform	Corresp. runtime option
RUNTIME-OPTIONS = NO		/PARAMETER CARD = NO
= YES(...)		= YES
	LINE-OVERPRINT = LASER	RUNOPT OVERPRINT = LASER
	= YES	= YES
	= NO	= NO
	SYSDTA-UNIT = (1,5,97)	Standard file numbers for SYSDTA: 1,5,97
	= list-poss: <integer 0..99>	SUBSTITUTE, DTA: n,n,...,END
	SYSOPT-UNIT = (7,98)	Standard file numbers for SYSOPT: 7,98
	= list-poss: <integer 0..99>	SUBSTITUTE, OPT: n,n,...,END
	SYSLST-UNIT = (6,99)	Standard file numbers for SYSLST: 6,99
	= list-poss: <integer 0..99>	SUBSTITUTE, LST: n,n,...,END
	SYSIPT-UNIT = 8	Standard file number for SYSIPT: 8
	= list-poss: <integer 0..99>	SUBSTITUTE, IPT: n,n,...,END
	SYSOUT-UNIT = 2	Standard file number for SYSOUT: 2
	= list-poss: <integer 0..99>	SUBSTITUTE, OUT: n,n,...,END

	FOR1-COUNT-UNIT = 6	Output of the %COUNT listing via SYSLST
	= list-poss: <integer 0..99>	SUBSTITUTE, CNT: n,n,...,END
	START = XS	RUNOPT START = XS
	= NOT-XS	No RUNOPT START
	EXPONENT-UNDERFLOW = UNCHANGED	RUNOPT EXPONENT-UNDERFLOW No RUNOPT EXPONENT-UNDERFLOW
	= YES	= YES
	= NO	= NO

Table 6-1: SDF form RUNTIME-OPTIONS and corresponding runtime options

6.3 Controlling program execution with runtime options

By specifying runtime options the user can influence program execution after the program has been called.

Runtime options enable the user to:

- change the predefined assignment of I/O units for the system files (standard FORTRAN files),
- control the interpretation of form feed characters for the printers,
- suppress FOR1 STXIT handling during FOR1 initialization,
- change the machine address mode to 31 during execution of a FOR1 program,
- update the settings of the exponent underflow handling facility.

6.3.1 Entering runtime options

Entry of runtime options is made possible by means of the CARD operand in the PARAMETER command:

/PARAM[ETER]	CARD = $\left\{\begin{matrix}\texttt{YES}\\ \underline{\texttt{NO}}\end{matrix}\right\}$

CARD = YES

After the program has been called, the user may enter runtime options by specifying CARD=YES.

The runtime options may be read either from

- a file with the linkname FOR1RUN (/SET-FILE-LINK LINK-NAME=FOR1RUN, FILE-NAME=filename) or
- from SYSDTA, if no linkname has been issued.

CARD = <u>NO</u>

Default. The user specifies none of the runtime options listed below. The default values are assumed for the program run.

The command remains in force until LOGOFF, SET-JOB-STEP or until the next PARAM CARD =... command.

Entry of the runtime options is concluded with END. The entry may comprise up to 320 characters in total.

Example:

```
     /PARAM CARD = YES
     /START-PROGRAM program
(out) GIVE RUNOPT OR END CARD OR ?       (1)
(in)  RUNOPT OVERPRINT = YES
(out) GIVE RUNOPT OR END CARD OR ?
(in)  A,DTA:11,END                       (2)
```

(1) In a batch job, output of the request "GIVE RUNOPT OR..." is suppressed, but input is still possible.

(2) If the standard assignment of BS2000 system files is changed, the corresponding runtime option must be the last RUNOPT entry since runtime options which change the assignment must be terminated by END.

If "?" is entered, the following help information is displayed:

```
STANDARD PRECONNECTION IS:
SYSDTA  :  1 , 5 , 97
SYSOPT  :  7 , 98
SYSLST  :  6 , 99
SYSIPT  :  8
SYSOUT  :  2
%COUNT  :  6
PLEASE CHANGE THIS PRECONNECTION IN THE FORM:
S<UBSTITUTE>/A<DD>/D<ELETE>/N<O>,(DTA:MM,NN,...)(,OPT:OO,PP,...)
(,LST:...)(,IPT:...)(,OUT:...)(,CNT:...),END
( ) MEANS OPTIONAL
```

6.3.2 Changing file numbers: SUBSTITUTE, ADD, DELETE and NO runtime options

```
[*]   │                     ⎡DTA⎤                        ⎡DTA⎤
      │    ⎧S[UBSTITUTE]⎫   ⎢OPT⎥                        ⎢OPT⎥
      │    ⎪A[DD]       ⎪   ⎢LST⎥                        ⎢LST⎥
      │    ⎨            ⎬, ⎨    ⎬ :n [,n] ... [,⎨    ⎬ :n [,n] ... ] ...,END
      │    ⎪D[ELETE]    ⎪   ⎢IPT⎥                        ⎢IPT⎥
      │    ⎩N[O]        ⎭   ⎢OUT⎥                        ⎢OUT⎥
      │                     ⎣CNT⎦                        ⎣CNT⎦
```

S[UBSTITUTE]
: The predefined file numbers (see section 8.3.3.1) for the specified standard files are replaced by the specified numbers.

Restriction:
No file number can be simultaneously assigned to more than one BS2000 system file.

A[DD]
: The predefined file numbers for the specified standard files are extended to include the specified numbers.

D[ELETE]
The specified file numbers are removed from the set of predefined file numbers. If the UNIT parameter of an I/O statement is an asterisk ("*") and the file number preset for the asterisk is removed, the number is replaced by another predefined file number. If all preset file numbers for a standard file are deleted, an error (IC02) occurs when "*" is specified as the UNIT parameter in an I/O statement.

N[O]	No changes to the files numbers are requested.
n	Integer value, 0 ≤ n ≤ 99

Note

Since these entries terminate with an END, any other RUNOPT entries must precede them.

Examples:

Input	*Effect*
`SUBSTITUTE,DTA:10,37,END`	The system file SYSDTA is assigned the file numbers 10 and 37. The predefined numbers 1, 5 and 97 are no longer valid for SYSDTA.
`ADD,OPT:11,12,OUT:13,END`	In addition to the predefined numbers 7 and 98, numbers 11 and 12 are also valid for the system file SYSOPT. Accordingly, the numbers 2 and 13 are valid for SYSOUT.
`S,CNT:15,END`	The dynamic listing (%COUNT) is not output via file number 6 (SYSLST), but via file number 15.
`DELETE,DTA:1,END`	Predefined file number 1 is deleted. The FORTRAN statement "READ *,A" may now result in input via file number 5, for example.

6.3.3 Controlling form feed character generation for output to SYSLST: RUNOPT OVERPRINT

For output to SYSLST, RUNOPT OVERPRINT controls the generation of printer control characters; FORTRAN control characters are converted into hexadecimal form feed characters (see table 6-2). These hexadecimal form feed characters are then interpreted during printing.

```
                               ⎡Y[ES]    ⎤
[*]RUNOPT        O[VERPRINT] [=[⎨N[O]     ⎬]]
                               ⎣L[ASER]  ⎦
```

O=YES A FORTRAN control character other than "+" generates an additional data line that contains the hexadecimal form feed characters (two-line solution). "+" signifies only that no additional data line is generated; the predecessor record is not suppressed, however.

O=NO Overprinting of records is excluded. For records with a form feed character other than "+", there is a definite saving in CPU and elapsed time because only one line per FORTRAN form feed character is output (single-line solution).
The form feed character "+" is interpreted as "␣".

O=<u>LASER</u>
Default: Physical overprinting of a record on laser printer is possible exactly once. Records are not immediately output but are buffered (single-line solution with buffering). A record output with the form feed character "+" physically overprints its predecessor record. An attempt to output two succeeding records with the form feed character "+" is rejected and results in abortion of the print process.

If more than one RUNOPT OVERPRINT entry is specified, the last one applies. If the operand portion of the OVERPRINT parameter is empty, e.g. RUNOPT O or RUNOPT O =, it is regarded as not specified and the default value is assumed.

The following table shows the conversion of FORTRAN control characters:

FORTRAN form feed character	Form feed	Conversion to control character		
		OVERPRINT		
		YES	NO	LASER
+	No form feed	00aaaa	40aaaa	00aaaa
’␣’	1 line	41 00bbbb	40bbbb	01bbbb
0	2 lines	42 00cccc	41cccc	02cccc
1	1 page (1st line on next page	C1 00dddd	C1dddd	81dddd

`{aaaa|bbbb|cccc|dddd}          Data`

Table 6-2: Conversion of FORTRAN control characters

Form feed control with RUNOPT OVERPRINT=LASER

[*] RUNOPT OVERPRINT=LASER causes the physical output of a record to be delayed until the next record is available. This record starts the output of the next buffered record, using the control character X’0n’ for form feed after printing. The value for n is calculated from the control character of the successor record, whose output in turn is delayed.
When buffering the first record, a dummy record is output using X’C1’ (single-page form feed) or X’4n’, where n is calculated using the control character of the first record.

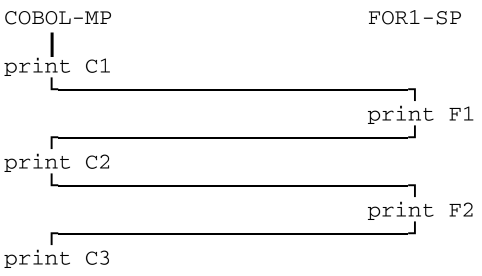
Buffer output

The user can output a record buffered because of OVERPRINT=LASER by calling the FOR1 RTS routine I$PRINT.

FORTRAN statement: CALL I$PRINT

Appropriate use of this statement prevents any shift of the print format, which may occur as a result of the use of non-FORTRAN program units.

Example 1



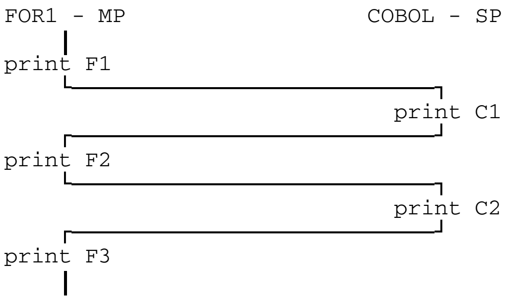
The following shift will occur (only the form feeds of FORTRAN records are shown):

Anticipated print format:	Generated print format (0=LASER):
C1	C1
(form feed of F1)	(form feed of F1)
F1	C2
C2	F1
(form feed of F2)	(form feed of F2)
F2	C3
C3	

The form feed character for F1 is output as a dummy record. Record F1, on the other hand, is buffered until record F2 is available and is then output with the form feed character of F2. Record F2 remains in the buffer because no FORTRAN record follows it.

Remedy: The correct output of FORTRAN records must be controlled through the statement for buffer output (see above).

Example 2



The print format is as follows:

Anticipated print format:	Generated print format (0=LASER)
(form feed of F1)	(form feed of F2)
F1	C1
C1	F1
(form feed of F2)	(form feed of F2)
F2	C2
C2	F2
(form feed of F3)	(form feed of F3)
F3	F3

Because of the print delay of the FORTRAN records due to buffering, the correct sequence has not been observed.

Remedy: Either the user changes the default RUNOPT OVERPRINT = LASER or the correct output of FORTRAN records is controlled through the statement for buffer output (see above).

6.3.4 Suppressing the STXIT error handling routine: RUNOPT STXIT

`[*]RUNOPT`	`STXIT = {YES|NO}` (YES underlined)

STXIT=YES

By default, an STXIT error handling routine is requested when FOR1 is initialized (see appendix A.8.1).

STXIT=NO

The request for an STXIT error handling routine is suppressed. Thus in the case of language interfaces which do not execute in a standard linkage environment it is possible to prevent mutual overwriting of STXIT requests. In ILCS environments, STXIT=NO has no effect since STXITs are requested not by FOR1 itself but by the ILCS initialization.

6.3.5 Setting the machine address mode: RUNOPT START

The address mode determined by the loader can be changed by means of a runtime option while the program is being started:

`[*]RUNOPT`	`START=XS`

START=XS
Sets machine address mode 31.

When machine address mode 24 has been set or determined by the loader, it can only be changed by RUNOPT START=XS if the program concerned is an XS program. (As of FOR1 Version 2.2A, XS modules are always generated. For FOR1 Versions < 2.2A, compilation would need to be effected with EXTENDED-SYSTEM=YES in order to generate XS modules.)

Even when RUNOPT START = XS is specified, a program will run in the lower address space; however, dynamic arrays can still be generated in the upper address space.

Notes

If a non-XS program or a mixed program consisting of non-XS and XS program units which can be executed in machine address mode 24 exists, no error message will be issued after RUNOPT START = XS is entered. In such a case errors which are extremely hard to diagnose may occur during the program run.

In the case of non-XS systems, the START=XS runtime option is simply ignored (although a WARNING is output).

6.3.6 Setting the exponent underflow handling: RUNOPT EXPONENT-UNDERFLOW

Using the EXPONENT-UNDERFLOW runtime option, the setting of the program interrupt due to an exponent underflow (bit 2 in the program mask) can be updated at runtime. Handling of exponent underflow is defined by means of the EXPUNDERFLOW compiler option (default: NOEXPUNDERFLOW).

`[*]RUNOPT`	`EXPONENT-UNDERFLOW={YES|NO}`

NO	The program is not interrupted when an exponent underflow occurs. The data item with which an exponent underflow has occurred is set to the value 0.
YES	The program is interrupted when an exponent underflow occurs and an error message is issued.

6.4 Internal procedures for initializing and terminating programs

6.4.1 Program initialization

A series of initialization measures must be taken internally at the beginning of program execution. This happens in a separate initialization routine.

The initialization routine causes output of a start message, among other things. The start message is output via SYSOUT and has the following format:

```
BS2000 F O R 1 : FORTRAN PROGRAM "name" STARTED ON date AT time
```

name	Name of main program
date	Current date in the form `yyyy-mm-dd`, for example: 1991-08-30
time	Time of loading in the form hh:mm:ss

The start message can be suppressed by setting job switch 4 (/MODIFY-JOB-SWITCHES ON=4).

6.4.2 Program termination

The program termination routine is called by the object program at the end of execution, i.e. when a STOP or END statement occurs or when the EXIT subprogram is called.

Calling the prefabricated EXIT subprogram for program termination can be written in the FORTRAN source program. It has the same effect as a STOP statement. In addition, the program termination routine is called by Error Handling if an error is encountered which makes it impossible to continue execution.

The program termination routine performs the following functions:

1. It closes the input/output operations if the call was issued in an invalid input/output statement.

   If the routine was activated by Error Handling, the first step is to check for any improperly closed input/output operation. Such an operation is then closed by the input/output termination routine. Before that, it may be necessary to output the contents of the input/output buffer.

On the other hand, if program termination was caused by an input/output error, the input/output operation cannot be properly closed, but the corresponding file is closed.

2. It cancels internally generated TFT entries.
3. It calls any termination procedures from other language program sections which may be active.
4. It closes those files which may still be open.

   The program termination routine closes any files still open by calling the CLOSE routine.
   This routine is called with the parameter value KEEP so that the files remain available after the CLOSE routine is executed. Only the temporary work files are closed with the parameter value DELETE and are subsequently no longer available.
   If a file cannot be closed, the following message is output to SYSOUT:

   ```
   FILE name COULD NOT BE PROPERLY CLOSED
   ```

name	Name of file

5. An end message is output.

   At the end of the routine, an end message is issued via SYSOUT.

   ```
                                            ⎧PROPERLY⎫
   BS2000 FOR1:FORTRAN PROGRAM "name" ENDED ⎨        ⎬ AT time
                                            ⎩BADLY   ⎭

               CPU-TIME USED: nn.nnn SECONDS

               ELAPSED TIME: nn.nnn SECONDS
   ```

name	Name of main program
time	Time of program termination
nn.nnn	CPU time used / elapsed time of the object program

   Output of the end message is suppressed if job switch 4 is set to on (/MODIFY-JOB-SWITCHES ON=4).

6.5 Error handling at runtime

6.5.1 Structure of the error messages

When a runtime error occurs, an error message is output through SYSOUT.

The error message generally has the following basic form:

```
type ERROR DETECTED IN MODULE "module" AT hh:mm:ss
WHILE EXECUTING STMT statementno/SEG segno IN PROGRAM UNIT "unit"

xxnn: error message text
```

type designates the type of error. Depending on their severity and/or location of their occurrence, the FOR1 runtime errors are subdivided into the following five types:

FATAL (see 6.5.4)
I/O (see 6.5.5)
LIBRARY (see 6.5.6)
PROGRAM (see 6.5.7)
EXECUTION (see 6.5.8)

xx designates the error condition code, *nn* the hexadecimal error number. The following error condition codes are differentiated:

- Input/output errors

IC	Initial call errors
OP	Open errors
CL	Close errors
IQ	Inquire errors
PO	Positioning erros
IO	Record I/O errors
CO	Conversion errors
FC	Formatted control errors
UC	Unformatted control errors
VS	Value separator errors
NC	Namelist control errors
PA	Pause errors

- Library program errors

SH	"Short" functions (e.g. ABS, IMAG)
PO	Exponentiations
NU	Numeric functions
CH	Character functions
VS	XS-related messages

- Program errors

PR	Program monitoring
UR	Unrepairable error
AR	ARITHMOS error

6.5.2 Program continuation on runtime errors

Errors occurring at runtime (i.e. during execution of the program) do not necessarily lead to abortion of that program. In some cases it is desirable to continue the program, e.g. in order to resume with corrected values or to detect any further errors in a program run. Program continuation is precarious, e.g. if an unrecovered error provokes unrecoverable consequential errors, or if it makes diagnosis difficult in a loop and greatly increases the computer time requirement. Thus, it is up to the users themselves to make a decision on a case by case basis.

An error message is output in the basic form described above for errors which lead to abortion of the program. Afterwards an end message is output by the program termination routine.

In the case of runtime errors which permit continuation of the program, an error message is first output in the basic form. Subsequent messages and possible ways of recovering errors depend on the following conditions:

- Interactive or batch mode
- Type of error
- ERR and IOSTAT parameters in the case of input/output statements
- PARAM DEBUG command
- Debug subprograms OVERFL, DVCHK, FIXOV

If the program is executed in interactive mode, the user can decide whether or not to continue the program in interactive mode at the terminal. The FOR1 runtime system asks the user whether

- program execution is to be continued
- the hierarchy of calls is to be output
- a program interrupt (BREAKPOINT) is to be set.

Users wishing to continue a program may do so - depending on the type of error - by making further entries pertaining to the type of program continuation. In the following sections, the flowchart for the Error Handling Facility and the interaction between FOR1 and the user is displayed for each type of error.

If the program is executed in **batch mode**, the user cannot make a decision about program continuation once an error has occurred. In batch mode, the user can however activate error recovery in the event of an error by issuing the command /PARAM DEBUG=YES or by means of the SDF operand OBJECT-CONTINUATION=YES before the program starts. If possible, the runtime system then continues execution of the program, using internally predefined values ("default values"). If /PARAM DEBUG=NO or the SDF operand OBJECT-CONTINUATION=NO is specified, runtime errors will result in program termination.

In **interactive mode**, the error message in its basic form is additionally output via SYSLST. In batch mode, the error message appears in the log of the ENTER job, the log being output via SYSOUT. In addition, the error messages and information for diagnosing the error are output via SYSLST. The information for error diagnosis corresponds to the information which can be called from the terminal in interactive mode (output of parameters, hierarchy of calls, registers).

The ERR, END or IOSTAT parameter in input/output statements causes a branch to the user's own Error Handling Facility in the event of an error. In the case of input/output errors, the program is continued at the statement label specified in the ERR or END parameter.

6.5.3 Monitoring program execution with job variables

The execution of a FORTRAN program can be monitored with the aid of the software product JV (job variables, see "Job Variables" manual [24]). A previously defined job variable is specified as an operand in a LOGON, ENTER-JOB or START-PROGRAM command. Information on the current status of the program ("status indicator") and further information on the execution of the program ("return code indicator") are entered in this job variable. The user can interrogate this information and control further jobs and programs as a function of this information. The meaning of the indicators of job variables is described in section 4.9.

The status indicator in the first 3 bytes of the job variable can contain the following information:

$T␣	Program terminated normally.
$A␣	Program terminated abnormally. This indicator is likewise set by the system when the program is aborted.
$R␣	Once a program is started, the status indicator is set to "$R".

The termination code in the 4th byte of the job variable can contain the following information after a FORTRAN program run:

0	The object program run was free of error.
2	The object program has been terminated in a controlled manner due to the occurrence of errors.

The program information in bytes 5 through 7 of the job variable contains 3 blanks after the FORTRAN program run.

6.5.4 Fatal errors

These errors make abnormal termination of the program unavoidable, since meaningful results cannot be expected and the program cannot be continued when they are encountered.

Fatal errors occur, for example, in the following instances:

- Memory exhausted (NO DYNAMIC STORAGE AVAILABLE)
- Endless loop in a program (PROGRAM LOOP DETECTED)
- Errored call of the ALLOC subprogram ALLOC (WRONG LIMITS IN ALLOCATION CALL)

If a fatal error occurs, the program is terminated in a controlled manner (end handling).

6.5.5 Input/output errors

These errors appear in connection with input/output statements.

Input/output errors occur, for example, in the following instances:

- Device errors
- Format errors
- Data errors (types)
- DMS errors
- Illegal operations
- Sequence errors.

The user can program his own error recovery by specifying the ERR parameter in input/output statements. When an error occurs, the program then continues at the statement label specified in the ERR parameter.

If end of file is reached during execution of a READ statement or positioning statement, the user can prevent the program from being interrupted by specifying the END parameter. When the end of the file is reached, the program is continued at the statement label specified in the END parameter.

If no ERR or END parameter is specified, but an IOSTAT parameter is, the program is continued in the event of an error or when the end of file is reached, starting with the next executable statement.
If neither an ERR nor an IOSTAT parameter is specified, an error message is output when an error occurs and the user is queried as to whether or not the program is to be continued. If neither an END nor an IOSTAT parameter is specified, the message "EOF^x80ON^x80UNIT dsetno (pathname)" is output when the end of file is reached, and the program is aborted.

The various possibilities are summarized in the following table.

Specified parameters	Condition	
	Error	End-of-file
-	Error message; query: Continue program?	EOF message Abort program
ERR	ERR statement label	EOF message Abort program
END	Error message; query: Continue program?	END statement label
IOSTAT	Next executable statement	Next executable statement
ERR END	ERR statement label	END statement label
ERR IOSTAT	ERR statement label	Next executable statement
END IOSTAT	Next executable statement	END statement label
END/ERR IOSTAT	ERR statement label	END statement label

Table 6-3: Program continuation as a function of ERR, END, and IOSTAT parameters

The error message begins with an error number, in hexadecimal form, corresponding to the IOSTAT error number. For example, the hexadecimal error number 15 in the message

```
OP15: FILE COULD NOT BE OPENED
```

corresponds to IOSTAT error number 21.

The program can resume with the executable statement immediately following the input/output statement which caused the error. However this requires that the presetting COMOPT TESTOPT=(STNR) set by default applies, so that the compiler will generate a statement table in the object which the FOR1 runtime system can execute.

Program continuation is possible both in interactive and batch mode, as the following flowchart illustrates. The second diagram shows the prompting sequence in interactive mode.

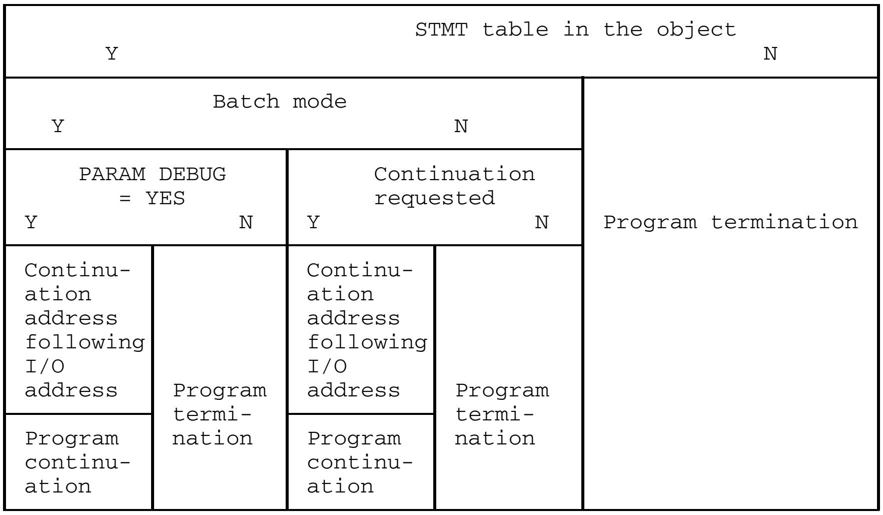


Fig. 6-1: Flowchart for input/output errors

Interaction for input/output errors:

FOR1 display	User response	FOR1 reaction
OBJECT CONTINUATION? Y/N	Y	Program run continues after illegal input/output statement.
	N	Dialog is continued.
DO YOU WANT PARAMETERS DISPLAYED? Y/N	Y	Parameters are displayed.
DO YOU WISH CALLING SEQUENCE DISPLAYED? Y/N	Y	Call hierarchy is output.
DO YOU WISH A BREAKPOINT? Y/N	Y	Program is interrupted.

6.5.6 Errors in mathematical library programs

These errors occur if FOR1 intrinsic functions are invoked and actual arguments are incorrectly supplied.

Such errors may occur in the following instances, for example:

- Argument too large or too small
- Argument illegally negative
- Invalid argument type
- Division by zero
- Base is zero.

Resumption may be advisable if the user is simulating results which the function causing the error should have produced. There is a difference between interactive and batch mode.

In interactive mode, users see the invoked function and the expected type of result so that they can specify the appropriate result with which they wish to resume.
In batch mode, such values can be assumed where the danger of consequential errors is minimal (e.g. 1 or 1.0).

The flowchart and the sequence of prompts that may appear in interactive mode are shown below.

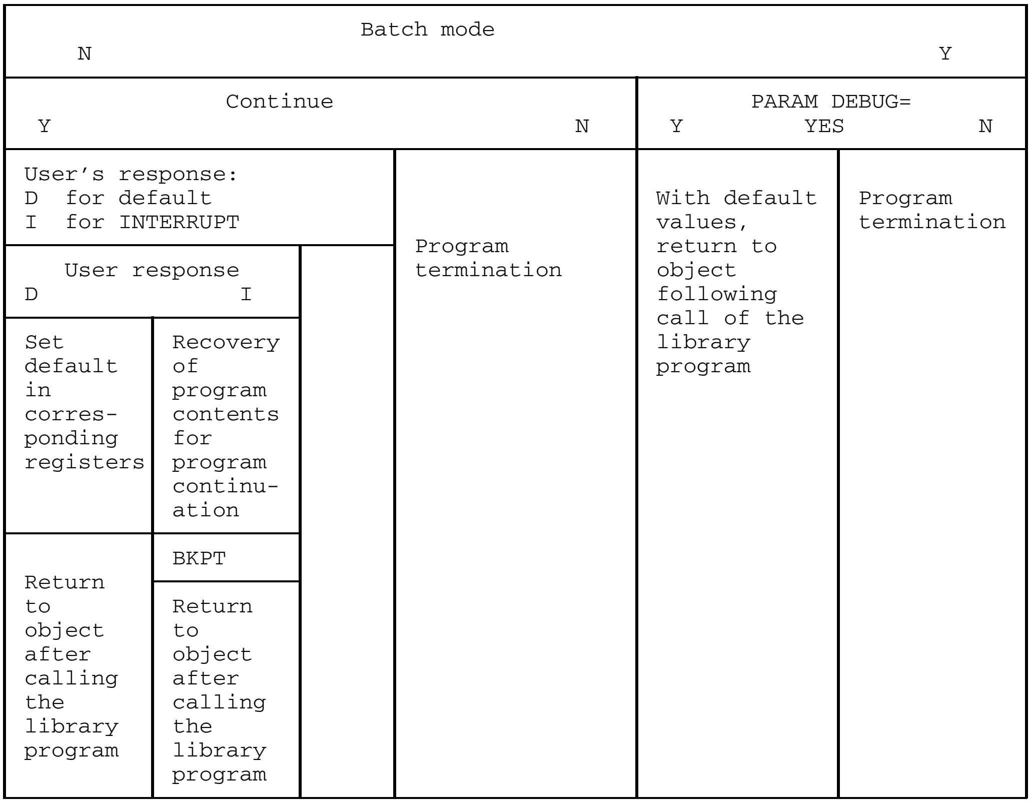

Batch mode
N
Y
Continue
Y
N
PARAM DEBUG= YES
Y
N
User's response: D for default I for INTERRUPT
User response
D
I
Set default in corres-ponding registers
Recovery of program contents for program continu-ation
BKPT
Return to object after calling the library program
Return to object after calling the library program
Program termination
With default values, return to object following call of the library program
Program termination

Fig. 6-2: Flowchart for library errors

Interaction for library errors:

FOR1 display	User response	FOR1 reaction
OBJECT CONTINUATION? Y/N	N	Continue dialog for (1).
	Y	Prompt user for type of continuation.
DEFAULT VALUE (D) or INTERRUPT (I)	D	Default values are loaded into corresponding registers. Execution continues after CALL.
	I	Branch (BKPT) to system mode so that the user can set the desired value in corresponding registers by means of AID.
BKPT	AID cmds.; /RESUME -PROGRAM	Execution continues after CALL.
(1) DO YOU WISH CALLING SEQUENCE DISPLAYED? Y/N	Y	Call hierarchy is output.
DO YOU WISH A BREAKPOINT? Y/N	Y	Program is interrupted.

Example:

The following program contains an invalid argument: negative radicand. A library error is reported when the program is executed.

```
/SET-TASKLIB LIB=$FOR1MODLIBS                                              (01)
/DEL-SYS-FILE OMF
/START-PROG $FOR1
 %  BLS0500 PROGRAM 'FOR1', VERSION '2.2A00' OF '91-06-05' LOADED.
 %  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG. 1991 ...
  FOR1:    V2.2A00 READY, GIVE COMPILER OPTION
 *COMOPT LISTFILE=LIST(OBJECT),END
 *       PROGRAM SQ                                                        (02)
 *       R=-16
 *       X=SQRT(R)
 *       WRITE *,X
 *       END
 */*                                                                       (03)
  FOR1: NO ERRORS DURING COMPILATION OF P.U. SQ
  END OF  F O R 1  COMPILATION;  CPU TIME USED: 0.191 SEC.
```

```
Excerpt from the OBJECT listing:

****  STATEMENT 3 (ASSIGNMENT) ********
         X = SQRT(R)
         LA    1,264(0,13)
         USING SQ@@@@@@+264,1
         L     15,136(0,12)
         LA    0,1(0,0)
         BALR  14,15                    IF@Q                       (04)
         STE   0,256(0,13)              X                          (05)
```

```
/START-PROG FROM-FILE=*MODULE(*OMF)                                (06)
 %  BLS0001 ### DBL VERSION 070 RUNNING
 %  BLS0517 MODULE 'SQ' LOADED
 BS2000  F O R 1 : FORTRAN PROGRAM "SQ"
 STARTED ON 1991-08-30 AT 13:57:38
  LIBRARY ERROR DETECTED IN MODULE "IF@Q    " AT 13:57:38          (07)
 WHILE EXECUTING STMT        3/SEG        1  IN PROGRAM UNIT "SQ      "
 NU1A: ARGUMENT X < 0
 OBJECT CONTINUATION ? Y/N
 Y
 DEFAULT VALUE "D" OR INTERRUPT "I" ? D/I
 I
 %  IDA0199 PROGRAM BREAK AT 02C956, AMODE = 24
 /%AID CHECK=ALL
 /%SET 4 INTO %0E                                                  (08)
  OLD CONTENT:
  -.1600000 E+002
  NEW CONTENT:
  +.4000000 E+001
 %  IDA0129 CHANGE? (Y=YES; N=NO)
 Y
 /%R
 0.40000000E+01
 BS2000  F O R 1 : FORTRAN PROGRAM "SQ      " ENDED PROPERLY AT 13:58:04
 CPU - TIME USED :       0.0491 SECONDS
 ELAPSED TIME    :      26.2960 SECONDS
```

Explanation of example:

(01) Runtime system assignment.
(02) Program input directly at the terminal.
(03) End of input.
(04) The BALR instruction calls root function IF@Q.
(05) The result is moved from floating point register 0 to the result field using the STE instruction.
(06) Program execution.
(07) Error message.
(08) Modification of memory contents (floating point register 0; single precision).

6.5.7 Program errors

These errors, recognized during the execution of machine instructions, are supplied via STXIT routines to the FOR1 runtime system for further handling. The errors are the results of interruptions in the STXIT event classes "Program check" and "Unrecoverable program error" (see "Executive Macros" manual [26]).

The following are examples of program errors:

- Address translation error
- Illegal SVC or illegal operation code
- Addressing error
- Data error
- Exponent overflow/underflow
- Divide error
- Decimal overflow
- Fixed point overflow.

The type of error determines the point where the program run resumes. A distinction is made between arithmetic errors and other errors.

Arithmetic errors:

These are characterized by hardware-produced interrupt weights of X’64’ -X’78’ (see "Assembler" manual [8]). The user, in the result register of the interrupted machine command instruction, may simulate a result which is suitable for continuing the computation. For some errors, the appropriate debug subprograms (OVERFL, FIXOV or DVCHK) can be linked into the program to prevent abnormal termination and supply defaults. If they are not linked in, the simulated result of computation may be

- entered by the user (in response to prompts) in interactive mode, or
- specified by the runtime system in batch mode.

Program execution continues after the interrupted machine instruction.

Other errors:

These are characterized by interrupt weights of X’48’ - X’60’. No results need be supplied here. There is only one difference: whether the interrupt is caused by the runtime system or by the FOR1 object:

- interrupt in the runtime system: execution is resumed after the call of the routine concerned.
- interrupt in the object: the interrupting FORTRAN statement is skipped and processing resumes with the next statement.

When program errors occur, an error message in the basic form is issued. Afterwards the absolute address, the address relative to the beginning of the module and the machine instruction for which the error occurred are output:

```
PROGRAM COUNT AT INTERRUPT OCCURENCE: absolute address CC=n

PROGRAM COUNT RELATIVE TO ENTRYPOINT: relative address

STATEMENT CODE: machine instruction
```

Subsequent to this the contents of the general-purpose and floating point registers are output. In interactive mode, the user can then decide about continuing the program.

The flowchart is shown below. The interactive prompts are the same as those for library errors, except that the program run continues after the illegal machine instruction. The simulated result of the illegal operation is loaded into the result register, whose number is obtained from the instruction code.

Nested error handling is possible, so that errors which occur in the error routine can be handled.

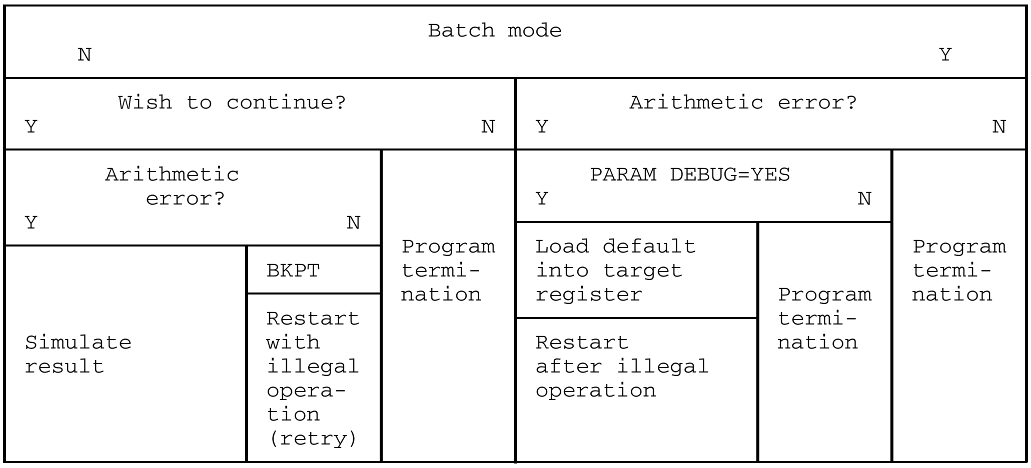


Fig. 6-3: Flowchart for program errors

Result simulation:

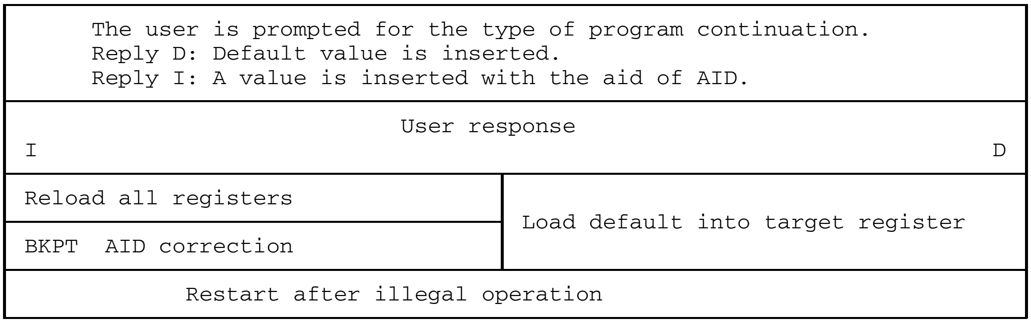

The user is prompted for the type of program continuation.
Reply D: Default value is inserted.
Reply I: A value is inserted with the aid of AID.
User response
I
D
Reload all registers
BKPT AID correction
Load default into target register
Restart after illegal operation

Fig. 6-4: Result simulation for program errors

6.5.8 Errors in debug options, debug statements, irregular flow of control

In the error message, these errors are designated "EXECUTION ERROR". There are three different types of errors.

Errors as a consequence of TESTOPT

As these errors are only reported if the debug options were activated at compile time, the user can, of course, treat the error messages like warnings and, following the display of these messages, continue executing the program in interactive mode in a totally normal fashion. The object ensures that a usable return address is provided in register 14. Usually
the system soon reports another error as the result of the first error.
In batch mode, this leads to abortion of the program.

Errors in the %COUNT evaluation

As the %COUNT statement is evaluated at the very end of program execution - when the production run is over - the user can simply cancel the evaluation and report generation process and let the object continue with the rest of the program termination routines (file close, closedown procedure calls, runtime calculation).

Errors due to irregularities in the flow of control

These include the following cases:

- illegal value in a DO loop
- illegal reference in assigned GOTO

- recursive subprogram call
- invalid END or ERR entry
- invalid format specification
- format group to be repeated does not contain conversion format
- zero step width in DO loop
- DO control variable overwritten outside loop

For any of these errors, users must decide individually where they wish to continue program execution. There is a difference between interactive mode and batch mode: In interactive mode, the user decides whether and from which address, label or FORTRAN statement he wishes to continue execution. In batch mode, the program is aborted.

Shown below are the flowchart and the prompting system.

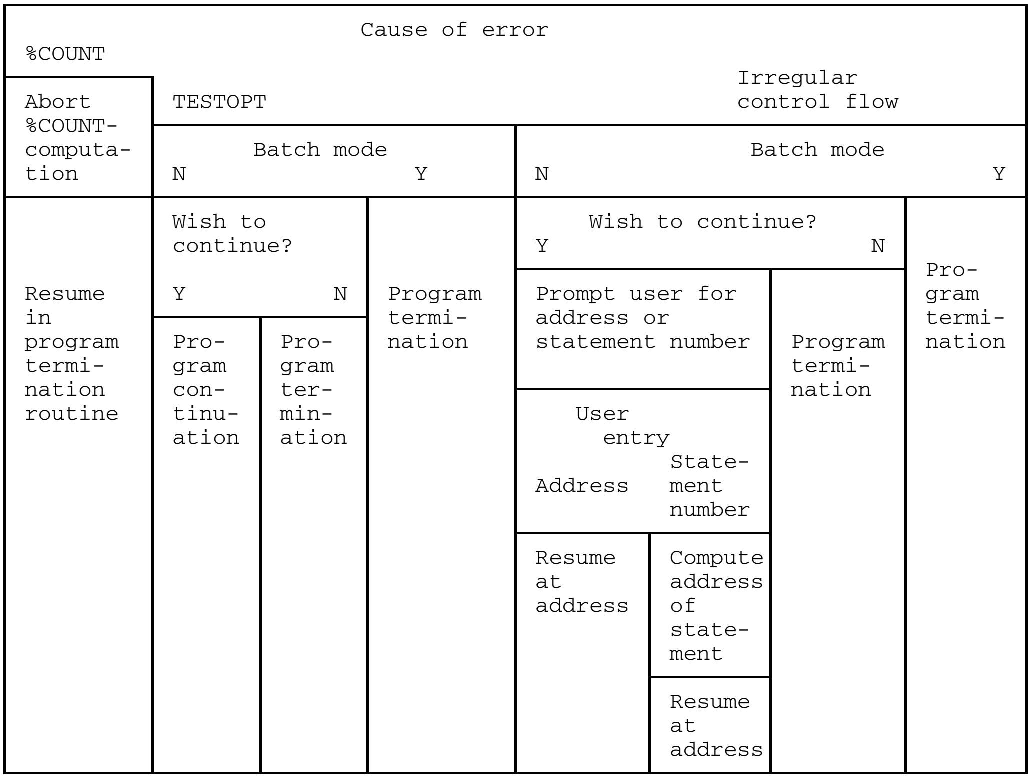


Fig. 6-5: Flowchart for errors in debug options, debug statements, irregular flow of control

Interaction for irregular flow of control:

FOR1 display	User response	FOR1 reaction
OBJECT CONTINUATION? Y/N	N	Continue dialog for (1).
	Y	Prompt user for address or statement number.
GIVE STORAGE ADDRESS (NNNNNN) OR STMT-NR (%NNN) WHERE TO CONTINUE	NNNNNN	Continue program run at specified address.
	%NNN	Compute address of specified statement by means of statement table and continue execution there.
(1) DO YOU WISH CALLING SEQUENCE DISPLAYED? Y/N	Y	Call hierarchy is displayed.
DO YOU WISH A BREAKPOINT? Y/N	Y	Program is interrupted.

7 Debugging aids

A FORTRAN program which has been compiled without error can possibly still be erro-red, and this is not noticeable until the program is executed. The following debugging aids are available to the user for locating runtime errors:

- debug options in the form of compiler options,
- debug statements in the form of compile time statements,
- debug subprograms in the form of ready-made subprograms,
- the debugging aid AID (as of AID Version 1.0C).

7.1 Controlling the debugging aids: SDF operand TEST-SUPPORT

```
START-FOR1-COMPILER
--------------------------------------------------------------
,TEST-SUPPORT = STD / NO / PARAMETER(...)

   PARAMETER(...)
   |
   |     STATEMENT-TABLE = YES / NO
   |
   |    ,TOOL-SUPPORT = NO / AID
   |
   |    ,CHECK-CODE = NO / ALL / YES(...)
   |
   |       YES(...)
   |       |    PROCEDURE-ARGUMENTS = NO / YES
   |       |   ,ARRAY-BOUNDS = NO / YES
   |       |   ,ARRAY-SUBSCRIPTS = NO / YES
   |       |   ,SUBSTRING-BOUNDS = NO / YES
   |       |   ,BRANCH-STMTS = NO / YES
   |       |   ,VARIABLE-ASSIGNMENT = NO / YES
   |       |   ,USER-DEBUG-STMTS = NO / YES
```

The SDF operands and corresponding compiler options are shown in table 2-10.

7.2 Summary: SDF operand TEST-SUPPORT and corresponding compiler options

In the following summary, the compiler option is given in the first line with the corresponding SDF operand in the second line, followed by the meaning.

```
TESTOPT=(ALL)
CHECK-CODE=ALL
```

All tests listed in the following are activated.

```
TESTOPT=(STNR)
STATEMENT-TABLE=YES
```

For all error messages the appropriate source program statement number is output.
The standard default TESTOPT=(STNR) is always valid, unless it has been deactivated by means of NOTESTOPT [=(ALL)] (see examples at the end of this section).

```
TESTOPT=(ARG)
CHECK-CODE=YES(PROCEDURE-ARGUMENTS=YES)
```

On entry into a subprogram the arguments passed are checked for number, type and length.
If an ILCS program in a different language calls a FOR1-ILCS object, only the number of parameters is compared.

```
TESTOPT=(BOUNDS)
CHECK-CODE=YES(ARRAY-BOUNDS=YES)
```

With array elements a check is made as to whether each subscript is within its declared bounds (see section 9.3.4, "Subscript computation").

```
TESTOPT=(SUBSCR)
CHECK-CODE=YES(ARRAY-SUBSCRIPTS=YES)
```

With array elements a check is made as to whether the element address calculated in accordance with the usual formula is within the array. This check is less rigid than the previous one.

```
TESTOPT=(STRING)
CHECK-CODE=YES(SUBSTRING-BOUNDS=YES)
```

With substring accesses a check is made as to whether the reference substring is entirely within the associated variables.

```
TESTOPT=(CNTRL)
CHECK-CODE=YES(BRANCH-STMTS=YES)
```

All set GOTOs and all GOTOs into loops are checked for correctness. The compiler checks whether the value of the loop variable has been changed.

```
TESTOPT=(UNDEF)
CHECK-CODE=YES(VARIABLE-ASSIGNMENT=YES)
```

When using the values of variables a check is made as to whether they have defined values at all.

```
TESTOPT=(DEBUG)
CHECK-CODE=YES(USER-DEBUG-STMTS=YES)
```

The debug statements (see section 7.4) are also compiled by FOR1.

Complementary quantities can be specified by using the prefix NO (see section 2.3.1).

7.3 Controlling the debugging aids with the TESTOPT compiler option

The user uses the TESTOPT compiler option to generate tests to locate runtime errors in the object program. On the basis of these tests, errors which could result in incorrect results, inexplicable behavior or abnormal program termination are detected and reported at compile time or runtime. Such errors are referred to as "EXECUTION ERROR" in the message output. The semantics of the program are not affected by debug options.

Debug options also control the output of statement numbers (STMT) in the event of runtime errors and the efficacy of the debugging statements (see section 7.4).

Debug options are specified in the TESTOPT option.

```
[*]COMOPT        [NO] TESTOPT [=([testparameter][,...])]

                 testparameter:={ALL|STNR|ARG|BOUNDS|
                                 SUBSCR|STRING|CNTRL|UNDEF|
                                 DEBUG}
```

ALL All actions described in this section are carried out.

STNR For runtime errors, error message output also includes the statement number (STMT) of the statement that caused the error (default). Execution is not burdened by additional code.

ARG For function and subprogram calls, the number, type and length of the actual and dummy arguments are checked for consistency.

The check is not possible with subprograms that are called by programs in other languages (except in the case of Assembler, if the corresponding macros have been used, and in the case of PLI1).
If an ILCS program in a different language calls a FOR1-ILCS object, only the number of parameters is compared.

For program linkage of OLD, non-XS and XS object programs (see section A.7), the following checks are performed and corresponding messages output (if required) at runtime:

- For the linkage of OLD with non-XS programs, no dynamic array can be passed as the argument;
- a dynamic array as the actual parameter must, when used as a dummy argument, likewise correspond to a dynamic array with the same dimension number;
- calls of OLD and non-XS subprograms can only take place in machine

address mode 24;
- in the XSTONXS call, no XS program can be used as the subprogram;
- no OLD parameter list must be passed to an XS subprogram;
- no XS parameter list must be passed to an OLD subprogram.

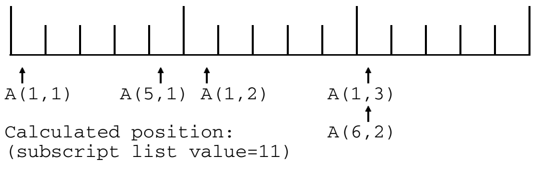
BOUNDS Each array element name is checked as to whether the subscript values are within the subscript bounds of that array. If BOUNDS is specified, errors will be reported for the examples described under SUBSCR.
This check is only possible with optimization deactivated. If BOUNDS is specified in conjunction with OPTIMIZE>0, FOR1 switches on SUBSCR and issues a message.

SUBSCR Each array element name is checked as to whether the calculated position of the array element (subscript list value) is within the bounds of the array (see formula in section 9.3.4).

Example:

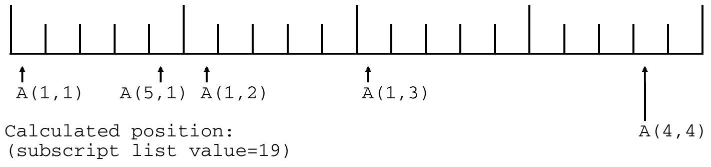
```
DIMENSION A(5,3)
I=6
J=2
A(I,J)= ...
```

This element is within the bounds of the array; no error is reported.



```
DIMENSION A(5,3)
I=4
J=4
A(I,J)= ...
```

In this case, the array element is outside the bounds of the array; an error message is therefore issued.



STRING When using CHARACTER substrings, a check is made as to whether the specified values for the bounds of the substring are within the associated CHARACTER variable. An error message also appears if the value of the lower bound of the substring is greater than the value of the upper bound.

Example:

```
CHARACTER*20 B
I=20
J=22
B(I:J) = ...
```

An error is reported because the end of the substring is outside the variable.

CNTRL Transfers of control within a program unit are checked for validity.

For transfers of control into the range of DO loops, the compiler generates the test code only if a simultaneous branch from the loop range is intended (extended range for relocating parts of the loop range). At runtime, a check is made as to whether the iteration counter of these loops and any existing comprehensive loops shows a value greater than zero and whether the running variable of these loops has changed.

The compiler checks whether the value of the loop variable within the loop range has changed. If an assignment, multiple assignment, ASSIGN, READ or DECODE statement exists, a warning message (SA 168) is issued.

For assigned GOTOs, a check is made as to whether the corresponding variable was assigned a valid branch label by an ASSIGN statement. The same checks are also performed for assigned labels in the ERR and END parameters of input/output statements and for assigned format.

UNDEF Whenever the value of an item is to be used, a check is made as to whether it has already been assigned any value. To enable these checks to be carried out, the first step is to initialize to hexadecimal ’80’ the data sections of all program unit sections which have not been initialized by user entries. For items whose value shows this bit pattern (X’80’) during the execution of a program, it is assumed that they have not yet been assigned a value; therefore an error message appears if their value is accessed. In the case of INTEGER*1 items, initialization with X’80’ is performed, however the check is not, since there are only 256 possible values and X’80’ represents the value -128. This check is also not performed for a CHARACTER item with variable length, since the length 0 is in this case permissible (see "FOR1" manual [21]). Likewise dynamic arrays are not checked for X’80’.

If an UNDEF test pattern is detected, the user is informed by an error message (at program runtime). It is up to the user whether or not the program run is continued (see section 6.5.8).
Variables which, at compile time, are known to have a defined value (e.g.

through initialization in a DATA statement) are not checked at runtime. If at least one array element is defined in a field, the entire array is regarded as being defined at compile time. Non-defined array elements are rejected at runtime.

DEBUG	Instruction sequences are generated for the debug statements specified in the source program (see section 7.4). If this parameter is omitted, the debug statements are ignored.

Restrictions:

- The TESTOPT and OPTIMIZE=3,4 compiler options are mutually exclusive. The most recently specified compiler option is applicable.
- If TESTOPT is specified in conjunction with PROCEDURE-OPTIMIZATION=YES or PROCEDURE-OPTIMIZATION=SPECIAL, PROCEDURE-OPTIMIZATION is then reset to NO.

Examples:

Option specified:	Effect:
[*]COMOPT TESTOPT=(ALL)	All checks are carried out.
[*]COMOPT NOTESTOPT=(STRING)	All checks are carried out except those described for STRING.
[*]COMOPT NOTESTOPT	No checks whatsoever are carried out. Statement number output for runtime errors is also suppressed.
[*]COMOPT NOTESTOPT=(STNR)	Since the default TESTOPT=(STNR) is still valid, provided it has not been deactivated by means of NOTESTOPT[=(ALL)], all checks including STNR are performed.

7.4 Debug statements (controlling the debugging aids through statements in the source program)

Debug statements are written in the FORTRAN source program and provide assistance in troubleshooting programs. At compile time, the debug statements are translated into instruction sequences that issue information about the structure and execution of the program as well as the dynamic values of the data in that program.

Unlike the debug options (see section 7.4), which apply to the entire process of compilation, debug statements may be written at specific points in the source program. They are valid only in the program unit in which they were defined and they are valid for an area determined by the user.

Debug statements begin with a percentage sign.
Debug statements must not be preceded by statement labels.

The compiler option COMOPT TESTOPT = (DEBUG) is a prerequisite for compiling the debug statements. If this compiler option is omitted, the debug statements are simply ignored.

7.4.1 Overview: Debug statements

Format	**Meaning**
`%DISPLAY [(unit)] name [,name]...`	Output of values of variables and arrays.
`%CHECK` $\begin{Bmatrix}\text{ON}\\ \text{OFF}\end{Bmatrix}$ `[(unit)][name[,name]...]`	Output of new values of variables and array elements with assignments between ...ON and ...OFF.
`%CALLTRACE` $\begin{Bmatrix}\text{ON}\\ \text{OFF}\end{Bmatrix}$ `[(unit)]`	Output of call information, calling or called program unit, statement number of call point, entry point, parameter addresses and values for each call between ...ON and ...OFF.
`%JUMPTRACE` $\begin{Bmatrix}\text{ON}\\ \text{OFF}\end{Bmatrix}$ `[(unit)]`	Logging of GOTO's performed to statement labels between ...ON and ...OFF.
`%FULLTRACE` $\begin{Bmatrix}\text{ON}\\ \text{OFF}\end{Bmatrix}$ `[(unit)]`	Logging of the numbers of statements executed as well as output of new values of variables and array elements for assignments between ...ON and ...OFF.
`%COUNT` $\begin{Bmatrix}\text{ON}\\ \text{OFF}\end{Bmatrix}$	Count of statement execution frequency and timing between ...ON and ...OFF.

unit	Variable or constant
name	Name of an item

The parameter value *unit* may be used to specify the file number of the file to which output is to be made. Standard output is to SYSLST.

If a debug statement contains a file number, it remains in effect for all further occurrences of the same debug statement until a further occurrence of the same debug statement specifies another file number.

The debug statement is activated by the parameter ON and deactivated by the parameter OFF.

If no statement for deactivating the debug function is given, the debug function is deactivated at the end of the program unit.

If the OFF parameter is specified in a debug statement along with a file number that matches the file number currently in effect for that debug statement, the statement deactivates the debug function and resets output for these debug statements to SYSLST.

If no FILE command was given for the specified file number, a file with the following standard name is generated:

```
DBG.FOR1.stmt.prog.unit[.tsn[.time]]
```

where

stmt	Name of the debug statement in the form #DISPL, #CHECK, #FULL, #CALL or #JUMP
prog	Name of the program unit
unit	File number
tsn	Task sequence number, 4-digit
time	File creation time in the form hhmmss

The qualifications *tsn* and *time* are only performed if the entry would otherwise be ambiguous.

7.4.2 %DISPLAY statement

The %DISPLAY statement may be used to output the names and current values of items.

```
%DISPLAY [(unit)] name [,name]...
```

unit	Variable or constant.
name	Name of an item, i.e. of a variable or an array.

Form of output for variable and array:

```
 %DISPLAY/prog/ddddd:
name = value,

 %DISPLAY/prog/dddddd:
name = value1, value2, value3, ...
```

prog	Name of the program unit.
ddddd	Statement number of the %DISPLAY statement.
name	Name of an item which was specified in the %DISPLAY statement.

The items are displayed in the order in which they were supplied in the associated %DISPLAY statement. The values of each item are shown in the standard format for the type concerned. The standard formats are described in the "FOR1" manual [21]. If an array name is specified, then all elements of the array are shown.

7.4.3 %CHECK statement

The %CHECK statement may be used to trace changes to values of variables.

```
        ⎧ON ⎫
%CHECK  ⎨   ⎬ [(unit)] [name [,name]...]
        ⎩OFF⎭
```

unit	Variable or constant.
name	Name of a an item, i.e. of a variable or an array.

%CHECK ON (unit) name,name,...name
activates the debug function for the items specified.

%CHECK OFF (unit) name,...name
deactivates the debug functions for the items specified.

%CHECK OFF (unit)
While this statement without specification of items deactivates the check, a subsequent %CHECK ON statement activates the check again for the newly specified items *and* the old items.

The %CHECK ON (unit) statement without specification of items as the first %CHECK statement is meaningless.

If the value of a specified item changes, the current value of the item is displayed at the end of the corresponding statement.

Position where values may change	Data concerned
Left side of assignment statement, input/output list in input statement, storage input/output unit in output statements	All data occurring
ENTRY statement	All dummy arguments of the entry and COMMON data of the program unit
Function call and CALL statement	Actual arguments which occur, if variable or array name, and all COMMON data of the (calling) program unit.

Form of output for the variable or array:

```
 %CHECK/prog/aaaaa :
name = value ,

 %CHECK/prog/bbbbb :
name = value1 , value2 , value3 , ...
```

Form of output for the LABEL variable:

```
 %CHECK/prog/aaaaa:
name = 'ZZZZ (LABEL)',
```

prog	Name of program unit
aaaaa,bbbbb	Numbers of those statements where the value change has taken place.
name	Name of an item specified in the %CHECK statement.
ZZZZ	Statement label

The values of items are displayed in the standard format for the type concerned. The standard formats are described in the "FOR1" reference manual [21].

7.4.4 %CALLTRACE statement

The %CALLTRACE statement provides information on the subprogram and function calls within a program unit.

```
            ⎧ON ⎫
%CALLTRACE  ⎨   ⎬ [(unit)]
            ⎩OFF⎭
```

The name of the entry point, the name of the associated program unit, and the transmitted argument values are shown for each call issued within the range between %CALLTRACE ON and %CALLTRACE OFF.

If further calls are made in a called program unit, no information will be displayed about these calls unless a %CALLTRACE statement was also specified for the called program unit.

Form of output:

```
   %CALLTRACE/prog/cccc:      CALLED ENTRY : <entry           subprogram"
 SUBMITTED PARAMETERS :
ARG.  ABSOLUTE   PROGRAM  ARGUMENT   DATA   HEXADECIMAL VALUE                VALUE
NO.   LOCATION   ADDRESS   OFFSET    TYPE
____  ________  ________  ________  ______  ________________________________  ______________
  .       .      .         .         .       .        .
  .       .      .         .         .       .        .
  .       .      .         .         .       .        .
  .       .      .         .         .       .        .
```

prog	Name of the calling program unit
ccccc	Statement number of each statement where the call is made
entry	Name of the entry point
subprogram	Name of the called program unit

7.4.5 %JUMPTRACE statement

The %JUMPTRACE statement provides information on transfers of control within a program unit.

```
            ⎧ON ⎫
%JUMPTRACE  ⎨   ⎬ [(unit)]
            ⎩OFF⎭
```

For each transfer of control within a program unit, the segment and statement number of the exit and entry point are shown, provided that the branch point is within the range between %JUMPTRACE ON and %JUMPTRACE OFF.

This applies to explicit branches caused by a GOTO statement as well as implicit branches such as those caused by DO statements or IF statements.

Form of output:

```
%JUMPTRACE/prog/jjjjj:SEG-NR=SSSSS;STMT-NR=sssss
```

prog	Name of the program unit
jjjjj	Statement number of the %JUMPTRACE statement
SSSSS	Segment number of the destination point
sssss	Statement number of the destination point

7.4.6 %FULLTRACE statement

The %FULLTRACE statement provides information on the execution sequence of statements and value changes of all variables in the respective statements.

```
            ⎧ON ⎫
%FULLTRACE  ⎨   ⎬ [(unit)]
            ⎩OFF⎭
```

For each statement executed and any value change of the variables within the range between %FULLTRACE ON and %FULLTRACE OFF the statement number, segment number and the current value of the item are shown at the end of the statement concerned.

Form of output:

```
%FULLTRACE/prog/fffff:SEG-NR=SSSSS;STMT-NR=sssss
```

prog	Name of the program unit
fffff	Statement number of the associated %FULLTRACE ON statement
SSSSS	Segment number
sssss	Statement number

When %FULLTRACE is specified, a %CHECK listing is output for all value changes. The %FULLTRACE statement should only be used for smaller ranges since it considerably increases compile time, the size of the object program generated, the execution time, and the amount of data output. The results that may be achieved with the two following statements are usually sufficient for tracing the execution of the program.

7.4.7 %COUNT statement

The %COUNT statement provides information on the frequency with which the various program parts are executed as well as the time each part takes.

```
        ⎧ON ⎫
%COUNT  ⎨   ⎬
        ⎩OFF⎭
```

For each segment whose beginning is in the range between %COUNT ON and %COUNT OFF, the number of passes through the segment is traced for the whole object program execution.

In addition, the approximate time needed for each statement in the specified area is also determined approximately. At the end of program execution, two statistics are issued for each program unit in which these counts were taken. Output is not to a random file number as in the case of the other debug statements, but to SYSLST. Output to another unit can be defined by means of the runtime options (see section 6.3).

If such statistics are output for two or more program units of a program system, an overall statistics list is produced at the end, covering all these program units.

Form of output:

Statistic 1

```
DYNAMIC COUNT PROFILE OF PROGRAM UNIT prog

SEG-NR  STMT-NR  COUNT  PROFILE
SSSSS   sssss      n    ******
  .        .       .      .
  .        .       .      .
  .        .       .      .
```

prog	Name of the program unit
SSSSS	Segment number
sssss	Statement number of the first statement in the segment concerned
n	Number of passes through that segment
******	Histogram illustrating the relationship of the passes between the individual segments for which the count was performed

Statistic 2

```
DYNAMIC TIME PROFILE OF PROGRAM UNIT prog

SEG-NR  STMT-NR  TIME   PROFILE
SSSSS   sssss     n     ******
  .       .       .        .
  .       .       .        .
  .       .       .        .
```

prog	Name of program unit
SSSSS	Segment number
sssss	Statement number
n	Relative ratio between the time required for all executions of the respective statements and the total time requirements of all statements recognized in that program unit
******	Histogram illustrating the relative ratio

Statistic 3

```
DYNAMIC TIME PROFILE ON ALL COUNTED PROGRAM UNITS
PROGRAM UNIT   TIME    PROFILE
prog            n     ******
 .              .        .
 .              .        .
 .              .        .
```

prog	Name of the program unit.
n	Relative ratio between the time requirements of that program unit and the total time requirements of all statements for which the corresponding %COUNT statements were given.
******	Histogram illustating the relative ratio

The number of passes registered by %COUNT and the generation of the listings are evaluated only at the end of the program run. At this time all program units concerned must be in main memory and must not be overlaid by other program units.

Full use of the information about the proportionate times of program units can be made only if all statements of each program unit involved are in the counting range. If a program unit calls other program units, the time requirements of such called program units are not allowed for in the time requirements shown for the calling program unit. If the called program units include %COUNT statements of their own, these time requirements may be inferred from the appropriate lines of Statistic 3.

7.4.8 Example: Using debug statements

The program INV is contained in the file QUELL.MAT and is compiled using the following statements:

```
/START-PROG $FOR1
 %  BLS0500 PROGRAM 'FOR1', VERSION '2.2A00' OF '91-06-05'LOADED.
 %  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG. 1991 ...
  FOR1:    V2.2A00 READY, GIVE COMPILER OPTION
*COMOPT TESTOPT=(DEBUG),SOURCE=QUELL.MAT,END                          (9)
```

SOURCE LISTING for the source program:

```
 ****  SOURCE  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-26 TIME = 12:36:48   PAGE   1
                                                   PROGRAM UNIT: INV
DO/IF SEG   STMT  I/H LINE        SOURCE-TEXT                                                    COL73-80 RECORD-ID.
----------------------------------------------------------------------------------------------------------------------
      1/1      1         1 |      PROGRAM INV                                                 |
        1      2         2 |      COMMON IR                                                   |
        1      3         3 |      %COUNT ON                                        (1)        |
        1      4         4 |      %CALLTRACE ON (2)                                (2)        |
        1      5         5 |    1 WRITE (2,10)                                                |
        1      6         6 |      %CHECK ON (2) IR                                 (3)        |
        1      7         7 |      READ (1,11) IR                                              |
        1      8         8 |      %CHECK OFF (2)                                              |
        1      9         9 |      IF (IR.GT.3) THEN                                           |
  1     2     10        10 |        GO TO 1                                                   |
  1     2     11        11 |      ELSE IF (IR.EQ.0) THEN                                      |
  1     3     12        12 |        GO TO 100                                                 |
  1     3     13        13 |      ELSE                                                        |
  1     4     14        14 |        CALL DETS ()                                              |
  1     4     15        15 |      END IF                                                      |
        4     16        16 |      %CALLTRACE OFF                                              |
        4     17        17 |      %COUNT OFF                                                  |
        4     18        18 |   10 FORMAT (' ENTER RANK OF MATRIX')                            |
        4     19        19 |   11 FORMAT (I1)                                                 |
        5     20        20 |  100 END                                                         |

 ****  SOURCE  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-26 TIME = 12:36:48   PAGE    1
                                                   PROGRAM UNIT: DETS
DO/IF SEG   STMT  I/H LINE        SOURCE-TEXT                                                    COL73-80 RECORD-ID.
----------------------------------------------------------------------------------------------------------------------
      1/1      1         1 |      SUBROUTINE DETS ()                                          |
        1      2         2 |      COMMON IR                                                   |
        1      3         3 |      REAL A(:,:)                                                 |
        1      4         4 |      DIMENSION B(3,3)                                            |
        1      5         5 |      DIMENSION DET(3,3)                                          |
        1      6         6 |      CALL ALLOC(A,1,IR,1,IR,'ANY')                               |
        1      7         7 |      %COUNT ON                                        (1)        |
        1      8         8 |      WRITE (2,10)                                                |
        6      9         9 |      READ (1,*) ((A(IA,IB),IB=1,IR),IA=1,IR)                     |
        6     10        10 |      %JUMPTRACE ON (2)                                (4)        |
        6     11        11 |      IF (IR.EQ.2) THEN                                           |
  1     7     12        12 |        SDET=A(1,1)*A(2,2)-A(2,1)*A(1,2)                          |
  1     7     13        13 |        %DISPLAY (2) SDET                              (5)        |
  1     7     14        14 |        %CHECK ON (2) B                                (6)        |
  1     7     15        15 |        B(1,1)=A(2,2)/SDET                                        |
  1     7     16        16 |        B(2,2)=A(1,1)/SDET                                        |
  1     7     17        17 |        B(1,2)=(-A(1,2))/SDET                                     |
  1     7     18        18 |        B(2,1)=(-A(2,1))/SDET                                     |
  1     7     19        19 |        %CHECK OFF (2)                                            |
  1     7     20        20 |      ELSE                                                        |
  1     8     21        21 |        DET(1,1)=A(2,2)*A(3,3)-A(3,2)*A(2,3)                      |
  1     8     22          22 |        DET(1,2)=A(2,1)*A(3,3)-A(3,1)*A(2,3)
  1     8     23          23 |        DET(1,3)=A(2,1)*A(3,2)-A(3,1)*A(2,2)
  1     8     24          24 |        DET(2,1)=A(1,2)*A(3,3)-A(3,2)*A(1,3)
  1     8     25          25 |        DET(2,2)=A(1,1)*A(3,3)-A(3,1)*A(1,3)
  1     8     26          26 |        DET(2,3)=A(1,1)*A(3,2)-A(3,1)*A(1,2)
  1     8     27          27 |        DET(3,1)=A(1,2)*A(2,3)-A(2,2)*A(1,3)
  1     8     28          28 |        DET(3,2)=A(1,1)*A(2,3)-A(2,1)*A(1,3)
  1     8     29          29 |        DET(3,3)=A(1,1)*A(2,2)-A(2,1)*A(1,2)
  1     8     30          30 |        SDET=-A(1,1)*DET(1,1)+A(1,2)*DET(1,2)-A(1,3)*DET(1,3)
  1     8     31          31 |        %CHECK ON (2) B                                    (7)
  1     8     32          32 |        DO 6 I=1,IR
  2    11     33          33 |        DO 6 J=1,IR
  3    12     34          34 |   6    B(I,J)=(-1)**(I+J+1)*DET(J,I)/SDET
  1    12     35          35 |        %CHECK OFF (2)
  1    13     36          36 |      END IF
       13     37          37 |      %FULLTRACE ON (2)                                    (8)
       14     38          38 |      CALL DVCHK (IS)
       14     39          39 |      IF (IS.NE.1) THEN
  1    15     40          40 |        WRITE (2,11)
  1    15     41          41 |        %FULLTRACE OFF
  1    15     42          42 |        %JUMPTRACE OFF
  1    15     43          43 |        DO 9 K=1,IR
  2    19     44          44 |   9    WRITE (2,*) (B(K,J),J=1,IR)
  1    19     45          45 |  10    FORMAT (' ENTER MATRIX LINE BY LINE')
  1    19     46          46 |  11    FORMAT (' INVERSE MATRIX:')
  1    19     47          47 |  12    FORMAT (' INVERSE DOES NOT EXIST')
  1    20     48          48 |        RETURN
  1    20     49          49 |      END IF
       21     50          50 |      WRITE (2,12)
       21     51          51 |      %COUNT OFF                                           (1)
       21     52          52 |      CALL DEALLOC(A)
       21     53          53 |      RETURN
       21     54          54 |      END
```

Explanation of example:

(1) The number of executions and computation time required for all statements between %COUNT ON and %COUNT OFF are output to SYSLST.

(2) For the DETS subprogram called, the called program unit, the statement number of the calling location, the entry point and parameters are output on the terminal.

(3) The value change of variable IR is logged at the terminal.

(4) All transfers of control to statements between here and %JUMPTRACE OFF are logged at the terminal.

(5) The value of the variable SDET is output at the terminal.

(6) The value change of B is logged at the terminal.

(7) The value change of B is logged at the terminal.

(8) Up to statement %FULLTRACE OFF, all executed statements are logged at the terminal.

(9) The TESTOPT=(DEBUG) option causes the debug statements in the source program to be compiled.

The following data is output at the terminal during program execution, for example:

```
/START-PROG FROM-FILE=*MODULE(*OMF)
 %  BLS0001 ### DBL VERSION 070 RUNNING
 %  BLS0517 MODULE 'INV' LOADED
 BS2000  F O R 1 : FORTRAN PROGRAM "INV"
 STARTED ON 1991-08-26 AT 15:33:02
 ENTER RANK OF MATRIX
 2
    %CHECK/INV/7 :
   IR = 2 ,
    %CALLTRACE/INV/14 :       CALLED ENTRY : <DETS                DETS"
                              NO PARAMETERS SUBMITTED
 ENTER MATRIX LINE BY LINE
 1 2
 2 1
    %DISPLAY/DETS/13 :
  SDET = -0.30000000E+01 ,
    %CHECK/DETS/15 :
   B = -0.33333331E+00 , 8*0.00000000E+00 ,
    %CHECK/DETS/16 :
   B = -0.33333331E+00 , 3*0.00000000E+00 , -0.33333331E+00 , 4*0.00000000E+00 ,
    %CHECK/DETS/17 :
    %CHECK/DETS/17 :
   B = -0.33333331E+00 , 2*0.00000000E+00 , 0.66666663E+00 , -0.33333331E+00 , 4*0.00000000E+00 ,
    %CHECK/DETS/18 :
   B = -0.33333331E+00 , 0.66666663E+00 , 0.00000000E+00 , 0.66666663E+00 , -0.33333331E+00 , 4*0.00000000E+00 ,
    %JUMPTRACE/DETS/10 :      SEG-NR = 13    ; STMT-NR = 36
    %FULLTRACE/DETS/37 :      SEG-NR = 14    ; STMT-NR = 38
    %CHECK/DETS/38 :
   IS = 2 ,
    %FULLTRACE/DETS/37 :      SEG-NR = 14    ; STMT-NR = 39
    %FULLTRACE/DETS/37 :      SEG-NR = 15    ; STMT-NR = 40
 INVERSE MATRIX:
 -0.33333331E+00 , 0.66666663E+00
 0.66666663E+00 , -0.33333331E+00
 BS2000  F O R 1 : FORTRAN PROGRAM "INV     " ENDED PROPERLY AT 15:33:17
 CPU - TIME USED :       0.0318 SECONDS
 ELAPSED TIME    :      15.2920 SECONDS
```

The DYNAMIC COUNT PROFILE of the program units INV and DETS is output to SYSLST.

```
                                         DYNAMIC COUNT PROFILE OF PROGRAM UNIT DETS
                                         ==========================================
SEG-NR  STMT-NR    COUNT    PROFILE

     1        1        1    *************************
     1        8        1    *************************
     2        9        2    **************************************************
     3        9        2    **************************************************
     4        9        4    ****************************************************************************************************
     5        9        2    **************************************************
     6        9        1    *************************
     7       11        1    *************************
     8       21        0
     9       33        0
    10       34        0
    11       34        0
    12       34        0
    13       36        0
    14       37        1    *************************
    15       39        1    *************************
    16       41        2    **************************************************
    17       44        2    **************************************************
    18       44        4    ****************************************************************************************************
    19       44        2    **************************************************
    20       48        1    *************************
    21       49        0
                                      DYNAMIC TIME  PROFILE OF PROGRAM UNIT DETS
                                      ==========================================
SEG-NR  STMT-NR     TIME    PROFILE

     1        1       83    ***********************************************************************************
     1        8        9    *********
     1        9       24    ************************
     2        9       30    ******************************
     2        9       20    ********************
     3        9       14    **************
     4        9      100    ****************************************************************************************************
     5        9       26    **************************
     6        9        8    ********
     6       10        5    *****
     7       11       48    ************************************************
     7       12       16    ****************
     7       15       27    ***************************
     7       16       26    **************************
     7       17       28    ****************************
     7       18       27    ***************************
     7       19        1    *
     7       20       15    ***************
     8       21        0
     8       22        0
     8       23        0
     8       24        0
     8       25        0
     8       26        0
     8       27        0
     8       28        0
     8       29        0
     8       30        0
     8       32        0
     9       33        0
     9       34        0
    10       34        0
    11       34        0
    11       34        0
    12       34        0
    13       36        0
    14       37       31    *******************************
    14       38       19    *******************
    15       39       30    ******************************
    15       40       11    ***********
    16       41       44    ********************************************
    16       44       16    ****************
    17       44        12    ************
    18       44        88    ****************************************************************************************
    19       44        16    ****************
    19       44        16    ****************
    20       48         8    ********
    21       49         0
                                      DYNAMIC COUNT PROFILE OF PROGRAM UNIT INV
                                      =========================================
SEG-NR  STMT-NR    COUNT   PROFILE
------  -------  --------  -------
     1        1         1    ****************************************************************************************************
     2       10         0
     3       12         0
     4       13         1    ****************************************************************************************************
                                      DYNAMIC TIME  PROFILE OF PROGRAM UNIT INV
                                      =========================================
SEG-NR  STMT-NR    TIME    PROFILE
------  -------  --------  -------
     1        1        35    ****************************************************************************************************
     1        4         9    **************************
     1        7        21    ************************************************************
     1        9         5    **************
     2       10         0
     2       11         0
     3       12         0
     4       13        21    ************************************************************
                                      DYNAMIC TIME  PROFILE OF ALL COUNTED PROGRAM UNITS
                                      ==================================================
PROGRAM UNIT     TIME     PROFILE
------------   --------   -------
       DETS         798    ****************************************************************************************************
        INV          91    ***********
```

7.5 Debug subprograms

In the FORTRAN source program, debug subprograms are called as follows:

```
CALL name (parameter)
```

Debug subprograms are an integral part of the runtime system, requiring merely the call, and no FORTRAN programming.

Debug subprograms are used for

- communication between program units (subprograms SLITE and SLITET),
- program continuation in the event of underflow, overflow, divide errors (subprograms OVERFLOW, FIXOV, DVCHK),
- output of information on the program status (DEBUG subprogram).

7.5.1 Overview: Debug subprograms

Subprogram	**Function**

SLITE and SLITET	Enable the setting, resetting, interrogation of 4 symbolic indicators
	
	`CALL SLITE(i)`
	`i: INTEGER expression, 0 ≤ i ≤ 4`
	
	`CALL SLITET(i,j)`
	`i: INTEGER expression, 1 ≤ i ≤ 4`
	`j: INTEGER variable`

OVERFL	Recognizes the internally set overflow and underflow indicators following arithmetic operations involving floating-point numbers
	
	`CALL OVERFL(j)`
	`j: INTEGER variable`

DVCHK	Recognizes the overflow in a division involving fixed-point and floating-point numbers (division by zero)
	
	`CALL DVCHK (j)`
	`j: INTEGER variable`

FIXOV	Recognizes the fixed-point overflow in arithmetic operations
	
	`CALL FIXOV(j)`
	`j: INTEGER variable`

DEBUG	Provides information about the program status (current statement number, call hierarchy, parameter lists), terminates the program run.
	
	`CALL DEBUG [(v)]`
	`v: CHARACTER item`

Note

For a PDUMP printout, see also the "DUMP" utility in the "Utilities" manual [17].

7.5.2 Subprograms SLITE and SLITET

The subprogram SLITE can be used to set and reset switches.
The subprogram SLITET can be used to interrogate and reset switches.

SLITE and SLITET permit communication between program units within an object run. For this purpose one byte is reserved in the runtime communication area, superseding the usual communication via COMMON areas or by transfer of parameters.

Subprogram calls

```
CALL SLITE(i)
```

i	INTEGER expression; value: $0 \leq i \leq 4$

If i=0, all switches are reset. If i=1,2,3 or 4 then the appropriate switch is set.

```
CALL SLITET(i,j)
```

i	INTEGER expression; value: $1 \leq i \leq 4$
j	INTEGER variable

The switch i (equal to 1,2,3 or 4) is tested and, if appropriate, reset.
Variable j is set equal to 1 if i was set, or equal to 2 if i was not set.

7.5.3 Subprogram OVERFL

The subprogram OVERFLOW

– recognizes the internally set overflow and underflow indicators following arithmetic operations involving floating-point numbers,
– prevents program abortion when an exponent overflow occurs.

If the OVERFL subprogram is not used, an exponent overflow or underflow will cause the program run to be abnormally terminated. Information about the error and the location of the error will be displayed.

On the other hand, if the OVERFL subprogram was linked into the code module (based on a call made in the FORTRAN source program), no abnormal program termination occurs. The result of the operation concerned is assigned the maximum or minimum value, taking into account the sign.

Call:

```
CALL OVERFL (j)
```

j	INTEGER variable

An overflow indicator is internally set if an exponent overflow occurs, i.e. if the result of an arithmetic operation involving floating-point numbers is greater than 16**(+63) (mantissa$\neq$0).
The underflow indicator is set if an exponent underflow occurs, i.e. if the result of an arithmetic operation involving floating-point numbers is less than 16**(-64) (mantissa$\neq$0).

When the subprogram is called, the parameter j is set as follows, depending on the indicators set internally:

j	Overflow indicator	Underflow indicator	Reset indicator
0	Set	Not set	Overflow indicator reset
1	Set	Set	Overflow indicator reset
2	Not set	Not set	——
3	Not set	Set	Overflow indicator reset

After the variable j is set, the OVERFL subprogram resets the internal indicators. If both the overflow and underflow indicator are set, the OVERFL will reset only the internal overflow indicators, so that a subsequent call of OVERFL may examine the underflow indicator.

In the case of REAL and COMPLEX data types with all lengths as well as in the case of INTEGER*8 data type, the OVERFL subprogram recognizes overflows and/or underflows resulting from any arithmetic operations.

In the case of INTEGER*1 and INTEGER*2 data types, the OVERFL subprogram will not recognize any overflow and/or underflow resulting from an arithmetic operation.

In the case of the INTEGER*4 data type, leading places are truncated when a multiplication produces an overflow. The integer variable j has the value 2 after an overflow, since neither the internal overflow nor underflow indicator has been set. In the case of an overflow due to an addition or subtraction, abnormal program termination occurs in the case of the INTEGER*4 data type, and an error message is issued (FIXED POINT OVERFLOW).

Overflow or underflow caused by an arithmetic operation can be monitored by interrogating variable j. If the value of variable j is interrogated in the FORTRAN program, the call CALL OVERFLOW (j) should be immediately behind the FORTRAN statement for which overflow or underflow monitoring is to take place.

Example:

```
           .
           .
           .
      A = 0.72370E+76
      B = 1.E-50
      C = 1.E+30
      J = 5
C
      D = A+0.1E+76
      CALL OVERFL (J)
      IF (J.EQ.0) GO TO 200
C
      E = B/C
      CALL OVERFL (J)
      IF(J.EQ.3) GO TO 300
           .
           .
           .
200   WRITE (20,'(''OVERFLOW : D = '',E20.5,'' J='',I4)')D,J

300   WRITE (20,'(''UNDERFLOW : E = '',E20.5,'' J='',I4)')E,J
```

Display in the event of overflow:

```
OVERFLOW: D = 0.72370E+76 J=0
```

Display in the event of underflow:

```
UNDERFLOW: E = 0.00000E+00 J=3
```

7.5.4 Subprogram DVCHK

The subprogram DVCHK recognizes overflow in a division involving fixed-point and floating-point numbers. Division overflow occurs if the second operand (divisor) has a value of 0.

If the DVCHK subprogram is not used, division overflow causes the program run to be terminated, and messages about the error and the location of the error are output.

However, if the DVCHK subprogram was linked into the load module (owing to its use in the FORTRAN source program), no program abortion takes place. The result of the operation concerned is set to the maximum possible value, except when the first operand is equal to 0. In this case, the result is also equal to zero.

Call:

```
CALL DVCHK (j)
```

j	INTEGER variable of length 4

When subprogram DVCHK is called parameter j is set as follows, depending on the division overflow indicator:

j	Division overflow indicator
1	Set
2	Not set

After the variable j is set, the division overflow indicator is reset.

Overflow caused by division can be monitored by interrogating variable j. If the value of j is interrogated in the FORTRAN program, the call CALL DVCHK (j) should immediately follow the division for which overflow is to be monitored.

7.5.5 Subprogram FIXOV

The FIXOV subprogram recognizes the fixed-point overflow in arithmetic operations, with the exception of multiplications (see note below). Fixed-point overflow will occur if a carry from the highest-ranking bit position occurs with arithmetic fixed-point commands or if valid bits are lost during arithmetic shift-left commands.

If the FIXOV subprogram is not used, the program run is abnormally terminated when a fixed-point overflow occurs. Messages indicating the error and its location are output.

If the FIXOV subprogram was linked into the load module (e.g. by using the FORTRAN source program), the program run is not terminated. The result of the corresponding operation is set to the highest value possible. The operation following the one which caused the fixed-point overflow is processed next.

Call:

```
CALL FIXOV (j)
```

j	INTEGER variable of length 4

When calling the FIXOV subprogram the parameter is set as follows, depending on the overflow indicator for fixed-point overflow:

j	Fixed-point overflow indicator
1	Set
2	Not set

After the variable j has been set, the internal indicator for fixed-point overflow is reset.

Fixed-point overflow caused by an arithmetic operation can be monitored by interrogating variable j. If the value of j is interrogated in the FORTRAN program, the call CALL FIXOV (j) should immediately follow the FORTRAN statement for which fixed-point overflow is to be monitored.

Note

The FIXOV subprogram can recognize a fixed-point overflow only if interrupt weight X’78’ has been set by the processor. This happens when a fixed-point overflow occurs during the execution of the instructions A, AR, LCR, LPR, S, SR, SH, SLA or SLDA.
Therefore, a fixed-point overflow as a result of multiplication (M, MR and MH instructions) cannot be recognized.

7.5.6 Subprogram DEBUG

The DEBUG subprogram provides information on the program status.

Call:

```
CALL DEBUG[(v)]
```

v	CHARACTER item

When this statement is executed, the program run terminates. SYSLST output includes the first 133 characters of "v" as well as information about the program status, in particular the current statement number, call hierarchy and the parameter lists of all currently active (i.e. called and not yet terminated) program units.

This information is also output via the standard error routine in the event of a runtime error.

7.5.7 Example: Use of the debugging subprograms

```
    PROGRAM INV
    COMMON IR
  1 WRITE (2,10)
    READ (1,11) IR
    IF (IR.GT.3) THEN
      GO TO 1
    ELSE IF (IR.EQ.0) THEN
      GO TO 100
    ELSE
      CALL DETS ()
    END IF
    CALL SLITET (1,IV)                                            (1)
    IF (IV.NE.1) GO TO 100
 10 FORMAT (' ENTER RANK OF MATRIX')
 11 FORMAT (I1)
    CALL DEBUG                                                    (2)
100 END
    SUBROUTINE DETS ()
    COMMON IR
    REAL A(3,3)
    DIMENSION B(3,3)
    DIMENSION DET(3,3)
    WRITE (2,10)
    READ (1,*) ((A(IA,IB),IB=1,IR),IA=1,IR)
    IF (IR.EQ.2) THEN
      SDET=A(1,1)*A(2,2)-A(2,1)*A(1,2)
      B(1,1)=A(2,2)/SDET
      B(2,2)=A(1,1)/SDET
      B(1,2)=(-A(1,2))/SDET
```

```
          B(2,1)=(-A(2,1))/SDET
        ELSE
          DET(1,1)=A(2,2)*A(3,3)-A(3,2)*A(2,3)
          DET(1,2)=A(2,1)*A(3,3)-A(3,1)*A(2,3)
          DET(1,3)=A(2,1)*A(3,2)-A(3,1)*A(2,2)
          DET(2,1)=A(1,2)*A(3,3)-A(3,2)*A(1,3)
          DET(2,2)=A(1,1)*A(3,3)-A(3,1)*A(1,3)
          DET(2,3)=A(1,1)*A(3,2)-A(3,1)*A(1,2)
          DET(3,1)=A(1,2)*A(2,3)-A(2,2)*A(1,3)
          DET(3,2)=A(1,1)*A(2,3)-A(2,1)*A(1,3)
          DET(3,3)=A(1,1)*A(2,2)-A(2,1)*A(1,2)
          SDET=-A(1,1)*DET(1,1)+A(1,2)*DET(1,2)-A(1,3)*DET(1,3)
          DO 6 I=1,IR
          DO 6 J=1,IR
  6       B(I,J)=(-1)**(I+J+1)*DET(J,I)/SDET
        END IF
        CALL DVCHK (IS)                                         (3)
        IF (IS.NE.1) THEN
          CALL OVERFL (IT)                                      (4)
          IF (IT.NE.2) GO TO 30
          WRITE (2,11)

          DO 9 K=1,IR
  9       WRITE (2,*) (B(K,J),J=1,IR)
 10       FORMAT (' ENTER MATRIX LINE BY LINE')
 11       FORMAT (' INVERSE MATRIX:')
 12       FORMAT (' INVERSE DOES NOT EXIST')
 13       FORMAT (' OVERFLOW')
          RETURN
        END IF
 20     WRITE (2,12)
        RETURN
 30     WRITE (2,13)
        CALL SLITE (1)                                          (5)
        RETURN
        END
```

Explanation of example:

(1) Switch 1 is tested. If it is not set, the program transfers control to the statement with the branch label 100, otherwise it resumes with the next statement.

(2) The DEBUG subprogram provides information about the program status and terminates the program.

(3) If a division by zero has occurred in the execution of the program, IS receives the value 1 and the statement with the branch label 20 is executed.

(4) If the overflow or underflow indicator is set, the program transfers control to the statement with the branch label 30.

(5) Switch 1 is set.

7.6 Advanced Interactive Debugger (AID)

FOR1 programs can be tested with the Advanced Interactive Debugger AID (as of AID Version 1.0C) (see the manual "AID Debugging of FORTRAN Programs" [3]). Using AID, F0R1 programs executing in the 31-bit address space can also be tested.

7.6.1 Prerequisite for debugging using AID: SYMTEST option

A prerequisite for symbolic debugging with AID is the generation of LSD information (LSD=List for Symbolic Debugging) by means of the compiler. Generation of this information is controlled by the FOR1 user and by means of the SYMTEST option:

```
                          ⎡NO ⎤
[*]COMOPT      SYMTEST = ⎨MAP⎬
                          ⎣ALL⎦
```

NO	No LSD information is generated. AID cannot be used for symbolic debugging of FOR1 programs; only machine-oriented debugging is possible.
MAP	SYMTEST=MAP only permits a restricted range of symbolic debugging functions: Program names can be addressed and call hierarchies traced.
ALL	With SYMTEST=ALL, the compiler generates LSD information. FOR1 programs can be symbolically debugged with AID.

It is advisable to work with COMOPT OPTIMIZE=NO (see chapter 9).
Although it is possible to combine the SYMTEST=ALL specification with with all optimization levels, it is then no longer possible to use the source listing of an optimized program as a clear-cut basis for debugging with AID. Optimization can, for example, change the order of the statements; a statement can be split up into several statements or can be omitted entirely. If a program is to be debugged even with the optimization function activated, a decompiler listing (see 4.7.9) can be of assistance; this can be requested when OPT=3 or 4 is activated. The decompiler listing provides a detailed description of the object code, which is intended to facilitate tracing and setting of test points with AID.

For debugging purposes the user can load the LSD information in the following ways. He can

- load it together with the program (TEST-OPTIONS operand for calling the DBL or TSOSLNK and the ELDE)
- loaded as required, provided that the associated object modules are in a PLAM library (AID command %SYMLIB).

7.6.2 Functional scope of AID

AID is a high-performance debugging system for testing, diagnosing errors and temporarily correcting programming errors in BS2000.

AID supports not only the symbolic debugging of FORTRAN (FOR1) programs but also the symbolic debugging of PL/1 (PLI1) programs, COBOL programs, Assembler programs and C programs, and also the testing on machine code level for all programming languages under BS2000.

When symbolic debugging is carried out on a FORTRAN program, the symbolic names from a FORTRAN source program are used for addressing purposes. Testing on machine code level is available in instances where this is not possible.

In AID commands, the following can be symbolically addressed:

- executable FORTRAN statements identified by the statement number
- executable FORTRAN statements identified by the statement label
- symbolic constants, variables, arrays and array elements
- dynamic arrays
- initiation of procedures.

Symbolic constants, variables, arrays and array elements are addressed using the name defined in the source program. FORTRAN statements are addressed using S'nnnnn', statement labels are addressed as L'nnnnn' (where nnnnn is the statement number or statement label, represented by a maximum of 5 digits).

The following AID commands are available:

- Commands for execution monitoring
  - specific types of statements in the source program (%CONTROLn)
  - selected events in the execution of the program (%ON)
  - declared program addresses (%INSERT)

  The user can define that AID is to interrupt program execution at specific addresses or upon execution of selected statement types or when defined events occur, and then execute subcommands. A subcommand is an individual command or string of AID and BS2000 commands. It is defined as the operand of an AID command.

- Commands for tracing and logging (%TRACE) and for skipping statements (%JUMP)

  %TRACE can be used to select the program area for tracing as well as the number of type of statements which are logged.

  %JUMP can be used within a program unit to define a statement to which a GOTO is made after completion of the command. Using %JUMP, the execution sequence can be changed, for example by replacing illegal statements with an AID command string and then defining resumption of the program at a specific statement, using %JUMP. By means of %RESUME or %TRACE, execution of the program is continued starting with the specified statement. The %JUMP command is supported only when OPTIMIZE=NO is specified.

- Commands for output and updating of memory contents

  – output of the values of symbolic constants, variables, array elements and arrays (%DISPLAY),
  – updating of the values of variables, array elements and arrays (%SET),
  – output of call hierarchies (%SDUMP %NEST),
  – to decompile the memory contents in Assembler (%DISASSEMBLE).

  With the %NEST operand of the %SDUMP command, the user can display the level at which the call hierarchy of the program was interrupted and which modules are CALL nested. In the call hierarchy, the names of FORTRAN subprograms and the name of the main program are output.

- Commands for management of AID input files (%DUMPFILE, %SYMLIB) and AID output media (%OUTFILE)

  AID can be used to edit a running program or diagnose a memory dump in a disk file. Within a debug session the user can alternate between these two options, e.g. in order to compare data inventories between the current program and a memory dump.

- Commands for definition of output data sets (%OUT) and global declarations (%BASE, %AID).

- The use of AID is supported by means of the %HELP function

  – in conjunction with all AID commands and operands
  – to explain the AID messages and possible responses to AID messages.

7.6.3 Example: Use of Advanced Interactive Debugger AID

```
  ****  SOURCE  LISTING  ****    SIEMENS-NIXDORF FORTRAN COMPILER    FOR1 V2.2A00 DATE = ...
                                                  PROGRAM UNIT: INV
 DO/IF SEG   STMT  I/H LINE      SOURCE-TEXT
--------------------------------------------------------------------------------------------
      1/1      1          1 |     PROGRAM INV
        1      2          2 |     COMMON IR
        1      3          3 |   1 WRITE (2,10)
        1      4          4 |     READ (1,11) IR
        1      5          5 |     IF (IR.GT.3) THEN
   1    2      6          6 |       GO TO 1
   1    2      7          7 |     ELSE IF (IR.EQ.0) THEN
   1    3      8          8 |       GO TO 100
   1    3      9          9 |     ELSE
   1    4     10         10 |       CALL DETS ()
   1    4     11         11 |     END IF
        4     12         12 |  10 FORMAT (' ENTER RANK OF MATRIX')
        4     13         13 |  11 FORMAT (I1)
        5     14         14 |100 END
  ****  SOURCE  LISTING  ****    SIEMENS-NIXDORF FORTRAN COMPILER    FOR1 V2.2A00 DATE = ...
                                                   PROGRAM UNIT: DETS
 DO/IF SEG   STMT  I/H LINE      SOURCE-TEXT
--------------------------------------------------------------------------------------------
      1/1      1          1 |     SUBROUTINE DETS ()
        1      2          2 |     COMMON IR
        1      3          3 |     REAL A (3,3)
        1      4          4 |     DIMENSION B(3,3)
        1      5          5 |     DIMENSION DET(3,3)
        1      6          6 |     WRITE (2,10)
        5      7          7 |     READ (1,*) ((A(IA,IB),IB=1,IR),IA=1,IR)
        5      8          8 |     IF (IR.EQ.2) THEN
   1    6      9          9 |       SDET=A(1,1)*A(2,2)-A(2,1)*A(1,2)
   1    6     10         10 |       B(1,1)=A(2,2)/SDET
   1    6     11         11 |       B(2,2)=A(1,1)/SDET
   1    6     12         12 |       B(1,2)=(-A(1,2))/SDET
   1    6     13         13 |       B(2,1)=(-A(2,1))/SDET
   1    6     14         14 |     ELSE
   1    7     15         15 |       DET(1,1)=A(2,2)*A(3,3)-A(3,2)*A(2,3)
   1    7     16         16 |       DET(1,2)=A(2,1)*A(3,3)-A(3,1)*A(2,3)
   1    7     17         17 |       DET(1,3)=A(2,1)*A(3,2)-A(3,1)*A(2,2)
   1    7     18         18 |       DET(2,1)=A(1,2)*A(3,3)-A(3,2)*A(1,3)
   1    7     19         19 |       DET(2,2)=A(1,1)*A(3,3)-A(3,1)*A(1,3)
   1    7     20         20 |       DET(2,3)=A(1,1)*A(3,2)-A(3,1)*A(1,2)
   1    7     21         21 |       DET(3,1)=A(1,2)*A(2,3)-A(2,2)*A(1,3)
   1    7     22         22 |       DET(3,2)=A(1,1)*A(2,3)-A(2,1)*A(1,3)
   1    7     23         23 |       DET(3,3)=A(1,1)*A(2,2)-A(2,1)*A(1,2)
   1    7     24         24 |       SDET=-A(1,1)*DET(1,1)+A(1,2)*DET(1,2)-A(1,3)*DET(1,3)
   1    8     25         25 |       DO 6 I=1,IR
   2    8     26         26 |       DO 6 J=1,IR
   3   10     27         27 |   6   B(I,J)=(-1)**(I+J+1)*DET(J,I)/SDET
   1   10     28         28 |     END IF
   1   10                29 |
       11     29         30 |     CALL DVCHK (IS)
       11     30         31 |     IF (IS.NE.2) THEN
   1   12     31         32 |       WRITE (2,11)
   1   13     32         33 |       DO 9 K=1,IR
   2   16     33         34 |   9   WRITE (2,*) (B(K,J),J=1,IR)
   1   16     34         35 |  10   FORMAT (' ENTER MATRIX LINE BY LINE')
   1   16     35         36 |  11   FORMAT (' INVERSE MATRIX:')
   1   16     36         37 |  12   FORMAT (' INVERSE DOES NOT EXIST')
   1   17     37         38 |       RETURN
   1   17     38         39 |     END IF
*      18     39         40 |  20 WRITE (2,12)
***** WARNING 1) (SA047) *******
       18     40         41 |     RETURN
       18     41         42 |     END
```

1) Except for a WARNING message (UNREFERENCED LABEL), no compilation errors are reported when the INV program is compiled.

```
/START-PROG FROM-FILE=*MODULE(LIBRARY=PLAM.LIB,ELEMENT=INV)                        (2)
%  BLS0001 ### DBL VERSION 070 RUNNING
%  BLS0517 MODULE 'INV' LOADED
 BS2000  F O R 1 : FORTRAN PROGRAM "INV"
 STARTED ON 1991-08-25 AT 10:59:06
 ENTER RANK OF MATRIX
*2
 ENTER MATRIX LINE BY LINE
*1 2
*2 1
INVERSE DOES NOT EXIST
BS2000  F O R 1 : FORTRAN PROGRAM "INV     " ENDED PROPERLY AT 10:59:22
CPU - TIME USED :       0.0480 SECONDS
ELAPSED TIME    :      17.2550 SECONDS
```

(2) The program executes without an execution error being reported. The result ("INVERSE DOES NOT EXIST"), however, is not correct, which suggests that a logical error may have occurred.

```
/START-PROG $FOR1                                                                  (3)
*COMOPT SOURCE=SRC.INV,MODULE-LIBRARY=PLAM.LIB
*COMOPT SYMTEST=ALL
*END
```

(3) INV is recompiled using the "SYMTEST=ALL" and "MODULE-LIBRARY=PLAM.LIB" options, thus writing the object module, together with the LSD information, to a PLAM library.

```
/LOAD-PROG FROM-FILE=*MODULE(LIBRARY=PLAM.LIB,ELEMENT=INV)                         (4)
%  BLS0001 ### DBL VERSION 070 RUNNING
%  BLS0517 MODULE 'INV' LOADED
/%INSERT PROG=DETS.S'13' <%SDUMP;%STOP>                                            (5)
I378 SYMBOLIC INFORMATION MISSING                                                  (6)
/%SYMLIB PLAM.LIB                                                                  (7)
```

(4) The program is loaded with the dynamic binder loader.

(5) After loading with the AID command %INSERT, a test point is defined for the statement with the number 13 in the program unit DETS. Prior to execution of this command, subcommand string "<%SDUMP;%STOP>" is to be executed. "%SDUMP" causes the values of all data items in the current call hierarchy to be output according to their declared data type. "%STOP" halts the program after "%SDUMP" is executed so that further AID commands can be entered, for example.

(6) AID rejects the command given in (5) and issues an error message. When the DBL was called, "SYMTEST=ALL" was not specified so that no LSD information was loaded.

(7) The PLAM library in which the LSD information is contained is is by the AID command %SYMLIB. AID reloads this information once a symbolic name is addressed, one for which no LSD information has been loaded.

```
/%INSERT PROG=DETS.S'13' <%SDUMP;%STOP>                                        (8)
/%TRACE                                                                        (9)
 BS2000  F O R 1 : FORTRAN PROGRAM "INV"
 STARTED ON 1991-08-24 AT 12:03:29
      3  1                    START    , I-O-ACCESS
 ENTER RANK OF MATRIX
      4                       I-O-ACCESS
*2
      5                       IF
      7                       THEN/ELSE
     10                       THEN/ELSE, CALL
 ENTER MATRIX LINE BY LINE
*1 2
*2 1
** ITN: #'00000041' *** TSN: 3113 ********************************************  (10)
SRC_REF:    13  SOURCE: DETS      PROC: DETS  ********************************
IR              =          2

A( 1: 3, 1: 3)
( 1, 1) +.1000000 E+01  ( 2, 1) +.2000000 E+01  ( 3, 1) +.0000000 E+00
( 1, 2) +.2000000 E+01  ( 2, 2) +.1000000 E+01  ( 3, 2) +.0000000 E+00
( 1, 3) +.0000000 E+00  ( 2, 3) +.0000000 E+00  ( 3, 3) +.0000000 E+00


B( 1: 3, 1: 3)
( 1, 1) -.3333333 E+00  ( 2, 1) +.0000000 E+00  ( 3, 1) +.0000000 E+00
( 1, 2) +.6666666 E+00  ( 2, 2) -.3333333 E+00  ( 3, 2) +.0000000 E+00
( 1, 3) +.0000000 E+00  ( 2, 3) +.0000000 E+00  ( 3, 3) +.0000000 E+00


DET( 1: 3, 1: 3)
( 1, 1) +.0000000 E+00  ( 2, 1) +.0000000 E+00  ( 3, 1) +.0000000 E+00
( 1, 2) +.0000000 E+00  ( 2, 2) +.0000000 E+00  ( 3, 2) +.0000000 E+00
( 1, 3) +.0000000 E+00  ( 2, 3) +.0000000 E+00  ( 3, 3) +.0000000 E+00


IA              =          3

IB              =          3

SDET            = -.3000000 E+01

I               =          0

J               =          0

IS              =          0

K               =          0

STOPPED AT SRC_REF: 13 , SOURCE: DETS , PROC: DETS                             (11)
```

(8) The %INSERT command is accepted.

(9) %TRACE starts the program and causes execution, with output of all the FORTRAN statements which have been executed during the program run. Output includes statement numbers, any existing statement label and type of statement. As the default, 10 executable statements at a time are handled and logged.

(10) Prior to execution of the statement with the number 13, the subcommand %SDUMP of the AID command given under (8) is executed; the values of all data items at this point in the program are output according to their data type. B(2,1) still has the value 0, since statement 13 has not yet been executed.

(11) As the result of the %STOP subcommand in (8), the program is in the STOP state. After the STOP message is displayed, new commands can be entered.

```
/%INSERT S'28' <%DISPLAY B(2,1);%STOP>                                        (12)
/%TRACE                                                                       (13)
 SRC_REF:    29 SOURCE: DETS      PROC: DETS  *********************************
 B( 2, 1)        = +.6666666 E+00                                             (14)

 STOPPED AT SRC_REF: 29 , SOURCE: DETS , PROC: DETS
```

(12) Prior to execution of statement 28, the value of the array element B(2,1) is output. The qualification "PROG=DETS." in the %INSERT command (8) is now no longer necessary, since DETS is the current program unit.

(13) %TRACE causes the program run to continue, tracing the execution.

(14) The value of array element B(2,1) has likewise been computed correctly.

```
/%TRACE
    29                      CALL
    30                      IF                                                (15)
    39  20                  I-O-ACCESS, LABEL                                 (16)
 INVERSE EXISTIERT NICHT
    40                      END
 BS2000  F O R 1 : FORTRAN PROGRAM "INV     " ENDED PROPERLY AT 16:56:51
 CPU - TIME USED :       0.7277 SECONDS
 ELAPSED TIME    :     141.7410 SECONDS
```

(15) %TRACE continues execution of the program from the interrupt point.

(16) Following IF statement 30 an erroneous branch is made to statement 39 and the message "INVERSE DOES NOT EXIST" displayed. The correct IF statement should be "IF (IS.NE.1)", since the debugging subprogram has the value 1 in the event of division overflow,

```
/LOAD-PROG FROM-FILE=*MODULE(LIBRARY=PLAM.LIB,ELEMENT=INV)                  (17)
%  BLS0001 ### DBL VERSION 070 RUNNING
%  BLS0517 MODULE 'INV' LOADED
/%INSERT PROG=DETS.S'30' <%SDUMP %NEST;%DISPLAY IS;%STOP>                   (18)
/%RESUME                                                                    (19)
 BS2000  F O R 1 : FORTRAN PROGRAM "INV"
 STARTED ON 1991-08-24 AT 16:00:22
 ENTER RANK OF MATRIX
*2
 ENTER MATRIX LINE BY LINE
*1 2
*2 1
 ** ITN: #'000000C7' *** TSN: 3704 *******************************************
 SRC_REF:    30 SOURCE: DETS      PROC: DETS  *********************************
 SRC_REF:    30 SOURCE: INV       PROC: INV   **********************************
 IS              =           2                                              (20)

 STOPPED AT SRC_REF: 30 , SOURCE: DETS , PROC: DETS
/%AID CHECK=ALL                                                             (21)
/%SET 1 INTO IS;%RESUME                                                     (22)
 OLD CONTENT:                                                               (23)
           2
 NEW CONTENT:
           1
%  IDA0129 CHANGE? (Y=YES;N=NO)?
Y
 INVERSE MATRIX:                                                            (24)
 -0.33333331E+00 , 0.66666663E+00
 0.66666663E+00 , -0.33333331E+00
 BS2000  F O R 1 : FORTRAN PROGRAM "INV     " ENDED PROPERLY AT 16:15:04
 CPU - TIME USED :       0.3266 SECONDS
 ELAPSED TIME    :     862.3140 SECONDS
```

(17) INV is loaded again.

(18) A STOP point is set for statement 30. The subcommand %SDUMP %NEST causes output of the current call hierarchy.

(19) %RESUME starts the program.

(20) Execution of the %DISPLAY command.

(21) The %SET command is used to change the memory contents of the data item. If the %AID CHECK=ALL was specified beforehand, AID conducts an updating dialog prior to execution of a %SET command.

(22) The %SET command is used to change the value of data item IS to 1. After this, execution of the program continues with %RESUME.

(23) AID conducts an updating dialog before the %SET command is executed.

(24) After the value of IS is changed, the program executes correctly.

8 File processing

Using the input/output language elements defined in FORTRAN, data may be output to external files and read from them. To do so, the FORTRAN input/output concepts must be related to those of the Data Management System (DMS).

The I/O statements are discussed in the "FOR1" reference manual [21]. The various DMS functions and the commands required for file processing are described in the manuals "DMS Introductory Guide and Command Interface" [18] , "DMS Assembler Interface" [19] and "User Commands (SDF Format)" [12].

Sections 8.1 and 8.2 deal with certain important attributes and processing options of BS2000 files.
Section 8.3 describes how BS2000 files are linked with FOR1 programs. Section 8.4 finally explains the relationships between FORTRAN data records and DMS data records.

8.1 BS2000 system files

BS2000 employs logical system files for input of commands and data or for output of data or operating system messages (see "User Commands (SDF Format)" manual [12]).

A distinction is made between the logical input files of the operating system which use the standard file names

SYSCMD, SYSDTA, SYSIPT

and the logical output files with the standard file names

SYSOUT, SYSLST, SYSOPT.

System files are set up automatically for each task and need not be defined separately by the user.

System files usually already include an assignment to specific input or output devices, or files, namely the primary assignment. The user can change this assignment by issuing commands, in particular assigning cataloged files to the system files (by means of the commands ASSIGN-SYSDTA, ASSIGN-SYSIPT, ASSIGN-SYSOUT, ASSIGN-SYSLST, ASSIGN-SYSOPT).

The following table illustrates the predefined system file assignments:

System file	Primary assignments	
	Interactive mode	Batch mode
SYSCMD	Terminal	SPOOLIN or ENTER file
SYSDTA		
SYSIPT	No primary assignment	
SYSOUT	Terminal	SPOOLOUT file (EAM file), which is to be output to printer at the end of the task and then deleted.
SYSLST SYSLSTnn	SPOOLOUT file (EAM file) which is to be output to printer at the end of the task and then deleted.	
SYSOPT	SPOOLOUT file (EAM) file which is to be output to floppy disk or card punch at the end of the task and then deleted.	

Table 8-1: Primary assignments of system files

By default, FOR1 reads source programs, change lines and compiler options via system file SYSDTA and outputs the listings which have been created to system file SYSLST.

In FORTRAN, certain input/output units are assigned to system files SYSDTA, SYSOUT, SYSLST, SYSIPT and SYSOPT through presetting (see section 8.3.3.1).

8.2 BS2000 user files

In contrast to system files, which are created for each task without the need for any previous declaration, the user can himself define the characteristics of user files. The following file attributes can be defined:

- file name (see for example "DMS Introductory Guide and Command Interface" [18])
- file link name (8.3.1)
- access method (8.2.1)
- record format (8.2.2)
- record length (8.2.2)

Commands for defining the file attributes

The attributes of a file can be defined using the CREATE-FILE, MODIFY-FILE-ATTRIBUTES and SET-FILE-LINK commands.

The CREATE-FILE command creates an entry in the user catalog and assigns storage space to the file. The file attributes given in the CREATE-FILE command are incorporated into the catalog. The other attributes, e.g. access method or record length, are not entered in the catalog until the file is closed following its initial opening.

The MODIFY-FILE-ATTRIBUTES command can be used to modify the catalog attributes of files already contained in the catalog. Information about the catalog attributes of files can be requested by means of the SHOW-FILE-ATTRIBUTES command.

The SET-FILE-LINK command is used to define a file link name; the file attributes specified in the command are entered in the TFT (Task File Table) (see section 8.3.1). The SET-FILE-LINK command will only generate a new catalog entry if no catalog entry for the file name specified in the FILE-NAME operand yet exists.

Permanent and temporary files

User files can be either permanent or temporary files.
Permanent files are files created by the user using the SAM, ISAM, UPAM or BTAM access method and which are retained even after the task in which they were created and cataloged is terminated.

Temporary files are files created by the user using the SAM, ISAM, UPAM or BTAM access method and which are tied to the task creating them. They are automatically deleted as part of LOGOFF. Temporary files use a different nomenclature than permanent files (through the prefix "#" or "@"). The term "temporary file" refers only to files which satisfy the above definition and not to other task-related file types such as system files or EAM files. In contrast to temporary files, EAM files and system files are not kept in the user catalog and are not subject to pubspace checks.

8.2.1 Access methods of the DMS

For a detailed description of the access methods see the manuals "DMS Introductory Guide and Command Interface" [18] and "DMS Assembler Interface" [19]. Only a brief overview is given below.

To access files, FOR1 programs make use of the logical access methods of DMS. The DMS logical access method transfers data between a file and the address space of an active program. The file organization and the type of access to records is defined by the access method.

The FOR1 runtime system supports the following access methods:

- the common access methods ISAM, SAM and BTAM
- special access method EAM for job-oriented files.

The access method can be defined by the ACCESS-METHOD operand of the SET-FILE-LINK command (see section 8.3.1). ACCESS-METHOD=BY-PROGRAM is the default. A FOR1 program opens files using the ISAM access method as standard.

ISAM	(Indexed Sequential Access Method) The Indexed Sequential Access method makes it possible to process records sequentially and non-sequentially. Here records can be inserted on the basis of their logical sequence in the file. In the case of non-sequential processing of records, each record has a key. This key is in the same position for each record of a file. As standard, the key has a length of 8 bytes and precedes the data portion of the record.
SAM	(Sequential Access Method) The sequential access method makes it possible to process records sequentially starting at a defined point. Updating and rewriting of records to a SAM file is not supported by FOR1.
BTAM	(Basic Tape Access Method) The Basic Tape Access method makes it possible to store the records of a sequentially organized tape file and retrieve blocks of a sequentially organized tape file.

EAM (Evanescent Access Method)
With the Evanescent Access Method, work files which are only available for the duration of the program run can be accessed. FOR1 creates EAM files if the STATUS='SCRATCH' operand has been specified in the OPEN statement.

For all access methods, the file attributes are defined when a file is generated, i.e. opened as an output file. The file attributes are transferred to the catalog entry and apply for all subsequent declarations.

8.2.2 Record format and record length

DMS distinguishes between three types of *record formats*:

- record format FIXED, fixed length
- record format VARIABLE, variable length
- record format UNDEFINED, undefined length.

A file can only consist of records with the same record format. With record format FIXED, all records of a file have identical lengths. With record formats VARIABLE and UNDEFINED, the records may have different lengths.

Table 8-2 shows the permissible record formats and device types for the various access methods.

Access method	Record formats			Device types	Files which use other access methods also permissible for input
	F	V	U		
SAM	X	X	X	Disk, tape	
ISAM	X	X		Disk	
BTAM	X	X 1)	X	Tape	SAM
EAM	X			Disk	

1) Treated like record format U

Table 8-2: Access methods, record formats and device types

Defining the record format

The record format is defined in the SET-FILE-LINK command with the RECORD-FORMAT operand. BY-PROGRAM is set by default. As standard, FOR1 programs create files with variable record format.

Defining the record length

The *length of a record* (RECSIZE) can be defined with the RECORD-SIZE operand in the SET-FILE-LINK command. The specification in the RECORD-SIZE operand refers to the length of the FORTRAN data record plus any existing administrative information (see section 8.4).

Record length field

In the case of record format VARIABLE the information on the length of the individual record is indicated by means of an additional information field at the beginning of the record, namely the record length field (RLF). The record length field has a length of 4 bytes and the length of the record is given in the higher-value bytes (including the record length field). The two remaining bytes contain a pointer to the next record. In the case of record format UNDEFINED, the specification of the length is made via a register.

8.2.3 Data block and buffer

The access methods of DMS transfer data between the file and the address space of the job.
Transfer of data is not record by record. DMS combines records to form data blocks and transfers these blocks of data between peripheral storage and main memory. A *data block* or *logical block* is understood to be the unit which DMS transfers between peripheral storage and main memory when a file is accessed. The portion of main memory which accepts a data block or whose contents are transferred to peripheral storage is known as a *buffer*. The buffer is part of the address space of the program which initiated the I/O operation.

The size of a logical block is described in terms of PAM pages. One *PAM page* (also known as a standard block, halfpage or, as opposed to the logical block, physical block) is a unit of storage encompassing 2048 bytes and which may contain a PAM key (depending on the file format).

The size of the logical block and thus that of the buffer is defined by the BUFFER-LENGTH operand of the SET-FILE-LINK command. A logical block can have the following sizes:

- in the case of files on disk, the standard block size (2048 bytes) or an integral multiple of a standard block size (up to 16 PAM pages)
- in the case of files on tape, the nonstandard block size with a length of up to 32767 bytes.

ISAM and EAM work with standard blocks. SAM and BTAM can process both standard and nonstandard blocks (in the case of magnetic tape files). In the case of disk files, SAM works with standard blocks.

The usable data area of a logical block is the result of the size of the logical block (defined by the BUFFER-LENGTH operand) minus any existing administrative information. The size of the administrative information varies, depending on the access type and the file format (cf. table 8-3 and table 8-4).
When defining the logical block the FOR1 user must note that the usable area for data in a data block

– must be at least as long as the longest record of the file
– must wherever possible be an integral multiple of the record length, in order to achieve optimal utilization of the data area.

Example: Optimum block size (file format NK-SAM, variable record length)

File definition:

```
/SET-FILE-LINK LINK-NAME=DSET12, FILE-NAME=DAT, ACCESS-METHOD=SAM,
RECORD-FORMAT=FIXED(RECORD-SIZE=1500)
```

This combination of BUFFER-LENGTH and RECORD-SIZE would be inadvisable:
The default block length is 2048 bytes, of which 2044 bytes are usable for DMS records (cf. table 8-4). Each block can accommodate only one record; 544 bytes are lost in each case.
A better block size would be three PAM pages: BUFFER-LENGTH=STD(SIZE=3). With this block size (6144 bytes, 6140 usable), each block can accommodate 4 records; only 140 bytes are lost.

8.2.4 Keyed and no-key file formats

As of BS2000 Version 10, the no-key disk periphery is supported. As of this version a distinction is made between two file formats:

- the previous keyed format (also called K format in the following)
- the newly introduced no-key format (also called NK format in the following).

The no-key file format for ISAM files is already supported by operating system V9.5.

On keyed disks it is possible to work both with files in K format and with files in NK format.
On no-key disks, however, it is possible to work only with files in NK format.

The file format is controlled by the BLOCK-CONTROL-INFO operand of the SET-FILE-LINK command:

BLOCK-CONTROL-INFO=PAMKEY (file processing in keyed format) BLOCK-CONTROL-INFO=WITHIN-DATA-BLOCK (file processing in no-key format)

The administrative information contained in a separate PAM key for the K format is relocated to the data area in the case of the NK format. Relocation of PAM key information to the data area reduces the area available for the records, which results in **modification of the optimum and maximum record lengths** for the no-key format, a consideration which the FOR1 user must take into account.

Keyed and no-key ISAM files

If ISAM files in K format which utilize the maximum record length are to be converted to NK format with the same block size, DMS will create overflow blocks.

The formation of overflow blocks is accompanied by the following problems:

- the overflow blocks increase the disk space required and thus increase the number of I/O operations required for file processing
- the ISAM key may in no case be located within an overflow block.

Overflow blocks can be avoided if care is taken to ensure that the longest record in the file does not exceed the length of the usable area of a logical block in the case of NK-ISAM files.

The following table illustrates how it is possible in the case of ISAM files to calculate how much space is available per logical block for DMS records.

File format	**Record format**	**Usable area**
K-ISAM	VARIABLE	Block size
	FIXED	Block size - (r*4) r = Number of records per logical block
NK-ISAM	VARIABLE	Block size - (n*16) - 12 - (r*2) (rounded down to next number divisible by 4) n = Blocking factor r = Number of records per logical block
	FIXED	Block size - (n*16) - 12 - (r*2) - (r*4) (rounded down to next number divisible by 4) n = Blocking factor r = Number of records per logical block

Table 8-3: Usable block area with ISAM files

Explanation of formulae:
With RECORD-FORMAT=FIXED, although a 4-byte record length field is present for each record in the case of both K-ISAM and NK-ISAM files, it is not included in the block size. In these cases therefore 4 bytes must be deducted per record. With NK-ISAM files, each PAM page of a logical block contains 16 bytes of administrative information. The logical block additionally contains a further 12 bytes of administrative information and a 2-byte record pointer for each record.

Example: Maximum record length of a NK-ISAM file (fixed-length records)

File definition:

```
/SET-FILE-LINK LINK-NAME=DSETnn, FILE-NAME=file, RECORD-FORMAT=FIXED,
 BUFFER-LENGTH=STD(SIZE=2), BLOCK-CONTROL-INFO=WITHIN-DATA-BLOCK
```

Maximum record length (as per formula in table 8-3):
4096 - (2*16) - 12 - 1*2 - 1*4 = 4046, rounded down to the next number divisible by four: 4044 (bytes).

Keyed and no-key SAM files

With SAM files there are no overflow blocks. Therefore SAM files in K format which utilize the maximum record length cannot be converted into NK-SAM files. FOR1 programs which work with such maximum record lengths for K-SAM files are no longer executable with NK-SAM files.

The following table illustrates how much space is available per logical block with SAM files for DMS records.

File format	**Record format**	**Usable area**
K-SAM	VARIABLE	Block size - 4
	FIXED / UNDEFINED	Block size
NK-SAM	VARIABLE / FIXED UNDEFINED	Block size - 16

Table 8-4: Usable area with SAM files

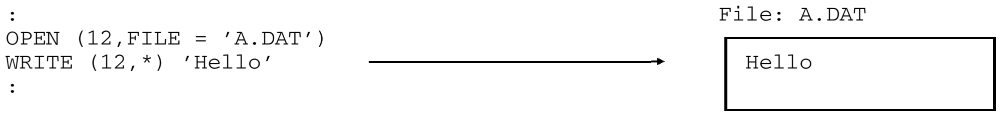
The deduction of 4 bytes for K-SAM files with variable-length records results from the fact that the logical blocks of such files contain a block length field having this length that is not included in BLKSIZE.

8.3 Linkage of BS2000 files and FOR1 programs

If a FOR1 program is to process files, then a link must be established between files and program. The FORTRAN input/output units form the basis for this. The link can be established in different ways:

- directly, without file link names
  FOR1 programs can access BS2000 files directly via the FORTRAN input/output units without the need to define file link names. By specifying the file name in the FILE operand of the FORTRAN statement OPEN it is possible to define the relevant user file which is to be accessed.

  Example:

  ```
  :
  OPEN (12,FILE = 'A.DAT')
  WRITE (12,*) 'Hello'
  :
  ```

  However, since the OPEN statement has no parameters for determining file format, record format, record length, block size and OPEN mode, these file attributes cannot be explicitly defined in the case of direct linkage. It is therefore advisable to always reference input/output files via file link names (see below).

  BS2000 system files are already defaulted to certain input/output units. System files are always addressed directly, i.e. it is not possible to use file link names here. The OPEN and CLOSE statements are not required.

- through file link names
  The file link names DSET00, DSET01,..., DSET99 are available, corresponding to the FORTRAN file numbers. File link names are defined by means of the BS2000 SET-FILE-LINK command (see following section).

8.3.1 Defining file link names: BS2000 command SET-FILE-LINK

By using the SET-FILE-LINK command it is possible to link BS2000 user files with FOR1 programs via a file link name. This "indirect" link has the advantage of being variable: Without changing the source code, it is possible to define for each program run which files are to be assigned to the program. The SET-FILE-LINK command can also be used to define the attributes for the files to be linked. A detailed description of the SET-FILE-LINK command may be found in the "User Commands (SDF Format)" manual [12].

FOR1-specific file link names

File link names which link files to FOR1 programs each relate to a particular input/output unit. They are formed as follows:

```
LINK-NAME = DSETnn
```

where *nn* is the number of an input/output unit. The file link names DSET00, DSET01, DSET02, ..., DSET99 are therefore available for linking files to FOR1 programs.

Example:

```
/SET-FILE-LINK LINK-NAME=DSET12, FILE-NAME=B.DAT, ACCESS-METHOD=SAM
```

TFT entry

DMS creates an entry in the Task File Table (TFT) under the file link name specified in the LINK-NAME operand of the SET-FILE-LINK command. The TFT is a temporary table which is set up automatically for each task. All the file attributes specified in the SET-FILE-LINK command are included in the TFT entry. If an entry already exists in the TFT for the specified file link name, this will be overwritten. The SHOW-FILE-LINK command can be used to obtain information about the contents of the TFT. TFT entries can be deleted using the REMOVE-FILE-LINK command, or modified using the CHANGE-FILE-LINK command.

Access to files via the TFT entry

If a FOR1 program attempts to open a file via a particular input/output unit, DMS checks whether an entry exists in the TFT under the corresponding file link name. If such an entry is found, all subsequent input/output statements which address this unit are related to the linked file - as long as the corresponding TFT entry exists.

Assignment through file link names takes precedence over direct assignment, i.e. if a corresponding TFT entry exists, the assignment will then also be made through the file link name if the name of a different file is specified in the FILE operand of the OPEN statement.

Example:

```
:                                  TFT:
:                                 LINK=DSET12              File B.DAT
OPEN (12, FILE='A.DAT')           FILE=B.DAT              ┌─────────────────┐
WRITE (12,*) 'Hello' ─────────────────────────────────►   │ Hello           │
                                                          └─────────────────┘
```

Updating the TFT during program execution

A TFT entry can be modified, deleted or overwritten during program execution (using the CHANGE-FILE-LINK, REMOVE-FILE-LINK or SET-FILE-LINK commands). The relevant file must be closed during command input.

There are two ways of updating the TFT during program execution:

– Interrupting execution with the FORTRAN statement PAUSE, issuing the desired command on the operating system level and continuing program execution with the R[ESUME] command.

  Example:

```
 CLOSE (UNIT=unit1,...)
 PAUSE [expression]
/SET-FILE-LINK LINK-NAME=DSETunit2, FILE-NAME=file,...
/R
 OPEN (UNIT=unit2,...)
```

– Issuing the desired command with interrupting the program by means of the FPOOL function FCMD (see section 12.2.5).

  Example:

```
COMOPT FPOOL = FOR1.FPOOL
            .
            .
      COMMAND='/SET-FILE-LINK LINK-NAME=DSETunit2, FILE-NAME=file, ...'
      CLOSE (UNIT=unit1,...)
      CALL FCMD(RETCODE, COMMAND, RESPONSE, 'Y', SCRATCH)
      OPEN (UNIT=unit2,...)
```

8.3.2 Definition of file attributes by DMS

On opening a file, DMS obtains information about file attributes from three different sources:

- the catalog entry (if present)
- the TFT entry (if present)
- explicit and implicit declarations in the program

With regard to the above, values from the TFT entry, i.e. specifications made by a SET-FILE-LINK command, take priority over declarations in the program. Only file attributes that are defined neither by the program nor by the TFT entry or that were specified with BY-CATALOG in the SET-FILE-LINK command are taken from the catalog entry

If the different sources contain contradictory information, this may give rise to incompatibility:

An execution error will occur for example when the information concerning the form of file organization (FCBTYPE) differs in the catalog and in the TFT, or if it is not consistent with the declarations made in the program. Should say the ACCESS operand of the OPEN statement be specified as ’DIRECT’ in the program, then neither the catalog entry nor the TFT entry may contain an FCBTYPE other than ISAM. (The FCBTYPE contained in the TFT entry is defined by the ACCESS-METHOD operand of the SET-FILE-LINK command.)

8.3.3 FORTRAN input/output units

FORTRAN input/output statements make reference to files through logical input/output units.

The logical input/output units for external files are the file numbers 0 - 99. File numbers can be assigned in different ways:

- by default (for BS2000 system files)
- by the OPEN statement (for user files)
- by implicit OPEN (if there is neither a default nor an OPEN statement for the corresponding file number)

In addition to the external files, there are also internal files in FORTRAN (see "FOR1" reference manual [21]). Internal files are FORTRAN data items of the type CHARACTER. Input/output units for internal files are not file numbers but the names of the data items. Internal files do not need to be spearately assigned; accordingly there is no OPEN statement for them.

8.3.3.1 Standard assignment of BS2000 system files

The system files SYSDTA, SYSOUT, SYSLST, SYSOPT and SYSIPT are linked to specific input/output units by means of default values. In the FORTRAN program, merely the relevant file number need be specified in the input/output statements.

Table 8-5 shows the assignment of system files to input/output inputs:

File number	File name	Macro used	Record format	Length of the data portion	Length of the DMS record
1	SYSDTA	RDATA	V	2024	2028
2	SYSOUT	WROUT	V	2024	2028
5	SYSDTA	RDATA	V	2024	2028
6	SYSLST	WRLST	V	133	137
7	SYSOPT	WRTOT	F	80	80
8	SYSIPT	RDCRD	F	80	80
97	SYSDTA	RDATA	V	2024	2028
98	SYSOPT	WRTOT	F	80	80
99	SYSLST	WRLST	V	133	137

Table 8-5: Standard assignment of BS2000 system files

Read-only access is allowed for input/output units 1, 5, 8 and 97, write-only access is allowed for input/output units 2, 6, 7, 98 and 99.

The default assignments can be changed by the user with the aid of the SUBSTITUTE, ADD and DELETE runtime options (see 6.3.2).

8.3.3.2 OPEN statement

The FORTRAN statement OPEN can be used to

- link a cataloged file to an input/output unit
- catalog a file which does not yet exist and link it to an input/output unit
- define or update the FORTRAN access type and further attributes.

The FORTRAN statement DEFINE FILE has the same effect as a restricted OPEN statement for files with direct access.

The OPEN statement is described in detail in the "FOR1" reference manual [21]. The following sections describe the format of the OPEN statement and explain the RECL, STATUS and FILE operands.

`OPEN`	`([ACCESS=caccess] [,ASSOVAR={iassovar|cassovar}] [,BLANK=cblank] [,ERR=ierr] [,FILE=cfile] [,FORM=cform] [,IOSTAT=iostat] [,MAXREC={imaxrec|cmaxrec}] [,RECL=irecl] [,STATUS=cstatus] [,UNIT=] iunit )`

The first letter of the individual operands indicates the type.

i	INTEGER
c	CHARACTER

RECL

The RECL operand defines the length of a FORTRAN record in bytes. The RECL=irecl operand is only interpreted when ACCESS='DIRECT,...' is specified. Input/output of a longer record results in an error.

If the RECL operand is omitted, the following applies:

- if an OPEN or an implicit OPEN was performed on this file during the course of the program run, RECL retains the value defined therein;
- in all other cases RECL is provided with the value entered for RECORD-SIZE or the default value minus the length of the corresponding administrative information (see tables 8-6 and 8-7 in section 8.4.2).

Note

In ISAM file records, the key at the beginning of the record is not part of the data portion. However if the key is not at the beginning of the record, it is part of the data portion of that record and must be included in the count (see section 8.4.2).

STATUS

The STATUS operand specifies the existence of a file or its use as a job-related work file (FCBTYPE=EAM). The criterion for entry in the STATUS operand is not the physical existence of the file, but whether or not it has a corresponding entry in the system catalog. For example, a catalog entry may be generated by the CREATE-FILE command, although the file does not physically exist until an OPEN or CLOSE is executed.

The following values may be specified for the STATUS operand:

$$\text{STATUS} = \left\{\begin{array}{l}\text{'OLD'}\\ \text{'NEW'}\\ \text{'}\underline{\text{UNKNOWN}}\text{'}\\ \text{'SCRATCH'}\\ \text{cstatus}\end{array}\right\}$$

'OLD'	A catalog entry for the referenced file already exists. If 'OLD' is specified but there is no entry in the system catalog, abnormal termination takes place.
'NEW'	No entry exists in the system catalog, an ISAM file is created which has the specified file name. If 'NEW' is specified and an entry already exists in the system catalog under the specified name, an execution error will occur.
'SCRATCH'	Creates a task-related work file (EAM file), which is deleted when the program ends. SET-FILE-LINK commands specified for this file are not interpreted.
<u>'UNKNOWN'</u>	If the FILE operand has been specified and the file specified therein already exists, or a SET-FILE-LINK command is in force for the specified input/output unit, the STATUS operand has the value 'OLD'. If no FILE operand is specified and if no SET-FILE-LINK command is in force for the specified input/output unit, the STATUS operand has the value 'NEW' and a file with a standard file name (see FILE operand) is created. After the OPEN statement is executed, the STATUS operand is set to 'OLD'.
cstatus	CHARACTER expression, which must have the value 'OLD', 'NEW', 'SCRATCH' or 'UNKNOWN' at OPEN time.

If STATUS='OLD' or 'NEW' is specified, the FILE operand must also be specified.

FILE

The FILE operand is used to specify the name of the file. If no file with the specified file name exists, a file with this name is created, provided that STATUS='OLD' has not been specified. If no name has been specified for a file (FILE operand omitted, FILE='' or FILE='   '), and neither a SET-FILE-LINK command nor a preceding OPEN statement with FILE='name' has been given for the specified input/output unit, a file with the following standard file name is created:

```
UNIT.FOR1.prog.unit [.tsn[.time]]
```

prog	Name of the program unit
unit	File number
tsn	Task sequence number, four-digit number
time	Current time the file was created, in the form hhmmss.

The two latter qualifications are omitted unless they are need for the uniqueness of the system catalog entry.

If neither an OPEN nor DEFINE FILE statement is used for a given file, that file will be opened implicitly when the first input/output statement is used. The values of the implicit OPEN depend on the operand values of the first input/output statement (see "FOR1" reference manual [21]).

8.4 Mapping of FORTRAN Records to DMS

8.4.1 FORTRAN record and DMS record

A **FORTRAN record** is understood to include a record defined by FORTRAN language elements. A FORTRAN record corresponds to the data portion of one or more DMS records. The length of a FORTRAN record can be specified, if ACCESS='DIRECT', in the RECL operand of the OPEN statement or in the DEFINE FILE statement (as positional specification).

A **DMS record** is a record which DMS uses for its operations. In addition to the FORTRAN record, this DMS record can also contain administrative information, such as:

- the record length field in the case of variable format records
- the record key in the case of ISAM files
- the Green Control Word (GCW) in the case of unformatted input/output.

The length of a DMS record, or, for variable-length records the maximum record length, is specified by the RECORD-SIZE operand of the SET-FILE-LINK command.

The relationship between the FORTRAN record and the DMS record is important for the FOR1 user

- in order to control the RECORD-SIZE operand of the SET-FILE-LINK command, based on the length of the FORTRAN record
- in order to select the most favorable size of the logical block, based on the length of the DMS record.

Record length field

In the case of record format VARIABLE, the record length field with a length of 4 bytes is at the beginning of the record (see 8.2.2).

Record key

As a standard procedure, the key precedes the data portion of the record and has a length of 8 bytes.

If the key is at the beginning of the record, FORTRAN does not count it as part of the data portion of the record and it is not transferred during the I/O operation. However if the key does not start at the beginning of a record, it counts as the data portion of a record and is transferred during the input/output operation. The specification in the RECL operand in this case includes the key length. A record key contained within the data must be taken into account during the input/output: either a dummy variable must be provided in the input/output list for the record key, or it is skipped by using the tabulator format key X in the format specification. If a record key contained within the data is not taken into account, then output data will be overwritten by the key during the output, for example.

In the case of sequential output to an ISAM file, the first record is given key value 1 and the consecutive key value is incremented by 1 each time a record is written.

Green Control Word

An unformatted record can be of any length. DMS can only process records with a maximum length of 32 Kbytes. To permit mapping of FORTRAN records of any length to DMS, a Green Control Word is required. The Green Control Word (GCW) contains administrative information on the structure of the data records for unformatted input/output. The GCW has a length of 4 bytes and immediately precedes the data portion of a subrecord.

If a FORTRAN record is distributed over more than one DMS record in the case of unformatted input/output, these records are combined to form packages. A package may comprise up to 255 DMS records. Since the length of an unformatted FORTRAN record is unlimited, any number of packages can be formed per FORTRAN record.

Format of the Green Control Word:

```
| Indica-  | Indica-  | Number of          |                        |
| tor for  | tor for  | data bytes         |      FORTRAN data      |
| record   | package  | in the subrecord   |                        |
------------------------------------------------------------------------

←-byte 0——→←-byte 1-→←-byte 2——→←-byte 3-→
```

Byte 0

0	Subrecord is not last record in package
Number of subrecords in the package	Subrecord is last record in package.

Byte 1

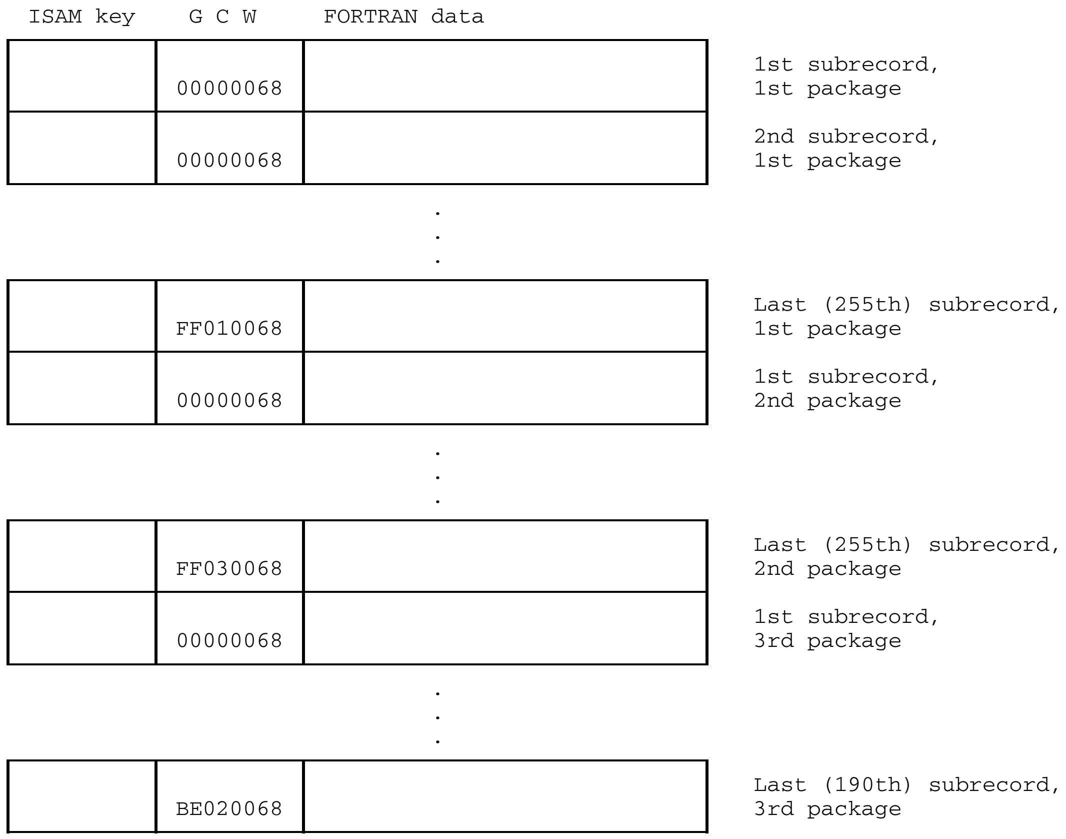
0	Subrecord is not last record in package (byte 0 = 0) or the FORTRAN record consists of only one package
1	Subrecord is last record in first package
2	Subrecord is last record in last package
3	Subrecord is last record and package is not first and not last package

Byte 2, byte 3

Bytes 2 and 3 contain the number of bytes in the subrecord.

Example:

ISAM file with RECORD-SIZE=120. A FORTRAN record with 700 subrecords is written.


ISAM key
G C W
FORTRAN data
00000068
1st subrecord, 1st package
00000068
2nd subrecord, 1st package
FF010068
Last (255th) subrecord, 1st package
00000068
1st subrecord, 2nd package
FF030068
Last (255th) subrecord, 2nd package
00000068
1st subrecord, 3rd package
BE020068
Last (190th) subrecord, 3rd package

8.4.2 Summary: Relationship between DMS and FORTRAN records

Tables 8-6 and 8-7 show how the length of a DMS record is derived from the length of the FORTRAN data plus any existing administrative information, depending on access type, record format and input/output format. The administrative information can consist of the following components:

KEY	ISAM key, default 8 bytes
RLF	Record length field, 4 bytes
GCW	Green Control Word, 4 bytes
DAT	Designates the length of the FORTRAN data record. When ACCESS='DIRECT' is specified in the OPEN statement, DAT corresponds to the entry in the RECL operand.

For access type ISAM, the representation of the record format only applies for KEY-POSITION=1 with RECORD-FORMAT=FIXED or KEY-POSITION=5 with RECORD-FORMAT=VARIABLE.

DMS records for formatted input/output			
ACCESS TYPE	RECORD FORMAT FIXED	VARIABLE	UNDEFINED
S A M	DAT	RLF + DAT	DAT
I S A M	KEY + DAT	RLF + KEY + DAT	-
B T A M	DAT	RLF + DAT	DAT
E A M	2048 bytes DAT DAT ≤ 2048 bytes	-	-
SYSTEM FILES	FOR IPT AND LST: DAT	FOR DTA,OUT AND LST: RLF DAT	-

Table 8-6: Format and length of the DMS record for formatted input/output

DMS records for unformatted input/output			
ACCESS TYPE	RECORD FORMAT FIXED	VARIABLE	UNDEFINED
S A M	GCW + DAT	RLF + GCW + DAT	GCW + DAT
I S A M	KEY + GCW + DAT	RLF + KEY + GCW + DAT	-
B T A M	GCW + DAT	RLF + GCW + DAT	GCW + DAT
E A M	2048 bytes GCW + DAT GCW+DAT≤2048 bytes	-	-
SYSTEM FILES	FOR IPT AND LST: GCW + DAT	FOR DTA,OUT AND LST: RLF + GCW + DAT	-

Table 8-7: Format and length of the DMS record for unformatted input/output

8.4.3 Examples: FORTRAN/DMS record

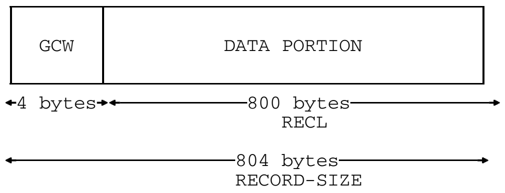
Example 1: FORTRAN/DMS record for SAM file

```
/CREATE-FILE FILE-NAME=DAT, SUPPORT=PUBLIC-DISC(SPACE=RELATIVE(PRIMARY-
 ALLOCATION=4))
/SET-FILE-LINK LINK-NAME=DSET17, FILE-NAME=DAT, ACCESS-METHOD=SAM,
 RECORD-FORMAT=FIXED(RECORD-SIZE=804), BUFFER-LENGTH=STD(SIZE=2)
```

Input/output statement for this file:

```
.
.
.
INTEGER*8 FELD(100)
OPEN(UNIT=17,FORM='UNFORMATTED')
WRITE(17)FELD
```

A record in this file has the following format:



RECORD-SIZE=804 is specified in the SET-FILE-LINK command, since the record length for unformatted input/output comprises the data portion of 800 bytes and the 4 bytes of the Green Control Word.

Example 2: FORTRAN/DMS record for ISAM file

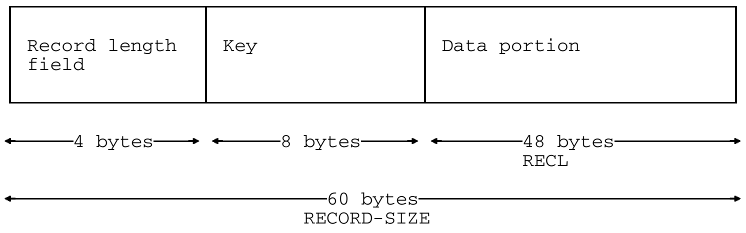
```
/SET-FILE-LINK LINK-NAME=DSET20, FILE-NAME=DAT1, ACCESS-METHOD=ISAM,
 RECORD-FORMAT=VARIABLE(RECORD-SIZE=60)
```

Input/output statements for this file may have the following format:

```
      Sequential access                      Direct access

      OPEN(UNIT=20,ACCESS='SEQUENTIAL',      OPEN(UNIT=20,ACCESS='DIRECT',
     *     STATUS='OLD',FILE='DAT1')        *     RECL=48,STATUS='OLD',
                                            *     FILE='DAT1')
      WRITE(20,100)I,J,K,L                   READ (20,REC=16,FMT=100)I,J,K,L
100   FORMAT (4I12)                    100   FORMAT (4I12)
               .                                 .
               .                                 .
               .                                 .
```

Since RECORD-FORMAT=VARIABLE, each record has a record length field in addition to the data portion and key. In this example a record has the following format:



Example 3: FORTRAN/DMS record for ISAM file with key in data portion

```
/SET-FILE-LINK LINK-NAME=DSET21, ACCESS-METHOD=ISAM(KEY-LENGTH=4,
 KEY-POSITION=21), RECORD-FORMAT=VARIABLE(RECORD-SIZE=30)
```

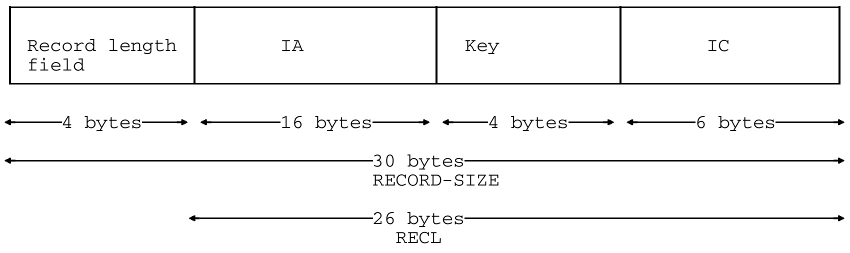
Input/output statements for this file may have the following format:

```
      Sequential access                       Direct access

      OPEN (UNIT=21,ACCESS='SEQUENTIAL',      OPEN (UNIT=21,RECL=26,ACCESS=
     *      STATUS='OLD',                    *     'DIRECT',STATUS='OLD',
     *      FILE='DAT2')                     *     FILE='DAT2')
      READ(21,100)IA,IB,IC                    WRITE(21,REC=18,FMT=100)IA,IB,IC
100   FORMAT(I16,A4,I6)                 100   FORMAT(I16,I4,I6)

      After execution of the                  The value specified by IB is
      READ statement                          overwritten by the binary
      variable IB contains                    representation of key
      the value of the key.                   value 18.
```

A record has the following format:



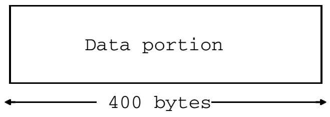
Example 4: FORTRAN/DMS record for BTAM file

```
/CREATE-FILE FILE-NAME=DAT2, SUPPORT=TAPE(VOLUME=volume, DEVICE-TYPE=
 device-type)
/SET-FILE-LINK LINK-NAME=DSET50, FILE-NAME=DAT, ACCESS-METHOD=BTAM,
 RECORD-FORMAT=UNDEFINED, BUFFER-LENGTH=400
```

Input/output statements for this file may have the following format:

```
    INTEGER*8 A(50)
    OPEN (UNIT = 50)
    WRITE (50,100)A
100 FORMAT (50 I8)
```

A record in this file has the following format:



9 Optimization

The optimization of the FOR1 compiler system is tailored to speeding up the execution of object programs. The term "optimization" as it is used below covers all measures that, by means of additional analyzes and additional transformations, are instrumental in effecting shorter object program execution.

The FOR1 programmer basically has two optimization options at his disposal:

- manual optimization, i.e. shortening the runtime thanks to a more efficient style of programming (section 9.1);
- steps taken by FOR1 which are controlled by means of the SDF operand OPTIMIZATION in the START-FOR1-COMPILER command (see section 9.2.1) or the OPTIMIZE compiler option (see section 9.2.2).

Optimization primarily affects program segments which are to be executed frequently (loops) and slow program sections (input/output).

The following optimization measures are frequently undertaken by the FOR1 compiler:

- computation of constant arithmetical expressions at compile time
- optimization of logical expressions
- reuse of common subexpressions
- optimization of subscript computation
- loop optimization
- elimination of superfluous code
- register optimization.

Optimization can also be carried out for calling subprograms and functions controlled using the PROCEDURE-OPTIMIZATION compiler option (see section 9.2.3).

The user can select one of the optimization levels 0 through 4 or NO. The program area via which optimization takes place is different, depending on which level was selected. In the case of optimization levels 1 and 2, optimization is via basic blocks and DO loops. A basic block is a string of commands with as few branches as possible and having exactly one entry point and one exit point. In the case of optimization levels 3 and 4, however, optimization is by means of complete loops of any type.

The optimization levels of FOR1 are described in detail in section 9.2.

9.1 Manual optimization

Manual optimization of programs is an efficient means of running a program in less time, irrespective of compilation. Some of the optimization measures which the FOR1 optimization performs may also be included by the user when writing the program, such as the reuse of previously calculated (sub)expressions or the extraction of loop-invariant parts from the range of a loop. These optimization functions are not described at this point. The optimization measures described below supplement the FOR1 optimization.

Branches

For logical IF statements and BLOCK-IF statements, the optimum object code is generated if the test expression is a LOGICAL*1 variable. For arithmetic IF statements, an INTEGER*2 variable will be the most efficient choice.

Subprograms

If efficiency is the sole objective, FUNCTION or SUBROUTINE subprograms may be unsuitable for frequently executed but small tasks if the effort of call and return is relatively high with respect to the actual task. There are two alternatives to this:

- Code implemented as an %INCLUDE item; that is, in-line programming of the respective statements rather than calling a subprogram.
- Use of a statement function (if possible).

Transfer of parameters via COMMON blocks is more efficient than through parameter lists; however it cannot be covered by the debug option ARG.

Arithmetic expressions

For small integer exponents, A*A*A . . . is preferable to A**I since iterative multiplication is much faster than exponentiation. For a similar reason, A+A+A . . . is faster than A*N for small N’s. By continuing the computation with intermediate results, execution may be speeded up even further.

For example, the exponentiation `X=A**8` can be split up into the following statements:

```
X1=A*A
X2=X1*X1
X =X2*X2
```

Converting to multiplication is also performed by FOR1 optimization.

Since multiplication takes less time to execute than division, division should be replaced by multiplication whenever possible.

Example:

`A/(B*C*D)` instead of `(((A/B)/C)/D)`
`SQRT(X)` is more efficient than `X**.5`

Conversions

Conversions between different types of data may take more time than the actual operation and therefore should be avoided as far as possible. This may be achieved by reordering operands in arithmetic expressions.

Example:

Statement sequence: better:

```
      INTEGER I1,I2,I3
      REAL R1,R2,R3
      A=I1+R1+I2-R2-I3+R3                A=(I1+I2-I3)+(R1-R2+R3)
```

DO loops

Successive DO loops with the same structure should be combined.

Example:

```
      DO 1 I=1,10
1     A(I)=B(I)+C(I)
      DO 2 I=1,10
2     AA(I)=F(I)
```

This statement sequence can be transformed as follows:

```
      DO 1 I=1,10
      A(I)=B(I)+C(I)
1     AA(I)=F(I)
```

This loop is faster than the original two loops since the loop control effort is required only once.

Multi-dimensional arrays

When using multi-dimensional arrays, overlaying with one-dimensional arrays may be used in certain cases to speed up execution.

Example:

```
     REAL A (10,50,50)

     DO 1 I=1,50
        .
        .
        .
     DO 1 J=1,50
     DO 1 K=1,10,2
1    A(K,J,I)=A(K,J,I)+B
     X=A(L,M,N)
```

This statement sequence can be transformed as follows:

```
     REAL A(10,50,50),AA(25000)
     EQUIVALENCE(A,AA)
        .
        .
        .
     DO 1 I=1,25000,2
1    AA(I)=AA(I)+B
     X=A(L,M,N)
```

Input/output

Unformatted input/output is the fastest kind of input/output. Intermediate results which will be reread only by the same program should therefore always be output unformatted. In addition, unformatted data usually takes less space on external storage.

The number of executions of input/output statements should be kept as low as possible.

The following program part, for example, is inefficient since the WRITE statement is executed ten times:

```
    REAL A(10),B(10)
    DO 1 I=1,10
1   WRITE(21) A(I),B(I)
```

The following statement sequence is better since here the WRITE statement is executed only once:

```
    REAL A(10),B(10)
    WRITE(21) (A(I),B(I),I=1,10)
```

Better still are the following statement sequences since here the elements of A and B are processed contiguously rather than individually:

```
REAL A(10),B(10)
WRITE(21) (A(I),I=1,10),(B(I),I=1,10)
```

or

```
REAL A(10),B(10)
WRITE(21) A,B
```

or

```
REAL A(10),B(10)
WRITE(21) A(1)...A(10),B(1)...B(10)
```

Programming for virtual memory

Virtual memory with its large address space provides, in many cases, a capability for holding relatively large files in memory instead of in external files. This may save a large number of executions of input/output statements and simplifies programming, for access to data is either direct or associative (variable names) rather than via input/output statements, and no program parts are required for file management or error handling.

There are, however, further aspects to the virtual memory concept which are important to program execution time and system throughput. In the BS2000 operating system, each task has its own address space of a few megabytes (varying with system generation). This address space is divided into "pages" of 4096 bytes each. The system ensures that the pages being accessed by the current task are indeed available in real memory of the mainframe, whereas those pages that are not being used are, as far as possible, kept away from real memory or stored externally on a paging device. The following schematic diagram illustrates the concept:

This figure is not any longer available for the online pdf.

Fig. 9-1: Schematic diagram of virtual storage

Pages A1, A3 or B3 are stored externally.

If a task now wants to access a page in its address space that does not exist in real memory, the task is interrupted and the system must read in the page from the paging device into real memory. If there is no space available in real memory, some other page must be removed from real memory. Thus, access to a page not currently present in real memory involves interrupts and, albeit preferential and relatively fast, input/output operations.

In order to save run time, the program should be designed in such a way that the required pages are kept in real memory to the greatest possible extent. The basic rules are:

- Avoid any unnecessary reference to other pages by keeping, as far as possible, to the memory areas currently in use.
- Statements that follow one another in execution, or data which is successively accessed should be arranged in adjacent object program locations.

There are various ways of allowing for that in the FORTRAN program:

With algorithms, "long-distance" jumps over large areas should be avoided wherever possible.

Parts of algorithms which are executed only seldom, e.g. error handling routines, should be taken out of the ordinary context of processing and placed separately.

Parts of algorithms which are executed regularly should be written "in-line" rather than by subprogram calls.

In subprograms, access to global data items (transfer in COMMON blocks or by addresses) may result in paging so that it is advisable to use a transfer of values.

When processing large volumes of data, the physical arrangement of the data in memory should be taken into consideration.

The following program part, for example, is inefficient since FOR1 arrays are stored in columnar fashion:

```
      REAL*4 A (1000,100)
        .
        .
        .
      DO 1 I = 1,1000
      DO 1 J = 1,100
1     A(I,J) = A(I,J) + I*J
```

With a page size of 4096 bytes, the element referenced by statement 1 will be in a different page for almost every iteration of the inner DO loop, in all probability creating a very high paging frequency. In the worst case, almost 100000 paging operations take place.

The following arrangement of DO loops is more efficient:

```
      DO 1 J = 1,100
      DO 1 I = 1,1000
1     A(I,J) = A(I,J) + I*J
```

The elements of A are referenced in the order in which they are arranged in the virtual address space, creating a paging frequency of about 100 in the worst case, an improvement which may be in the order of many minutes.

For nested DO loops it is generally better to keep the memory area referenced by the inner loops very small, since frequent execution of the inner loops would imply a high paging frequency.

At link-edit time, explicit entries of the INCLUDE control statement may be used to arrange those program units which follow one another in execution in adjacent locations. In doing this it is generally advisable to sort the most frequently used program units towards the "middle" rather than at the beginning.

Compile time improvements

The order of the specification statements in the FORTRAN program can influence the time a program needs for compiling. If there are is a large number of specification statements, significant savings can be achieved.

It is advantageous to arrange all specification statements of the same type in adjacent locations. The following statement types should each be grouped together:

IMPLICIT statements
INTEGER, REAL, LOGICAL statements
DIMENSION, CHARACTER statements
PARAMETER statements
COMMON statements
EQUIVALENCE statements
DATA statements

The order of the individual groups does not affect the time needed for compilation.

PARAMETER statements that refer to other symbolic constants should follow the PARAMETER statements for these symbolic constants.

The following is inefficient:

```
PARAMETER(A=B+5)
PARAMETER(B=7)
```

Better would be:

```
PARAMETER(B=7)
PARAMETER(A=B+5)
```

DATA statements which use an implicit DO loop should be placed at the end of the DATA statement group.

9.2 Controlling optimization

FOR1 provides suitable optimization levels for each of the various phases of program development.

The main purpose in the debugging phase is to locate errors in a program rapidly and easily. Optimization level NO is appropriate in this case. At this optimization level, the program is converted into machine code directly statement by statement. The compiler thus requires only a small amount of time for the compilation. In addition, the resulting object can be optimally examined using the symbolic debugging aid AID.

When the program has been fully tested and is to be used, it should run as quickly as possible and also occupy a minimum of storage space. It is therefore the purpose of high-level optimization, which can be activated with optimization level 3, to generate an object code that could not be produced significantly more efficiently even by an Assembler programmer. To do so, extensive analysis must be performed on the program; this then becomes apparent through a noticeable increase in compilation time.

A compromise between optimization levels NO and 3 is offered by the default optimization level 1. Here only limited analysis is performed and, accordingly, fewer optimization measures are carried out. A slightly increased compilation time provides some enhancement in object runtime.

With the aforementioned optimization levels NO, 1 and 3, the only transformations performed on the program are those which do not alter its external behavior. The objects generated with these optimization levels always yield exactly the same results as non-optimized objects.

The object runtime may possibly be reduced still further than with optimization level 3 through the use of level 4 and the optimization parameters REORDER, PARAMETER-SIDEEFFECT and FUNCTION-SIDEEFFECT. However, since a degradation in object runtime or differing results may occur here in rare instances, the user should employ these more extensive optimization measures only if he knows their effect and is in a position to judge whether they are suitable for his program. (The same applies to optimization level 2.) The FOR1 optimization measures are thus described in detail in section 9.3.

9.2.1 SDF operand OPTIMIZATION

```
START-FOR1-COMPILER
```

```
,OPTIMIZATION = NO / LOW / MEDIUM(...) / HIGH(...)

   MEDIUM(...)
     |
     |  CONDITIONAL-LOOPS = IGNORED / RISK-OPTIMIZED
     |
     | ,OPTIMIZE-PROCEDURES = NO / YES / SPECIAL

   HIGH(...)
     |
     |  CONDITIONAL-LOOPS = IGNORED / RISK-OPTIMIZED
     |
     | ,OPTIMIZE-PROCEDURES = NO / YES / SPECIAL
     |
     | ,OPTIMIZATION-HINTS = STD / PARAMETER(...)
     |
     |    PARAMETER(...)
     |      |  REORDER-EXPRESSIONS = YES / NO
     |      | ,FUNCTION-SIDEEFFECTS = YES / NO
     |      | ,ARGUMENT-SIDEEFFECTS = NO / YES
```

The SDF operands and corresponding compiler options are shown in table 2-11.

9.2.2 OPTIMIZE compiler option

The COMOPT OPTIMIZE compiler option controls optimization. Optimization level NO, 1, 2, 3 or 4 may be selected. The default value is optimization level 1.

The optimization level selected with OPTIMIZE influences the defaults for the SAVE-CONSTANT (see section 4.1.2.7) and PROCEDURE-OPTIMIZATION (see section 9.2.3) compiler options:

with OPTIMIZE=NO, 0, 1, 2:
- SAVE-CONSTANT=YES is the default
- with PROCEDURE-OPTIMIZATION, the default PROCEDURE-OPTIMIZATION=STD is interpreted as PROCEDURE-OPTIMIZATION=NO

with OPTIMIZE=3, 4:
- SAVE-CONSTANT=NO is the default
- with PROCEDURE-OPTIMIZATION, the default PROCEDURE-OPTIMIZATION=STD is interpreted as PROCEDURE-OPTIMIZATION=YES (unless LINKAGE=STD is specified *explicitly*)

PROCEDURE-OPTIMIZATION=YES is incompatible with LINKAGE=STD (see 4.2.2.6). Therefore when OPTIMIZE=3,4 is entered, the default LINKAGE=STD is converted to LINKGAGE=FOR1-SPECIFIC and the following warning is issued:

```
MA43 LINKAGE=FOR1-SPECIFIC EXPECTED
```

There are nevertheless two ways of generating ILCS modules, in spite of OPTIMIZE=3,4:
- LINKAGE=STD is specified *explicitly*. PROCEDURE-OPTIMIZATION=STD will then be interpreted as PROCEDURE-OPTIMIZATION=NO in spite of OPTIMIZE=3,4, a fact to which the following warning draws attention:
  ```
  MA42 PROC-OPT=NO BECAUSE LINK=STD
  ```
- Specifying PROCEDURE-OPTIMIZATION=NO *before* entering OPTIMIZE=3,4

Optimization causes changes to and rearrangement of the code. Values of variables may under certain circumstances no longer be locatable with reference to the source program, or statement execution may no longer be traceable when a program is debugged using the interactive debugging aid AID. It is therefore advisable to deactivate optimization by means of OPTIMIZE=NO during the program debugging phase.

If a program is to be debugged using an interactive debugging aid, although it has been optimized and can therefore no longer be debugged under defined conditions, a decompiler listing (see section 4.7.9) can be of assistance. A decompiler listing can be requested if optimization level 3 or 4 has been used. The decompiler listing shows the changes optimization has caused, as compared to the original source program. As a result, tracing and the setting of test points on a source program level are facilitated.

`[*]COMOPT`	`OPT[IMIZE]=NO`

deactivates all optimization measures. OPTIMIZE=NO is advisable when debugging with AID, in order to prevent the code from being altered. Where OPT=NO, there are no variables in the register at the statement bounds. As a result, variables can be updated by means of AID and still remain unique. The AID statement $JUMP (skip statements) is only supported when OPT=NO is specified.

`[*]COMOPT`	`OPT[IMIZE]=0`

deactivates most optimization measures. The following are optimized

- logical expressions,
- arithmetic expressions containing operands which are all constant,
- register usage.

The range for these optimizations encompasses the entire program unit. The compilation phase "global optimization" (see appendix 2) is not activated when OPTIMIZE=0 is specified.

`[*]COMOPT`	`OPT[IMIZE]=1`

All optimizations activated by means of OPTIMIZE=0 are performed.
In addition, the following are optimized:

- the parts of arithmetic expressions consisting of constant operands
- common arithmetic expressions
- subscript computations
- basic blocks of explicit DO loops, which are executed exactly once per loop cycle. Loop-invariant computations are relocated to precede the loop. Multiplications of an INTEGER loop variable with a pseudo-constant quantity (see section 9.3.5) are replaced by additions.
- exponentiations with a power of 2 or 3 are replaced by multiplications.

`[*]COMOPT`	`OPT[IMIZE]=2`

All optimizations activated by OPTIMIZE=1 are also additionally performed for conditionally-executed sections of loops.

```
                          ⎧3                           ⎫
[*]COMOPT      OPT[IMIZE]=⎨                            ⎬
                          ⎩(3,parameter[,...])         ⎭
```

```
                ⎧                                ⎧YES⎫⎫
                ⎪FUNC[TION-SIDEEFFECT]=          ⎨   ⎬⎪
                ⎪                                ⎩NO ⎭⎪
                ⎪                                ⎧YES⎫⎪
parameter:=     ⎨PARAM[ETER-SIDEEFFECT]=         ⎨   ⎬⎬
                ⎪                                ⎩NO ⎭⎪
                ⎪                                ⎧YES⎫⎪
                ⎪REORDER                =        ⎨   ⎬⎪
                ⎩                                ⎩NO ⎭⎭
```

(Defaults underlined: FUNCTION-SIDEEFFECT=YES, PARAMETER-SIDEEFFECT=NO, REORDER=NO)

All optimizations activated by OPTIMIZE=1 are performed, yet a wider area is covered. If *parameter* is not specified, the defaults FUNCTION-SIDEEFFECT=YES, PARAMETER-SIDEEFFECT=NO and REORDER=NO are applicable. The compile time can be significantly reduced by changing these defaults.

When OPTIMIZE=3 is set, the following optimizations are additionally performed:

- In addition to explicit DO loops, implicit loops are optimized like loops formed by means of IF statements, GOTO statements or ERR and END parameters in input/output statements. Only loops which have no more than one entry point are optimized, and only those parts of the loops which are executed exactly once each time the loop is executed.
- "Superfluous" code, i.e. statements whose results are no longer required, is eliminated. Superfluous code is usually a consequence of optimization measures. As a result of loop optimization, for example, incrementation of a loop variable may be unnecessary. The loop variable is set to its final value or eliminated entirely.
- Exponentiations with constant integer exponents are resolved into a string of multiplications. Up to an exponent of 120, this breakdown is more expedient than calling a runtime routine. With large exponents, rounding discrepancies may occur as a result of this optimization.
- In the case of unformatted input/output statements, access to adjacent array elements is superseded by access to array areas. To make this type of optimization possible, access to the array elements must follow the same order the elements have in memory (see "FOR1" reference manual [21]).

  Example:

  `READ(X) ((A(I,J),I=1,100),J=1,100)` is converted to
  `READ(X) A(1,1): A(100,100)`

If the array elements are not arranged adjacently and in ascending order, a warning is issued: `UNFAVOURABLE INCREMENTATION PREVENTS ARRAY OPTIMIZATION.`
If the order of the array elements is rearranged, access optimization can then be performed.

– Isolated constant operands in arithmetic expressions are converted to the resulting data type at compile time. Special cases, e.g. multiplying by 1 or 0, or addition/subtraction of 0, are recognized and appropriately simplified. If a division by zero is detected when interpreting constant expressions, the following warning is issued:
`DIVISION BY ZERO IGNORED`
Division is not performed until runtime.

– Intrinsic and standard functions are computed before the loop if their arguments are pseudoconstant (see section 9.3.5).

FUNC[TION-SIDEEFFECT]=NO

All functions are assumed to be standard (normal) functions.

In the case of standard functions, optimization measures are performed via function calls. The following attributes are assumed of a function defined as a standard function:

– The function changes none of its arguments, it merely returns the computed function value to the calling program unit. The function may not use or change any COMMON data.

– For each call with the same argument values the function will provide the same function value.

– The function performs no input/output and calls neither SUBROUTINE subprograms nor abnormal functions.

The compiler does not check whether a function declared as "standard" really has the specified characteristics. Declaration of a function as "standard", despite the fact that it does not have the characteristics of a standard function, may result in optimization errors.

Entry of FUNCTION-SIDEEFFECT=NO has the same effect as the FORTRAN statement ABNORMAL without parameters (see "FOR1" reference manual [21]).
If individual functions have been declared "abnormal" by means of an ABNORMAL statement, then FUNCTION-SIDEEFFECT=NO cancels the effect of the ABNORMAL statement.

FUNC[TION-SIDEEFFECT]=YES

All functions are assumed to be abnormal.

PARAM[ETER-SIDEEFFECT]=NO
: It is assumed that the dummy arguments of a subprogram are not associated with other dummy arguments or COMMON variables.

PARAM[ETER-SIDEEFFECT]=YES
: It is assumed that the dummy arguments of a subprogram are associated with other dummy arguments or COMMON variables. When a dummy argument or COMMON variable is modified, it is assumed that all dummy arguments and COMMON variables are modified, so many operations are not performed.

REORDER=NO
: Identically-ranked commutative operations are performed from left to right, as prescribed by the ANS FORTRAN 77 standard vom Standard ANS FORTRAN 77 vorgeschrieben ist (see "FOR1" reference manual [21]).

REORDER=YES
: Identically-ranked commutative operations can be reordered as a result of optimization. This reordering may affect the overflow of intermediate results. Rounding differences may occur when computing REAL and COMPLEX quantities.

Example:

In the expression A = 3. + B + 4. the constants 3 and 4 are first added as a result of reordering, producing: A = 7.+ B

`[*]COMOPT`	$\texttt{OPT[IMIZE]=}\left\{\begin{matrix}\texttt{4}\\ \texttt{(4,parameter[,...])}\end{matrix}\right\}$

All optimizations activated by OPTIMIZE=3 are additionally performed for loop sections executed on the basis of conditions. The specifications FUNCTION-SIDEEFFECT, PARAMETER-SIDEEFFECT and REORDER have the same meaning as with OPTIMIZE=3.

9.2.3 PROCEDURE-OPTIMIZATION compiler option

The PROCEDURE-OPTIMIZATION compiler option controls optimizations when calling procedures, i.e. subprograms and functions. These optimizations result in an improvement of runtime thanks to gradual shortening of the ENTRY/EXIT code when procedures are called. The runtime improvement is particularly important for programs which use a large number of short procedures.

The effect of the PROCEDURE option is subject to several restrictions, the number of which increases as a function of the degree of optimization.

Procedure optimization limits the options available for error diagnosis, depending on the particular degree of optimization. In particular, the call hierarchy in the event of program abortion often cannot be traced when procedure optimization is activated.

To establish priorities for pinpointing errors, procedure optimization is only performed if no debugging options except the default value TESTOPT=(STNR) are specified. If any other debugging option besides the default value is set, procedure optimization is deactived.

If ILCS modules are to be generated during the compilation, it is necessary to work with PROCEDURE-OPTIMIZATION=NO (see section 4.2.2.6, LINKAGE option).

```
                                               ⎧STD                   ⎫
                                               ⎪NO                    ⎪
[*]COMOPT      PROCEDURE[-OPTIMIZATION] =     ⎨                      ⎬
                                               ⎪YES                   ⎪
                                               ⎩SPECIAL[-ATTEMPTS]    ⎭
```

STD Default. Procedure optimization is defined by means of the optimization level specified in the OPTIMIZE option:

For OPT=NO,0,1,2	PROCEDURE-OPTIMIZATION=NO
For OPT=3,4	PROCEDURE-OPTIMIZATION=YES

If the user explicitly sets LINKAGE=STD, PROCEDURE-OPTIMIZATION=STD is interpreted as PROCEDURE-OPTIMIZATION=NO for all optimization levels. The following message is output:

```
MA42 PROC-OPT=NO BECAUSE LINK=STD
```

NO No shortening of the ENTRY/EXIT code takes place. If any debugging option other than TESTOPT=(STNR) is activated, PROCEDURE=NO is set and a warning issued.

If ILCS modules are to be generated during the compilation, it is necessary to work with PROCEDURE-OPTIMIZATION=NO (see section 4.2.2.6, LINKAGE option).

YES Procedure calls are optimized depending on the current conditions of the ENTRY environment. This results in savings in machine instructions for each subprogram call in the case of ENTRY and RETURN. These savings are achieved by gradually dispensing with

- chaining of the save areas
- recursivity check
- copying of arguments
- skipping over address constants
- allocate separate registers for addressing constants and data
- recopying arguments
- recovering old base addresses for constants and data
- recursion marking
- marking of BGFOR-compatible reentry points

When ENTRY or RETURN expression statements are used, they reduce the amount of code shortening which can be achieved.

When PROCEDURE=YES is specified, the most important intrinsic functions are calculated by calling optimized runtime system functions. The ENTRY names of the optimized runtime system functions are formed from the ENTRY names of the non-optimized functions and a $ symbol as the last character.

Intrinsic function	ENTRY name	ENTRY name of the optimized function
SIN	IF@S	IF@S$
COS	IF@C	IF@C$
ATAN	IF@AT	IF@AT$
SQRT	IF@Q	IF@Q$
EXP	IF@E	IF@E$
ALOG	IF@AL	IF@AL$
DSIN	IF@DS	IF@DS$
DCOS	IF@DC	IF@DC$
DATAN	IF@DAT	IF@DAT$
DSQRT	IF@DQ	IF@DQ$
DEXP	IF@DE	IF@DE$
DLOG	IF@DL	IF@DL$

Table 9-1: ENTRY names of optimized functions

The optimized runtime system functions can only be called if no functions with the names of the intrinsic functions are being used.

The optimized runtime functions for the intrinsic functions DSIN, DCOS, DATAN, DSQRT, DEXP and DLOG are referenced only by objects generated by a FOR1 Version ≤ 2.1. Objects generated as of FOR1 Version 2.2 use the high-precision mathematical routines.

SPECIAL-[ATTEMPTS]

More extensive shortening of the ENTRY/EXIT code as compared with PROCEDURE=YES. This leads to an improvement in runtime, particularly in the case of many short procedures.

An improvement as compared with optimization when PROCEDURE=YES is specified only occurs when procedures are limited as follows:

- The total length of the procedure object code may not exceed 4096 bytes. The compiler estimates the expected length of the object code, in which case 12 bytes are assumed for each use of a variable. Starting at an estimated length of approximately 4000 bytes, optimization with PROCEDURE=SPECIAL is no longer performed. If the actual length of the object code exceeds 4096 bytes contrary to the estimate, compilation is aborted and an error message issued.
- The procedure may not have any side entry points, i.e. no ENTRY statements with different parameters. All entries must include the same type of calculated functional value no matter what the function.
- The procedure must not call any other procedure.
- No mathematical function may be called in a subprogram, no exponentiations, calculations with complex numbers, relational operations, quadruple precision floating point divisions or compile time statements may be executed.
- No input/output operations may be performed in the procedure.

If a procedure does not satisfy these conditions, a change is made from PROCEDURE=SPECIAL to PROCEDURE=YES, without any message being issued.

If shortening an ENTRY/EXIT in the case of PROCEDURE=SPECIAL results in an error although the program without procedure optimization is free of errors, the error is reported in the diagnostic listing. The program can then be recompiled with PROCEDURE=YES instead of PROCEDURE=SPECIAL and optimized without error.

9.3 FOR1 optimization measures

9.3.1 Computation of constant expressions at compile time

Expressions whose operands have values that are known are computed at compile time. This causes the execution of instructions to be shifted from the object run to the time of compilation and reduces the runtime of the program.

Compile time computations cover all arithmetic, logical and relational operations.

Evaluation takes place not only of expressions whose constants are specified explicitly, but also of those expressions which contain variables whose values are known at compile time.

Example:

Original	Effect of optimization

`I = 17/2`
`J = I + 1`
	`I = 8`
	`J = 9`

Expressions with exclusively explicitly specified constants are also evaluated at compile time even when optimization level 0 is specified. Consequently, the first statement from the above example is transformed in any case, while the second statement is transformed only if optimization is activated.

Should an error be encountered during evaluation of an expression containing variables, evaluation is not carried out and a warning message is issued. In the case of an invalid expression with only explicit constants, computation continues with 1.

For "hidden" value assignments to variables (READ statement, subprogram call parameters), it is always assumed that the variable concerned will change its value.

Value tracing of COMMON variables at optimization levels 1 through 4 ceases once a subprogram is called. For OPT=3/4, information on COMMON variables is collected again until the subprogram is recalled.
Expressions in which equivalent variables (made equal by the EQUIVALENCE statement) occur are not interpreted when OPT=1/2.

Intrinsic functions with constant arguments are never computed at compile time no matter what the optimization level.

9.3.2 Optimization of logical expressions

Logical expressions in the logical IF statement are resolved into a string of conditional GOTOs. Once the logical value of the entire expression is ascertained, a transfer of control takes place and the remainder of the expression remains unprocessed.

Example:

Original	Effect of optimization

`IF(A.LT.B.OR.C.GT.F(0).OR.X.EQ.Y)`
`GOTO 10`
	`   IF(A.LT.B) GOTO 10`
	`   IF(C.GT.F(0))GOTO 10`
	`   IF(X.EQ.Y)GOTO 10`

`IF(A.EQ.B.AND.C.GT.D)X=Y`	`   IF(A.NE.B)GOTO 10`
	`   IF(C.LE.D)GOTO 10`
	`   X = Y`
	`10 CONTINUE`

The user may employ the order of individual operands to influence the time required for evaluation of an expression. If A is true more often than B, it is more efficient to write an expression in the form of A.OR.B than B.OR.A. By analogy, B.AND.A would be more efficient than A.AND.B. If the evaluation of a logical expression does not cover all of the operands, FUNCTION subprogram calls contained therein may remain unexecuted. In the first example, F(O) is not executed if .TRUE. is already provided by A.LT.B.

Logical expressions are optimized even when optimization level 0 is activated.

9.3.3 Recognition of common subexpressions

The instruction sequences generated during compilation are enhanced by computing frequently used common (sub)expressions only once and resorting to the existing value as required.

Example:

Original	Effect of optimization

`X=(A+B)*C`
`Y=(A+B)*D`
	`%T1=A+B`
	`X=%T1*C`
	`Y=%T1*D`

The optimization of common subexpressions covers the arithmetic operators +,-,*,/, when OPT=3/4 they also cover the arithmetic operation ** as well as intrinsic functions. According to the Standard Language the distributive law does not apply.

At each optimization level, common subexpressions are only recognized as being common when their internal representation is identical. The internal representation is based on the order in which the expressions are evaluated. In FORTRAN, this order is defined by means of parentheses and by means of the priorities set for operators.

Example:

```
X=A+B*C*D
Y=E+B*C
```

In the first statement the subexpression "B*C*D" and in the second statement the subexpression "B*C" are internally represented as a unit. Since the internal representation of the two subexpressions does not match, no optimization takes place.
If "B*C" is explicitly parenthesized, the same subexpression B*C is represented internally in both statements:

Original	Effect of optimization

`X=A+(B*C)*D`
 	
`Y=E+B *C`
	`%T1=B*C`
	`X=A+%T1*D`
	`Y=E+%T1`

Common subexpressions relate to values rather than to variable names. If a variable changes its value before the repeated occurrence of the same expression, the computed value can no longer be used.

For example, the following instruction sequence is not optimized because the value of A has changed:

```
X=A+B
A=A/3
Y=A+B
```

Tracing of a variable is governed by the following rules:

- For "hidden" value assignments to variables (READ statement, subprogram call parameters) the assumption is always that the variable concerned changes its value.
- COMMON variables are taken into account when OPT=1/2 only to the extent that no SUBROUTINE subprogram call or ABNORMAL FUNCTION subprogram occurs. For OPT=3/4, COMMON variables in the area between the two subprogram calls are taken into account.
- The area of tracing covers a basic block when OPT=1/2, and a loop when OPT=3/4.

Equivalenced quantities

If an expression contains equivalenced quantities (EQUIVALENCE statement), this expression is not optimized when OPT = 1/2. When OPT = 3/4, this expression is optimized only if none of the equivalenced quantities has changed its value in the meantime.

Example:

```
EQUIVALENCE (A,C)
X = A+B              (1)
   .
   .
   .
Y = A+B              (2)
```

For OPT = 1/2, the common subexpression A+B is not recognized, since A is equivalenced with C and equivalenced quantities are not interpreted at these optimization levels.
For OPT =3/4, the common subexpression A+B is optimized if neither A nor B nor the quantity C equivalenced with A changes its value between (1) and (2).

Overlaid variables

Partly overlaid variables are not taken into account in the recognition of common subexpressions.

Completely overlaid variables may be covered, but a change to the value of one variable causes the value of the overlaid variable also to be changed.

In the following example, however, optimization does take place:

Example:

Original	Effect of optimization

`X = (A+B)*D`
`C = A`
`Y = C+B`
	`%T1 = A+B`
	`X   = %T1*D`
	`C   = A`
	`Y   = %T1`

9.3.4 Subscript computation

In order that the compiler may reference an array element, it must determine the address of the array element from the specified subscript values. To do this, the compiler expands the subscript list into a sequence of arithmetic operations.

This arithmetic generated during subscript expansion is optimized as follows:

- when OPTIMIZE=0, no optimization takes place.
- when OPTIMIZE=1/2, optimization takes place, e.g. by the use of common subexpressions, by employment of the distributive law, by reduction of multiplication to additions, and by shifting instruction code (see section 9.3.5).
- when OPTIMIZE=3/4, all optimization measures activated by OPTIMIZE=1/2 are performed. In addition, an attempt is made to break down the address computation into three parts:
  - a constant part computed at compile time,
  - a loop-invariant part shifted to precede the loop,
  - a variable part which stays in the loop.

The address A_s of an array element $A(s_1, ..., s_n)$ of an array $A (p_1:q_1,p_2:q_2, ..., p_n:q_n)$ denotes the first byte of $A (s_1,...,s_n)$. This address is calculated as follows:

Assuming that

$$d_k = q_k - p_k + 1 \qquad (k=1,\ldots,n)$$

are the sizes of the individual dimensions and

$$m_0 = 1,\ m_1 = d_1\ ,\ldots,\ m_i = d_1 * d_2 *\ldots* d_i \qquad (i=1,\ldots,n)$$

are the sizes of the i-dimensional subarrays.

Then the subscript value I_s of A $(s_1,\ldots,s_n)$ is obtained from

$$I_s = (s_n-p_n)\ m_{n-1} + \ldots + (s_2-p_2)\ m_1 + (s_1-p_1)\ m_0 + 1,$$

and the following applies:

$$A_s = A + l(I_s-1),$$

where A is the address of the array
l the length of the element.

Thus

$$A_s = A + l(s_n-p_n)\ m_{n-1} + \ldots + l(s_2-p_2)\ m_1 + l(s_1-p_1)\ m_0$$
$$= A - l(p_1m_0 + \ldots + p_nm_{n-1}) + l(s_1m_0 + \ldots + s_nm_{n-1})$$

The following applies in particular:

$$A_0 = A(0,\ \ldots,0) = A-l(p_1m_0 + \ldots + p_nm_{n-1})$$

and consequently

$$A_s = A_0 + l(s_1m_0 + \ldots + s_nm_{n-1})$$

This formula offers the best chances in respect of the occurrence of common subexpressions and therefore for optimization. That is why it is used for subscript expansions in FOR1.

The address A_0 and the multipliers m_i are computed at compile time if the subscript boundaries are constant; otherwise at the beginning of the program unit.

Example:

`Array`	$m_0 = 1$
`INTEGER*8 A(3,7,5)`	$m_1 = 3$
	$m_2 = 21$

Original	Effect of subscript expansion
`      DO 1  I = M1,M2`	`      DO 1  I  = M1,M2`
`      DO 1  J = N1,N2`	`      DO 1  J  = N1,N2`
`      DO 1  K = L1,L2`	`      DO 1  K  = L1,L2`
`            .`	`      %T1 = S1(I)`
`            .`	`      %T2 = S2(J)*3`
`            .`	`      %T3 = %T1+%T2`
`1     A(S1(I),S2(J),S3(K))=`	`      %T4 = S3(K)*21`
`      F(I,J,K)`	`      %T5 = %T3+%T4`
	`      %T6 = %T5*8`
	`1     A (%T6) = F(I,J,K)`

%T6 is the displacement to the address calculated for A(0,0,0).

9.3.5 Loop optimization

The range of a loop is usually executed repeatedly; an optimization of such program parts is therefore particularly effective.

Optimization levels 1 and 3 perform optimization measures throughout the entire loop range only for "ideal" loops. "Ideal" loops are loops whose ranges are without transfers of control, i.e.:
- no GO TO statement,
- no END, ERR parameters in input/output statements,
- no IF statements,
- no branch label parameters in subprogram calls.

Non-ideal loops have transfers of control inside their loop range. The loop range consists of several basic blocks. In the case of optimization levels 1 and 3, loop optimization is limited to those basic blocks for which it is possible to recognize at compile time that they are executed exactly once for each loop cycle. In the case of optimization level 2, the optimization measures of level 1 are also executed in conditionally executable loop sections; in the case of level 4, also the optimization measures of level 3. Optimizations in loop sections which can be conditionally executed in most cases include a further gain in runtime; however in some cases these can lead to runtime loss or unwanted interrupts.

Example 1:

```
   LOGICAL BREAK
   DO 10 I = 1,100
   IF(BREAK) GOTO 10
   A(K,I)=A(K,I)+B(L,I)
10 CONTINUE
```

With OPTIMIZE = 2/4, linear subscript incrementation becomes effective in the A(K,I)... statement:

- If the logical variable BREAK has the value .FALSE., the result is a significant improvement in runtime.
- If, however, the variable BREAK has the value .TRUE., unnecessary auxiliary calculations are performed for a subscript increment which would be omitted with OPTIMIZE = 0/1 or OPTIMIZE = 3, and result in an increase in runtime.

Example 2:

```
   DO 10 I = 1,100
10 IF(A.GT.O)B = SQRT(A)
```

At optimization levels 3 or 4, intrinsic functions are calculated before the loop for loop-invariant arguments. With OPTIMIZE=4, loop opimization is also executed in the conditionally executed loop section: SQRT(A) is calculated outside the loop. In the case of a negative argument A, an error interrupt occurs.

Loop-invariance

An explanation of the term "loop-invariance" requires that the term "pseudo-constant" be explained first. A data item is pseudo-constant within a loop when the following conditions are satisfied:

- the value of the data item is not changed in the loop after a first assignment, or no assignment is made.
- the initial assignment, if any, is made before the value is used for the first time and a pseudo-constant entity or an expression that contains only pseudo-constant entities is assigned. This assignment may only be made in a part which is executed exactly once per loop cycle.

Constants are therefore also pseudo-constant.

Data that is not pseudo-constant in a given loop range is also not pseudo-constant in any encompassing loop.

Example:

```
  DO 1 I = 1,10
  IF(I.GT.3)M = 3          J   is pseudo-constant since the value of
  %T1 = J                      J is not changed in the loop
  %T2 = K                  K   is not pseudo-constant since the value
  K  = 5                       is assigned after it is used
  L  = 5                   L   is pseudo-constant, the value is assigned
  %T3 = L                      before it is used
  %T4 = M                  M   is not pseudo-constant, since the value
  N  = F(I)                    assignment takes place in a conditionally
  %T5 = N                      executed part
1 CONTINUE                 N   is not pseudo-constant since a value is
                               assigned that is not pseudo-constant
```

An operation in a loop range is loop-invariant if the following conditions are true:

For an arithmetic operation:	All operands are pseudo-constant
For a function:	No abnormal FUNCTION, all arguments are pseudo-constant
For a value assignment:	The assigned entity is pseudoconstant; the variable assigned a value was not previously used

Operations which satisfy these conditions are moved out of the range of the loop to the outside.

Note

Since all functions are generally abnormal except instrinsic functions, use should be made of the ABNORMAL statement (see "FOR1" reference manual [21]).

Instruction code shifting

Loop-invariant parts are extracted from the loop range in order to reduce the number of cycles for these parts.

Example:

STMT	Original	Effect of optimization
6	`DO 1 I = 1,9,2`	`%T1    = F(K)`
7	`1 A(I)   = F(K)+I**2`	`DO 1 I = 1,9,2`
		`1 A(I)   = %T1+I**2`

The effect of optimization is shown as follows in the decompiler listing (see section 4.7.9):

```
     ***** STATEMENT 6 (DO) *****************
6          I=1
     ***** STATEMENT 7 (MOVED STMT) *********
7          %T00010154=F(K)
     ***** STATEMENT 7 (MOVED STMT) *********
7          %T00010330=4
7          %I1=5
     ***** STATEMENT 7 (ASSIGNMENT) *********
7    L3    CONTINUE
7    1     %T00010198=I*I
7          A(%T00010330/4)=%T00010154+%T00010198
     ***** STATEMENT 7 (INCR STMT) **********
7          %T00010330=%T00010330+8
     ***** STATEMENT 7 (DOEND) **************
7          I=I+2
7          %I1 = %I1- 1
7          IF (%I1 .NE. 0) GOTO L3
```

F must not be an abnormal function, i.e. the compiler option COMOPT OPT=(3,FUNCTION-SIDEEFFECT=NO) or the FORTRAN statement ABNORMAL without parameters must be specified.

The extracted parts are executed before the loop is processed. If it is not yet known at compile time whether this loop will be executed at least once, the shifted parts may only be executed in dependence on the iteration counter.

The iteration counter controls correct processing of the loop. Fig. 9-2 shows the execution of a DO loop (see the "FOR1" reference manual [21]).

If several loops are nested, optimization begins with the innermost loop. If the extracted parts are to be executed unconditionally, they may be included in the optimization for the outer loops. In this way loop-invariant parts may be shifted more and more to the outside through several nested loops.

This figure is not any longer available for the online pdf.

Fig. 9-2: Execution of a DO loop with instruction code shifting

Reduction to less complex operations

In loop ranges, optimization replaces complex operations with less complex operations. Exponentiations are reduced to multiplications, and multiplications to additions.

With OPT=1, multiplications in which the product is formed from a loop variable and a pseudo-constant quantity are replaced by additions. With OPT=3, multiplications in which programmed iteration variables occur as factors are also replaced by additions (see section 9.4.2).

Iteration variables

Iteration variables are variables whose values are changed by a pseudo-constant value for each cycle of the range of a loop.

The control variable of a loop is therefore always an iteration variable. Reduction to addition is performed by splitting up the operation into an initialization part and an incrementation part.

The initialization part is loop-invariant and precedes the loop (see above, Instruction code shifting); the incrementation part is placed in a separate incrementation block. Fig. 9-3 shows the execution of a DO loop in reducing multiplications to additions (see "FOR1" reference manual [21]).

This figure is not any longer available for the online pdf.

Fig. 9-3: Execution of a DO loop reducing multiplications to additions

Example:

```
  PROGRAM OPT
  ABNORMAL
  INTEGER I,A,B,X,F
  READ (*,*)B
  DO 1 I = 1,9,2
  A=B*I
1 X=F(A)
  WRITE (*,*) X,I
  END
```

F is defined as a normal function by means of the ABNORMAL statement without parameters. If F were an abnormal function, the multiplication of B*I for OPT=1/2 would not be reduced to an addition. This would only be the case when OPT=3/4. Loop initialization I=1 in the example is only eliminated in conjunction with optimization levels 3 and 4.

In the decompiler listing, the effect of optimization on the loop is evident (with OPT=3):

```
   |  ***** STATEMENT 5 (DO) *****************
 5 |  L5    I=1
   |  ***** STATEMENT 6 (MOVED STMT) *********
 6 |        %T00010220=B
 6 |        %T00010264=2*B
 6 |        I=11
 6 |        %I1=5

   |  ***** STATEMENT 6 (ASSIGNMENT) *********
 6 |  L3    A=%T00010220
   |  ***** STATEMENT 7 (ASSIGNMENT) *********
 7 |  1     X=F(A)
   |  ***** STATEMENT 6 (INCR STMT) **********
 6 |        %T00010220=%T00010220+%T00010264
   |  ***** STATEMENT 7 (DOEND) **************
 7 |        %I1 = %I1- 1
 7 |        IF (%I1 .NE. 0) GOTO L3
```

This figure is not any longer available for the online pdf.

Fig. 9-4: Execution of a DO loop before and after optimization (OPT=1)

Elimination of "unnecessary" incrementations of iteration variables

If an iteration variable no longer occurs within the range of a loop except in the iteration statement, its continuous incrementation is unnecessary. Continuous incrementation of the iteration variables can be eliminated by assigning to the iteration variable its final value before processing the loop and if the iteration variable is no longer used after the loop. The "final value" of the iteration variable is the value it would have reached by single incrementation after a loop is processed.

For loops that include a transfer out of their range, such a "final value" cannot be assigned, and single incrementation can only be eliminated if the iteration variable is no longer used after the loop.

"Unnecessary" incrementations are often the result of reducing multiplications to additions.

Example:

In the example shown in Fig. 9-4, the statement **I** = **I**+2 is removed and replaced by the "final value" assignment **I** = 11.

9.3.6 Global register allocation

The basic concept in global optimization is to save memory accesses. Variables that are used frequently and compiler-generated temporary auxiliary entities should be left in registers.

When optimization is activated, the various entities are examined as to the number of accesses. Furthermore an analysis is performed in respect of the time that will be saved if a particular entity is contained in a register. This information is then evaluated as part of register allocation.

9.4 Examples of optimization

9.4.1 Effect of optimization on a program loop

```
  DO     SEG     STMT     I     LINE           SOURCE-TEXT
----------------------------------------------------------------------------
         1/1       1               1 |          PROGRAM OPT
           2       2               2 |          DO 1 I=1,5
   1       2       3               3 |          L=7
   1       2       4               4 |          M=M+N*L
   1       2       5               5 |          K=I*3+L*4
   1       3       6               6 |        1 N=N*7+K
           3       7               7 |          END
```

The excerpts from the object listings in Fig. 9-5 show the effect of OPTIMIZE=0 and OPTIMIZE=1 on the above program loop:

(1)	Shifting of constant assignment from the iteration range to (9).
(2)	Replacing the variable L with the constant 7 (13).
(3)	Reducing the multiplication to an addition (12).
(4)	Computing the constant expression (L*4) at compile time
(5)	Computing the start value of K at compile time (10).
(6)	Reuse of the common subexpression N*7 from statement 4.
(7)	Eliminating the incrementation of the iteration variable. Assigning the final value outside the loop (8).
(11)	Register load points outside the loop.
(13),(14)	Use of machine instructions of type RR.
(15)	Register storage points outside the loop.

This figure is not any longer available for the online pdf.

Fig. 9-5: Effect of optimization on a program loop

9.4.2 Differences between optimization levels 1 and 3

Original	Before optimization	After OPTIMIZE=1	After OPTIMIZE=3
SUBROUTINE OPTIM (A, 1END,INCR)			
INTEGER A(100),END, 1INCR,IV			

IV=1	IV=1
	%V1=END
	J=2 (1)
	%I1=%V1-1
	IF %I1≤0 GOTO L13
		IV=1
		%V1=END (3)
		J=2
		%I1=END-1
		IF %I1≤0 GOTO L13
			J=2
			%I1=END-1
			IF %I1≤0 GOTO L13

			%T4=4 %T5=INCR*4 (6)
DO 10 J=2,END A (IV)=J IV=IV+INCR 10 CONTINUE	L3 %T3=IV %T1=%T3*4 A(%T1)=J IV=IV+INCR J=J+1 %I1=%I1-1 (2) IF(%I1.NE.0) .GOTO L3	L3 %T3=IV %T1=IV*4 (4) A(%T1)=J IV=IV+INCR (5) J=J+1 %I1=%I1-1 IF(%I1.NE.0) .GOTO L3	L3 A(%T4)=J %T4=%T4+%T5 (7) J=J+1 %I1=%I1-1 IF(%I1.NE.0) .GOTO L3
END	L13 END	L13 END	L13 END

Table 9-2: Differences between OPTIMIZE=1 and OPTIMIZE=3

Using subprogram OPTIM as an example, various differences in the effect of optimization levels 1 and 3 become apparent. Quantities with a % symbol as the first character refer to internal representation. In the initialization section (01) of the loop, the loop parameters are evaluated and iteration counter %I1 tested. In (2) the iteration counter is decremented by one and a branch is made to L3 if %I1>0.

Replacing complex operations with less complex operations

Optimization levels 1 and 3 differ in their treatment of iteration variables. Iteration variables are integer variables which change their value in a loop in a linear manner. In the example, the DO variable J and the programmed iteration variable IV are the iteration variables.

With OPT=1, only DO variables are recognized as iteration variables. For this reason only those multiplications containing a DO variable as a factor are simplified. As a result of subscript expansion (cf. section 9.3.4) multiplication (4) with the programmed iteration variable IV takes place. This multiplication is not simplified when OPT=1.

With OPT=3 on the other hand, the multiplication (4) with programmed iteration variable IV is also simplified. The loop-invariant operations (6) are placed ahead of the loop; the multiplication (4) is reduced to an addition (7). As a result of this simplification, the term (5) becomes superfluous and is removed by means of "dead code elimination".

Calculation of constant expressions at compile time

Following loop optimization, the program section before the loop consists of the first basic block (1) and the second basic block.

```
%T4=IV*4
%T5=INCR*4
```

This program section before the loop is further optimized when OPT=3. In the first and second basic block, IV has a value of 1. With OPT=1, optimization is only performed within each basic block. Thus, when the second basic block is optimized, it is not taken into account that IV has a value of 1, and no optimization takes place.

With OPT=3, on the other hand, the values are traced throughout the entire loop, as a result of which `%T4=IV*4` can be reduced to `%T4=4`.

Elimination of superfluous code

At the two optimization levels 1 and 3, value tracing can be used in the expression

```
   %I1=%V1-1
```

to replace %V1 by means of END:

```
   %I1=END-1
```

The statement (3) %V1=END becomes superfluous as a result. This superfluous statement is, however, only eliminated by optimization techniques using "dead code elimination" when OPT=3 is specified.

10 Programming considerations

10.1 Considerations for individual FOR1 language elements

See also "FOR1" reference manual [21].

WAIT statement
The WAIT statement is implemented in such a way that only the specified parameters are provided with the appropriate values. The WAIT statement does not, however, cause the task to be suspended since waiting for the end of data transfer is already included in the asynchronous READ and WRITE statements.

SAVE statement
The SAVE statement, although accepted, has no additional effect. With FOR1, all data items of a FUNCTION or SUBROUTINE subprogram generally retain their values after exiting from the subprogram.

FIND statement
The FIND statement is supported to the extent that the specified parameters are supplied with the appropriate values and the specified checks are carried out. However actual positioning of the referenced file is always a function of the READ or WRITE statement.

ENCODE, DECODE statements
The statements are retained for reasons of compatibility. It is recommended, however, to apply the FOR1-supplied language elements of input/output to internal files for data transfer within memory.

"Dangerous" language elements

The following describes certain language elements and constructions that should not be used unless there are good reasons for doing so. These elements make the program logic less transparent and, since they are known to be frequent sources of hidden programming errors, they reduce the reliability of a program. In addition, they impair or reduce the optimization of parts of the program (see chapter 9).

– Extended range of DO loops
  The extended range may be replaced by execution of a subprogram or by in-line programming of the relevant statements in the loop range. If this involves copying the extended range, it is best converted into an %INCLUDE item.

– Duplicate use of ASSIGN variables
  ASSIGN variables should be used only for their original function as carriers of statement numbers, and not in arithmetic operations. Any errors resulting from such mixed use are usually difficult to pinpoint.

– Overlaying data items of differing length.

10.2 FOR1 extensions no longer supported by the Fortran90 compiler

Almost all FOR1 extensions are also supported by the Fortran90 compiler. There are however a few exceptions. The user has the capability by specifying the FORTRAN90-CHECK = YES compiler option (see section 4.1.2.8) to check whether a source program contains FOR1 extensions that are no longer supported by the Fortran90 compiler.

The following section contains a summary of these language elements that are incompatible with the Fortran90 compiler. Also given in each case is the message used to mark the occurrence of such language elements during the Fortran90 check.

Cyclical range specification in the IMPLICIT statement

Within the scope of FOR1 it is permissible to define a range of letters cyclically in an IMPLICIT statement. The following is possible, for example:

```
IMPLICIT INTEGER*2 (Y-D)
```

This will no longer be supported by the Fortran90 compiler. In the Fortran90 check, the compiler marks IMPLICIT statements containing cyclical range specifications with the message:

```
FA300  FORTRAN90 DEVIATION:  CYCLIC RANGE-SPECIFICATION
```

Complex expressions as lower bound, upper bound or step size in loops:

Within the scope of FOR1, expressions of data type COMPLEX are permissible as lower bound, upper bound and step size in DO loops and implicit DO loops.
The Fortran90 compiler will no longer support this. If complex expressions are used for such purposes, the compiler marks the corresponding positions during the Fortran90 check with one of the following warnings:

```
SA250  FORTRAN90 DEVIATION:  LOWER BOUND OF TYPE COMPLEX
SA251  FORTRAN90 DEVIATION:  UPPER BOUND OF TYPE COMPLEX
SA252  FORTRAN90 DEVIATION:  STEP SIZE OF TYPE COMPLEX
```

Nesting of logical IF statements

Nested logical IF statements are permissible within the scope of FOR1, i.e. one logical IF statement can in turn contain a further logical IF statement.
This will no longer be supported by the Fortran90 compiler. Any construction of this type will be marked with the following message during the Fortran90 check:

```
FA301  FORTRAN90 DEVIATION:  NESTED BOOLEAN IF-STATEMENTS
```

Free statement sequence

The FOR1 compiler permits a completely free sequence of declarations and executable statements. This holds true only to a limited extent for the Fortran90 compiler: declarations must precede the first instance of use. Since it would be extremely complicated for the FOR1 compiler to check this in each case, a warning will be issued during the Fortran90 check as soon as executable statements are found to precede a declaration statement:

```
FA302  FORTRAN90 DEVIATION:  POSSIBLE USE BEFORE DECLARATION
```

Temporary updating of source programs

The temporary updating of source programs using the UPD compiler option will no longer be supported by the Fortran90 compiler. Use of the UPD option will be marked with the following warning during the Fortran90 check:

```
MA34  FORTRAN90 DEVIATION: UPD
```

*DELETE statements occurring in the program will not be marked, however.

Missing END statement

Within the FOR1 language scope it is permissible to omit the END statement at the end of a program unit. The Fortran90 compiler will tolerate this only at the end of the entire source program file. If an END statement is missing, the compiler issues the following message during the Fortran90 check.

```
FA304  FORTRAN90 DEVIATION:  END-STATEMENT MISSING
```

RETURN statements in main program

FOR1 replaces RETURN statements in a main program with STOP statements. The Fortran90 compiler will no longer support this. If a RETURN statement occurs in a main program, the following message will be issued during the Fortran90 check.

```
FA305  FORTRAN90 DEVIATION: RETURN-STATEMENT IN MAIN-PROGRAM
```

End mark for LINE-END comments

The Fortran90 compiler will also support LINE-END comments. However, it should be borne in mind that the Fortran90 character set includes four characters more than the FOR1 character set. These characters are as follows:

- quotation marks (")
- semicolon (;)
- greater than (>)
- less than(<)

If one of these characters has been specified as the end mark, the following message will be output during the Fortran90 check:

```
MA36  FORTRAN90 DEVIATION:  FORTRAN90 CHARACTER AS LINEEND
```

FPOOL

The Fortran90 compiler will no longer support FPOOL since language elements are available directly in Fortran90 for checking call interfaces. Although the %FPOOL statements will be accepted syntactically, their semantics will be ignored. If %FPOOL statements occur in a source program, they will be marked with one of the following messages during the Fortran90 check:

```
FA308  FORTRAN90 DEVIATION:  %FPOOL COMPILER DIRECTED STATEMENT
                             WILL BE IGNORED
FA309  FORTRAN90 DEVIATION:  %NOFPOOL COMPILER DIRECTED STATEMENT
                             WILL BE IGNORED
```

When the FPOOL compiler option is used, the following warning is issued:

```
MA35  FORTRAN90 DEVIATION:  FPOOL
```

The functions of the central FPOOL can still also be utilized by the Fortran90 compiler, but without interface checking by FPOOL.

Debugging statements

Within the FOR1 language scope there are debugging statements that are specified in the source program.
The symbolic debugging aid AID is available in BS2000. AID encompasses all the essential functions of the debugging statements and thus makes these dispensable. The Fortran90 compiler will ignore debugging statements. If debugging statements are used in a source program, then the compiler will issue one of the following messages during the Fortran90 check:

```
FA310  FORTRAN90 DEVIATION:  %CALLTRACE COMPILER DIRECTED STATEMENT
                             WILL BE IGNORED
FA311  FORTRAN90 DEVIATION:  %CHECK COMPILER DIRECTED STATEMENT
                             WILL BE IGNORED
FA312  FORTRAN90 DEVIATION:  %COUNT COMPILER DIRECTED STATEMENT
                             WILL BE IGNORED
FA313  FORTRAN90 DEVIATION:  %DISPLAY COMPILER DIRECTED STATEMENT
                             WILL BE IGNORED
FA314  FORTRAN90 DEVIATION:  %FULLTRACE COMPILER DIRECTED STATEMENT
                             WILL BE IGNORED
FA315  FORTRAN90 DEVIATION:  %JMPTRACE COMPILER DIRECTED STATEMENT
                             WILL BE IGNORED
```

Parameters for the ENCODE and DECODE statements

ENCODE and DECODE statements have the following format within the FOR1 language scope:

```
ENCODE (intreclength, format, charname [,intname])   or
DECODE (intreclength, format, charname [,intname])
```

The parameter *intname* is present only for reasons of compatibility and is ignored by the FOR1 compiler. The Fortran90 compiler will also no longer accept this parameter syntactically. If the parameter occurs in a source program, the following message will be output during the Fortran90 check:

```
SA253  FORTRAN90 DEVIATION:  FORTH PARAMETER OF ENCODE/DECODE
```

Overlapping in the CHARACTER assignment

Whilst it is impermissible according to ANS FORTRAN 77 for left and right side to overlap in a CHARACTER assignment, this is possible both according to the Fortran90 standard and with the FOR1 compiler. However, the assignment is effected differently in each case.

Example:

```
CHARACTER*5 CHAR
CHAR      = 'ABCDE'
CHAR(2:5) = CHAR(1:4)
```

According to the Fortran90 standard, CHAR must contain the value 'AABCD' after the assignment. With FOR1 however, it has the value 'AAAAA'.

If overlapping of the two operands is possible in a CHARACTER assignment, the following message is issued during the Fortran90 check.

```
SA254 FORTRAN90 DEVIATION:  DIFFERENT SEMANTICS BY OVERLAPPING FROM
                            SOURCE AND TARGET
```

10.3 High-precision mathematical intrinsic functions

Enhanced routines for the following intrinsic functions of the DOUBLE PRECISION (or REAL*8) type are provided as of FOR1 Version 2.2A:

- DSIN
- DLOG
- DSQRT
- DTAN
- DASIN
- DLOG2
- DCOS
- DEXP
- DATAN
- DCOTAN
- DACOS
- DLOG10

The enhanced routines are naturally also used if the generic names of the above functions are applied with arguments of the DOUBLE PRECISION (or REAL*8) data type.

The new routines are high-precision, i.e.:

a) no representable double-precision floating-point number exists between the calculated result of the function and the exact result. (However, there can be a double-precision floating-point number which is *close* to the exact result.)

b) exactly representable function values are calculated exactly.

In spite of this considerably increased accuracy, the new routines offer a level of performance that is at least equally as good as the corresponding old FOR1 routines.

The new routines are accessed via the same interface as the old ones.

However, the new routines have internal names different to those of the corresponding old routines. In the case of program systems containing both new FOR1 V2.2A programs and FOR1 programs compiled with a FOR1 Version < 2.2A, it may happen that one subprogram computes using the old routine and another computes using the new routine.

Since the new routines supply different function values to the old FOR1 routines, incompatibilities may occur - for example in the case of test packages whose correctness is checked on the basis of function results. Therefore, if the program contains calls for the new routines, the following message is issued when the program starts:

```
IMPROVED MATHEMATICAL ACCURACY
```

When supplying values for job variables which monitor execution, this message is not taken into consideration.

10.4 Floating-point arithmetic and fixed-point arithmetic

Floating-point arithmetic

An interrogation for equality of values involving REAL and COMPLEX type data might not have the desired effect, since the propagation of rounding errors in different computations may lead to different approximations of the same arithmetic result, so that bit equality is no longer ensured. It is advisable to define a nonstandard relational operator EQ by means of a statement function, such as

```
LOGICAL*1 EQ
PARAMETER(EPSILON = 0.000001)
EQ(A,B)=ABS(A-B).LE.EPSILON
```

Fixed-point arithmetic

An overflow is only recognized after additions or subtractions with data items of type INTEGER*4 (error message: FIXED POINT OVERFLOW). Overflows following multiplications with INTEGER*4 data items, overflows or underflows after arithmetic operations with INTEGER*1 and INTEGER*2 data items are not recognized. The higher-order places are lost, and computation continues with wrong values. The sign bit of an overflowing data item is overwritten. The branch of an arithmetic IF statement examining the value of that data item is no longer indicative of the value of the data item.

10.5 Alignment of data items

In FOR1, data items are aligned on byte, halfword, word or doubleword boundaries, according to their type. Table 10-1 shows the rules for alignment of each data type.

Data type	Aligned on
INTEGER*1	byte
INTEGER*2	halfword
INTEGER*4	word
INTEGER*8	doubleword
REAL*4	word
REAL*8	doubleword
REAL*16	doubleword
COMPLEX*8	word
COMPLEX*16	doubleword
COMPLEX*32	doubleword
CHARACTER*N	byte
LOGICAL*1	byte
LOGICAL*4	word

Table 10-1: Alignment of data items

For data items of type COMPLEX, alignment relates separately to the real portion and the imaginary portion.

In arrays, the first element of the array is aligned according to the rules.

For COMMON blocks and areas formed by overlaying with the EQUIVALENCE statement, the user must allow for this alignment of data items. The beginning of a COMMON block or of an overlaid area is always aligned on a doubleword boundary. Alignment of the individual data items of the COMMON block may create gaps.

The occurrence of gaps in COMMON blocks may be prevented by arranging the data items according to descending length specification (cf. example 1). Since CHARACTER data items are aligned on a byte boundary, it is advisable to place this at the end of the COMMON block.

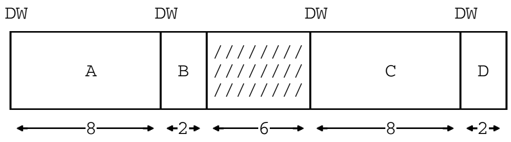
When overlaying storage areas (EQUIVALENCE statement), alignment errors may occur (cf. example 2).

Example 1:

a) COMMON block with gap:

```
REAL*8 A,C
INTEGER*2 B,D
COMMON/LIST/A, B, C, D
```

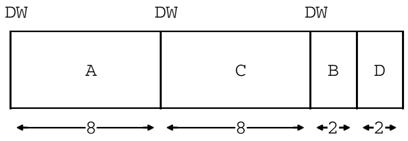
Arrangement in COMMON block (DW = doubleword boundary):



b) COMMON block with enhanced arrangement:

```
REAL*8 A,C
INTEGER*2 B,D
COMMON/LIST/A, C, B, D
```

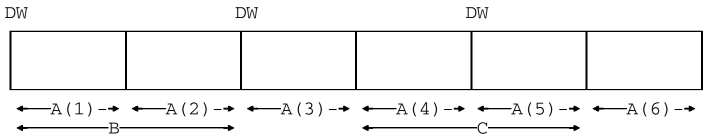
Arrangement in the COMMON block (DW = doubleword boundary):



Example 2:

```
REAL*4 A(6)
REAL*8 B,C
EQUIVALENCE (A(1),B),(A(4),C)
```

This statement sequence will result in an alignment error because C cannot be aligned on a doubleword boundary (DW):



The following error message is issued:

```
ERROR  (SA124)  ENTITY C MISALIGNED DUE TO EQUIVALENCE
```

10.6 Creating dynamic memory for arrays

As of FOR1 Version 2.0A, the user can employ the extended address space of extended systems (XS systems), permitting processing of large amounts of data by means of FORTRAN programs.

Dynamic arrays can be used on these systems. With dynamic arrays, memory allocation is not performed until program runtime. The user must request the memory required for a certain array by invoking the subprogram ALLOC. The memory provided is released explicitly by calling the subprogram DEALLOC, or automatically on program termination.

ALLOC and DEALLOC are ready-made subprograms within the runtime system; they are invoked by CALL.

The dynamic storage management subprograms DYNARA and DYNAST which were available up until now are no longer supported by the compiler as of FOR1 Version 2.0A. Programs containing DYNARA or DYNAST calls can still be executed using a runtime system ≥ 2.0A.

In order to be able to utilize the extended address space above 16 Mbytes, a program must be executed on a system with XS capabilities. Dynamic allocation of memory by means of the ALLOC and DEALLOC subprograms is, however, also advisable for users of systems without XS capabilities. Arrays which are not dynamically created are created when the load module is loaded and not deallocated until it is unloaded. However, memory for a dynamically stored array is only required in the times between when ALLOC and DEALLOC (or program end) are invoked, i.e. by means of these subprograms the user of a system without XS capabilities can also relieve the burden on the address space.

A prerequisite for the use of the extended address space above 16 Mbytes is that the object module concerned is an XS module. (As of FOR1 Version 2.2A, XS modules are always generated. With FOR1 Versions < 2.2A, compilation would need to be performed with EXTENDED-SYSTEM=YES in order to generate XS modules.)

The program attributes are evaluated by the linking and loading system or they can still be modified by the linking and loading system as well as by using the runtime option START=XS. Appendix A.7 describes the interaction of the possible specifications in accordance with which an XS program above 16 Mbytes, or an XS or non-XS program below 16 Mbytes is executed.

The following language elements are available for the creation of dynamic memory:

- the format (:[,:] [,...])
  for the dimension bound list of a dynamically created array;
- the ALLOC subprogram, to request memory for a dynamically created array;
- the DEALLOC subprogram, to release this memory;
- the GETSHAPE subprogram, to interrogate the dimension bounds of a dynamically created array.

10.6.1 Declaring dynamic arrays

Dynamically created arrays must be declared in the FORTRAN program similar to the way static arrays are. The upper and lower bounds of a dimension which are still open when this declaration is made need only be specified in the form of a colon in the dimension bound list. A dynamically created array is thus declared using a type, DIMENSION or COMMON statement with the format:

```
arrayname (:[,:] [,...])
```

The number of colons in the dimension bound list is equal to the number of dimensions of the array. A maximum of 7 dimensions is permissible.

10.6.2 Allocating memory (CALL ALLOC)

Prior to the first reference to a dynamically created array (or to an array item), the requi-red memory must be allocated by invoking the ALLOC subprogram:

$$\texttt{CALL ALLOC (arrayname,}l_1\texttt{,}u_1\texttt{[,}l_2\texttt{,}u_2\texttt{][,...][,}l_n\texttt{,}u_n\texttt{][,}\begin{Bmatrix}\texttt{'NXS'}\\ \underline{\texttt{'ANY'}}\end{Bmatrix}\texttt{])}$$

arrayname
: Name of the array dynamically specified in the type, DIMENSION or COMMON statement.

l_i,u_i	Arithmetic expressions of type INTEGER*4. l_i is the smallest, u_i the largest subscript of the i-th dimension ($1 \leq i \leq 7$) of the dynamic array *arrayname*. The number of dimensions n ($1 \leq n \leq 7$) must be equal to the number of dimensions in the associated type, DIMENSION or COMMON statement.
'NXS'	The memory for the array *arrayname* is created below 16 Mbytes.
'ANY'	The memory for the array *arrayname* is created dependent on the current machine addressing mode (see appendix A.7):

– If 24 is the current machine addressing mode, the memory for the dynamically created array is stored below 16 Mbytes.

– If 31 is the current machine addressing mode, an attempt is made to place the memory for the dynamically created array above 16 Mbytes. If this is not successful, the memory is created below 16 Mbytes.

Response in the event of an error

The number and type of parameters in the ALLOC call are checked at compile time for conformance with the dynamic array declaration.

If a call of the ALLOC subprogram is followed by a second call with the same actual arguments, then this second invocation is ignored. In all other error cases a runtime error (fatal error) will be the result.

If a dynamically declared array is addressed but no memory has been assigned to this array by means of CALL ALLOC, undefined execution is the result. Such arrays can only be detected if TESTOPT=(BOUNDS) has been defined as the debug option.

10.6.3 Releasing memory (CALL DEALLOC)

The memory for a dynamically created array *arrayname* is released by calling the DEALLOC subprogram:

```
CALL DEALLOC (arrayname)
```

arrayname
Name of the array defined dynamically in the type, DIMENSION or COMMON statement.

Response in the event of an error

If, in a program unit, an array created dynamically by calling ALLOC is not released by the DEALLOC call, memory is not deallocated until the program is terminated. If the memory of a dynamically created array has not yet been allocated or has already been released by the DEALLOC call, the DEALLOC call is ignored and a LIBRARY WARNING message is issued. In all other cases a runtime error (fatal error) will occur.

10.6.4 Interrogating the dimension bounds (CALL GETSHAPE)

Subprogram GETSHAPE is provided for interrogating the dimension bounds of a dynamically declared array. Interrogation of the current dimension bounds is advisable for example in subprograms with dynamically created arrays used as dummy arguments.

CALL GETSHAPE (arrayname,l_1,u_1[,l_2,u_2][,...][,l_n,u_n])

arrayname
Name of the array dynamically specified in the type, DIMENSION or COMMON statement.

l_i,u_i INTEGER*4-Variable. l_i contains as its value the smallest subscript, u_i the largest subscript of the i-th dimension ($1 \le i \le 7$) of the dynamic array *arrayname*. The number n ($1 \le n \le 7$) of the must be equal to the number defined for the dynamic array in the same program unit.

Response in the event of an error

Runtime error messages (LIBRARY ERRORs) are output in the following cases:

- when *arrayname* is not defined as a dynamically created array in the current program unit;
- when no memory was yet assigned to the dynamic array *arrayname* by means of the ALLOC subprogram;
- when the number n of the interrogated lower bounds l_i or upper bounds u_i does not match the number of dimensions defined in the program unit.

10.6.5 Restrictions on programming using dynamically created arrays

Initialization
A dynamically created array cannot be initialized by means of a type statement or a DATA statement. Initialization must take place by means of value assignments or input assignments after the ALLOC subprogram is called.

Overlaying (EQUIVALENCE statement)
A dynamically created array cannot be overlaid by means of the EQUIVALENCE statement using another data item of the same program unit. When a dynamically created array is overlaid with the aid of the EQUIVALENCE statement an error message is output at compile time.

Overlaying (COMMON statement)
In the case of dynamically created arrays in non-initialized COMMON areas, only memory for an array descriptor is reserved. The memory space for the dynamically created array is not reserved until the ALLOC subprogram is called.

In the case of dynamically created arrays in COMMON areas the user must take the following restriction into account:

A dynamic array in a COMMON area can be defined in other program units only by using a dimension bound list with the format (:[,:][,...]). Dimension number and type of the dynamic array must match in all program units in which the array is referenced.

Array elements in the form of actual arguments
If an actual argument is an array element to which an array corresponds as the dummy argument, this array cannot be a dynamic array.

Debugging aids
Dynamic arrays can be referenced with the debugging aid AID (as of V1.0C).

Dynamic arrays as dummy arguments
Only a dynamic array of the same data type with the same number of dimensions and used as the actual argument may be assigned to a dynamic array as the dummy argument.

11 Program interfacing

A program system consists of a main program (the program that is called at system level) and one or more subprograms, which can be written either in the language of the main program or in other languages.

As of FOR1 version 2.2A there are two different ways of providing the requisite program interfacing:
- in accordance with the previous conventions
- in accordance with the program communication interface ILCS (= Inter Language Communication Services).

The LINKAGE={STD|FOR1-SPECIFIC} option allows the user to define the manner in which the linkage is to be effected (see section 4.2.2.6). If compilation is performed using the default value STD, interfacing is in accordance with ILCS.

Contents of this chapter

Section 11.1 describes the new program communication interface ILCS.

Section 11.2 provides information on compatibility when interfacing programs which were generated using different FOR1 versions, and on compatibility when interfacing programs written in different languages.

Section 11.3 describes the execution of and conventions associated with program interfacing.

Section 11.4 explains what the user must take into consideration when link-editing program systems which contain FOR1 subprograms but no FOR1 main program.

Sections 11.5 - 11.7 describe the interfacing of FOR1 programs with COBOL programs (11.5), PLI1 programs (11.6) and C programs (11.7). These sections list the parameter types which are possible in addition to those generally guaranteed by ILCS in each case.

FOR1 programs can also be linked with RPG3 programs and Pascal-XT programs (as of Pascal-XT V2.2A) via ILCS. These language interfaces are not however described separately in the present manual.

FOR1 provides interfacing macros for linking FOR1 and assembly language programs. However, since when using the macros only a restricted ILCS interfacing is possible, they are described in the appendix (A.9) rather than in this chapter. Unrestricted ILCS interfacing of FOR1 and assembly language programs is possible through the linkage macros offered by ASSEMBH as of version 1.1A. These are described in the "ASSEMBH" Reference Manual [10].

11.1 The program communication interface ILCS

ILCS standardizes and simplifies the main functions of communication between the programs of a runtime unit and between runtime unit and operating system in a language-independent fashion.

ILCS is a combination of software and interface convention:
On the one hand it contains runtime routines which are combined in a PLAM library, whilst on the other ILCS also guarantees the communication interface corresponding to the "Standard Linkage Conventions in BS2000"; i.e. each object module generated by a compiler with the ILCS capability is prepared in accordance with the standard linkage conventions for interfacing with programs written in the same language and in different languages.

The library for the ILCS runtime routines is supplied with every compiler having the ILCS capability - as an additional runtime system so to speak.

ILCS offers the following individual functions:

- multilateral convention for interfacing of programs in different languages
- uniform guidelines for event handling
- storage management (stack and heap storages)
- handling of the program mask
- processing of non-local branches

11.1.1 Initialization of the program system

The initialization of a program system takes place in two stages:

- First the main program initiates calling of the ILCS initialization routine. With FOR1 program systems, as long as the main program itself is not an ILCS object, the presence of FOR1 ILCS subprograms also causes the ILCS initialization routine to be called (see examples, program system C).

  Examples:

  Program system A:

  ```
  FOR1 ILCS main program              ──────→   ILCS called
  COBOL ILCS subprogram
  ```

  Program system B:

  ```
  COBOL main program (non-ILCS)       ──────→   ILCS not called
  FOR1 ILCS subprogram
  ```

  Program system C:

  ```
  FOR1 main program (non-ILCS)        ──────→   ILCS called
  FOR1 ILCS subprogram
  ```

- The ILCS initialization routine called then in turn calls all the requisite language-specific initializations so that the language environments required for the entire program system are set up prior to execution of the first program statement.

11.1.2 ILCS environment

Program systems in which the ILCS initialization routine is called, thereby activating ILCS, execute in the ILCS environment.

In ILCS environments, the event handling is performed in accordance with the ILCS convention.

Program systems containing no ILCS module execute according to the previous conventions.

11.1.3 Prosys common data area (PCD)

For internal control of program interfacing in ILCS environments, a common data area PCD which is available to programs in any programming language is provided in addition to the save areas for the individual programs. The size of the PCD is 4096 bytes. The first part of the PCD contains the data areas used by ILCS, including the "program mask" field (in byte 148), which is preset to the value X’0C’. The second part of the PCD contains the programming language areas, each 128 bytes long, which are available to the runtime systems of the different languages.

11.1.4 Program mask handling by ILCS

The program mask for program execution is set to the value of the PCD field "program mask" (preset to X’0C’) during the course of initialization. If it is changed during program execution, it must be reset prior to the next program call or transfer of control to the value of the PCD field "program mask".

11.1.5 Parameter transfer in ILCS program systems

The semantics of the data types exhibit significant differences for the programming languages that can be interfaced by ILCS. Illustrated below are those data types which have the same form of data representation in the individual programming languages and can therefore be transferred as parameters without problems. When using other data types as parameters, a precise knowledge of the relevant form of data storage is essential in order to ensure correct program execution.

Compiler	Data types			
	Binary Word	Floating-point Word	Floating-point Doubleword	String
FOR1	INTEGER*4	REAL*4	REAL*8	CHARACTER*i (fixed length)
COBOL85	PIC S9(i) COMP SYNCHRONIZED 5<=i<=9	COMP-1	COMP-2	USAGE DISPLAY
Pascal-XT	long_integer	short_real	long_real	packed array [<range>]of char
PLI1	BIN FIXED(31) ALIGNED	BIN FLOAT(21) DEC FLOAT(6)	BIN FLOAT(53) DEC FLOAT(16)	CHAR(i) NONVARYING
C	long	float	double	char <var> [<size>]
Columbus-Assembler	F	E	D	C
RPG3	Binary array with 0 decimal places	——	——	Alphanum. array (fixed length)

Table 11-1: Data types which can be exchanged without problems between ILCS programs in different languages

The data must always be stored aligned; i.e. 32-bit integers in binary representation are aligned on a word boundary, floating-point numbers on a word or doubleword boundary, strings on a byte boundary. The lengths of strings are constant and known to the called program.

It is always the addresses of the data that are transferred and not the values themselves. Since addresses are also transferred internally during FOR1 value transfer ("call by value"), nothing changes in the case of this transfer mode either.

The calling program creates a list of the transferred addresses. The number of parameters is transferred in register 0, the address of the list in register 1 (see section 11.3.3 "Register conventions").

FOR1 parameter types not generally guaranteed by ILCS

The following FOR1 parameter types are not generally guaranteed in the case of interfacing of ILCS programs in different languages:

- Parameters with FOR1 data types not contained in table 11-1:

INTEGER *1	COMPLEX*16
INTEGER *2	COMPLEX*32
INTEGER *8	LOGICAL*1
REAL*16	LOGICAL*4
COMPLEX*8	CHARACTER*(n,V)

- NAMELIST lists, statement labels, EXTERNAL subprograms, multi-dimensional arrays

Since, however, parameter transfer by FOR1 remains unchanged, the parameter types not generally guaranteed by ILCS can be transferred to programs written in other languages. The previous restrictions are applicable.

Arrays as parameters for the interfacing of ILCS programs in different languages

When interfacing programs written in different languages, it is possible to transfer arrays with fixed bounds since with such arrays only the start address is transferred and no array descriptor is required.

The transfer of one-dimensional arrays with fixed bounds is generally guaranteed by ILCS if the arrays have an ILCS-compatible data type (cf. table 11-1).

Multi-dimensional arrays can also be used as parameters as previously in the interfacing of programs in different languages. However, since multi-dimensional arrays are not of uniform structure in the different languages and the maximum permissible number of dimensions is different, the transfer of multi-dimensional arrays is not guaranteed by ILCS.

11.1.6 Notes concerning linking of ILCS program systems

In addition to the language-specific runtime libraries, the library SYSLNK.ILCS containing the ILCS runtime routines is required.

Static linking

During static linking it is sufficient as previously to assign the FOR1 runtime library by means of the RESOLVE statement of TSOSLNK. This contains the ILCS initialization routine IT0INITS.

However, since the language-specific runtime libraries can each contain different versions of the ILCS initialization routine, it is advisable to use the ILCS initialization routine from the ILCS library SYSLNK.ILCS for language interfacing. This always has the latest version.

Access to the initialization routine is effected from the SYSLNK.ILCS library when:

(a) this routine is linked in explicitly by means of an INCLUDE statement (INCLUDE IT0INITS, $TSOS.SYSLNK.ILCS), or

(b) the SYSLNK.ILCS library is assigned with the last RESOLVE statement (RESOLVE ,$TSOS.SYSLNK.ILCS).

Dynamic linking

If the program system exclusively contains programs written in the same language, it is sufficient during dynamic linking to assign the FOR1 runtime library by means of SET-TASKLIB, as previously.

Program systems containing programs in different languages can be linked dynamically using DBL (as of BS2000 version 10.0) since with DBL in RUN-MODE=ADVANCED more than one library can be assigned (via the LINK name BLSLIBnn, 00≤nn≤99).

In order to ensure that the very latest ILCS initialization routine is used, the ILCS library SYSLNK.ILCS should also be assigned in addition to the language-specific runtime systems. The LINK name of the ILCS library must receive a lower number here than the LINK names of the language-specific libraries since DBL searches the assigned libraries on ascending numbers.

11.2 Compatibility

11.2.1 Explanation of terms

OLD programs

Programs generated with a FOR1 version ≤ 1.6A.

NXS programs

Programs generated with a FOR1 compiler of version 2.0A/2.1A with the EXTENDED-SYSTEM=NO option.

XS program

Programs generated with a FOR1 compiler as of version 2.2A (as of V2.2A, only XS modules are generated)
or
FOR1 programs compiled using a FOR1 compiler of version 2.0A/2.1A with the EXTENDED-SYSTEM=YES option.

ILCS programs

FOR1 programs compiled using a FOR1 compiler as of version 2.2A with the LINKAGE=STD option. Since only XS modules are generated as of version 2.2A, ILCS programs are always XS programs.

11.2.2 Compatibility when interfacing FOR1 programs

NXS programs, XS (non-ILCS) programs and ILCS programs can be interfaced without problems.

OLD programs can, however, only be interfaced with NXS programs (and with other OLD programs): Direct interfacing with XS programs - and thus with ILCS programs - is not possible.

11.2.3 Compatibility when interfacing programs in different languages

In ILCS environments, all programs participating in language interfacing should be ILCS programs. Only in this way is it possible to guarantee correct error handling and parameter transfer.

In non-ILCS environments, FOR1 ILCS programs and non-ILCS programs in other languages can be combined using the old interface for the other language - but with the following restrictions:

- C programs generated using older C compiler versions (<V2.0) assume that register R1 is retained when INTEGER functions without parameters are called. ILCS INTEGER functions, however, return the function value both in R0 and also in R1.
- In the case of assembly language programs which simulate the FOR1 interface with the aid of FOR1 macros, the parameter values FIRST=1 and LAST=12 should be selected for the IFART (or IFARTO) macro.

11.3 Subprogram interface: Execution and conventions

Once a subprogram is invoked, a series of measures is taken which are executed by the calling program and the called program.

In the case of language interfacing of ILCS programs in an ILCS environment and also a FORTRAN - FORTRAN call, these measures are executed correctly without user intervention

When interfacing FORTRAN programs with non-ILCS programs in other languages, the user must ensure that the interfacing is performed correctly by calling ready-made macros or routines.

11.3.1 Program interfacing sequence

After a subprogram is invoked the following actions are taken:

Timing	Actions taken by calling program	Actions taken by called program
Before entry	- Provision of a data area in which the parameter addresses and information on the parameters are transferred (parameter address list) - Provision of a save area used for storing the contents of registers - Provision of the entry address and the return address - Branch to subprogram	
After entry		- Storage of the contents of registers in the save area provided by the program - Chaining of the save areas to trace the call hierarchy - Only with ILCS: Storage of the address of the save area of the called program in the PCD (Prosys Common Data Area); saving of the old PCD contents - Setting of an indicator which shows whether the subprogram is currently active
Before return		- Only with ILCS: Resetting of address of the save area in the PCD - Setting of the DO loops to inactive, if applicable - Provision of the function value, if applicable - Storage of the return code - Resetting of the old register contents with the exception of R0 and R1 (intended for function values) - Return
After the return	- On computed return: Evaluation of the return code	

Table 11-2: Actions taken by calling and called programs

11.3.2 Structure of the save area

The save area is a buffer area in which the contents of registers are stored when the subprogram is called. The save area is located at the beginning of the data section of a program unit. Before branching to the subprogram, the calling program supplies register 13 with the address of the save area. Register 13 is saved in the save area of the program which has been called.

The format of a save area is as follows:

Byte	Contents

1-4	Byte 1:
	   Bit 1: activity bit (1: program active, 0: program inactive)
	   Bits 2-7: reserved
	   Bit 8 = normally 0
	Byte 2 (only with ILCS):
	   Version = X'01'
	Bytes 3 and 4 (only with ILCS):
	   X'FEFF'

5-8	Start address of the save area of the calling program. In the **first** calling program, this field contains -1.
9-12	Start address of the next (chained) save area, if applicable.
13-16	Contents of register 14
17-20	Contents of register 15
21-24	Contents of register 0
25-28	Contents of register 1
29-32	Contents of register 2
.	
.	
69-72	Contents of register 12
73-76	Address of the Runtime Communication Area (RTCA)
77-80	With ILCS: Address of the PCD; otherwise reserved
81-84	With ILCS: Address of the EHL (Event Handler List): If no EHL is defined, the field contains the value -1; otherwise reserved
85-88	Reserved

Table 11-3: Structure of the save area

The first bit of the save area is an indicator bit with a value of 1, if the program unit is currently active, and 0 if the program unit is inactive. Active program units are those units which have been called but are not yet finished.

Chaining of save areas

By forward and backward chaining of the save areas of the active program units, the call hierarchy can be output in the event of runtime errors. In ILCS environments, the call hierarchy is reproduced only up to a language limit.

Save area of the calling program unit:

Save area of the called program unit:

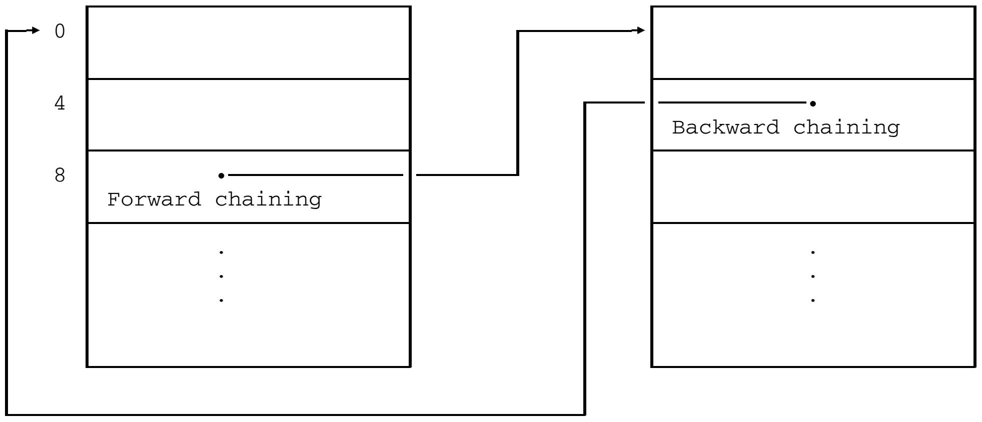


Fig. 11-1: Chaining of the save areas

Forward chaining: Word 2 of the save area of the calling program unit contains the start address of the save area of the called program.

Backward chaining: Word 1 of the save area of the called program unit contains the start address of the save area of the calling program. In the first calling program this word contains 0.

11.3.3 Register conventions

Register loading on program call

The following table gives an overview of the register loading performed by the calling program before the called program is entered.

Register number	Contents
0	Number of parameters
1	Start address of the parameter address list
2 - 12	Program data
13	Start address of the save area of the calling program
14	Address of the return point to the calling program
15	Address of the entry point in the called program
PM	Program mask: Value from PCD field "program mask" (with ILCS)

Table 11-4: Register loading on a subprogram call

The calling program loads these registers before branching to the called program. The register contents of the general registers are stored in the save area provided by the calling program - except for the contents of R13 which are stored in the save area of the called program (backward chaining).

Return code with statement label parameters

Statement label parameters in the form "{&|*}statement-label" are not transferred in the parameter address list, rather a return code is stored in register 1 as follows:
- in the case of a simple return (RETURN statement), 0 is stored
- in the case of a computed return (RETURN expression), the value of "expression" is stored.

Register loading on returning to calling program

The following table gives an overview of the register loading performed by the called program on returning to the calling program. Floating-point registers are not restored.

Register number	Contents
0 - 1	Return values of functions or undefined
2 - 14	As under loading on call
15	Undefined
PM	Program mask: Value from PCD field "program mask" (with ILCS)

Table 11-5: Register loading on return to calling program

Transfer of a function value

On returning from a FUNCTION subprogram, the function value is stored in the following registers:

Function value type	Register
LOGICAL*1	R0
LOGICAL*4	R0
INTEGER*1	R0
INTEGER*2	R0
INTEGER*4	R0
INTEGER*8	F0
REAL*4	F0
REAL*8	F0
REAL*16	F0,F2
COMPLEX*8	F0,F2
COMPLEX*16	F0,F2
COMPLEX*32	F0,F2,F4,F6
CHARACTER*{n|(n,v)}	R1

Table 11-6: Register conventions for various types of functions

ILCS functions of data type INTEGER*{1|2|4} store the function value not only in R0, but also in R1.

In R1 the address of the descriptor for a data item of type CHARACTER is stored. In the case of data items of the type LOGICAL*1, INTEGER*1 and INTEGER*2, the value is stored right-justified in register R0. If, in an assembly language subprogram, such function values calculated by FOR1 are to be taken from register R0, the corresponding instruction "STC R0, <address>" or "STH R0, <address>" must be issued.

11.3.4 Parameter address lists

When a FUNCTION or SUBROUTINE subprogram is called, information can be supplied to the subprogram via a parameter list. The parameter address list corresponds internally to this parameter list.

The parameter address list contains:
- the addresses of the data items to be transferred
- the addresses of descriptors of the data items to be transferred
- information on the type and length of the data items (type indicators) as well as further attributes such as constant, variable, field, etc. (attribute indicators).

The number of parameters transferred is stored in register 0, the address of the parameter address list is stored in register 1. All this information is added to the save area by the calling program.

The addresses of the parameters and information on the type and attributes are generated for all parameters. If a subprogram does not contain any parameters, no parameter address list is generated.

Structure of parameter address list

Fig. 11-2 shows the structure of the parameter address list for an odd number of parameters.

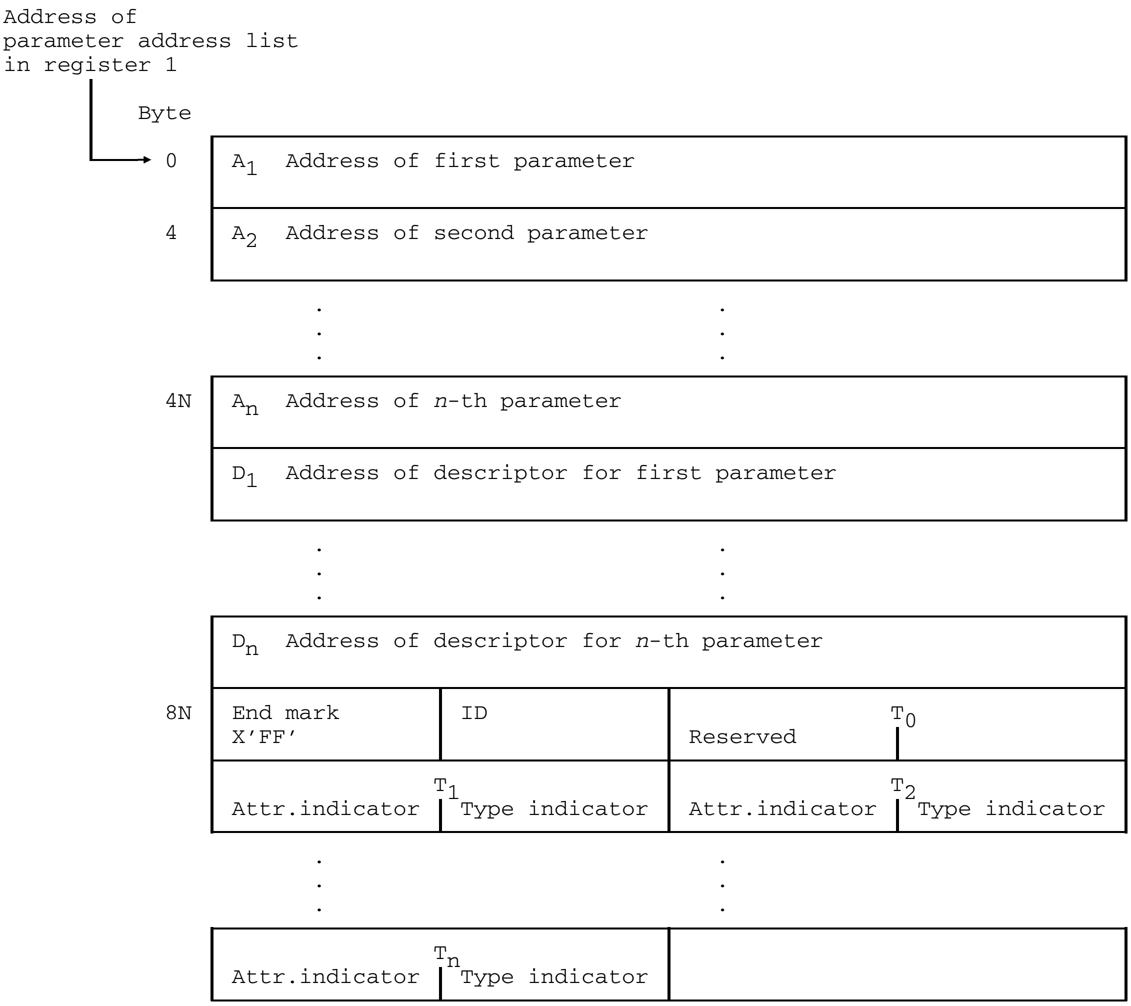

Address of parameter address list in register 1
Byte
0
A_1 Address of first parameter
4
A_2 Address of second parameter
4N
A_n Address of *n*-th parameter
D_1 Address of descriptor for first parameter
D_n Address of descriptor for *n*-th parameter
8N
End mark X'FF'
ID
Reserved
T_0
Attr.indicator
T_1
Type indicator
Attr.indicator
T_2
Type indicator
Attr.indicator
T_n
Type indicator

Fig. 11-2: Structure of parameter address list

The term "reserved" in this chapter implies that the contents of such a field must not be altered (e.g. by assembly language programs). If a non-FOR1 program generates a parameter address list for transfer to a FOR1 program, reserved fields must be deleted in advance by overwriting with binary zeros.

A_i Address of the *i*-th parameter. The most significant bit has the value 0. In the case of parameters of the type INTEGER and LOGICAL with a length less than 4 bytes, a modified address is transferred. When INTEGER*1 and LOGICAL*1 are specified, this is "address -3"; when INTEGER*2, this is "address -2".

D_i Address of the descriptor of the *i*-th parameter. The most significant bit has the value 0. If the parameter does not require a descriptor, the associated word is reserved.

ID

Bits 0-5	Bits 0-5 5 of this byte contain a version identifier for the parameter address list. The identifier for FOR1 versions ≥ V2.0A is B’000000’.
Bit 6	0: No descriptors exist 1: Descriptors exist
Bit 7	0: No attribute and type information exists 1: Attribute and type information exists

T_0 The first byte of this halfword is reserved, the second byte contains:
- (when a FUNCTION subprogram is called) the type of the returned function value
- (when a SUBROUTINE subprogram is called) the type entry NIL (bit pattern 10000)

T_i $(1 \leq i \leq n)$
The high-order byte of this halfword is the attribute indicator; the low-order byte is the type indicator of the corresponding parameter.

Attribute indicator values of a parameter address list:

Bit pattern	Value	Meaning of the attribute indicator
0000	0	Temporary auxiliary variable for actual argument expressions
0001	1	Constant
0010	2	Variable
0011	3	Subprogram specified in an EXTERNAL statement
0100	4	Array
0101	5	Array element (descriptor of an array element or array)
0110	6	Substring
0111	7	FOR1-specific intrinsic function
1000	8	Reserved
1001	9	NAMELIST name
1010	10	Reserved
1011	11	Dynamic array
1100	12	Direct value

Table 11-6: Attribute indicator values

Type indicator values of a parameter address list

Bit pattern	Value	Meaning of the type indicator
00000	0	LOGICAL*1
00001	1	LOGICAL*4
00010	2	INTEGER*1
00011	3	INTEGER*2
00100	4	INTEGER*4
00101	5	INTEGER*8
00110	6	REAL*4
00111	7	REAL*8
01000	8	REAL*16
01001	9	COMPLEX*8
01010	10	COMPLEX*16
01011	11	COMPLEX*32
01100	12	CHARACTER fixed length, Hollerith
01101	13	CHARACTER variable length
01110	14	Subprogram specified in an EXTERNAL statement
01111	15	NAMELIST name
10000	16	NIL (set only in field T_0 for a SUBROUTINE subprogam, not for a FUNCTION subprogram

Table 11-7: Type indicator values of a parameter address list

Hollerith data items are transferred as the CHARACTER field, whose number of items is equal to the length of the Hollerith data item and whose item length is equal to 1.

11.3.5 Descriptors

If at least one descriptor is required for invoking a routine, memory is created for each of the *n* parameters. No descriptors are required for a simple variable. Descriptors are, however, required for arrays, array elements and strings as the actual arguments in the following cases:

- for arrays employing an array as a dummy argument, with open subscript upper bound (*) in the uppermost dimension;
- for array elements employing an array as a dummy argument, with open subscript upper bound (*) in the uppermost dimension;
- for CHARACTER variables of fixed length as the dummy argument;
- for CHARACTER variables with length (*) as the dummy argument (the length is taken from the actual argument);
- for arrays of the CHARACTER type employing as the dummy argument an array of CHARACTER variables with length (*).

Depending on the type of actual argument, different descriptors are generated:

- for arrays and array elements an array descriptor (ADS)
- for array element parameters:
  in addition to the array descriptor (ADS), an array element descriptor (EDS) is generated if the array element parameter is a CHARACTER substring.
- for character strings a string descriptor (SDS)

If a dynamic array is supplied to a subprogram, the entire descriptor is copied into the input code of the subprogram.

Structure of descriptors

String descriptor (SDS)

Byte		
0	—— Reserved ——	
4	—— Reserved ——	
8	—— Reserved ——	
12	Maximum length	Actual length

Array descriptor (ADS)

Byte		
0	—— Reserved ——	
4	Address of first byte following end of array	
8	—— Reserved ——	
12	Reserved	Length of an array element

Descriptor for an array element of type CHARACTER substring (EDS)

Byte		
0	—— Reserved ——	
4	Address of first byte following end of array	
8	—— Reserved ——	
12	Reserved	Actual length of an array element

11.4 Linking program systems without a FOR1 main program

When linking program systems which contain FOR1 subprograms but no FOR1 main program, the following should be noted:

In each generated END statement of a FOR1 subprogram there is an external pointer to the start address of the associated FOR1 main program with the entry `IF@@MPI` (Main Program Initializer).

If FOR1 subprograms are to be linked without a FOR1 main program, the linkage editor will accordingly report an unsatisfied FOR1 external reference `IF@@MPI`.

Linking without a FOR1 main program can be achieved by

- the statements BIND or CONTINUE of TSOSLNK. These statements initiate immediate linkage, even if not all of the external references can be satisfied.
- the LET statement of TSOSLNK or the LET=Y operand in the PROGRAM statement which likewise initiate linking, even if not all external references can be satisfied.
- entering a dummy module `IF@@MPI` in the user library. This dummy module can be generated, for example, by assembling the following statements:

```
IF@@MPI CSECT
IF@@MPI AMODE ANY
IF@@MPI RMODE ANY
        END
```

If a dummy module has been entered in the user library, a FOR1 main program must be linked explicitly with INCLUDE, since the linkage editor might otherwise also link the dummy module in the course of an automatic search.

11.5 Interfacing of FOR1 with COBOL programs

In the following sections a COBOL program is understood to mean a COBOL program compiled with the aid of the COBOL85 compiler (see "COBOL85 User Guide" [15]).

Additional permissible parameter types

In addition to the parameter types generally guaranteed by ILCS (cf. table 11-1), parameters of the following data type are also permissible for FOR1/COBOL interfacing:

COBOL	FOR1
COMP-2 SYNC	INTEGER*8

Table 11-8: COBOL/FOR1 interface: Additional permissible parameter type

Restrictions

- The FOR1 program cannot contain any debugging options.
- It is not possible to work with COB1 and FOR1 components interleaved on the same file since the runtime systems do not exchange information.

11.5.1 FOR1 program calls COBOL subprogram

No precautions are required when calling COBOL subprograms from FOR1 programs.

Call: `CALL subprog (par`$_1$`,...,par`$_n$`)`

11.5.2 COBOL program calls FOR1 subprogram

No precautions are required in ILCS environments when calling a FOR1 subprogram from a COBOL program.

Example of calling a FOR1 subprogram:

```
CALL "FOR1SUB" USING AP1,AP2,AP3
```

Example: COBOL program calls FOR1 subprogram

COBOL program:

```
 ID DIVISION
 PROGRAM-ID.     COBFOR.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION
 01  TABLE-1.
     02  INTEGER1-4
     02  INTEGER1-8                      COMP-2 SYNC.
     02  REAL1-4                         COMP-1 SYNC.
     02  REAL1-8                         COMP-2 SYNC.
     02  LOGICAL1-4          PIC S9(5)  COMP   SYNC.
     02  LOGICAL1-RED        REDEFINES LOGICAL1-4.
         03  LOG1            PIC X.
         03  FILLER          PIC X(3).
     02  CHARACTER1          PIC X(15).
*
 01  TABLE-2.
     02  INTEGER2-8                      COMP-2 SYNC.
     02  INTEGER2-4          PIC S9(5)  COMP   SYNC.
     02  REAL2-8                         COMP-2 SYNC.
     02  REAL2-4                         COMP-1 SYNC.
     02  LOGICAL2-4          PIC S9(5)  COMP   SYNC.
     02  LOGICAL2-RED        REDEFINES LOGICAL2-4.
         03  LOG2            PIC X.
         03  FILLER          PIC X(3).
     02  CHARACTER2          PIC X(15).
*
*
 77  LOGICAL1                PIC X(5).
 77  LOGICAL2                PIC X(5).
*
*
 01  OUTPUT-1               PIC X(25)  VALUE IS
         " COBOL:    INTEGER*4:".
 01  OUTPUT-2               PIC X(25)  VALUE IS
         " COBOL:    INTEGER*8:".
 01  OUTPUT-3               PIC X(25)  VALUE IS
         " COBOL:    REAL*4:".
 01  OUTPUT-4               PIC X(25)  VALUE IS
         " COBOL:    REAL*8:".
 01  OUTPUT-5               PIC X(25)  VALUE IS
         " COBOL:    LOGICAL*4:".
 01  OUTPUT-6               PIC X(25)  VALUE IS
         " COBOL:    CHARACTER*15:".
*
*
*
 PROCEDURE DIVISION.
 BEG.
     MOVE 314             TO INTEGER1-4        INTEGER2-4.
     MOVE 3141592654535  TO INTEGER1-8        INTEGER2-8.
     MOVE 3.141592654    TO REAL1-4           REAL2-4.
     MOVE 3141592654.533 TO REAL1-8           REAL2-8.
     MOVE LOW-VALUES     TO LOG1              LOG2.
```

```
         MOVE "ABCDEFGHIJKL" TO CHARACTER1 CHARACTER2.
         PERFORM OUTP.
*
     FORTRAN.
         CALL "UFOR"        USING    TABLE-1  TABLE-2.
         CALL "UFOR" USING
            INTEGER1-4  INTEGER1-8  REAL1-4  REAL1-8  LOGICAL1-4
            CHARACTER1
            INTEGER2-8  INTEGER2-4  REAL2-8  REAL2-4  LOGICAL2-4
            CHARACTER2.
     OUTP.
     IF LOG1 EQUAL LOW-VALUES THEN
             MOVE "FALSE" TO LOGICAL1
         ELSE
             MOVE "TRUE " TO LOGICAL1.
         IF LOG2 EQUAL LOW-VALUES THEN
             MOVE "FALSE" TO LOGICAL2
         ELSE
             MOVE "TRUE " TO LOGICAL2.
*
         DISPLAY OUTPUT-1-1 INTEGER1-4 " ; " INTEGER2-4 UPON TERMINAL.
         DISPLAY OUTPUT-2-2 INTEGER1-8 " ; " INTEGER2-8 UPON TERMINAL.
         DISPLAY OUTPUT-3-3 REAL1-4    " ; " REAL2-4    UPON TERMINAL.
         DISPLAY OUTPUT-4-4 REAL1-8    " ; " REAL2-8    UPON TERMINAL.
         DISPLAY OUTPUT-5-5 LOGICAL1   " ; " LOGICAL2   UPON TERMINAL.
         DISPLAY OUTPUT-6-6 CHARACTER1 " ; "
                           CHARACTER2 UPON TERMINAL.
     END1.
         STOP RUN.
```

FOR1 subprogram:

```
      SUBROUTINE UFOR (I4,I8,R4,R8,L4,C1,K8,K4,S8,S4,M4,C2)
C
      INTEGER*4   I4,K4
      INTEGER*8   I8,K8
      REAL*4      R4,S4
      REAL*8      R8,S8
      LOGICAL*4   L4,M4
      CHARACTER*15 C1,C2
C
      WRITE(2,10) I4,K4
      WRITE(2,20) I8,K8
      WRITE(2,30) R4,S4
      WRITE(2,40) R8,S8
      WRITE(2,50) L4,M4
      WRITE(2,60) C1,C2
C
      I4 = I4 * 2
      K4 = K4 * 2
      I8 = I8 * 2
      K8 = K8 * 2
      R4 = R4 * 2.
      S4 = S4 * 2.
      R8 = R8 * 2.
      S8 = S8 * 2.
      L4 = .NOT. L4
      M4 = .NOT. M4
      C1='MNOPQRSTUVWX'
      C2='YZ!"@$%&/()>'
C
      WRITE(2,10) I4,K4
      WRITE(2,20) I8,K8
      WRITE(2,30) R4,S4
      WRITE(2,40) R8,S8
      WRITE(2,50) L4,M4
      WRITE(2,60) C1,C2
C
 10   FORMAT (1X,'FORTRAN:   INTEGER*4:  ',I18,' ; ',I18)
 20   FORMAT (1X,'FORTRAN:   INTEGER*8:  ',I18,' ; ',I18)
 30   FORMAT (1X,'FORTRAN:   REAL*4   :  ',G18.8,' ; ',G18.8)
 40   FORMAT (1X,'FORTRAN:   REAL*8   :  ',G18.8,' ; ',G18.8)
 50   FORMAT (1X,'FORTRAN:   LOGICAL*4:  ',L18,' ; ',L18)
 60   FORMAT (1X,'FORTRAN:   CHARACTER*15:  ',A,' ; ',A)
C
      RETURN
      END
```

The following data is output when the COBFOR program is executed (the program COBFOR is contained in a file of the same name):

```
/**START-PROG COBFOR**
%  BLS0500 PROGRAM 'COBFOR' VERSION '' OF '91-07-30' LOADED
 COBOL:    INTEGER*4:    00314 ; 00314
 COBOL:    INTEGER*8:    +.314159265453500E+13 ; +.314159265453500E+13
 COBOL:    REAL*4:       +.314159E+01 ; +.314159E+01
 COBOL:    REAL*8:       +.314159265453300E+10 ; +.314159265453300E+10
 COBOL:    LOGICAL*4:    FALSE ; FALSE
 COBOL:    CHARACTER*15: ABCDEFGHIJKL    ; ABCDEFGHIJKL
FORTRAN:   INTEGER*4:                314 ;                314
FORTRAN:   INTEGER*8:        3141592654535 ;      3141592654535
FORTRAN:   REAL*4   :        3.1415930     ;      3.1415930
FORTRAN:   REAL*8   :       0.31415927D+10 ;     0.31415927D+10
FORTRAN:   LOGICAL*4:                  F ;                  F
FORTRAN:   CHARACTER*15:  ABCDEFGHIJKL    ; ABCDEFGHIJKL
FORTRAN:   INTEGER*4:                628 ;                628
FORTRAN:   INTEGER*8:        6283185309070 ;      6283185309070
FORTRAN:   REAL*4   :        6.2831860     ;      6.2831860
FORTRAN:   REAL*8   :       0.62831853D+10 ;     0.62831853D+10
FORTRAN:   LOGICAL*4:                  T ;                  T
FORTRAN:   CHARACTER*15:  MNOPQRSTUVWX    ; YZ!"@$%&/()>
 COBOL:    INTEGER*4:    00628 ; 00628
 COBOL:    INTEGER*8:    +.628318530907000E+13 ; +.628318530907000E+13
 COBOL:    REAL*4:       +.628319E+01 ; +.628319E+01
 COBOL:    REAL*8:       +.628318530906600E+10 ; +.628318530906600E+10
 COBOL:    LOGICAL*4:    TRUE  ; TRUE
 COBOL:    CHARACTER*15: MNOPQRSTUVWX    ; YZ!"@$%&/()>
```

11.6 Interfacing of FOR1 with PLI1 programs

Language interfacing of FOR1 with PLI1 programs is accomplished with the aid of parameters and external data (see "PLI1" User Guide [37]). Up to 255 parameters can be transferred.

Additional permissible parameter types

In addition to the parameter types generally guaranteed by ILCS (cf. table 11-1), parameters of the following data type are also permissible for FOR1/PL1 interfacing:

FOR1	PLI1 (only ALIGNED data)	
REAL * 16	FLOAT BINARY (p) FLOAT DECIMAL (p)	with 53 < p ≤ 109 with 16 < p ≤ 33
COMPLEX * 8	COMPLEX FLOAT BIN (p) COMPLEX FLOAT DEC (p)	with p ≤ 21 with p ≤ 6
COMPLEX * 16	COMPLEX FLOAT BIN (p) COMPLEX FLOAT DEC (p)	with 21 < p ≤ 53 with 6 < p ≤ 16
COMPLEX * 32	COMPLEX FLOAT BIN (p) COMPLEX FLOAT DEC (p)	with 53 < p ≤ 109 with 16 < p ≤ 33
LOGICAL * 1	BIT (8)	
LOGICAL * 4	BIT (32)	
CHARACTER n variable length	CHARACTER (n) VARYING	

Table 11-10: PLI1/FOR1 interfacing: Additional permissible parameter types

Special considerations

1. If a procedure is EXTERNAL, it must also be a procedure in the PLI1 program.
2. PLI1 fields (DIMENSION) can be passed to FOR1 fields as long as they share memory. If the items are strings of characters, they must have the attribute NONVARYING. Fields cannot be passed as functional values.

   Multi-dimensional arrays in FOR1 are stored by line, as opposed to the fields in PLI1, which are stored by column. Index overlaying of an array in PLI1, e.g. using B (i, j, k) results in the same item being accessed in FOR1 when A (i, j, k) was used.

3. Declarations which use * are also permissible in certain places for the transfer of parameters to FOR1. Entry of current values in the data descriptions to be passed is then taken care of.
4. A COMMON block in FOR1 and a PLI1 variable with the attribute STATIC EXTERNAL are stored in static memory. If the two of them have the same names, they are arranged one above the other and have the same effect as two STATIC EXTERNAL variables in PLI1: Allocation of a value to one of the variables therefore means that the same value is also assigned to the other variable.

11.6.1 FOR1 program calls PLI1 subprogram

No precautions are required in ILCS environments when calling a PLI1 program from a FOR1 program.

– Statement in FOR1:

```
CALL name (par1,...,parn)
```

– Statement in PLI1:

```
      ⎡PROCEDURE⎤
name: {          } (par1,...,parn) OPTIONS (ILCS);
      ⎣ENTRY    ⎦
```

Example: FOR1 program calls PLI1 subprogram

A PLI1 subprogram UPROG called from a FOR1 program could have the following structure:

```
UPROG:    PROC (A)    OPTIONS (ILCS);

       DCL A  DIMENSION (4,5,6) PARAMETER...;
       DCL B  DIM (6,5,4) DEF A (3SUB,2SUB,1SUB)...;

       /* B IS USED IN PLI */

END;
```

In the calling FOR1 program the field to be transferred is declared with
`DIMENSION A (6,5,4)`

DEFINED variables with iSUB entry **cannot** be used together with GET DATA and PUT DATA.

11.6.2 PLI1 program calls FOR1 subprogram

No precautions are required in ILCS environments when calling a FOR1 subprogram from a PLI1 main program.

- Statements in PLI1:

```
DCL forspro ENTRY OPTIONS (ILCS);
CALL forspro (par1,...parn);
```

`forspro`	Name of the FOR1 subprogram

- Statements in FOR1:

```
⎡SUBROUTINE forspro (par1,...parn)⎤
⎨                                 ⎬
⎣FUNCTION forspro (par1,...parn)  ⎦
```

Example: PLI1 main program calls FOR1 subprogram

A PLI1 program that calls a FOR1 subprogram could have the following structure:

```
PLIROUT:             PROC OPTIONS(MAIN);

       DCL    FORSP ENTRY (DIM(4,5,6)...) OPTIONS(ILCS);
       DCL A  DIMENSION (4,5,6)...;
       DCL B  DIMENSION (6,5,4)DEF A (3SUB,2SUB,1SUB)...;

       /* B IS USED IN PLI */
       CALL FORSP (A);

END;
```

In the called FOR1 program the field to be transferred is declared with
`DIMENSION A (6,5,4)`

DEFINED variables with iSUB entry **cannot** be used together with GET DATA and PUT DATA.

11.7 Interfacing of FOR1 with C programs

Language interfacing of FOR1 and C programs relates to FOR1 programs compiled with a FOR1 compiler version as of V2.0A.

Additional permissible parameter types

In addition to the parameter types generally guaranteed by ILCS (cf. table 11-1), parameters with the following data types are also permissible for FOR1/C program interfacing:

FOR1	C
`INTEGER*8`	`double`
`LOGICAL*1`	`char`
`LOGICAL*4`	`int`
`COMPLEX*8`	`struct {float; float;}`
`COMPLEX*16`	`struct {double; double;}`
`Array` [1]	`Array`

[1] C arrays are arranged by lines, FORTRAN arrays, however, are arranged in columns. This different internal organization must be taken into account when multi-dimensional arrays are used; for example, if an array in a C program was declared using "type array [2] [3]", a dimension must be specified by "ARRAY (3,2)" (see also example in section 11.7.1).

Table 11-11: FOR1/C interfacing: Additional permissible parameter types

Notes on linkage

When linking a C main program to FOR1 subprograms, the PROG statement with parameter LET=Y or the BIND statement should be used.

When mathematical functions are used, the following must be noted: Mathematical functions (ATAN, ASIN etc.) are provided in both the C and FOR1 runtime systems. The parameter supply and the data types are different. The entry point not being used at the time must be excluded with an EXCLUDE statement. However if the same input name is to be used in both the C and also the FOR1 section, the C or FOR1 section in question must be linked into a prelinked module using the associated runtime function.

11.7.1 C program calls FOR1 subprogram

This section describes the connection of external FOR1 subprograms to C programs. The FOR1 subprogram called by the C program can be a FUNCTION or a SUBROUTINE subprogram.

Parameter transfer

In FOR1, the addresses of the parameters are always transferred to the parameter list, even though value transfer is explicitly requested. The distinction between the transfer modes "call by reference" and "call by value" does not go into effect until the FOR1 subprogram is called, and depends on the type of dummy arguments defined therein.

In C, the values of the parameters are always entered in the parameter list, with the exception of arrays (vectors) and pointers.
Therefore, when a FOR1 subprogram is called, the addresses of the data items to be transferred must be specified as the actual arguments in C (e.g. using the address operator &: &par). From a technical standpoint, the address of the required parameter is transferred as the value.
Array names can be directly specified as parameters, since the value of an array is, by definition, its address.

For the mode of transfer "call by value" in FOR1, the current values of the parameters are returned to the actual arguments at the end of the FOR1 subprogram (see "FOR1" Reference Manual [21]). The result is that the actual arguments can be modified in this transfer mode as well, after returning from the FOR1 subprogram.

When interfacing C with FOR1, the FOR1 language element for computed returns (RETURN expression) may not generally be used.

Restrictions on the transfer of parameters

In addition to the different transfer modes, there are further differences pertaining to the format of the parameter lists in C and FOR1.

In the case of character strings, arrays and array elements, in certain cases FOR1 transfers and expects the addresses of descriptors in addition to the addresses of the actual data items (see section 11.3.5). Attribute and type indicators are also transferred.

As a result of these differences, the following restrictions result for the FOR1 subprogram:

– no DEBUG output (CALL DEBUG or error situation),

– no arrays as dummy arguments, whose dimension bound is supplied by the calling program (* character as the upper dimension bound),

– no dummy arguments in conjunction with the TESTOPT operands ARG, BOUNDS and SUBSCR,

– no CHARACTER substrings as dummy arguments.

If the FOR1 subprogram fulfills the above-mentioned conditions, it can be called without problems, as FOR1 then does not require any type indicators or aggregate descriptors.

Example: C program calls FOR1 subprogram

C program CMAIN

```
#include <stdio.h>

main()
{
 char carray [2] [3];
 int  lc,lc1,lc2;                    /* loop counter */
 for (lc = 0;lc <= 2;++lc) {
    carray [0] [lc] = 'A';
    carray [1] [lc] = 'B';
}for (lc1 = 0; lc1 <= 1; ++lc1)      /* output carray */
   for (lc2 = 0; lc2 <= 2; ++lc2)
     printf("C:   carray(%d,%d) = %c\n", lc1, lc2, carray [lc1] [lc2] );
 forsub(carray);                     /* Call FOR1 subprogram forsub */
 for (lc1 = 0; lc1 <= 1; ++lc1)      /* output carray */
   for (lc2 = 0; lc2 <= 2; ++lc2)
     printf("C:    carray(%d,%d) = %c\n", lc1, lc2, cfeld [lc1] [lc2] );
}
```

FOR1 program FORUP

```
C     FORTRAN SUBPROGRAM

      SUBROUTINE FORSUB(CARRAY)
      CHARACTER*1  CARRAY(3,2)
      INTEGER*4    I,J
* OUTPUT CARRAY
      DO 1 I=1,2
       DO 1 J=1,3
1         WRITE (2,10) J, I, CARRAY(J,I)
* SUPPLY CARRAY WITH NEW VALUES
      DO 2 I=1,3
            CARRAY(I,1) = 'C'
2           CARRAY(I,2) = 'D'
* OUTPUT CARRAY
      DO 3 I=1,2
       DO 3 J=1,3
3         WRITE (2,10) J, I, CARRAY(J,I)
10    FORMAT (' FOR1: CARRAY(',I1,',',I1,') = ',A)
      RETURN
      END
```

Tracer log for compiling, linking and program execution

```
(IN)     START-PROG $FOR1
(OUT)    %  BLS0500 PROGRAM 'FOR1', VERSION '2.2A00' OF '91-06-05' LOADED
(OUT)    %  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG ...
(OUT)     FOR1:   V2.2A00 READY, GIVE COMPILER OPTION
(IN)     COMOPT SRC=FORUP,MODULE-LIBRARY=PLAM.MODFOR1,END
(OUT)     FOR1: COMPILER NOT PRELOADED (BAD LOAD PERFORMANCE)
(OUT)     FOR1: NO ERRORS DURING COMPILATION OF P.U. FORSUB
(OUT)     END OF  F O R 1  COMPILATION;  CPU TIME USED: 0.611 SEC.

(IN)     START-PROG $C
(OUT)    %  BLS0500 PROGRAM 'C', VERSION '2.0A' OF '91-05-27' LOADED
(OUT)    %  CCM9992  BEGIN C  V2.0A00
```

```
(OUT)    %  CCM9993  Copyright (C) Siemens Nixdorf Informationssysteme AG 1991.
(OUT)    %  CCM9994  All rights reserved.
(IN)     COMPILE SOU=CMAIN,MODULE-LIBRARY=PLAM.MODC
(IN)     END
(OUT)    %  CCM9995  NOTES: 0 WARNINGS: 0 ERRORS: 0
(OUT)    %  CCM9997  MODULES GENERATED
(OUT)    %  CCM9998  END   C TIME USED = 4.1822

(IN)     START-PROG $TSOSLNK
(OUT)    %  BLS0500 PROGRAM 'TSOSLNK', VERSION '21.0D17' OF '91-04-25' LOADED
(IN)     PROG CFOR1,FILENAM=C.FORUP,LOADPT=*XS,LET=Y
(IN)     INCLUDE (CMAIN#,CMAIN@),PLAM.MODC
(IN)     RESOLVE ,PLAM.MODFOR1
(IN)     RESOLVE ,$CLIB
(IN)     RESOLVE ,$FOR1MODLIBS
(IN)     RESOLVE ,$TSOS.SYSLNK.ILCS
(IN)     BIND
(OUT)     UNRESOLVED EXTRNS:
(OUT)     IF@@MPI
(OUT)    %  LNK0055 PROGRAM BOUND IN SPITE OF UNRESOLVED EXTERN'S
(OUT)    %  LNK0062 THE PHASE CAN BE LOADED ON XS SYSTEM ONLY
(OUT)    %  LNK0503 PROGRAM FILE 'C.FORUP' WRITTEN
(OUT)    %  LNK0504     18 PAM PAGES USED. TSOSLNK RUN FINISHED

(IN)     SET-TASKLIB $FOR1MODLIBS

(IN)     START-PROG C.FORUP
(OUT)    %  BLS0500 PROGRAM 'CFOR1', VERSION ' ' OF '91-07-30' LOADED
(OUT)    C:    carray(0,0) = A
(OUT)    C:    carray(0,1) = A
(OUT)    C:    carray(0,2) = A
(OUT)    C:    carray(1,0) = B
(OUT)    C:    carray(1,1) = B
(OUT)    C:    carray(1,2) = B
(OUT)    FOR1: CARRAY(1,1) = A
(OUT)    FOR1: CARRAY(2,1) = A
(OUT)    FOR1: CARRAY(3,1) = A
(OUT)    FOR1: CARRAY(1,2) = B
(OUT)    FOR1: CARRAY(2,2) = B
(OUT)    FOR1: CARRAY(3,2) = B
(OUT)    FOR1: CARRAY(1,1) = C
(OUT)    FOR1: CARRAY(2,1) = C
(OUT)    FOR1: CARRAY(3,1) = C
(OUT)    FOR1: CARRAY(1,2) = D
(OUT)    FOR1: CARRAY(2,2) = D
(OUT)    FOR1: CARRAY(3,2) = D
(OUT)    C:    carray(0,0) = C
(OUT)    C:    carray(0,1) = C
(OUT)    C:    carray(0,2) = C
(OUT)    C:    carray(1,0) = D
(OUT)    C:    carray(1,1) = D
(OUT)    C:    carray(1,2) = D
(OUT)    %  CCM0998  used CPU-time 0.0280 seconds
```

The message concerning the unresolved external reference IF@@MPI is the result of linkage without the FOR1 main program and can be ignored in this case.

11.7.2 FOR1 program calls C function

This section describes the connection of C functions to FOR1 programs.

C functions which provide a function value in accordance with their data type can be called in FOR1 both within expressions and also by using the CALL statement.

C functions of the type "void" should only be called with the CALL statement.

Calling a main function

It is possible to call a main function by using MAIN as the entry address.
If more than one main function exists, the selection of the required main function can be ensured by explicit linking-in of the corresponding object module.

Redirection of the standard input/output files and parameter transfers to the main function are not possible.

Parameter transfer

In FOR1, the addresses of the parameters are always entered in the parameter list. The dummy arguments of the C function are therefore to be defined as pointers to the data items to be transferred (<type> *par); array names can be directly specified since the value of an array is, by definition, its address.

When interfacing C with FOR1, the FOR1 language element for computed returns (RETURN expression) may not generally be used.

Example: FOR1 program calls C function

FOR1 program FORMAIN

```
C     FORTRAN-MAINPROGRAM

      PROGRAM MAIN
      INTEGER*4 X, Y, Z
      INTEGER*4 ADR
      X = 5
      Y = 4
      WRITE(2,100) X, Y, Z
100   FORMAT('  X = ', I4, '  Y = ', I4, ' Z = ', I4)
      CALL CSUB(X, Y, Z)
      WRITE(2,200) Z
200   FORMAT('  SUM = ', I4)
      STOP
      END
```

C program CUP

```
void
csub(a, b, c)
int *a, *b, *c;
{
  printf("c-program: a = %d, b = %d, c = %d\n", *a, *b, *c);
  *c = *a + *b;
  printf("c-program: sum = %d\n", *c);
}
```

Tracer log for compiling, linking and program execution

```
(IN)      START-PROG $FOR1
(OUT)     %  BLS0500 PROGRAM 'FOR1', VERSION '2.2A00' OF '91-06-05' LOADED
(OUT)     %  BLS0552 COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG ...
(OUT)      FOR1:   V2.2A00 READY, GIVE COMPILER OPTION
(IN)      COMOPT SRC=FORMAIN,MODULE-LIBRARY=PLAM.MODFOR1,END
(OUT)      FOR1: COMPILER NOT PRELOADED (BAD LOAD PERFORMANCE)
(OUT)      FOR1: NO ERRORS DURING COMPILATION OF P.U. MAIN
(OUT)      END OF  F O R 1  COMPILATION;  CPU TIME USED: 0.598 SEC.

(IN)      START-PROG $C
(OUT)     %  BLS0500 PROGRAM 'C', VERSION '2.0A' OF '91-05-27' LOADED
(OUT)     %  CCM9992  BEGIN C  V2.0A00
(OUT)     %  CCM9993  Copyright (C) Siemens Nixdorf Informationssysteme AG 1991.
(OUT)     %  CCM9994  All rights reserved.
(IN)      COMPILE SOU=CUP,MOD-LIB=PLAM.MODC
(IN)      END
(OUT)     %  CCM9995  NOTES: 0 WARNINGS: 0 ERRORS: 0
(OUT)     %  CCM9997  MODULES GENERATED
(OUT)     %  CCM9998  END   C TIME USED = 3.8963
```

```
(IN)     START-PROG $TSOSLNK
(OUT)    %  BLS0500 PROGRAM 'TSOSLNK', VERSION '21.0D17' OF '91-04-25' LOADED
(IN)     PROG PROG,FILENAM=C.FORMAIN
(IN)     INCLUDE MAIN,PLAM.MODFOR1
(IN)     INCLUDE (CUP#,CUP@),PLAM.MODC
(IN)     RESOLVE, $CLIB
(IN)     RESOLVE ,$FOR1MODLIBS
(IN)     RESOLVE ,$TSOS.SYSLNK.ILCS
(IN)     BIND
(OUT)    %  LNK0500 PROGRAM BOUND
(OUT)    %  LNK0503 PROGRAM FILE 'C.FORMAIN' WRITTEN
(OUT)    %  LNK0504     17 PAM PAGES USED. TSOSLNK RUN FINISHED

(IN)     SET-TASKLIB $FOR1MODLIBS

(IN)     START-PROG C.FORMAIN
(OUT)    %  BLS0500 PROGRAM 'PROG', VERSION ' ' OF '91-07-30' LOADED
(OUT)    BS2000  F O R 1  :  FORTRAN PROGRAM "MAIN"
(OUT)    STARTED ON 1991-07-30 AT 13:47:15
(OUT)     X =    5  Y =    4 Z =    0
(OUT)    c-program: a = 5, b = 4, c = 0
(OUT)    c-program: sum = 9
(OUT)     SUM =    9
(OUT)    STOP AT STMT 11 IN MAIN
(OUT)    BS2000  F O R 1 : FORTRAN PROGRAM "MAIN   " ENDED PROPERLY AT 13:47:16
(OUT)    CPU - TIME USED :       0.0091 SECONDS
(OUT)    ELAPSED TIME    :       0.0150 SECONDS
```

11.7.3 Common file processing

Standard input/output files

The standard input/output files can be addressed both in the C section and in the FOR1 section of the program.

Non-standard files

Files processed together must be opened in both the C section and the FOR1 section. Their processing is accomplished internally with the aid of different FCBs.

Since processing of a shared file takes place via separate FCBs, interleaved reading to the C and FOR1 sections is not possible. All characters of the file are supplied to the C section and the FOR1 section.

12 Function pool FPOOL

The FPOOL concept makes it possible to extend the call interface check, which was previously performed on call interfaces for intrinsic functions only. If, for example, a source program contains the statements

```
CHARACTER *5   X
Y = SIN(X)
```

FOR1 will issue an error message, because an argument of the type CHARACTER is not allowed in the SIN function call. Furthermore arguments in user-own subprogram calls (CALL...) have not yet been checked by the compiler.

By accessing FPOOL files containing information on call interfaces, the compiler is now able to include subprogram interfaces (SUBROUTINEs and FUNCTIONs) in its error analysis. In the case of FUNCTIONs, only the parameters are checked and not the type and length of the FUNCTION.

The FPOOL file is compiler independent, unlike the intrinsic table, which can be chekked on the basis of the intrinsic function calls. Thus the FPOOL concept makes it possible to create a central FPOOL with properties similar to intrinsic type properties, but which need not be implemented in the FOR1 compiler or FOR1 runtime system.

More specifically, the user may create private FPOOLs, thus enabling call interfaces of user-own subprograms to be checked by the compiler (see also section 12.3).

A function pool is implemented in two files: an object module library from which FPOOL routines are linked into the load module, and the FPOOL file proper which contains descriptions of the call interfaces for these routines.

A call interface is defined by number, sequence, type, length, alignment, dimension, transfer and return type of parameters.

The interface description in an FPOOL file may also contain information about language inhibitions. With regard to language interfacing, such inhibitions may be useful if two languages are incompatible, e.g. in their call and return mechanisms.

The Fortran90 compiler will no longer support interface checking by FPOOL since language elements are available in Fortran90 itself for this purpose. The functions of the central FPOOL can, however, still also be utilized by the Fortran90 compiler, but without interface checking by FPOOL.

Example:

In the subprogram DIALOG, the arguments X and Y must be used, where X must be of the type INTEGER * 1, and Y of the type CHARACTER * 10. During compilation of a source program containing the statements

```
REAL * 4          X         (1)
CHARACTER * 10    Y
CALL DIALOG  (X,Y)          (2)
```

the FOR1 compiler, without FPOOL processing, will not issue an error message. The FPOOL connection, however, makes it possible to check in statement (2) whether DIALOG belongs to a user-defined FPOOL. If so, the interfaces will be checked on the basis of the FPOOL file.
If it is specified in this FPOOL file that the first argument must be of the INTEGER * 1 type, FOR1 will issue an error message (SEVERE) on account of statement (1).

12.1 Controlling FPOOL processing

The inclusion of FPOOL files for checking subprogram call interfaces is controlled by way of the SDF operand FPOOL-LIBRARY or via the FPOOL compiler option.

12.1.1 SDF operand FPOOL-LIBRARY

```
START-FOR1-COMPILER
```

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

The SDF operands and corresponding compiler options are shown in table 2-4.

12.1.2 FPOOL compiler option

```
*COMOPT     FPOOL [=⎧fpoolname                     ⎫]
                    ⎨                              ⎬
                    ⎩([fpoolname[,fpoolname]...])  ⎭
```

fpoolname   Name of an FPOOL file

Only if this option is specified will the compiler carry out an FPOOL operation.

When the COMOPT FPOOL contains a list of FPOOL files, their sequence from left to right represents a search hierarchy; FOR1 searches for the subprogram call name in the FPOOL file which is specified as the first name in the option. If it is not found, the search is continued in the FPOOL file which appears next in the list, etc.

If, in the COMOPT FPOOL, no list or only an empty list of FPOOL files is specified, FOR1 anticipates specification of the FPOOL files to be considered by way of a %FPOOL statement in the source program.

The FPOOL option is valid for one compiler run and, consequently, for all program units involved. It may, however, be changed by the %FPOOL and %NOFPOOL statements.

12.1.3 %FPOOL statements in source program

%FPOOL statement

Only if the FPOOL option is specified will the following source program statements be processed by the FOR1 compiler.

```
%FPOOL        fpoolname [(sproname[,sproname]...)],...
```

fpoolname	Name of an FPOOL file
sproname	Call name of a subprogram whose interfaces are defined in the FPOOL file *fpoolname*.

A %FPOOL statement is only valid in the program unit concerned.

Any search hierarchy set up by the FPOOL option is changed by the %FPOOL statement:
If a list of call names is specified in conjunction with *fpoolname*, the interface descriptions associated with the call names will be sought only in the file *fpoolname*. If a subprogram name *sproname* specified in the %FPOOL statement is not found in the FPOOL file *fpoolname*, a warning is given. In this case no further files are searched and no interface check is performed. If a subprogram name *sproname* is found in two or more %FPOOL statements, first the file containing the first %FPOOL statement is searched and then the file containing second %FPOOL statement etc.

If *fpoolname* is specified without a list of call names, this FPOOL file takes precedence over the search hierarchy defined by the FPOOL option.

The following summary serves to clarify the search hierarchy of the FOR1 compiler:

Assume that a program unit contains the call

```
CALL  function (A,B).
```

If COMOPT FPOOL has been set, FOR1 will search for the interface description associated with *function* in the following way:

1) FOR1 checks whether *function* is included in a list of call names in a %FPOOL statement of this program unit.
   If so, a search is only performed in the appropriate FPOOL file *fpoolname*.

2) If *function* is not associated with any listing of call names, FOR1 will search the FPOOL files specified without a listing of call names in the %FPOOL statements of this program unit.

3) If *function* is not found in any of these FPOOL files, the FPOOL files mentioned in the FPOOL option will be searched.

Only the names of the FPOOL files specified explicitly in the FPOOL option (and not those specified in the %FPOOL statement) are listed in the options list.

%NOFPOOL statement

```
%NOFPOOL (sproname[,sproname]...)
```

sproname	Subprogram call name

A %NOFPOOL statement is only valid in the program unit concerned.

The call names specified in the %NOFPOOL statement are excluded from FPOOL processing.
If call names occurring in one of the specified FPOOL files are specified for other subprograms, then the corresponding call must be excluded from FPOOL processing, since the compiler will otherwise use the corresponding FPOOL entries for checking the call interface.

If subprograms not associated with any of the specified FPOOL files are used, the %NOFPOOL statement may be used to avoid unsuccessful search operations on all specified FPOOL files.

The following diagram illustrates FPOOL processing by the FOR1 compiler:

This figure is not any longer available for the online pdf.

Fig. 12-1: FPOOL implementation in connection with the compilation and linkage of a FOR1 program

12.2 The central FPOOL

The central FPOOL consists of the object module library FOR1.FPOOLLIB and the associated file of interface descriptions FOR1.FPOOL.

All functions available in FOR1.FPOOLLIB are described in alphabetical order.
Each description is subdivided into the following sections:

- Call name (in the title)
- Generic name
- Connect name
- Interface description
- Implementation
- Example of call

All functions are invoked with a CALL statement.

If the COMOPT FPOOL option was specified at compile time, an FPOOL function can be invoked using either the call name or the generic name. The call name is the one used to store the function in FPOOL and perform the interface check.

When a generic name is used, FOR1 will identify the correct call name by the parameters passed (for formats of generic and call names see the summary in section 12.2.18, table 12-1).

FOR1 converts the call name in connection code into the connect name defined in the FPOOL file. This connect name and the entry name of the corresponding object module are identical.

The macro calls used for the implementation are described in the "Executive Macros" manual [26].

Note

If FPOOL functions are used which have parameters of data type INTEGER*1, COMOPT FPOOL must be specified for the compilation since such parameters are transferred differently in the case of FPOOL functions than with FOR1 programs.

12.2.1 FPOOL function ACCOUNTNO

```
ACCOUNTNO
```

Function:	Requests the task's account number
Generic:	TMODE
Connect:	TMODACC

Interface

Number of parameters: 2

1st parameter

Type:	CHARACTER*8
Usage:	OUT
Meaning:	After invocation, contains the account number left-justified, padded with blanks if applicable.

2nd parameter

Type:	CHARACTER*45
Usage:	SCRATCH
Meaning:	Work area of the function; contents before and after invocation not subject to any regulation

Implementation

System macro used: TMODE
Shareable

Example:

```
Parameters:
   CHARACTER*8  ACCOUNTNO
   CHARACTER*45 SCRATCH

Specific call:
   CALL ACCOUNTNO (ACCOUNTNO,SCRATCH)

Generic call:
   CALL TMODE (ACCOUNTNO,SCRATCH)
```

12.2.2 FPOOL Function DIALOG

When using this function, compilation must take place with COMOPT=FPOOL since parameters of data type INTEGER*1 are transferred differently with FPOOL functions than with FOR1 programs.

```
DIALOG
```

Function:	Requests the task type
Generic:	TMODE
Connect:	DIALOG

Interface

Number of parameters: 2

1st parameter

Type:	INTEGER*1
Usage:	OUT
Meaning:	After invocation, contains the task type in numeric representation. The individual values have the following meanings:

0	Batch job	
2	Terminal	8103
4	Video terminal	8150
17	TRANSDATA	8418,8415
21	Video terminal	8151
22	Video terminal	8152
23	Printer terminal 8110	
24	Data terminal	8161/54
25	Data terminal	8161/64
26	Data terminal	8161/80
44	Data terminal	8162
45	Data terminal	8160/80
53	Data terminal	9750

2nd parameter

Type:	CHARACTER*10
Usage:	SCRATCH
Meaning:	Work area of the function; contents before and after invocation not subject to any regulation

Implementation

System macro used: TMODE
Shareable

Example:

```
Parameters:
   INTEGER*1    TASKTYP
   CHARACTER*10 SCRATCH

Specific call:
   CALL DIALOG (TASKTYP,SCRATCH)

Generic call:
   CALL TMODE (TASKTYP,SCRATCH)
```

12.2.3 FPOOL function ELIMCHR

```
ELIMCHR
```

Function: Eliminates a record by specifying the CHARACTER key
Generic: IDELETE
Connect: ELIMCHR

Interface
Number of parameters: 3

1st parameter
Type: INTEGER [1)]
Usage: IN
Meaning: Number of the input/output unit with which the file must be linked.

2nd parameter
Type: CHARACTER*n [2)]
Usage: IN
Meaning: CHARACTER-ISAM key of the record to be eliminated

3rd parameter
Type: INTEGER*4
Usage: OUT
Meaning: IOSTAT return code via which information about the result of the DELETE call is provided

1) Length arbitrary, however when called via the connect name without specifying COMOPT FPOOL=FOR1.FPOOL, the first parameter must be of type INTEGER*4.

2) Length of the second parameter (see "FOR1" Reference Manual [21]), CHARACTER-ISAM key.

Implementation

System macro used: ELIM
Shareable

When the second parameter is used, note that the file must be opened in accordance with the ISAM key, and using `ACCESS= 'DIRECT, CHARACTER'`

Example:

```
        INTEGER*4   UNIT/20/
        CHARACTER*8 CKEY
        INTEGER*4   RETCODE

        OPEN (UNIT, ACCESS='DIRECT,C',FILE='FILE')
        CKEY='AAAAAAAA'

Specific call:
        CALL ELIMCHR (UNIT,CKEY,RETCODE)

Generic call:
        CALL IDELETE (20,'AAAAAAAA',RETCODE)
```

12.2.4 FPOOL function ELIMINT

```
ELIMINT
```

Function:	Eliminates a record by specifying the INTEGER key
Generic:	IDELETE
Connect:	ELIMINT

Interface

Number of parameters: 3

1st parameter

Type:	INTEGER 1)
Usage:	IN
Meaning:	Number of the input/output unit, with which the file must be linked

2nd parameter

Type:	INTEGER 1)
Usage:	IN
Meaning:	INTEGER-ISAM key of the record to be eliminated

3rd parameter

Type:	INTEGER*4
Usage:	OUT
Meaning:	IOSTAT return code which provides information about the result of the DELETE call

1) INTEGER of any length, however when calling via the connect name without specifying COMOPT FPOOL=FOR1.FPOOL, data type INTEGER*4 must be used.

Implementation

System macro used: ELIM
Shareable

When the second parameter is used, note that the file must be opened in accordance with the ISAM key and using `ACCESS= 'DIRECT,[,I]'`

Example:

```
        INTEGER*4   UNIT/20/
        INTEGER*4   IKEY
        INTEGER*4   RETCODE

        OPEN (UNIT, ACCESS='DIRECT',FILE='COLLECTION')
        IKEY=5

Specific call:
        CALL ELIMINT (20,10*5+3, RETCODE)

Generic call:
        CALL IDELETE (UNIT,10*IKEY+3, RETCODE)
```

12.2.5 FPOOL function FCMD

```
FCMD
```

Function:	Issues BS2000 commands
Generic:	-
Connect:	FP@CMD

Interface

Number of parameters: 5

1st parameter

Type: CHARACTER*1

Usage: OUT

Meaning: After invocation, contains the error code (of SVC 88). The individual values have the following meanings:

0	BS2000 command successfully completed
1	Insufficient storage space
2	Memory addresses invalid
3	System message truncated
4	Error on execution of the BS2000 command (e.g. BS2000 command with invalid format)
5	BS2000 command does not correspond to BS2000 syntax

2nd parameter

Type:	CHARACTER*512
Usage:	IN
Meaning:	BS2000 command (maximum length 512) (e.g. /SET-FILE-LINK LINK-NAME=DSET20, FILE-NAME=FILE.20, ACCESS-METHOD=SAM)

3rd parameter

Type:	CHARACTER*1024
Usage:	OUT
Meaning:	After invocation, contains the system message.

Structure:

1 - 2	Length of the 3rd parameter
3 - 4	Empty
5 - 1024	System message

4th parameter

Type:	CHARACTER*1
Usage:	IN
Meaning:	Control of system message output. Possible entries: Y/N

Y	System messages are output to SYSOUT and in the output field (3rd parameter).
N	System messages are output in the output field only.

This parameter is only effective for BS2000 versions ≥ 8.0; with earlier versions, Y is assumed even if N is entered.

5th parameter

Type:	CHARACTER*1552
Usage:	SCRATCH
Meaning:	Work area of the function; contents before and after call not subject to any regulation

Implementation

System macro used: CMD
Shareable

Example:

```
   Parameters:

       CHARACTER CPAR1  * 1  /'0'/
       CHARACTER CPAR2  * 512       1)
       CHARACTER CPAR3  * 1024
       INTEGER   CPAR31 * 2
       CHARACTER CPAR32 * 2
       CHARACTER CPAR33 * 1020
       EQUIVALENCE (CPAR3,CPAR31),(CPAR3(3:4),CPAR32),(CPAR3(5:),CPAR33)
       CHARACTER CPAR4  * 1
       CHARACTER CPAR5  * 1552

       1) NOTE:  (512,V) not permitted !


   Specific call:

       CPAR2='/SET-FILE-LINK LINK-NAME=DSET20,FILE-NAME=FILE.20,
      *       ACCESS-METHOD=SAM'
       CPAR4='Y' 2)
      [%FPOOL FOR1.FPOOL(FCMD)]
       CALL FCMD (CPAR1,CPAR2,CPAR3,CPAR4,CPAR5)

       2) System message on SYSOUT desired

   Generic call:

      ____
```

12.2.6 FPOOL function GDATECHAR

```
GDATECHAR
```

Function:	Requests the current data in alpha form
Generic:	GDATE
Connect:	GDATCHR

Interface

Number of parameters: 5

1st parameter

Type:	CHARACTER*2
Usage:	OUT
Meaning:	After invocation, contains the current month in alpha form, e.g. '09' is supplied for the date 20.9.81.

2nd parameter

Type:	CHARACTER*2
Usage:	OUT
Meaning:	After invocation, contains the current day in alpha form, e.g. ’20’ is supplied for the date 20.9.81.

3rd parameter

Type:	CHARACTER*2
Usage:	OUT
Meaning:	After invocation, contains the current year in alpha form, e.g. ’81’ is supplied for the date 20.9.81.

4th parameter

Type:	CHARACTER*3
Usage:	OUT
Meaning:	After invocation, contains the current day of the year in alpha form, e.g. ’263’ is supplied for the date 20.9.81.

5th parameter

Type:	CHARACTER*12
Usage:	SCRATCH
Meaning:	Work area of the function; contents before and after invocation not subject to any regulation

Implementation

System macro used: GDATE
Shareable

Example:

```
Parameters:
   CHARACTER*2  DAY,MONTH,YEAR
   CHARACTER*3  DAYOFYEAR
   CHARACTER*12 SCRATCH

Specific call:
   CALL GDATECHAR (MONTH,DAY,YEAR,DAYOFYEAR,SCRATCH)

Generic call:
   CALL GDATE (MONTH,DAY,YEAR,DAYOFYEAR,SCRATCH)
```

12.2.7 FPOOL function GDATEINT

```
GDATEINT
```

Function:	Requests the current date in number form
Generic:	GDATE
Connect:	GDATINT

Interface

Number of parameters: 5

1st parameter

Type:	INTEGER*4
Usage:	OUT
Meaning:	After invocation, contains the current month in number form, e.g. the numeral 9 is supplied for the date 20.9.81.

2nd parameter

Type:	INTEGER*4
Usage:	OUT
Meaning:	After invocation, contains the current day in number form, e.g., the numeral 20 is supplied for the date 20.9.81.

3rd parameter

Type:	INTEGER*4
Usage:	OUT
Meaning:	After invocation, contains the current year in number form, e.g. the numeral 81 is supplied for the date 20.9.81.

4th parameter

Type:	INTEGER*4
Usage:	OUT
Meaning:	After invocation, contains the current day of the year in number form, e.g. the numeral 263 is supplied for the date 20.9.81.

5th parameter

Type:	CHARACTER*31
Usage:	SCRATCH
Meaning:	Work area of the function; contents before and after invocation not subject to any regulation

Implementation

System macro used: GDATE
Shareable

Example:

```
Parameters:
   INTEGER*4  DAY,MONTH,YEAR
   INTEGER*4  DAYOFYEAR
   CHARACTER*31 SCRATCH

Specific call:
   CALL GDATEINT (MONTH,DAY,YEAR,DAYOFYEAR,SCRATCH)

Generic call:
   CALL GDATE (MONTH,DAY,YEAR,DAYOFYEAR,SCRATCH)
```

12.2.8 FPOOL function GEPRTCHAR

```
GEPRTCHAR
```

Function:	Requests the task’s elapsed CPU time in alpha form
Generic:	GEPRT
Connect:	GPRTCHR

Interface

Number of parameters: 5

1st parameter

Type:	CHARACTER*2
Usage:	OUT
Meaning:	After invocation, contains the hour portion of the elapsed CPU time in alpha form.

2nd parameter

Type:	CHARACTER*2
Usage:	OUT
Meaning:	After invocation, contains the minutes portion of the elapsed CPU time in alpha form.

3rd parameter

Type:	CHARACTER*2
Usage:	OUT
Meaning:	After invocation, contains the seconds portion of the elapsed CPU time in alpha form.

4th parameter

Type:	CHARACTER*4
Usage:	OUT
Meaning:	After invocation, contains the seconds fraction of the elapsed CPU time in alpha form; unit of measurement: one ten-thousandth of one second.

5th parameter

Type:	CHARACTER*49
Usage:	SCRATCH
Meaning:	Work area of the function; contents before and after invocation not subject to any regulation

Implementation

System macro used: TMODE
Shareable

Example:

```
Parameters:
   CHARACTER*2  HRS,MIN,SEC
   CHARACTER*4  TTHSND
   CHARACTER*49 SCRATCH

Specific call:
   CALL GEPRTCHAR (HRS,MIN,SEC,TTHSND,SCRATCH)

Generic call:
   CALL GEPRT (HRS,MIN,SEC,TTHSND,SCRATCH)
```

12.2.9 FPOOL function GEPRTINT

```
GEPRTINT
```

Function:	Requests task’s elapsed CPU time in number form
Generic:	GEPRT
Connect:	GPRTINT

Interface

Number of parameters: 5

1st parameter

Type:	INTEGER*4
Usage:	OUT
Meaning:	After invocation, contains the hours portion of the elapsed CPU time in number form.

2nd parameter

Type:	INTEGER*4
Usage:	OUT
Meaning:	After invocation, contains the minutes portion of the elapsed CPU time in number form.

3rd parameter

Type:	INTEGER*4
Usage:	OUT
Meaning:	After invocation, requests the seconds portion of the elapsed CPU time in number form.

4th parameter

Type:	INTEGER*4
Usage:	OUT
Meaning:	After invocation, requests the seconds fraction of the elapsed CPU time in number form; unit of measurement: one ten-thousandth of one second.

5th parameter

Type:	CHARACTER*45
Usage:	SCRATCH
Meaning:	Work area of the function; contents before and after invocation not subject to any regulation

Implementation

System macro used: TMODE
Shareable

Example:

```
      Parameters:
         INTEGER*4    HRS,MIN,SEC,TTHSND
         CHARACTER*45 SCRATCH

      Specific call:
         CALL GEPRTINT (HRS,MIN,SEC,TTHSND,SCRATCH)

      Generic call:
         CALL GEPRT (HRS,MIN,SEC,TTHSND,SCRATCH)
```

12.2.10 FPOOL function GETDATE

`GETDATE`

Function: Requests the current data in alpha form in ISO4 format
Generic: -
Connect: FP@GTDAT

Interface

Number of parameters: 2

1st parameter

Type:	CHARACTER*10
Usage:	OUT
Meaning:	After invocation, contains the current date in alpha form in ISO4 format: `YYYY-MM-DD`.

2nd parameter

Type:	CHARACTER*3
Usage:	OUT
Meaning:	After invocation, contains the current day of the year.

Implementation

System macro used: GDATE
Shareable

Example:

```
      Parameters:
         CHARACTER*10  DATE
         CHARACTER*3   DAY

      Specific call:
         CALL GETDATE (DATE,DAY)

      Generic call:
         ————
```

12.2.11 FPOOL function GETMEMMAPLONG

```
GETMEMMAPLONG
```

Function:	Requests the memory map which shows which pages of class 5 and class 6 memories are unallocated or reserved for the task; a user address space of up to 8 Mbytes is taken into consideration.
Generic:	GETMEMORYMAP
Connect:	GTMAPL

Interface

Number of parameters: 5

1st parameter

Type:	INTEGER*4
Usage:	OUT
Meaning:	After invocation, contains the number of the last page of class 6 memory.

2nd parameter

Type:	INTEGER*4
Usage:	OUT
Meaning:	After invocation, contains the number of the last page of class 5 memory.

3rd parameter

Type:	LOGICAL*1, DIMENSION 2048
Usage:	OUT
Meaning:	After invocation, contains a memory page allocation table. The index of a table element corresponds to the number of a memory page. A table element has the value .TRUE. if the memory page involved is reserved for the task; otherwise the value is .FALSE.

4th parameter

Type:	CHARACTER*1
Usage:	OUT
Meaning:	Error code in accordance with the return information of the GTMAP macro: X’00’ → o.k., X’04’,X’08’ → error

5th parameter

Type:	CHARACTER*260
Usage:	SCRATCH
Meaning:	Work area of the function; contents before and after invocation not subject to any regulation

Implementation

System macro used: GTMAP
Shareable

Example:

```
Parameters:
   INTEGER*4     MAXPAGECL6,MAXPAGECL5
   LOGICAL*1     MEMTABLE
   DIMENSION     MEMTABLE(2048)
   CHARACTER*1   ERRCODE
   CHARACTER*260 SCRATCH

Specific call:
   CALL GETMEMMAPLONG (MAXPAGECL6,MAXPAGECL5,MEMTABLE,ERRCODE,SCRATCH)

Generic call:
   CALL GETMEMORYMAP (MAXPAGECL6,MAXPAGECL5,MEMTABLE,ERRCODE,SCRATCH)
```

12.2.12 FPOOL function GETMEMMAPSHORT

```
GETMEMMAPSHORT
```

Function:	Requests the memory map which shows which pages of class 6 memory are unallocated or reserved for the task; a user address space of up to 1 Mbyte is taken into consideration.
Generic:	GETMEMORYMAP
Connect:	GTMAPS

Interface

Number of parameters: 4

1st parameter

Type:	INTEGER*4
Usage:	OUT
Meaning:	After invocation, contains the number of the last page of class 6 memory.

2nd parameter

Type:	LOGICAL*1, DIMENSION 256
Usage:	OUT
Meaning:	After invocation, contains a memory page allocation table. The index of a table element corresponds to the number of a memory page. A table element has the value .TRUE. if the memory page involved is reserved for the task; otherwise the value is .FALSE.

3rd parameter

Type:	CHARACTER*1
Usage:	OUT
Meaning:	Error code in accordance with the return information of the GTMAP macro: X'00' → o.k., X'04',X'08' → error

4th parameter

Type:	CHARACTER*34
Usage:	SCRATCH
Meaning:	Work area of the function; contents before and after invocation not subject to any regulation

Implementation

System macro used: GTMAP
Shareable

Example:

```
Parameters:
   INTEGER*4     MAXPAGECL6
   LOGICAL*1     MEMTABLE
   DIMENSION     MEMTABLE(256)
   CHARACTER*1   ERRCODE
   CHARACTER*34  SCRATCH

Specific call:
   CALL GETMEMMAPSHORT (MAXPAGECL6,MEMTABLE,ERRCODE,SCRATCH)

Generic call:
   CALL GETMEMORYMAP (MAXPAGECL6,MEMTABLE,ERRCODE,SCRATCH)
```

12.2.13 FPOOL function GETODCHAR

```
GETODCHAR
```

Function:	Requests the time of day in alpha form
Generic:	GETOD
Connect:	GTODCHR

Interface

Number of parameters: 4

1st parameter

Type:	CHARACTER*2
Usage:	OUT
Meaning:	After invocation, contains the hours portion of the time of day in alpha form.

2nd parameter

Type:	CHARACTER*2
Usage:	OUT
Meaning:	After invocation, contains the minutes portion of the time of day in alpha form.

3rd parameter

Type:	CHARACTER*2
Usage:	OUT
Meaning:	After invocation, contains the seconds portion of the time of day in alpha form.

4th parameter

Type:	CHARACTER*6
Usage:	SCRATCH
Meaning:	Work area of the function; contents before and after invocation not subject to any regulation

Implementation

System macro used: GDATE
Shareable

Example:

```
Parameters:
   CHARACTER*2  HRS,MIN,SEC
   CHARACTER*6  SCRATCH

Specific call:
   CALL GETODCHAR (HRS,MIN,SEC,SCRATCH)

Generic call:
   CALL GETOD (HRS,MIN,SEC,SCRATCH)
```

12.2.14 FPOOL function GETODINT

```
GETODINT
```

Function:	Requests the time of day in number form
Generic:	GETOD
Connect:	GTODINT

Interface

Number of parameters: 4

1st parameter

Type:	INTEGER*4
Usage:	OUT
Meaning:	After invocation, contains the hours portion of the time of day in number form.

2nd parameter

Type:	INTEGER*4
Usage:	OUT
Meaning:	After invocation, requests the minutes portion of the time of day in number form.

3rd parameter

Type:	INTEGER*4
Usage:	OUT
Meaning:	After invocation, requests the seconds portion of the time of day in number form.

4 . Parameter

Type:	CHARACTER*23
Usage:	SCRATCH
Meaning:	Work area of the function; contents before and after invocation not subject to any regulation

Implementation

System macro used: GDATE
Shareable

Example:

```
Parameters:
   INTEGER*4    HRS,MIN,SEC
   CHARACTER*23 SCRATCH

Specific call:
   CALL GETODINT (HRS,MIN,SEC,SCRATCH)

Generic call:
   CALL GETOD (HRS,MIN,SEC,SCRATCH)
```

12.2.15 FPOOL function TASKANDUSERID

```
TASKANDUSERID
```

Function:	Requests the task sequence number (TSN) and the user ID of the LOGON command in alpha form
Generic:	TMODE
Connect:	TMODTSN

Interface

Number of parameters: 3

1st parameter

Type:	CHARACTER*4
Usage:	OUT
Meaning:	After invocation, contains the 4-digit task sequence number in alpha form, with leading zeros if applicable.

2nd parameter

Type:	CHARACTER*8
Usage:	OUT
Meaning:	After invocation, contains the user ID from the LOGON command in alpha form.

3rd parameter

Type:	CHARACTER*45
Usage:	SCRATCH
Meaning:	Work area of the function; contents before and after invocation not subject to any regulation

Implementation

System macro used: TMODE
Shareable

Example:

```
Parameters:
   CHARACTER*4  TASKNO
   CHARACTER*8  USERID
   CHARACTER*45 SCRATCH

Specific call:
   CALL TASKANDUSERID (TASKNO,USERID,SCRATCH)

Generic call:
   CALL TMODE (TASKNO,USERID,SCRATCH)
```

12.2.16 FPOOL function TMODEALL

When using this function, compilation must take place with COMOPT=FPOOL since parameters of data type INTEGER*1 are transferred differently with FPOOL functions than with FOR1 programs.

```
TMODEALL
```

Function:	Requests the task type, the task sequence number (TSN), the user ID from the LOGON command and the task account number
Generic:	TMODE
Connect:	TMODALL

Interface

Number of parameters: 5

1st parameter

Type:	INTEGER*1
Usage:	OUT
Meaning:	After invocation, contains the task type in number form. The individual values have the following meanings:

0	Batch job	
2	Terminal	8103
4	Video terminal	8150
17	TRANSDATA	8418,8415
21	Video terminal	8151
22	Video terminal	8152
23	Printer terminal	8110
24	Data terminal	8161/54
25	Data terminal	8161/64
26	Data terminal	8161/80
44	Data terminal	8162
45	Data terminal	8160/80
53	Data terminal	9750

2nd parameter

Type:	CHARACTER*4
Usage:	OUT
Meaning:	After invocation, contains the 4-digit task sequence number in alpha form, with leading zeros if applicable.

3rd parameter

Type:	CHARACTER*8
Usage:	OUT
Meaning:	After invocation, contains the user ID from the LOGON command in alpha form.

4th parameter

Type:	CHARACTER*8
Usage:	OUT
Meaning:	After invocation, contains the account number left-justified, padded with blanks if applicable.

5th parameter

Type:	CHARACTER*45
Usage:	SCRATCH
Meaning:	Work area of the function; contents before and after invocation not subject to any regulation

Implementation

System macro used: TMODE
Shareable

Example:

```
Parameters:
   INTEGER*1    TASKTYPE
   CHARACTER*4  TASKNO
   CHARACTER*8  USERID,ACCOUNTNO
   CHARACTER*45 SCRATCH

Specific call:
   CALL TMODEALL (TASKTYPE,TASKNO,USERID,ACCOUNTNO,SCRATCH)

Generic call:
   CALL TMODE (TASKTYPE,TASKNO,USERID,ACCOUNTNO,SCRATCH)
```

12.2.17 FPOOL function MEMOMAP

```
MEMOMAP
```

Function: Requests information on the size and occupancy of class 6 memory or of the memory pool in class 6 memory. This information can be requested by issuing a call in 31-bit address mode or, in the case of a class 6 memory with more than 8 Mbytes of storage, only by means of MEMOMAP (as of BS2000 V9.0).

Generic: -

Connect: FP@MINF

Interface

Number of parameters: 6

1st parameter

Type:	INTEGER*4		
Usage:	IN		
Meaning:	1	Information on the class 6 memory requested	
	2	Information on the memory pool requested	

2nd parameter

Type:	INTEGER*4	
Usage:	IN	
Meaning:	1	The virtual page number of the first page of memory and the number of memory pages of class 6 memory or of the memory pool.
	2	Outputs a table on the occupancy of the memory pages.

3rd parameter

Type:	Array of type INTEGER*4
Dimension:	4
Usage:	IN/OUT
Meaning:	The meaning of the third parameter (hereafter referred to as the TAB) depends on the values of the first and second parameters:

(a) 1st parameter=1, 2nd parameter=1 (class 6 memory occupancy)

In this case the third parameter is merely an output parameter.

TAB(1)		Contains the virtual page number of the first page of memory below 16 Mbytes.
TAB(2)		Contains the number of memory pages below 16 Mbytes.
TAB(3)	0	No class 6 memory exists above 16 Mbytes.
	>0	Virtual page number of the first page of memory above 16 Mbytes.
TAB(4)	0	No class 6 memory exists above 16 Mbytes.
	>0	Number of memory pages above 16 Mbytes.

(b) 1st parameter=2, 2nd parameter=1 (memory pool occupancy)

	Input	Output
TAB(1)	Virtual page number of any page of the requested memory pool. This entry identifies the memory pool.	Virtual page number of the first page of the memory pool
TAB(2)	Not used	Number of pages of the memory pool
TAB(3)	Not used	Unchanged
TAB(4)	Not used	Unchanged

(c) 1st parameter=1, 2nd parameter=2 (occupancy table of the class 6 memory)

	Input	Output
TAB(1)	Virtual page number of the 1st page of the area for which an occupancy table is requested. The specified page number must be a multiple of 16.	Unchanged
TAB(2)	Number of memory pages for which an occupancy table is requested.	Number of memory pages, for which the occupancy table is actually defined.
TAB(3)	Not used	Unchanged
TAB(4)	Not used	Unchanged

(d) 1st parameter=2, 2nd parameter=2 (occupancy table of the memory pool)

	Input	Output
TAB(1)	Virtual page number of the first page of the area for which an occupancy table is requested. The specified page number must be within the requested memory pool and must be a multiple of 16.	Unchanged
TAB(2)	Number of memory pages for which an occupancy table is requested.	Number of memory pages for which the occupancy table is actually defined.
TAB(3)	Not used	Unchanged
TAB(4)	Not used	Unchanged

If the 1st parameter=1 is selected, all memory pages of a memory pool are identified as being occupied. If the first parameter selected is parameter=2, only those pages requested with REQMP are identified as being occupied.

4th parameter

Type:	INTEGER*4
Usage:	IN
Meaning:	The fourth parameter specifies the size of the requested memory page occupancy table (see fifth parameter).

Permissible values: 1 ≤ fourth parameter ≤ 256 The parameter is also abbreviated to PTSIZE.

5th parameter

Type:	Array of type LOGICAL*1
Dimension	(PTSIZE*2048)
Usage:	OUT
Meaning:	After invocation, the fifth parameter contains a memory occupancy table. The dimension of the fifth parameter should be 2048 times that of the fourth parameter. The subscript of a table element corresponds to the number of a memory page (number of the page specified in TAB(1) minus 1). A table element has the value .TRUE. if the corresponding memory page is reserved by the task, otherwise it has the value .FALSE..

6th parameter

Type:	INTEGER*4
Usage:	OUT
Meaning:	The sixth parameter contains the return code of the MINF macro:

X’00’	Function executed
X’01’	Invalid entry for the first parameter
X’02’	Invalid entry for the second parameter
X’04’	Operand error
X’08’	Invalid virtual page number
X’0C’	Address error

Implementation

System macro used: MINF
Shareable

Example:

```
Parameters:
   INTEGER*4     INFORM1, INFORM2, PAGETAB(4),
  *              PTSIZE, ERRCODE
   LOGICAL*1     MEMTABLE
   PARAMETER (PTSIZE=10)
   INFORM1 = 1
   INFORM2 = 1
   DIMENSION     MEMTABLE(PTSIZE*2048)

Specific call:
   CALL MEMOMAP (INFORM1,INFORM2,PAGETAB,PTSIZE,MEMTABLE,ERRCODE)

Generic call:
   ____
```

12.2.18 Summary: Generic, call and connect names

The section on the central FPOOL referred to the use of generic names for FPOOL subprograms. Depending on the number, type and position of parameters, the following summary shows the linkage between generic names and the specific subprogram names, as well as the connect names.

Generic name	Parameter Total	Parameter No: Type	Specific subprogram (call name)	Connect name	Module name
-	5	1: CHARACTER*1 2: CHARACTER*512 3: CHARACTER*1024 4: CHARACTER*1 5: CHARACTER*1552	FCMD	FP@CMD	FP@CMD
GDATE	5	1: CHARACTER*2 2: CHARACTER*2 3: CHARACTER*2 4: CHARACTER*3 5: CHARACTER*12	GDATECHAR	GDATCHR	FP@GDCHR
	5	1: INTEGER*4 2: INTEGER*4 3: INTEGER*4 4: INTEGER*4 5: CHARACTER*31	GDATEINT	GDATINT	FP@GDINT
GEPRT	5	1: CHARACTER*2 2: CHARACTER*2 3: CHARACTER*2 4: CHARACTER*4 5: CHARACTER*49	GEPRTCHAR	GPRTCHR	FP@GPCHR
	5	1: INTEGER*4 2: INTEGER*4 3: INTEGER*4 4: INTEGER*4 5: CHARACTER*45	GEPRTINT	GPRTINT	FP@GPINT
-	2	1: CHARACTER*10 2: CHARACTER*3	GETDATE	FP@GTDAT	FP@GTDAT
GETMEMORYMAP	5	1: INTEGER*4 2: INTEGER*4 3: LOGICAL*1(2048) 4: CHARACTER*1 5: CHARACTER*260	GETMEMMAPLONG	GTMAPL	FP@GTAPL
	4	1: INTEGER*4 2: LOGICAL*1(256) 3: CHARACTER*1 4: CHARACTER*34	GETMEMMAPSHORT	GTMAPS	FP@GTAPS
GETOD	4	1: CHARACTER*2 2: CHARACTER*2 3: CHARACTER*2 4: CHARACTER*6	GETODCHAR	GTODCHR	FP@GTCHR
	4	1: INTEGER*4 2: INTEGER*4 3: INTEGER*4 4: CHARACTER*23	GETODINT	GTODINT	FP@GTINT

continued ▹

continued

Generic name	Parameter Total	No: Type	Specific subprogram (call name)	Connect name	Module name
IDELETE	3	1: INTEGER 2: CHARACTER*n 3: INTEGER*4	ELIMCHR	ELIMCHR	FP@ELM
	3	1: INTEGER 2: INTEGER 3: INTEGER*4	ELIMINT	ELIMINT	FP@ELM
-	6	1: INTEGER*4 2: INTEGER*4 3: INTEGER*4 4: INTEGER*4 5: LOGICAL*1 6: INTEGER*4	MEMOMAP	FP@MINF	FP@MINF
TMODE	2	1: CHARACTER*8 2: CHARACTER*45	ACCOUNTNR	TMODEACC	FP@TMACC
	2	1: INTEGER*1 2: CHARACTER*10	DIALOG	DIALOG	FP@DLOG
	3	1: CHARACTER*4 2: CHARACTER*8 3: CHARACTER*45	TASKANDUSERID	TMODTSN	FP@TMTSN
	5	1: INTEGER*1 2: CHARACTER*4 3: CHARACTER*8 4: CHARACTER*8 5: CHARACTER*45	TMODEALL	TMODALL	FP@TMALL

Table 12-1: Generic names, connect names and specific names of FPOOL subprograms

12.3 Setting up private FPOOLs (FPOOLITY utility routine)

The utility routine FPOOLITY enables users to prepare and process their own FPOOL files for their subprograms (cf. "FPOOLITY" manual [22]).
An interface description is entered in a new or in an expanded FPOOL file with the aid of the FPOOLITY function GENERATE and its Function Description Language (FDL).

FPOOL processing by the FOR1 compiler does not yet cover all FDL entries described in the "FPOOLITY" manual [22]. The following summary shows the section of the FDL interface that is interpreted by FOR1:

```
CALL NAME              :   name1                 ;
GENERIC                :   son1,son2,...,son20   ;
CONNECT NAME           :   name2                 ;
CONNECT MODE           :   STANDARD              ;
IMPL-LANGUAGE          :   FORTRAN               ;
ENVIRONMENT            :   FORTRAN               ;
FOR                    :   FORTRAN               ;
RETURNS                :   typename2             ;
P#n KEYWORD            :   parametername         ;
[P#n] DIRECTION        :   IN|OUT|INOUT|SCRATCH  ;
[P#n] MODE             :   REFERENCE             ;
[P#n] TYPE             :   typename1             ;
FDLEND    ;
```

name1	Call name of the subprogram, up to 15 characters long.
sonx	Call name of an FPOOL entry.
name2	In object code, FOR1 replaces the call name *name1* with the connect name *name2*. The connect name must be identical with the entry name of the object module. The connect name may be up to 8 characters in length.
P#n	n-th parameter of the transfer list in the CALL statement. Up to 30 parameters may be specified.
parametername	Name of the n-th parameter.
typename1	Data type (see table 12-2)
typename2	Data type of the RETURN value (see table 12-3)

The relationship between FDL type names and FORTRAN data types or FORTRAN RETURN values is shown in the following table:

FDL (TYPE:)	FORTRAN data type typename1
CHAR[ACTER]	CHARACTER
CHAR-STRING * n	CHARACTER * n (0 ≤ n ≤ 32000)
COMPLEX [*x└a8|16|32x└a]	COMPLEX *x└a8|16|32x└a
INTEGER [*x└a1|$\underline{2}$|4|8x└a]	INTEGER *x└a1|2|4|8x└a
LOGICAL	LOGICAL * 1
REAL [*{$\underline{4}$|8|16}]	REAL *{4|8|16}
SIGNED-INTEGER [*{1|2|$\underline{4}$|8}]	INTEGER *{1|2|4|8}

FOR1 treats all other FDL entries as comments.

Table 12-2: FDL type names and FORTRAN data types

FDL (RETURNS:)	Data type of the RETURN value typename2
CHAR[ACTER]	CHARACTER
CHAR-STRING * n	CHARACTER * n (0 ≤ n ≤ 4)
INTEGER [*{1|2|$\underline{4}$}]	INTEGER *{1|2|4}
LOGICAL	LOGICAL * 1
REAL [*4]	REAL *4
SIGNED-INTEGER [*{1|2|$\underline{4}$}]	INTEGER *{1|2|4}

Table 12-3: FDL and FORTRAN RETURN values

Restrictions

Checking of the interfaces with FORTRAN subprograms is subject to a number of restrictions:

- In the case of functions, only the parameters are checked, not the type and length of the function.
- Functions of the data type REAL*{8|16}, INTEGER*8, CHARACTER*n (n>5) and COMPLEX*{8|16|32} cannot be checked.
- Names of SUBROUTINEs, FUNCTIONs and LABELs in the form of parameters are not checked.
- Arrays are not checked.
- Data items and functions of the type LOGICAL*4 cannot be checked.

- In the case of CHARACTER data items, only one error (SEVERE ERROR) is output if the length of the actual argument is less than that of the dummy argument.
- Conversions are performed,
  - if it is possible to do so (e.g. actual arguments of the type REAL, dummy arguments of the type INTEGER), and
  - if "DIRECTION IN" is entered in the FPOOL (i.e. if they are purely input parameters and if only those are errored).

  In the above case a WARNING (SA151) is output. In all other cases no conversion takes place and a SEVERE ERROR is output.

Example: FDL entry

For a subprogram with the following statements

```
    SUBROUTINE DEVIATE (N,A,VALU)
    REAL A (2:101)
    REAL*8 VALU
```

the FDL entry has the following format:

```
CALL NAME:          DEVIATE ;
CONNECT NAME:       DEVIATE ;
CONNECT MODE:       STANDARD ;
IMPL-LANGUAGE:      FORTRAN ;
ENVIRONMENT:        FORTRAN ;
FOR:                FORTRAN ;
P#  1 KEYWORD:    N ;
      DIRECTION:  INOUT ;
      MODE:       REFERENCE ;
      TYPE:       INTEGER *4 ;
P#  2 KEYWORD:    A ;
      DIRECTION:  INOUT ;
      MODE:       REFERENCE ;
      TYPE:       REAL *4
                  ARRAY(100) ;
P#  3 KEYWORD:    WERT ;
      DIRECTION:  INOUT ;
      MODE:       REFERENCE ;
      TYPE:       REAL *8 ;
FDLEND ;
```

12.4 Example: Application of FOR1.FPOOLLIB interfaces

The application of FOR1.FPOOLLIB interfaces is illustrated in the following. The printout includes:

– source program
– commands/control statements for compilation, link-editing, execution
– result listing

1. Source program

```
      PROGRAM EXAMP
      IMPLICIT CHARACTER *2 (C-D)
      IMPLICIT INTEGER   *4 (I,J)
      INTEGER   *1   I1,I2,I3
      INTEGER   *1   UT /2     /
      LOGICAL   *1   L1(2048),L2(256)
      LOGICAL   *1   Q  /.TRUE./
      CHARACTER *260 SCR260
      CHARACTER *49  SCRATCH
      CHARACTER *8   C81,C82,C83,C84,D8(2:4)
      CHARACTER *4   C41,C42,C43,D4(2:3)
      CHARACTER *3   C31,D31
      CHARACTER *1   C1,C2
      CHARACTER *26  T(0:53)
      DIMENSION      I4(4)
      DIMENSION      D2(3)
      DATA T(0)  /'BATCH TASK'/,
     .     T(2)  /'INTERACTIVE TERMINAL    8103'/,
     .     T(4)  /'DATA DISPLAY TERMINAL 8150'/,
     .     T(17) /'TRANSDATA        8418,8415'/,
     .     T(21) /'DATA DISPLAY TERMINAL 8151'/,
     .     T(22) /'DATA DISPLAY TERMINAL 8152'/,
     .     T(23) /'PRINTER TERMINAL 8110'/,
     .     T(24) /'TERMINAL         8161/54'/,
     .     T(25) /'TERMINAL         8161/64'/,
     .     T(26) /'TERMINAL         8161/80'/,
     .     T(44) /'TERMINAL         8162'/,
     .     T(45) /'TERMINAL         8160/80'/,
     .     T(53) /'TERMINAL         9750'/
      DATA (T(I),I=5,16),
     .     (T(I),I=18,20),
     .     (T(I),I=27,43),
     .     (T(I),I=46,52),
     .      T(1),T(3) /41* 'DEVICE UNDEFINED'/
*
      CALL DIALOG(I1,SCRATCH(1:10))
         IF (I1 .EQ. 0) UT = 6
      WRITE (UT,1) I1,T(I1)
  1   FORMAT (' DIALOG:            ',I2,', D.H. ',A)
*
      CALL ACCOUNTNO(C81,SCRATCH(:45))
      WRITE (UT,2) C81
  2   FORMAT (' ACCOUNT NUMBER: ',A)
*
      CALL GDATECHAR(C21,C22,C23,C31,SCRATCH(:12))
      WRITE (UT,3) C21,C22,C23,C31

  3   FORMAT (' DATE (CHAR):       ',4(A,:', '))
*
      CALL GDATEINT(I41,I42,I43,I44,SCRATCH(:31))
      WRITE (UT,4) I41,I42,I43,I44
  4   FORMAT (' DATE (INT):        ',3(I2,', '),I3)
*
      CALL GEPRTCHAR(C24,C25,C26,C41,SCRATCH(:12))
      WRITE (UT,5) C24,C25,C26,C41
  5   FORMAT (' CPU TIME (CHAR):    ',4(A,:', '))
*
      CALL GEPRTINT(I45,I46,I47,I48,SCRATCH(:45))
      WRITE (UT,6) I45,I46,I47,I48
  6   FORMAT (' CPU TIME (INT):     ',3(I2,', '),I4)
*
      CALL GETODCHAR(C27,C28,C29,SCRATCH(:6))
      WRITE (UT,7) C27,C28,C29
  7   FORMAT (' TIME OF DAY (CHAR):  ',3(A,:', '))
*
      CALL GETODINT(I49,I4A,I4B,SCRATCH(:23))
      WRITE (UT,8) I49,I4A,I4B
  8   FORMAT (' TIME OF DAY (INT):   ',3(I2,:', '))
*
      CALL GETMEMMAPLONG(J1,J2,L1,C1,SCR260)
      DO 9 I = 2047,1,-1
  9      IF (L1(I).NEQV.L1(2048)) GOTO 10
 10   WRITE (UT,11) J1,J2,:L1(I)
 11   FORMAT (' MEMORYMAPLONG:     ',2(I9,', '),32(:/,10X,64L1))
      WRITE (UT,12) L1(I+1),C1
 12   FORMAT (10X,'REMAINDER ',L1,', ',Z2)
*
      CALL GETMEMMAPSHORT(J3,L2,C2,SCRATCH(:34))
      WRITE (UT,13) J3,L2,C2
 13   FORMAT (' MEMORYMAPSHORT:    ',I9,', ',4(/,10X,64L1),
     .        ', ',/,10X,Z2)
*
      CALL TASKANDUSERID(C42,C82,SCRATCH(2:46))
      WRITE (UT,14) C42,C82
 14   FORMAT (' TASKANDUSERID:     ',2(A,:', '))
*
      CALL TMODEALL(I2,C43,C83,C84,SCRATCH(3:47))
      WRITE (UT,15) I2,C43,C83,C84
 15   FORMAT (' TMODEALL:          ',I3,', ',3(A,:', '))
      IF (I1.NE.I2.OR.
     .    C42.NE.C43.OR.
     .    C82.NE.C83.OR.
     .    C81.NE.C84) THEN
         Q = .FALSE.
         WRITE (UT,16)
 16      FORMAT (' ERROR AFTER TMODEALL')
         END IF
*

      CALL TMODE(D8(1),SCRATCH(:45))
      IF (C81.NE.D8(1)) THEN
         Q = .FALSE.
         WRITE (UT,17)
 17      FORMAT (' ERROR IN ACCOUNT NUMBER = TMODE')
```

```
            END IF
*
         CALL TMODE(I3,SCRATCH(1:10))
         IF (I1.NE.I3) THEN
            Q = .FALSE.
            WRITE (UT,18)
 18         FORMAT (' ERROR IN DIALOG = TMODE')
            END IF
*
         CALL GDATE(D2(1),D2(2),D2(3),D31,SCRATCH(:12))
         IF (C21.NE.D2(1).OR.
        .    C22.NE.D2(2).OR.
        .    C23.NE.D2(3).OR.
        .    C31.NE.D31) THEN
            Q = .FALSE.
            WRITE (UT,19)
 19         FORMAT (' ERROR IN GDATECHAR = GDATE')
            END IF
*
         CALL GDATE(I4(1),I4(2),I4(3),I4(4),SCRATCH(:31))
         IF (I41.NE.I4(1).OR.
        .    I42.NE.I4(2).OR.
        .    I43.NE.I4(3).OR.
        .    I44.NE.I4(4)) THEN
            Q = .FALSE.
            WRITE (UT,20)
 20         FORMAT (' ERROR IN GDATEINT = GDATE')
            END IF
*
         CALL TMODE(D4(2),D8(2),SCRATCH(2:46))
         IF (D4(2) .NE.C42.OR.
        .    D8(2).NE.C82) THEN
            Q = .FALSE.
            WRITE (UT,21)
 21         FORMAT (' ERROR IN TASKANDUSERID = TMODE')
            END IF
*
         CALL TMODE(I2,D4(3),D8(3),D8(4),SCRATCH(3:47))
         IF (I1.NE.I2.OR.
        .    C42.NE.D4(3).OR.
        .    C82.NE.D8(3).OR.
        .    C81.NE.D8(4)) THEN
            Q = .FALSE.
            WRITE (UT,22)
 22         FORMAT (' ERROR IN TMODEALL = TMODE')
            END IF
*
         IF (Q) THEN
            WRITE (2,31)
            WRITE (6,31)
         ELSE
            WRITE (2,32)
            WRITE (6,32)
            END IF
 31      FORMAT (' FPOOLEXAMP SUCCESSFUL')
 32      FORMAT (' FPOOLEXAMP FAILURE   ')
         STOP
         END
```

2. Compilation

```
/DEL-SYS-FILE OMF
/START-PROG $FOR1
*COMOPT SRC=SRC.FPOOL,FPOOL=$TSOS.FOR1.FPOOL,END
```

where:

```
SRC.FPOOL           Source program file
$TSOS.FOR1.FPOOL    FPOOL file
```

3. Linkage

```
/START-PROG $TSOSLNK
 PROG FPOOL,FILENAM=L.FPOOL
 INCLUDE *
 RESOLVE ,$TSOS.FOR1MODLIBS
 RESOLVE ,$TSOS.FOR1.FPOOLLIB
 END
```

where:

```
L.FPOOL                 File name of the executable program
$TSOS.FOR1MODLIBS       FOR1 module library
$TSOS.FOR1.FPOOLLIB     FPOOL object module library
```

4. Execution

```
/SET-TASKLIB $TSOS.FOR1MODLIBS
/START-PROG L.FPOOL
```

The following listing is output to the terminal:

```
                %  BLS0500 PROGRAM 'FPOOL', VERSION ' ' OF '91-08-20' LOADED
                BS2000  F O R 1 : FORTRAN PROGRAM "EXAMP"
                STARTED ON 1991-08-20 AT 16:55:49
                DIALOG:            53, D.H. TERMINAL     9750
                ACCOUNT NUMBER:    ACCNT03
                DATE (CHAR):       08, 20, 91, 232
                DATE (INT):         8, 20, 91, 232
                CPU TIME (CHAR):   00, 00, 20, 7920
                CPU TIME (INT):     0,  0, 20, 7946
                TIME OF DAY (CHAR):16, 55, 49
                TIME OF DAY (INT): 16, 55, 49
                MEMORYMAPLONG:          2047,      3583,
                         TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
                         TTTTTTTTTTTTTTTTTTTTTT
                         REMAINDER F, 00
                MEMORYMAPSHORT:          239,
                         TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
                         TTTTTTTTTTTTTTTTTTTTTTTFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
                         FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
                         FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF,
                         00
                TASKANDUSERID:     9FIK, K3790052
                TMODEALL:           53, 9FIK, K3790052, ACCNT03
                FPOOLEXAMP SUCCESSFUL
                STOP AT STMT 112 IN EXAMP

FPOOLEXAMP SUCCESSFUL
                BS2000  F O R 1 : FORTRAN PROGRAM "EXAMP   " ENDED PROPERLY AT 16:55:55
                CPU - TIME USED :       0.0514 SECONDS
                ELAPSED TIME    :       5.9670 SECONDS
```

A Appendix

A.1 Abbreviations for FOR1 compiler options and option values

There are two ways of abbreviating the names of compiler options or option values:

1. Starting from the right, any number of letters may be dropped as long as the name is still unique.

   Example:

– instead of LIST	LIS or LI
– instead of LISTFILE	LISTFIL or LISTFI or LISTF

2. Special forms are considered declared:
   - abbreviations for composite names, e.g. instead of LISTFILE: LF
   - assignment of actually ambiguous abbreviations, e.g. instead of LIST: L

The following tables show abbreviations resulting from right truncation or from declaration.

Abbreviations of the compiler options:

CCOM	CC
CODE	CO
COLLECT	CL
COMPATIBLE	COMPAT, COM
DIALOG	D
NODIALOG	ND
DIALOG-SAVE	DIALOG-
EJECT	EJ
ERRKILL	EK
EXPAND	EX
EXPUNDERFLOW	EU
FORTRAN90-CHECK	F90
FPOOL	F
GEN	G
IMPLICIT	IM
INCLUDE-LIBRARY	INC
LANGUAGE	LA
LINECNT	LC

LINEEND	LE
LINKAGE	LNK
LIST	L
LISTFILE	LF
LIST-OUTPUT	LIST-
MAXERR	ME
MODULE-LIBRARY	MOD
MSGLEVEL	MSG
OBJECT	OBJ, O
OPTIMIZE	OPT
OPTIONS	OPTIO
OUTPUT	OU
PAD	P
PROCEDURE-OPTIMIZATION	PR
REAL	R
SAVE-CONSTANT	SAV
SHARE-LIBRARY	SH
SOURCE	SRC
SOURCE-FORMAT	SF
STANDARD-CHECK	STD
SUPPLIEDBOUND	SUP, SB
SYMTEST	SYM
TESTOPT	TO
TEXT-SEPARATOR	TEX
TRUNCONST	TC
UNIT	U
UPDATE	UPD

Abbreviations of the TESTOPT option values:

ALL	A
ARG	AR
BOUNDS	B
CNTRL	C
DEBUG	D
STNR	STN
STRING	STR
SUBSCR	SU
UNDEF	U

Abbreviations of the LIST option values:

ALL	A
ATR	AT
CHANGE	CH
DIAG	D
DECOMP	DE
ESD	E
LIST	L
MAP	M
MIN	MI
NONE	N
OBJECT	O
OPTIONS	OP
SOURCE	SRC

SUMMARY	S
XREF	X

Abbreviations of the OBJECT option values:

SHARE	S

Abbreviations of the UNIT option values:

PUNCH	PU
PRINT	PR
READ	R
WRITE	W

Abbreviations of the MSGLEVEL option values:

DIAG	D
ERROR	E
NOTE	N
SOURCE	S, SRC
WARNING	W

Abbreviations of the COMPATIBLE option values:

BGFOR	BGF
BS3FOR	BS3

The rules for abbreviating SDF operands are given in section 2.2.1.

A.2 Compiler phases

Compilation is divided into several phases, which are executed once for each program unit and loaded separately into virtual memory.
Information is exchanged between phases via virtual work files; in the event of an overflow, information is exchanged via external temporary work files. Interaction between individual phases is controlled by the Compiler Management Facility (see Fig. A.2-1).

The compiler phases perform the following functions:

Compiler initialization
Reading in the options, opening files, etc.

Formal analysis 1
Reading in the source program; classification of statements; lexicographical and syntactical analysis of specification statements;
configuration of constant and symbol tables.

Formal analysis 2
Identifying statement function definitions; lexicographical and syntactical analysis of executable statements; further configuration of constant and symbol tables.

Semantic analysis 1
Semantic checking and detailing of specification statements.

Semantic analysis 2
Semantic checking and detailing of executable statements.

Global optimization
Taking the necessary actions for optimization.

Code generation
Assigning addresses to all variables, constants and temporary auxiliary entities; assigning registers to operands; generating instruction sequences; assigning addresses to all branch labels.

Compiler output
Assigning displacements and addresses to forward branches; generating LSD, TXT, RLD and ESD records (see "System Conventions" manual [39]); generating standard and additional listings; output of the summary messages concerning the compilation process.

The initialization process serves to restart the compilation process for each program unit.

This figure is not any longer available for the online pdf.

Fig. A.2-1: FOR1 compiler phases

A.3 Naming convention for library modules

The names and ENTRY names of the mathematical library modules and of the FPOOL modules always begin with IF@ and FP@ respectively. The name or ENTRY name of a mathematical library module or of an FPOOL routine is thus not identical with the name of the function defined in FORTRAN. So, for example, the library module referenced by the intrinsic name ABS has the name IF@ABS, and the library module referenced by the intrinsic name DSIN has the name IF@DS.

Since the character "@" does not belong to the FORTRAN character set, accidental name identity with self-defined subprograms is thereby excluded. This is particularly important because some mathematical library modules themselves in turn call other library modules. This basic distinction ensures that with such "internal" function calls it is the required library functions that are accessed and not any identically named user routines that may be present.

However, care should be exercised when using names beginning with "ITS" or "IT0" since some runtime system modules begin with these character combinations. Accidental name identity is *not* excluded here, unlike mathematical library functions and FPOOL functions.

A.4 PARAMETER operands and corresponding compiler options

For reasons of compatibility, the PARAMETER command is also effective for the FOR1 compiler. It may be used to submit compilation operands and operands controlling program execution. However, the user should preferably insert compilation operands in COMOPT statements or specify them as SDF operands, which offer considerably more options.

In two cases there are no RUNOPT options corresponding to functions of PARAMETER operands, only corresponding SDF operands:

– CARD=YES for the input of runtime options (RUNOPTs, see section 6.3.1).
  Corresponding SDF operand: RUNTIME-OPTIONS=YES().

– DEBUG=YES for program continuation in the case of runtime errors in batch mode (see section 6.5.2).
  Corresponding SDF operand: OBJECT-CONTINUATION=YES.

A PARAMETER command is read from SYSCMD and, if it is needed, must be issued before calling the compiler. The command remains valid until the next LOGOFF or SET-JOB-STEP command is given (in procedures) or until the next change resulting from another PARAMETER command.

For FOR1, the PARAMETER command entries have the same effect as if corresponding compiler options preceded the actual compiler options. Thus the PARAMETER command entries will be effective only if they are not changed or overridden by the existing compiler options.

Example:

The command

```
/PARAMETER ERRFIL=YES,MAP=YES,SAVLST=SOURCE,OBJLST=YES,LIST=YES
```

corresponds to the compiler options

```
*COMOPT LISTFILE=(SOURCE,DIAG,ESD,XREF,SUMMARY,OPTIONS)
*COMOPT LIST=(MAP,SOURCE,OPTIONS,DIAG,SUMMARY,OBJECT).
```

The following table compares the PARAMETER operands relevant to FOR1 with the compiler options having the same meaning.

PARAM operand	* COMOPT...
CODE = 1	CODE = EBCDIC
CODE = 2	CODE = ISO
CODE = 3	CODE = BCD
DEBUG = YES	TESTOPT = (ALL)
DEBUG = NO	TESTOPT = (STNR)
DIAG = YES	LIST = (DIAG)
DIAG = NO	NOLIST = (DIAG)
DISC = YES	OBJECT = (*)
DISC = NO	NOOBJECT = (*)
ERRFIL = YES	LISTFILE = (DIAG)
ERRFIL = NO	LISTFILE = (NODIAG)
LIST = YES	LIST
LIST = NO	NOLIST
MAP = YES	LIST = (MAP)
MAP = NO	LIST = (NOMAP)
OBJLST = YES	LIST = (OBJECT)
OBJLST = NO	LIST = (NOOBJECT)
SAVLST = SOURCE	LISTFILE = (SOURCE, DIAG, ESD, XREF, SUMMARY, OPTIONS)
SAVLST = LOCMAP	LISTFILE = (MAP)
SAVLST = OBJECT	LISTFILE = (OBJECT)
SAVLST = ALL	LISTFILE = (ALL)
SAVLST = NO	NOLISTFILE
XREF = YES	LIST = (XREF)
XREF = NO	LIST = (NOXREF)

Table A.4-1: PARAM operands and corresponding compiler options

For list output to SYSLST controlled by PARAM commands, the specification of LIST=YES is always mandatory.

A.5 IOSTAT messages

IOSTAT messages consist of

- the IOSTAT code
- and the message text.

Both are accessible to the user by program, which requires:

- the IOSTAT parameter in the OPEN, READ or WRITE statement (see "FOR1" Reference Manual [21]).
- the INCLUDE item IFNIOS from the FOR1 macro library (FOR1MACLIB).

Use

Specify one of the following statements depending on the message text required:

a) for **German** messages:

```
%INCLUDE $FOR1MACLIB(IFNIOS),'*D'='␣␣'
```

b) for **English** messages:

```
%INCLUDE $FOR1MACLIB(IFNIOS),'*E'='␣␣'
```

c) for **German and English** messages at the same time:

```
%INCLUDE $FOR1MACLIB(IFNIOS),'*D'='␣␣','*E'='␣␣'
```

To suppress output of messages of the INCLUDE item IFNIOS in the source program listing, EXPAND mode must be deactivated by means of COMOPT NOEXPAND.

Example:

Access to the text of an IOSTAT message by program: Because of the invalid ACCESS-METHOD specification in the SET-FILE-LINK command, the program issues IOSTAT 19 with its associated message text.

The following are displayed:

- source program
- commands for compilation and execution
- IOSTAT message

1. Program (in file IO.SRC):

```
   PROGRAM IOSEXAM
   OPEN (UNIT=10,ACCESS='DIRECT',ERR=20,IOSTAT=IOS)
    .
    .
    .
   STOP
20 WRITE (6,'(A)') IOSTATDEUTXT(IOS)
   STOP
   %INCLUDE LIBNAME(IFNIOS),'*D'='  '
   END
```

2. Compilation:

```
/SET-FILE-LINK LINK-NAME=LIBNAME, FILE-NAME=$TSOS.FOR1MACLIB
/START-PROG $FOR1
*COMOPT INCLUDE-LIBRARY=LIBNAME, SOURCE=IO.SRC, NOEXPAND, END
```

3. Execution of the program linked by TSOSLNK:

```
/SET-TASKLIB LIB=$TSOS.FOR1MODLIBS
/SET-FILE-LINK LINK-NAME=DSET10, FILE-NAME=filename, ACCESS-METHOD=SAM
/START-PROG L.IOSEXAMP
```

4. Output via SYSLST:

```
IOSTAT=19: OPEN PARAMS DO NOT MATCH FILE/DEVICE ATTRIBUTES
```

List of IOSTAT messages

The file IFNIOS contains all IOSTAT messages. All lines marked "*D" are German messages; all lines marked "*E" are English messages.

The English IOSTAT messages from IFNIOS are listed in the following:

```
*E     CHARACTER IOSTATENGTXT*104 (-1:200)
*E     DATA IOSTATENGTXT/
*E    F 'IOSTAT= -1: END-OF-FILE CONDITION'
*E    F,'IOSTAT=  0: OPERATION ENDED PROPERLY'
*E    F,'IOSTAT=  1: PREVIOUS I/O OPERATION NOT PROPERLY TERMINATED'
*E    F,'IOSTAT=  2: UNIT NUMBER OUT OF RANGE'
*E    F,'IOSTAT=  3: NO FILE CONNECTED TO REQUESTED UNIT'
*E    F,'IOSTAT=  4: OPERATION NOT PERMITTED ON REQUESTED UNIT'
*E    F,'IOSTAT=  5: OPERATION NOT PERMITTED ON REQUESTED FILE'
*E    F,'IOSTAT=  6: SEQUENCE OF OPERATIONS NOT PERMITTED'
*E    F,'IOSTAT=  7: INITIAL CALL PARAMS INCONSISTENT'
*E    F,'IOSTAT=  8: ILLEGAL PARAM IN INITIAL CALL'
*E    F,'IOSTAT= 14: PAM FILE NOT OPENED'
*E    F,'IOSTAT= 15: OUTPUT OPERATION ON READ-ONLY FILE'
*E    F,'IOSTAT= 16: OPEN PARAMS INCONSISTENT'
*E    F,'IOSTAT= 17: ILLEGAL FILENAME IN OPEN STATEMENT'
*E    F,'IOSTAT= 18: ILLEGAL PARAM IN OPEN STATEMENT'
*E    F,'IOSTAT= 19: OPEN PARAMS DO NOT MATCH FILE/DEVICE ATTRIBUTES'
*E    F,'IOSTAT= 20: "OLD" FILE NOT IN CATALOGUE OR EMPTY'
*E    F,'IOSTAT= 21: FILE COULD NOT BE OPENED'
*E    F,'IOSTAT= 22: NO DEVICE AVAILABLE FOR PRIVATE VOLUME'
*E    F,'IOSTAT= 23: "NEW" FILE ALREADY IN CATALOGUE AND NOT EMPTY'
*E    F,'IOSTAT= 24: PHYSICAL RECSIZE LESS THAN FORTRAN RECLENGTH. TRUNCA
*E    FTION MAY APPEAR'
*E    F,'IOSTAT= 25: ILLEGAL OR MISSING PASSWORD FOR PROTECTED FILE'
*E    F,'IOSTAT= 26: OPEN ON A LOCKED FILE'
*E    F,'IOSTAT= 27: PRIVATE VOLUME HAS NO STD LABELS'
*E    F,'IOSTAT= 28: ILLEGAL PARAMS IN CLOSE'
*E    F,'IOSTAT= 29: CLOSE PARAMS DON''T MATCH FILE/DEVICE ATTRIBUTES'
*E    F,'IOSTAT= 30: CLOSE STATUS NEITHER KEEP NOR DELETE, KEEP ASSUMED'
*E    F,'IOSTAT= 31: UNIT TO BE CLOSED IS NOT CONNECTED. CLOSE IGNORED'
*E    F,'IOSTAT= 32: IRRECOVERABLE ERROR WHILE EXECUTING CLOSE'
*E    F,'IOSTAT= 33: ILLEGAL CLOSE STATUS, KEEP ASSUMED'
*E    F,'IOSTAT= 34: ILLEGAL FILENAME IN INQUIRE'
*E    F,'IOSTAT= 35: FILENAME MISSING IN INQUIRE BY FILE'

*E    F,'IOSTAT= 36: ILLEGAL PARAMS IN INQUIRE'
*E    F,'IOSTAT= 37: WRPASS MISSING, OPEN-MODE -→ INPUT, FURTHER OUTPUT
*E    FPERATIONS WILL BE REJECTED'
*E    F,'IOSTAT= 38: FILE LOCKED, OPEN-MODE -→ INPUT, FURTHER OUTPUT OPER
*E    FATIONS WILL BE REJECTED'
*E    F,'IOSTAT= 39: OPEN ON STANDARD-FORTRAN-FILE. REDIRECTION ONLY WIT
*E    FH RUNOPT'
*E    F,'IOSTAT= 40: ILLEGAL COUNT IN POSITIONING STATEMENT'
*E    F,'IOSTAT= 41: BACKFILE/SKIPFILE ONLY ON TAPE FILES'
*E    F,'IOSTAT= 42: BACKSPACE/SKIPREC/REWIND ONLY ON SEQ. FILES'
*E    F,'IOSTAT= 43: FIND ONLY ON D.A. FILES'
*E    F,'IOSTAT= 44: RECORD FORMAT TYPE UNKNOWN'
*E    F,'IOSTAT= 48: RECORD NUMBER EXCEEDS MAXREC ATTRIBUTE OR : END OF S
*E    FTORAGE FILE'
```

```
*E   F,'IOSTAT= 49: RECORD SIZE EXCEEDS BUFFER SIZE.RECORD TRUNCATED'
*E   F,'IOSTAT= 50: IRRECOV. ERROR WHILE PROCESSING A SYSTEM FILE'
*E   F,'IOSTAT= 51: IRRECOV. ERROR WHILE PROCESSING AN EAM FILE'
*E   F,'IOSTAT= 52: IRRECOV. ERROR WHILE PROCESSING A USER FILE'
*E   F,'IOSTAT= 53: ILLEGAL USE OF  D M S'
*E   F,'IOSTAT= 54: SPECIFIED RECORD NOT FOUND'
*E   F,'IOSTAT= 55: MULTIPLE KEY DETECTED ON ISAM FILE'
*E   F,'IOSTAT= 56: HARDWARE ERROR'
*E   F,'IOSTAT= 57: SPECIFIED ISAM RECORD LOCKED'
*E   F,'IOSTAT= 58: RECORD POINTER OUTSIDE I/O BUFFER'
*E   F,'IOSTAT= 59: RECORD NUMBER NOT GREATER 0'
*E   F,'IOSTAT= 60: NON-NUMERIC KEY OR KEY TOO LARGE'
*E   F,'IOSTAT= 61: NON-CHARACTER KEY'
*E   F,'IOSTAT= 62: STRING LENGTH EXCEEDS RECORD LENGTH'
*E   F,'IOSTAT= 64: B-FORMAT NOT ALLOWED FOR INTEGER  DATUM'
*E   F,'IOSTAT= 65: B-FORMAT NOT ALLOWED FOR REAL DATUM'
*E   F,'IOSTAT= 66: D (E,F,Q) FORMAT NOT ALLOWED FOR CHARACTER DATUM'
*E   F,'IOSTAT= 67: D (E,F,Q) FORMAT NOT ALLOWED FOR BIT DATUM'
*E   F,'IOSTAT= 68: D (E,F,Q) FORMAT NOT ALLOWED FOR LOGICAL DATUM'
*E   F,'IOSTAT= 69: I FORMAT NOT ALLOWED FOR CHARACTER DATUM'
*E   F,'IOSTAT= 70: I FORMAT NOT ALLOWED FOR BIT DATUM'
*E   F,'IOSTAT= 71: I FORMAT NOT ALLOWED FOR LOGICAL DATUM'
*E   F,'IOSTAT= 72: L FORMAT NOT ALLOWED FOR INTEGER DATUM'
*E   F,'IOSTAT= 73: L FORMAT NOT ALLOWED FOR REAL DATUM'
*E   F,'IOSTAT= 74: L FORMAT NOT ALLOWED FOR CHARACTER DATUM'
*E   F,'IOSTAT= 75: BIT DATA NOT YET IMPLEMENTED'
*E   F,'IOSTAT= 76: REAL CONVERSION WITH I FORMAT NOT IMPLEMENTED'
*E   F,'IOSTAT= 77: INT. CONVERSION WITH D,E,Q FORMAT NOT IMPL.'
*E   F,'IOSTAT= 78: INCONSISTENT FORMAT'
*E   F,'IOSTAT= 80: INTEGER VALUE TOO LARGE'
*E   F,'IOSTAT= 81: ILLEGAL INTEGER VALUE'
*E   F,'IOSTAT= 82: ILLEGAL REAL VALUE'
*E   F,'IOSTAT= 83: ILLEGAL BIT VALUE'
*E   F,'IOSTAT= 84: ILLEGAL CHARACTER VALUE'
*E   F,'IOSTAT= 85: ILLEGAL HEX. CHARACTER'
*E   F,'IOSTAT= 86: ILLEGAL LOGICAL VALUE'
*E   F,'IOSTAT= 96: FORMAT SYNTAX ERROR: MISSING LEFT PARENTHESIS '
*E   F,'IOSTAT= 97: FORMAT SYNTAX ERROR: FORMAT EMPTY'
*E   F,'IOSTAT= 98: FORMAT SYNTAX ERROR: ILLEGAL FORMAT ELEMENT'
*E   F,'IOSTAT= 99: FORMAT SYNTAX ERROR: ILLEGAL FACTOR ZERO'
*E   F,'IOSTAT=100: FORMAT SYNTAX ERROR: EMPTY FORMAT GROUP'
*E   F,'IOSTAT=101: FORMAT SYNTAX ERROR: MISSING RIGHT PARENTHESIS'
*E   F,'IOSTAT=102: FORMAT SYNTAX ERROR: REPETITION FACTOR WITHOUT FORMA
*E   FT ELEMENT'
*E   F,'IOSTAT=103: FORMAT SYNTAX ERROR: ILLEGAL POSITION OF SIGN '
*E   F,'IOSTAT=104: FORMAT SYNTAX ERROR: REPETITION FACTOR NOT ALLOWED'
*E   F,'IOSTAT=105: FORMAT SYNTAX ERROR: MISSING END QUOTE OF LITERAL'
*E   F,'IOSTAT=106: FORMAT SYNTAX ERROR: LITERAL OF LENGTH 0'
*E   F,'IOSTAT=107: FORMAT SYNTAX ERROR: REPETITION FACTOR TOO LARGE'
*E   F,'IOSTAT=108: FORMAT SYNTAX ERROR: DELIMITER MISSING'
*E   F,'IOSTAT=109: FORMAT SYNTAX ERROR: FIELD WIDTH MISSING'
*E   F,'IOSTAT=110: FORMAT SYNTAX ERROR: FRACTIONAL PART MISSING'
*E   F,'IOSTAT=111: FORMAT SYNTAX ERROR: EXPONENT DESIGNATOR ONLY WITH E
*E   F/G FORMAT'
*E   F,'IOSTAT=112: FORMAT SYNTAX ERROR: MISSING EXPONENT LENGTH'
*E   F,'IOSTAT=113: OBJECT FORMAT SYNTAX ERROR: FRACTIONAL PART NOT ALLO
*E   FWED'
*E   F,'IOSTAT=114: OBJECT FORMAT SYNTAX ERROR: MISSING NUMERIC COUNT'
*E   F,'IOSTAT=115: OBJECT FORMAT SYNTAX ERROR: FED SEQUENCE ERROR'
*E   F,'IOSTAT=116: OBJECT FORMAT SYNTAX ERROR: NUMERIC PART TOO LARGE'
*E   F,'IOSTAT=117: OBJECT FORMAT SYNTAX ERROR: FIELD WIDTH TOO LARGE'
*E   F,'IOSTAT=118: OBJECT FORMAT SYNTAX ERROR: FIELD WIDTH W=0'
*E   F,'IOSTAT=119: NO DATA FORMAT IN REVERSION PART.REVERSION REJECTED'
*E   F,'IOSTAT=120: UNFORMATTED I/O READS ONLY 1 RECORD!'
*D   F'
```

```
*E   F,'IOSTAT=121: ILLEGAL IDENTIFIER IN ASYNCHR. WAIT'
*E   F,'IOSTAT=122: GCW: TOO MANY SUBRECORDS'
*E   F,'IOSTAT=123: UNFORMATTED WRITE: RECORD SPLIT OCCURRED'
*E   F,'IOSTAT=128: ILLEGAL CHARACTER IN RECORD'
*E   F,'IOSTAT=129: ILLEGAL VALUE SEQUENCE'
*E   F,'IOSTAT=130: ILLEGAL COMPLEX VALUE'
*E   F,'IOSTAT=131: ILLEGAL LEFT PARENTHESIS'
*E   F,'IOSTAT=132: STRING WITHOUT QUOTES'
*E   F,'IOSTAT=144: ILLEGAL NAME IN NAMELIST RECORD'
*E   F,'IOSTAT=145: VARIABLE NOT IN NAMELIST'
*E   F,'IOSTAT=146: SIMPLE VARIABLE WITH INDICES'
*E   F,'IOSTAT=147: ILLEGAL NUMBER OF INDICES'
*E   F,'IOSTAT=148: NAMELIST NAME MISSING IN FIRST RECORD'
*E   F,'IOSTAT=149: NULL VALUE IN NAMELIST RECORD'
*E   F,'IOSTAT=150: TOO MANY VALUES FOLLOWING NAME OR NAME MISSING'
*E   F,'IOSTAT=151: END MISSING'
*E   F,'IOSTAT=152: ILLEGAL INDEX'
*E   F,'IOSTAT=153: REPETITION FACTOR TOO HIGH: (REPFAC * DATA LENGTH >
*E   F32 KBYTE)'
*E   F,'IOSTAT=156: IRRECOVERABLE ERROR DURING PAUSE/STOP'
*E   F,'IOSTAT=157: ILLEGAL DATA TYPE IN PAUSE/STOP'
*E   F,'IOSTAT=158: WARNING:PAUSE MESSAGE TRUNCATED'
*E   F,'IOSTAT=159: WARNING:PAUSE ANSWER TRUNCATED'
*D   F/
```

A.6 Examples of compiler listings

A.6.1 Source listing with diagnostic listing

```
   ****  SOURCE  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 16:35:28   PAGE    1
                                                     PROGRAM UNIT: INV
D DO/IF SEG   STMT  I/H LINE        SOURCE-TEXT                                                    COL73-80 RECORD-ID.

        1/1      1         1        PROGRAM INV                                                              07000000
          1      2         2        COMMON IR                                                                15000000
 *        1      3         3    1   WRITE (2,10)                                                             23000000            *
 ***** WARNING (SA047) *******                                                                    *UNREFERENCED LABEL************
          1      4         4        READ (1,11) IR                                                           30000000
          1      5         5        IF (IR.GT.3) THEN                                                        38000000
 *  1     2      6         6          GO TO 2                                                                46000000            *
 ***** SEVERE  (FA185) *******                                                                    *LABEL 2 NOT DECLARED**********
    1     2      7         7        ELSE IF (IR.EQ.0) THEN                                                   53000000
    1     3      8         8          CALL DETS (A)                                                          61000000
    1     3      9         9        END IF                                                                   69000000
          3     10        10   10   FORMAT('    ENTER RANK OF MATRIX')                                       76000000
          3     11        11   11   FORMAT (I1)                                                              84000000
          4     12        12        END                                                                      92000000
   *** DIAGNOSTIC  LISTING ***     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 16:35:28   PAGE    2
                                                     PROGRAM UNIT: INV
D DO/IF SEG   STMT  I/H LINE        SOURCE-TEXT                                                    COL73-80 RECORD-ID.

 *        1      3         3    1   WRITE (2,10)                                                             23000000            *
       WARNING (SA047)  UNREFERENCED LABEL #1
 *  1     2      6         6          GO TO 2                                                                46000000            *
       SEVERE  (FA185)  LABEL 2 NOT DECLARED
```

Note Statement 6 should read "GOTO 1".

A.6.2 Source listings (main program and subprogram)

```
   ****  SOURCE  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:41:01   PAGE    1
                                                     PROGRAM UNIT: DIALOG
  DO/IF SEG   STMT  I/H LINE        SOURCE-TEXT                                                        COL73-80 RECORD-ID.

        1/1      1         1        PROGRAM DIALOG                                                               02000000
          1      2         2        REAL A(10),B                                                                 03000000
          1      3         3        DIMENSION B(5)                                                               04000000
                           4  ****                                                                               05000000
          2      4         5        DO 10 I=1,5                                                                  06000000
    1     3      5         6        B(I)=A(I)+I                                                                  07000000
    1     4      6         7  10    A(I)=I                                                                       08000000
          4      7         8        WRITE (2,11) A                                                               09000000
          4      8         9  11    FORMAT ('***',5F5.2)                                                         10000000
          4      9        10        CALL SUBA(A(1))                                                              11000000
          4     10        11        END                                                                          12000000
   ****  SOURCE  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:43:33   PAGE    2
                                                     PROGRAM UNIT: SUBA
  DO/IF SEG   STMT  I/H LINE        SOURCE-TEXT                                                        COL73-80 RECORD-ID.

        1/1      1         1        SUBROUTINE SUBA(X)                                                           18000000
          1      2         2        Y=X*X                                                                        19000000
          1      3         3        END                                                                          20000000
```

A.6.3 Change listing

```
 ****  CHANGE  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30   TIME = 15:41:01   PAGE    1
                                                    PROGRAM UNIT: DIALOG
(OLD) @   3.0000:      REEL A(10),B
(NEW) @   3.0000:      REAL A(10),B
(IN)  @P
(IN)  @   5.0000:      DO 10 I=1,5
(OLD) @   5.0000:      DO 10 I=1.5
(NEW) @   5.0000:      DO 10 I=1,5
(IN)  @CON
(OUT)  FOR1: RECOMPILATION OF ACTUAL P.U. INITIATED
(IN)  @I3.5
(IN)  @IN3.5
(NEW) @   3.5000:      DIMENSION B(5)
(IN)  @R
(IN)  @P1-2
(IN)  @D2
(OUT)  FOR1: 1 LINE(S) DELETED
(IN)  @CON
(OUT)  FOR1: RECOMPILATION OF ACTUAL P.U. INITIATED
```

A.6.4 ESD listing

```
 ****  E S D   LISTING   ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:41:01   PAGE    1
                                                  PROGRAM UNIT: DIALOG
 IDENTIFIER ESID  TYPE  DISPL.  LENGTH     IDENTIFIER ESID  TYPE  DISPL.  LENGTH     IDENTIFIER ESID  TYPE  DISPL.  LENGTH
 _____________________________________     _____________________________________     _____________________________________
 DIALOG     0001   SD   000000  000250     IF@FCTL    000C   VC   000168             IF@XINI    0009   VC   00001C
 DIALOG@@   0002   SD   000250  000278     IF@FEDC    000D   VC   00016C             IF@XPRO    000A   VC   000160
 I@@@RTCA   0003   CM   000000  001000     IF@RETN    000F   VC   000000             IF@XTCA    0005   VC   0001E4
 IF@@MPI    0002   LD   000000             IF@SINI    000E   VC   000170             IT0PCD     0004   VC   000000
 IF@CTER    0010   VC   000000             IF@XFCO    0006   VC   0001E0             SUBA       0008   VC   0001D8
 IF@ERROR   000B   VC   000164             IF@XICA    0007   VC   0001DC
 ****  E S D   LISTING   ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:43:33   PAGE    2
                                                  PROGRAM UNIT: SUBA
 IDENTIFIER ESID  TYPE  DISPL.  LENGTH     IDENTIFIER ESID  TYPE  DISPL.  LENGTH     IDENTIFIER ESID  TYPE  DISPL.  LENGTH
 _____________________________________     _____________________________________     _____________________________________
 I@@@RTCA   0003   CM   000000  001000     IF@ERROR   0006   VC   000154             IT0PCD     0004   VC   000000
 IF@@MPI    0005   ER   000000             IF@RETN    0008   VC   000000             SUBA       0001   SD   000000  0001E8
 IF@CTER    0009   VC   000000             IF@SINI    0007   VC   000158             SUBA@@@@   0002   SD   0001E8  000138
```

A.6.5 Map listing

```
  ****  M A P  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:41:01   PAGE    1
CODE + CONSTANTS SECTION: DIALOG      ESID=0001     PROGRAM UNIT: DIALOG                                  SORTED BY ADDRESS
SYMBOL            TYPE         ADR            SYMBOL           TYPE        ADR            SYMBOL           TYPE        ADR
--------------------------------------------------------------------------------------------------------------------------------
10                STMNT_LABEL  00000080    |  11               FORMT_LABEL             |
  ****  M A P  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:41:01   PAGE    2
CODE + CONSTANTS SECTION: DIALOG      ESID=0001     PROGRAM UNIT: DIALOG                                  SORTED BY SYMBOL
SYMBOL            TYPE         ADR            SYMBOL           TYPE        ADR            SYMBOL           TYPE        ADR
--------------------------------------------------------------------------------------------------------------------------------
10                STMNT_LABEL  00000080    |  11               FORMT_LABEL             |
  ****  M A P  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:41:01   PAGE    3
L O C A L  DATA  SECTION: DIALOG@@    ESID=0002     PROGRAM UNIT: DIALOG                                  SORTED BY ADDRESS
SYMBOL            TYPE         ADR            SYMBOL           TYPE        ADR            SYMBOL           TYPE        ADR
--------------------------------------------------------------------------------------------------------------------------------
SUBA.A            EXT_SUBR     000001D8    |  IF@XTCA.A        EXT_SUBR    000001E4    |  I                I4          00000424
IF@XICA.A         EXT_SUBR     000001DC    |  A                R4(1)       000003E8    |  B.D              R4(1)       00000484
IF@XFCO.A         EXT_SUBR     000001E0    |  B                R4(1)       00000410    |  A.D              R4(1)       00000494
  ****  M A P  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:41:01   PAGE    4
L O C A L  DATA  SECTION: DIALOG@@    ESID=0002     PROGRAM UNIT: DIALOG                                  SORTED BY SYMBOL
SYMBOL            TYPE         ADR            SYMBOL           TYPE        ADR            SYMBOL           TYPE        ADR
--------------------------------------------------------------------------------------------------------------------------------
A                 R4(1)        000003E8    |  B.D              R4(1)       00000484    |  IF@XICA.A        EXT_SUBR    000001DC
A.D               R4(1)        00000494    |  I                I4          00000424    |  IF@XTCA.A        EXT_SUBR    000001E4
B                 R4(1)        00000410    |  IF@XFCO.A        EXT_SUBR    000001E0    |  SUBA.A           EXT_SUBR    000001D8
  ****  M A P  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:43:33   PAGE    5
L O C A L  DATA  SECTION: SUBA@@@@    ESID=0002     PROGRAM UNIT: SUBA                                    SORTED BY ADDRESS
SYMBOL            TYPE         ADR            SYMBOL           TYPE        ADR            SYMBOL           TYPE        ADR
--------------------------------------------------------------------------------------------------------------------------------
X                 R4           00000304    |  Y                R4          00000308    |
  ****  M A P  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:43:33   PAGE    6
L O C A L  DATA  SECTION: SUBA@@@@    ESID=0002     PROGRAM UNIT: SUBA                                    SORTED BY SYMBOL
SYMBOL            TYPE         ADR            SYMBOL           TYPE        ADR            SYMBOL           TYPE        ADR
--------------------------------------------------------------------------------------------------------------------------------
X                 R4           00000304    |  Y                R4          00000308    |
```

A.6.6 Cross-reference listing

```
  **** X R E F  LISTING ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:41:01   PAGE    1
                                                    PROGRAM UNIT: DIALOG
IDENTIFIER          DISPL   DESCR   SPEC   ATTRIBUTES AND REFERENCES (/ : SPECIFICATION;  = : ASSIGNMENT)
--------------------------------------------------------------------------------------------------------------------------------
A                   0003E8 000494      2   ARRAY, REAL*4(1) ------ /2 5 =6 7 =9
B                   000410 000484      2   ARRAY, REAL*4(1) ------ /2 /3 =5
DIALOG                                 *   PROGRAM NAME ------ 1
I                   000424             *   VARIABLE, INTEGER*4 ------ =4 5 5 5 6 6
SUBA                       0001D8      *   EXTERNAL SUBROUTINE ------ 9
10                  000080             6   STATEMENT LABEL ------ 4 /6
11                                     8   FORMAT LABEL ------ 7 8 /8
  **** X R E F  LISTING ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:43:33   PAGE    2
                                                    PROGRAM UNIT: SUBA
IDENTIFIER          DISPL   DESCR   SPEC   ATTRIBUTES AND REFERENCES (/ : SPECIFICATION;  = : ASSIGNMENT)
--------------------------------------------------------------------------------------------------------------------------------
SUBA                                   *   SUBROUTINE NAME(1) ------ 1
X                   000304             *   VARIABLE, REAL*4, DUMMY ARGUMENT ------ 1 2 2
Y                   000308             *   VARIABLE, REAL*4 ------ =2
```

A.6.7 Object listing

```
  ****  OBJECT  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:41:01   PAGE    1
                                                         PROGRAM UNIT: DIALOG
FLG DISPL   OPERATION       ADDR1  ADDR2   STMNT     ASSEMBLY CODE                           SYMB.ADDR1, SYMB.ADDR2
---------------------------------------------------------------------------------------------------------------------------------
                                               1     ***************************************
                                               2     **        C O D E    A R E A         **
                                               3     ***************************************
                                               4     *
    000000                                     5     DIALOG   EQU   *
    000000                                     6     *
                                               7     ***** STATEMENT 1 (ENTRY) *************
                                               8     *       PROGRAM DIALOG
                                               9     *                                                 **** SEGMENT 1 ****
    000000  17 EE                             10              XR    14,14
    000002  05 B0                             11              BALR  11,0
                                              12     * CODE SLICE BEGIN *                    (SLICE 1)
    000004                                    13              USING *,11
    000004  98 CD B010      000014            14              LM    12,13,16(11)
    000008  47 F0 B024      000028            15              BC    15,36(0,11)
    00000C  05                                16              DC    AL1(05)
    00000D  C4C9C1D3D6C740                    17              DC    CL7’DIALOG ’
    000014  000000D8                          18              DC    A(DIALOG##)
    000018  00000250                          19              DC    A(DIALOG@@)
    00001C  00000000                          20              DC    V(IF@XINI)
    000020  00000024                          21              DC    A(*+4)
    000024                                    22              DC    X’0000000C’
    0000D8                                    23              USING DIALOG##,12
    000250                                    24              USING DIALOG@@,13
    000028  58 F0 B018      00001C            25              L     15,24(0,11)
    00002C  D5 03 B018 C0A0 00001C 000178     26              CLC   24(4,11),160(12)
    000032  47 80 0001      000001            27              BC    8,1(0,0)
    000036  50 E0 D004      000254            28              ST    14,4(0,13)
    00003A  41 10 B01C      000020            29              LA    1,28(0,11)
    00003E  05 EF                             30              BALR  14,15
    000040  58 90 D04C      00029C            31              L     9,76(0,13)
    000044  50 D0 900C      00000C            32              ST    13,12(0,9)
                                              33     ***** STATEMENT 4 (DO) ****************
                                              34     *       DO 10 I=1,5
    000048  41 F0 0001      000001            35              LA    15,1(0,0)              1
    00004C  50 F0 D1D4      000424            36              ST    15,468(0,13)           I
    000050  41 A0 0005      000005            37              LA    10,5(0,0)              5
                                              38     ***** STATEMENT 5 (MOVED STMT) ********
    000054  41 10 0004      000004            39              LA    1,4(0,0)               4
    000058  41 30 0004      000004            40              LA    3,4(0,0)               4
    00005C  58 50 D1D4      000424            41              L     5,468(0,13)            I
    000060  47 F0 B062      000066            42              BC    15,98(0,11)            %L5
                                              43     ***** STATEMENT 5 (INCR STMT) *********           **** SEGMENT 2 ****
    000064  1A 13                             44     %L3      AR    1,3                    %V1, %V2
                                              45     ***** STATEMENT 5 (ASSIGNMENT) ********
                                              46     *       B(I)=A(I)+I
                                              47     *                                                 **** SEGMENT 3 ****
    000066  78 01 D194      0003E4            48     %L5      LE    0,404(1,13)            A
    00006A  68 20 C0E0      0001B8            49              LD    2,224(0,12)
    00006E  50 50 D0A4      0002F4            50              ST    5,164(0,13)            I
    000072  97 80 D0A4      0002F4            51              XI    164(13),128
    000076  6B 20 D0A0      0002F0            52              SD    2,160(0,13)
    00007A  3A 02                             53              AER   0,2                    I
    00007C  70 01 D1BC      00040C            54              STE   0,444(1,13)            B
                                              55     ***** STATEMENT 6 (ASSIGNMENT) ********
                                              56     * 10    A(I)=I
    000080  68 00 C0E0      0001B8            57     #10      LD    0,224(0,12)
  ****  OBJECT  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:41:01   PAGE    2
                                                   PROGRAM UNIT: DIALOG
 FLG DISPL    OPERATION       ADDR1  ADDR2    STMNT     ASSEMBLY CODE                            SYMB.ADDR1, SYMB.ADDR2

     000084  50 50 D0A4      0002F4             58               ST    5,164(0,13)             I
     000088  97 80 D0A4      0002F4             59               XI    164(13),128
     00008C  6B 00 D0A0      0002F0             60               SD    0,160(0,13)
     000090  70 01 D194      0003E4             61               STE   0,404(1,13)             A
                                                62      ***** STATEMENT 6 (DOEND) *************
                                                63      * 10    A(I)=I
     000094  1A 5F                              64               AR    5,15                    I, 1
     000096  46 A0 B060      000064             65               BCT   10,96(0,11)             %L3
                                                66      *                                                          **** SEGMENT 4 ****
     00009A  50 50 D1D4      000424             67               ST    5,468(0,13)             I
                                                68      ***** STATEMENT 7 (WRITE) *************
                                                69      *       WRITE (2,11) A
     00009E  41 10 D1F0      000440             70               LA    1,496(0,13)
     0000A2  58 F0 C104      0001DC             71               L     15,260(0,12)
     0000A6  41 00 0003      000003             72               LA    0,3(0,0)
     0000AA  05 EF                              73               BALR  14,15                   IF@XICA
     0000AC  58 F0 C108      0001E0             74               L     15,264(0,12)
     0000B0  41 10 D214      000464             75               LA    1,532(0,13)
     000464                                     76               USING DIALOG@@+532,1
     0000B4  41 00 0001      000001             77               LA    0,1(0,0)
     0000B8  05 EF                              78               BALR  14,15                   IF@XFCO
     0000BA  58 F0 C10C      0001E4             79               L     15,268(0,12)
     0000BE  17 00                              80               XR    0,0
     0000C0  05 EF                              81               BALR  14,15                   IF@XTCA
                                                82      ***** STATEMENT 9 (CALL) **************
                                                83      *       CALL SUBA(A(1))
     0000C2  58 F0 C100      0001D8             84               L     15,256(0,12)
     0000C6  41 10 D1E0      000430             85               LA    1,480(0,13)
     000430                                     86               USING DIALOG@@+480,1
     0000CA  41 00 0001      000001             87               LA    0,1(0,0)
     0000CE  05 EF                              88               BALR  14,15                   SUBA
                                                89      ***** STATEMENT 10 (END) **************
                                                90      *       END
     0000D0  58 F0 C088      000160             91               L     15,136(0,12)            IF@XPRO
     0000D4  05 EF                              92               BALR  14,15
                                                93      **************************************
                                                94      **     C O N S T A N T   A R E A     **
                                                95      **************************************
                                                96      *
     0000D6                                     97               ORG
     0000D8                                     98      DIALOG## DS    0D
     0000D8                                     99               USING *,12
                                               100      *                                      RUNTIME COMMUNICATION AREA
     0000D8  F3F0C1E4C7F9F1                    101               DC    '30AUG91'
     0000DF  5B081E0F29019135                  102               DC    X'5B081E0F'
     0000E7  40C6D6D9F140E74040E5F24BF2C1F0    103               DC    ' FOR1 X  V2.2A00'
     0000F7  A0                                104               DC    X'A0'
     0000F8  00000004                          105               DC    A(DIALOG)
     0000FC  FFFFFFFF                          106               DC    F'-1'
     000100  41000411                          107               DC    X'41000411'
     000104  58F0C08C                          108               DC    X'58F0C08C'
     000108  07FF                              109               DC    X'07FF'
     00010C                                    110               ORG   DIALOG##+52
     00010C  00000004                          111               DC    A(DIALOG+4)             CODE SLICE ADDRESS (SLICE 1)
     0001E4                                    112               ORG   DIALOG##+268
     0001E4  00000000                          113               DC    V(IF@XTCA)              EXTERNAL REFERENCE
  ****  OBJECT  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:41:01   PAGE    3
                                                   PROGRAM UNIT: DIALOG
FLG DISPL   OPERATION        ADDR1  ADDR2    STMNT     ASSEMBLY CODE                          SYMB.ADDR1, SYMB.ADDR2

    0001E0                                    114              ORG   DIALOG##+264
    0001E0  00000000                          115              DC    V(IF@XFCO)
    0001DC                                    116              ORG   DIALOG##+260
    0001DC  00000000                          117              DC    V(IF@XICA)
    0001D8                                    118              ORG   DIALOG##+256
    0001D8  00000000                          119              DC    V(SUBA)
    000160                                    120              ORG   DIALOG##+136
    000160  00000000                          121              DC    V(IF@XPRO)
    000164  00000000                          122              DC    V(IF@ERROR)
    000168  00000000                          123              DC    V(IF@FCTL)
    00016C  00000000                          124              DC    V(IF@FEDC)
    000170  00000000                          125              DC    V(IF@SINI)
    000178                                    126              ORG   DIALOG##+160
    000178  00000000                          127              DC    X'00000000'            0.0000000000000000000000000000000000+(
    00017C  00000000                          128              DC    X'00000000'
    000180  00000000                          129              DC    X'00000000'
    000184  00000000                          130              DC    X'00000000'
    000198                                    131              ORG   DIALOG##+192
    000198  00000001                          132              DC    X'00000001'            1
    00019C  00000002                          133              DC    X'00000002'            2
    0001A8                                    134              ORG   DIALOG##+208
    0001A8  4E000000                          135              DC    X'4E000000'            0.000000000000000000E+00
    0001AC  00000000                          136              DC    X'00000000'
    0001B0  4E000001                          137              DC    X'4E000001'            0.429496729600000000E+10
    0001B4  00000000                          138              DC    X'00000000'
    0001B8  CE000000                          139              DC    X'CE000000'            -0.214748364800000000E+10
    0001BC  80000000                          140              DC    X'80000000'
    0001C0  40                                141              DC    ' '
    0001CC                                    142              ORG   DIALOG##+244
    0001CC  00000005                          143              DC    X'00000005'            5
    0001D0  00000004                          144              DC    X'00000004'            4
    0001D4  01018600                          145              DC    X'01018600'            16877056
    0001EA                                    146              ORG   DIALOG##+274
    0001EA  4DF3C85C5C5C6BF5C6F54BF25D4040    147              DC    '(3H***,5F5.2)   '
                                              148       *                                   EVENTHANDLER LIST
    0001FC                                    149              ORG   DIALOG##+292
    0001FC  01E70000                          150              DC    X'01E70000'
    000200  00000214                          151              DC    X'00000214'            ADDR OF PU_NAME
    000204  00000000                          152              DC    V(IF@RETN)             TERMINATION ROUTINE
    000208  00000000                          153              DC    V(IF@CTER)             PROCHK ROUTINE
    00020C  00000000                          154              DC    V(IF@CTER)             ERROR ROUTINE
    000210  FFFFFFFF                          155              DC    X'FFFFFFFF'            OTHER EVENTS ROUTINE
    000214  06                                156              DC    X'06'
    000215  C4C9C1D3D6C7                      157              DC    'DIALOG'
    00014C                                    158              ORG   DIALOG##+116
    00014C  0000021C                          159              DC    A(DIALOG##+324)        ADDRESS OF STATEMENT TABLE
    00021C                                    160              ORG   DIALOG##+324
    00021C  0000003000FC0000                  161              DC    X'0000003000FC0000'
    000224  000124FC00000004                  162              DC    X'000124FC00000004'
    00022C  0608FC00000005FF                  163              DC    X'0608FC00000005FF'
    000234  01FC00000005FF0D                  164              DC    X'01FC00000005FF0D'
    00023C  0AFC00000006FF05                  165              DC    X'0AFC00000006FF05'
    000244  12FC000000090703                  166              DC    X'12FC000000090703'
  ****  OBJECT  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:41:01   PAGE    4
                                                          PROGRAM UNIT: DIALOG
FLG DISPL   OPERATION       ADDR1  ADDR2   STMNT     ASSEMBLY CODE                           SYMB.ADDR1, SYMB.ADDR2

                                            167      **************************************
                                            168      **        D A T A    A R E A        **
                                            169      **************************************
                                            170      *
    00024D                                  171               ORG
    000250                                  172               DS    0D
    000250                                  173      DIALOG@@ CSECT
    000250                                  174               USING *,13
    000250                                  175               DS    632C
    000250                                  176               ORG   DIALOG@@+0
    000250  0001FEFF                        177               DC    X'0001FEFF'              SAVE AREA
    000298                                  178               ORG   DIALOG@@+72
    000298  00000000                        179               DC    A(I@@@RTCA)              RUNTIME COMMUNICATION AREA
    00029C  00000000                        180               DC    V(IT0PCD)
    0002A0  000001FC                        181               DC    X'000001FC'              A(IT0EHL)
    0002A0                                  182               ORG   DIALOG@@+80
    0002A0  0000021C00000000                183               DC    X'0000021C00000000'      STMT TABLE AND COUNT TABLE ADDR
    0002F0                                  184               ORG   DIALOG@@+160
    0002F0  CE00000000000000                185               DC    X'CE00000000000000'
    0002FC                                  186               ORG   DIALOG@@+172
    0002FC  05                              187               DC    X'05'
    0002FD  C4C9C1D3D6C740                  188               DC    'DIALOG '
    0003DC                                  189               ORG   DIALOG@@+396
    0003DC  820004A4                        190               DC    A(DIALOG##+596)          ADDRESS OF FORMAT LABEL TABLE
    0003E0  000004C8                        191               DC    A(DIALOG@@+632)          DATA SLICE ADDRESS (ITA)
    000430                                  192               ORG   DIALOG@@+480
    000430  000003E8                        193               DC    X'000003E8'              A()             ARGUMENT LIST
    000434  00000494                        194               DC    X'00000494'
    000438  FF030010                        195               DC    X'FF030010'
    00043C  05060000                        196               DC    X'05060000'
    000440  000001D4                        197               DC    X'000001D4'              16877056        ARGUMENT LIST
    000444  0000019C                        198               DC    X'0000019C'              2
    000448  000004B4                        199               DC    X'000004B4'              '(3H***,5F5.2)
    00044C  00000000                        200               DC    X'00000000'
    000450  00000000                        201               DC    X'00000000'
    000454  00000000                        202               DC    X'00000000'
    000458  FF030010                        203               DC    X'FF030010'
    00045C  01040104                        204               DC    X'01040104'
    000460  010C0000                        205               DC    X'010C0000'
    000464  000003E8                        206               DC    X'000003E8'              A               ARGUMENT LIST
    000468  00000494                        207               DC    X'00000494'
    00046C  FF030010                        208               DC    X'FF030010'
    000470  04060000                        209               DC    X'04060000'
    000484                                  210               ORG   DIALOG@@+564
    000484  00000410                        211               DC    X'00000410'              ARRAY DESCRIPTOR
    000488  00000424                        212               DC    X'00000424'
    00048C  0000040C                        213               DC    X'0000040C'
    000490  00010004                        214               DC    X'00010004'
    000494  000003E8                        215               DC    X'000003E8'              ARRAY DESCRIPTOR
    000498  00000410                        216               DC    X'00000410'
    00049C  000003E4                        217               DC    X'000003E4'
    0004A0  00010004                        218               DC    X'00010004'
    0004B4                                  219               ORG   DIALOG@@+612
    0004B4  00000000                        220               DC    X'00000000'              OBJECT FORMAT DESCRIPTOR
    0004B8  00000000                        221               DC    X'00000000'
    0004BC  00000010                        222               DC    X'00000010'
    0004C0  000001EA                        223               DC    X'000001EA'
  ****  OBJECT  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:41:01   PAGE    5
                                                          PROGRAM UNIT: DIALOG
FLG DISPL   OPERATION       ADDR1  ADDR2   STMNT     ASSEMBLY CODE                           SYMB.ADDR1, SYMB.ADDR2

    0004C8                                  224               END   DIALOG
  ****  OBJECT  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:43:33   PAGE    6
                                                         PROGRAM UNIT: SUBA
 FLG DISPL   OPERATION       ADDR1  ADDR2   STMNT     ASSEMBLY CODE                                   SYMB.ADDR1, SYMB.ADDR2

                                               1      ***************************************
                                               2      **        C O D E    A R E A         **
                                               3      ***************************************
                                               4      *
     000000                                    5      SUBA      EQU   *
     000000                                    6      *
                                               7      ***** STATEMENT 1 (ENTRY) *************
                                               8      *       SUBROUTINE SUBA(X)
                                               9      *                                                         **** SEGMENT 1 ****
                                              10      * CODE SLICE BEGIN *                    (SLICE 1)
     000000                                   11              USING *,15
     000000                                   12              USING *,11
     000000  90 EC D00C      00000C           13              STM   14,12,12(13)
     000004  45 E0 F018      000018           14              BAL   14,24(0,15)
     000008  03                               15              DC    AL1(03)
     000009  E2E4C2C1404040                   16              DC    CL7'SUBA   '
     000010  000000C8                         17              DC    A(SUBA####)
     000014  000001E8                         18              DC    A(SUBA@@@@)
     0000C8                                   19              USING SUBA####,12
     0001E8                                   20              USING SUBA@@@@,13
     000018  18 BF                            21              LR    11,15
     00001A  18 9D                            22              LR    9,13
     00001C  58 D0 B014      000014           23              L     13,20(0,11)
     000020  58 C0 B010      000010           24              L     12,16(0,11)
     000024  50 D0 9008      000008           25              ST    13,8(0,9)
     000028  17 77                            26              XR    7,7
     00002A  58 70 D04C      000234           27              L     7,76(0,13)
     00002E  D2 03 D110 700C 0002F8 00000C    28              MVC   272(4,13),12(7)
     000034  50 D0 700C      00000C           29              ST    13,12(0,7)
     000038  92 EC 9000      000000           30              MVI   0(9),236
     00003C  95 00 D000      0001E8           31              CLI   0(13),0
     000040  47 80 B068      000068           32              BC    8,104(0,11)
     000044  90 C0 9068      000068           33              STM   12,0,104(9)
     000048  41 00 0403      000403           34              LA    0,1027(0,0)
     00004C  18 D9                            35              LR    13,9
     00004E  58 E0 900C      00000C           36              L     14,12(0,9)
     000052  58 C0 9044      000044           37              L     12,68(0,9)
     000056  D2 02 D001 B061 0001E9 000061    38              MVC   1(3,13),97(11)
     00005C  47 F0 C02C      0000F4           39              BC    15,44(0,12)
     000060  00000064                         40              DC    AL4(*+4)
     000064  98 C0 9068      000068           41              LM    12,0,104(9)
     000068  50 90 D004      0001EC           42              ST    9,4(0,13)
     00006C  58 A0 1000      000000           43              L     10,0(0,1)
     000070  D2 03 D11C A000 000304 000000    44              MVC   284(4,13),0(10)         X
     000076  45 90 B0AE      0000AE           45              BAL   9,174(0,11)             %L3
                                              46      *** END OF PROLOG
     00007A                                   47              USING *,15
     00007A  58 20 D004      0001EC           48              L     2,4(0,13)
     00007E  58 20 2018      000018           49              L     2,24(0,2)
     000082  58 40 2000      000000           50              L     4,0(0,2)
     000086  D2 03 4000 D11C 000000 000304    51              MVC   0(4,4),284(13)          X
     00008C  58 E0 D04C      000234           52              L     14,76(0,13)
     000090  58 F0 D110      0002F8           53              L     15,272(0,13)
     000094  58 D0 D004      0001EC           54              L     13,4(0,13)
     000098  50 F0 E00C      00000C           55              ST    15,12(0,14)
     00009C  92 00 D000      0001E8           56              MVI   0(13),0
     0000A0  58 E0 D00C      0001F4           57              L     14,12(0,13)
  ****  OBJECT  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:43:33   PAGE    7
                                                   PROGRAM UNIT: SUBA
FLG DISPL   OPERATION       ADDR1  ADDR2   STMNT     ASSEMBLY CODE                        SYMB.ADDR1, SYMB.ADDR2
-------------------------------------------------------------------------------------------------------------------------------

    0000A4  58 F0 D010      0001F8            58              L     15,16(0,13)
    0000A8  98 2C D01C      000204            59              LM    2,12,28(13)
    0000AC  07 FE                             60              BCR   15,14
                                              61              DROP  15
                                              62     *** END OF EPILOG
                                              63     *** BEGIN OF TRAILER
    0000AE  50 90 D0B4      00029C            64     %L3      ST    9,180(0,13)
                                              65     ***** STATEMENT 2 (ASSIGNMENT) ********
                                              66     *       Y=X*X
    0000B2  78 00 D11C      000304            67     %L4      LE    0,284(0,13)            X
    0000B6  7C 00 D11C      000304            68              ME    0,284(0,13)            X
                                              69     ***** STATEMENT 3 (END) ***************
                                              70     *       END
    0000BA  70 00 D120      000308            71              STE   0,288(0,13)            Y
    0000BE  58 F0 D0B4      00029C            72              L     15,180(0,13)
    0000C2  17 11                             73              XR    1,1
    0000C4  05 EF                             74              BALR  14,15
                                              75     ***************************************
                                              76     **     C O N S T A N T   A R E A     **
                                              77     ***************************************
                                              78     *
    0000C6                                    79              ORG
    0000C8                                    80     SUBA#### DS    0D
    0000C8                                    81              USING *,12
                                              82     *                                     RUNTIME COMMUNICATION AREA
    0000C8  F3F0C1E4C7F9F1                    83              DC    '30AUG91'
    0000CF  5B081E0F2B219135                  84              DC    X'5B081E0F'
    0000D7  40C6D6D9F140E74040E5F24BF2C1F0    85              DC    ' FOR1 X  V2.2A00'
    0000E7  A0                                86              DC    X'A0'
    0000E8  00000000                          87              DC    A(SUBA)
    0000EC  FFFFFFFF                          88              DC    F'-1'
    0000F0  41000411                          89              DC    X'41000411'
    0000F4  58F0C08C                          90              DC    X'58F0C08C'
    0000F8  07FF                              91              DC    X'07FF'
    0000FC                                    92              ORG   SUBA####+52
    0000FC  00000000                          93              DC    A(SUBA+0)              CODE SLICE ADDRESS (SLICE 1)
    000154                                    94              ORG   SUBA####+140
    000154  00000000                          95              DC    V(IF@ERROR)
    000158  00000000                          96              DC    V(IF@SINI)
    000160                                    97              ORG   SUBA####+152
    000160  00000000                          98              DC    X'00000000'            0.0000000000000000000000000000000000+0
    000164  00000000                          99              DC    X'00000000'
    000168  00000000                         100              DC    X'00000000'
    00016C  00000000                         101              DC    X'00000000'
    000180                                   102              ORG   SUBA####+184
    000180  00000001                         103              DC    X'00000001'            1
    000190                                   104              ORG   SUBA####+200
    000190  4E000000                         105              DC    X'4E000000'            0.000000000000000000E+00
    000194  00000000                         106              DC    X'00000000'
    000198  4E000001                         107              DC    X'4E000001'            0.429496729600000000E+10
    00019C  00000000                         108              DC    X'00000000'
    0001A0  CE000000                         109              DC    X'CE000000'            -0.214748364800000000E+10
    0001A4  80000000                         110              DC    X'80000000'
    0001A8  40                               111              DC    ' '
                                             112     *                                     EVENTHANDLER LIST
```

```
   ****  OBJECT  LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:43:33   PAGE    8
                                                       PROGRAM UNIT: SUBA
 FLG DISPL   OPERATION       ADDR1  ADDR2   STMNT     ASSEMBLY CODE                            SYMB.ADDR1, SYMB.ADDR2

     0001B8                                  113               ORG   SUBA####+240
     0001B8  01E70000                        114               DC    X'01E70000'
     0001BC  000001D0                        115               DC    X'000001D0'              ADDR OF PU_NAME
     0001C0  00000000                        116               DC    V(IF@RETN)               TERMINATION ROUTINE
     0001C4  00000000                        117               DC    V(IF@CTER)               PROCHK ROUTINE
     0001C8  00000000                        118               DC    V(IF@CTER)               ERROR ROUTINE
     0001CC  FFFFFFFF                        119               DC    X'FFFFFFFF'              OTHER EVENTS ROUTINE
     0001D0  04                              120               DC    X'04'
     0001D1  E2E4C2C1                        121               DC    'SUBA'
     00013C                                  122               ORG   SUBA####+116
     00013C  000001D8                        123               DC    A(SUBA####+272)          ADDRESS OF STATEMENT TABLE
     0001D8                                  124               ORG   SUBA####+272

     0001D8  0000000D00FC0000                125               DC    X'0000000D00FC0000'
     0001E0  0001590406                      126               DC    X'0001590406'
                                             127      ***************************************
                                             128      **        D A T A    A R E A        **
                                             129      ***************************************
                                             130      *
     0001E6                                  131               ORG
     0001E8                                  132               DS    0D
     0001E8                                  133      SUBA@@@@ CSECT
     0001E8                                  134               USING *,13
     0001E8                                  135               DS    312C
     0001E8                                  136               ORG   SUBA@@@@+0
     0001E8  0001FEFF                        137               DC    X'0001FEFF'              SAVE AREA
     000230                                  138               ORG   SUBA@@@@+72
     000230  00000000                        139               DC    A(I@@@RTCA)              RUNTIME COMMUNICATION AREA
     000234  00000000                        140               DC    V(IT0PCD)
     000238  000001B8                        141               DC    X'000001B8'              A(IT0EHL)
     000238                                  142               ORG   SUBA@@@@+80
     000238  000001D800000000                143               DC    X'000001D800000000'      STMT TABLE AND COUNT TABLE ADDR
     000288                                  144               ORG   SUBA@@@@+160
     000288  CE00000000000000                145               DC    X'CE00000000000000'
     000294                                  146               ORG   SUBA@@@@+172
     000294  03                              147               DC    X'03'
     000295  E2E4C2C1404040                  148               DC    'SUBA   '
     0002FC                                  149               ORG   SUBA@@@@+276
     0002FC  82000310                        150               DC    A(SUBA####+296)          ADDRESS OF FORMAT LABEL TABLE
     000300  00000320                        151               DC    A(SUBA@@@@+312)          DATA SLICE ADDRESS (ITA)
     000320                                  152               END
```

A.6.8 Summary listings for main program and subprogram

```
****  SUMMARY LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:41:01   PAGE    1
                                                   PROGRAM UNIT: DIALOG
OBJECT MODULES GENERATED
________________________
( NAME | LENGTH | TYPE )
                         DIALOG    |  0004C8 (    1224)  |  CODE & DATA MODULE OF MAIN PROGRAM
COMPILATION STATISTICS
______________________

    OPTIMIZATION :         2 COMMON SUBEXPRESSIONS
                           2 CONSTANT EVALUATIONS
                           1 STRENGTH REDUCTIONS
     DIAGNOSTICS :         0 NOTES                     MEMORY USAGE :       582 VIRTUAL MEMORY PAGES
                           1 WARNINGS                                         0 WORK FILE PAGES
                           0 ERRORS
                           0 SEVERE ERRORS
                           0 FAILURES
                                                       COMPILE TIME :       152 SECONDS ELAPSED TIME
  SOURCE PROGRAM :        11 SOURCE LINES                                  1167 LINES / MINUTE CPU TIME
                           1 COMMENT LINES                                 .565 SECONDS CPU TIME
                                                                                (COMPILER NOT PRELOADED)
****  SUMMARY LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:43:33   PAGE    2
                                                   PROGRAM UNIT: SUBA
OBJECT MODULES GENERATED
________________________
( NAME | LENGTH | TYPE )
                         SUBA      |  000320 (     800)  |  CODE & DATA MODULE OF SUBROUTINE
COMPILATION STATISTICS
______________________

    OPTIMIZATION :           ACTIVE
     DIAGNOSTICS :         0 NOTES                     MEMORY USAGE :       582 VIRTUAL MEMORY PAGES
                           0 WARNINGS                                         0 WORK FILE PAGES
                           0 ERRORS
                           0 SEVERE ERRORS
                           0 FAILURES
                                                       COMPILE TIME :         6 SECONDS ELAPSED TIME
  SOURCE PROGRAM :         3 SOURCE LINES                                   420 LINES / MINUTE CPU TIME
                           0 COMMENT LINES                                 .429 SECONDS CPU TIME
                                                                                (COMPILER NOT PRELOADED)
```

A.6.9 General summary listing

```
****  SUMMARY LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:43:39   PAGE    3
OBJECT MODULES GENERATED
________________________
( NAME | LENGTH | TYPE )
                         DIALOG    |  0004C8 (    1224)  |  CODE & DATA MODULE OF MAIN PROGRAM
                         SUBA      |  000320 (     800)  |  CODE & DATA MODULE OF SUBROUTINE
COMPILATION STATISTICS
______________________

    OPTIMIZATION :         2 COMMON SUBEXPRESSIONS
                           2 CONSTANT EVALUATIONS
                           1 STRENGTH REDUCTIONS
     DIAGNOSTICS :         0 NOTES                     MEMORY USAGE :       582 VIRTUAL MEMORY PAGES
                           1 WARNINGS                                         0 WORK FILE PAGES
                           0 ERRORS
                           0 SEVERE ERRORS
                           0 FAILURES
                                                       COMPILE TIME :       158 SECONDS ELAPSED TIME
  SOURCE PROGRAM :        14 SOURCE LINES                                   836 LINES / MINUTE CPU TIME
                           1 COMMENT LINES                                1.005 SECONDS CPU TIME
                                                                                (COMPILER NOT PRELOADED)
```

A.6.10 Options listing

```
****  OPTIONS LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:43:39   PAGE    1
ENVIRONMENT
__________
                          PROCESSOR          :     UNDEFINED
                          OPERATING SYSTEM   :     BS2000 V10.0
                          COMPILER           :     F O R  1    V2.2A00
                          USER-ID.           :     F3390052
                          TSN                :     91TE
                          CALLER             :     UNDEFINED
                          PREPROCESSOR       :     UNDEFINED
OPTIONS FILE
___________
                          |COMOPT SOURCE=S.DIALOG                                                     |
                          |COMOPT LISTFILE=L.DIALOG(SOURCE,OPTIONS,MAP,ESD,CHANGE,XREF,OBJECT,SUMMA|
                          |        RY)                                                                |
                          |COMOPT COLLECT=(LF)                                                        |
                          |END                                                                        |
OPTIONS IN EFFECT
________________
( D = DEFAULT )
( P = PARAM COMMAND )
                               SOURCE           = S.DIALOG
                          D    CODE             = EBCDIC
                               UPD              = SRCFCB
                          D    UPD CODE         = EBCDIC
                          D    INCLUDE-LIBRARY  = *NO
                          D    CCOM             = ''
                          D    LINEEND          = ''
                          D    IMPLICIT
                          D    STANDARD-CHECK   = NO
                          D    EXTENDED-SYSTEM  = YES
                          D    SYMTEST          = MAP
                          D    LANGUAGE         = ENGLISH
                          D  NODIALOG
                               LIST             = ( SOURCE, ESD, MAP, XREF, OBJECT, SUMMARY, OPTIONS )
                               LIST-OUTPUT      = L.DIALOG
                          D    LINECNT          = 64
                               COLLECT          = ( LISTFILE )
                          D    EJECT
                          D    EXPAND
                          D    TEXT-SEPARATOR   = |
                          D    OBJECT           = ( * )
                          D    MODULE-LIBRARY   = *OMF
                          D  NOFPOOL
                               PROCEDURE        = NO
                          D    ALIGN
                          D  NOSUPPLIEDBOUND
                          D    TRUNCONST
                          D  NOPAD
                          D  NOCOMPATIBLE
                          D    GEN
                          D    OPTIMIZE         = 1
****  OPTIONS LISTING  ****     SIEMENS-NIXDORF FORTRAN COMPILER   FOR1 V2.2A00  DATE = 1991-08-30 TIME = 15:43:39   PAGE    2
                          D    SAVE-CONSTANT    = YES
                          D    REAL             = 4
                          D    UNIT
                          D    TESTOPT          = ( STNR )
                          D  NOEXPUNDERFLOW
                          D    ERRKILL          = FAILURE
                          D    MAXERR           = 100
                          D    SOURCE-FORMAT    = FIXED
                          D    MSGLEVEL         = WARNING
                          D    FORTRAN90-CHECK  = YES
                          D    LINKAGE          = STD
```

A.7 Coexistence of OLD, NXS, XS programs

As of version 2.2A the FOR1 compiler always generates XS object modules. With FOR1 versions 2.0A and 2.1A, the EXTENDED-SYSTEM={YES|NO} option enabled the user to control whether the compiled program units were to be generated as XS or NXS modules (default was NO).

With NXS programs, the parameter address list has a format which is compatible both with the format of the parameter address list of OLD programs (= programs compiled with a FOR1 version ≤ 1.6A) and with the format of the parameter address list of XS programs. NXS subprograms therefore, in addition to processing NXS parameter address lists, also process OLD and XS parameter address lists, assuming that only 24 bits of the 31-bit addresses are relevant in the case of the latter.

A.7.1 31-bit address mode and 24-bit address mode

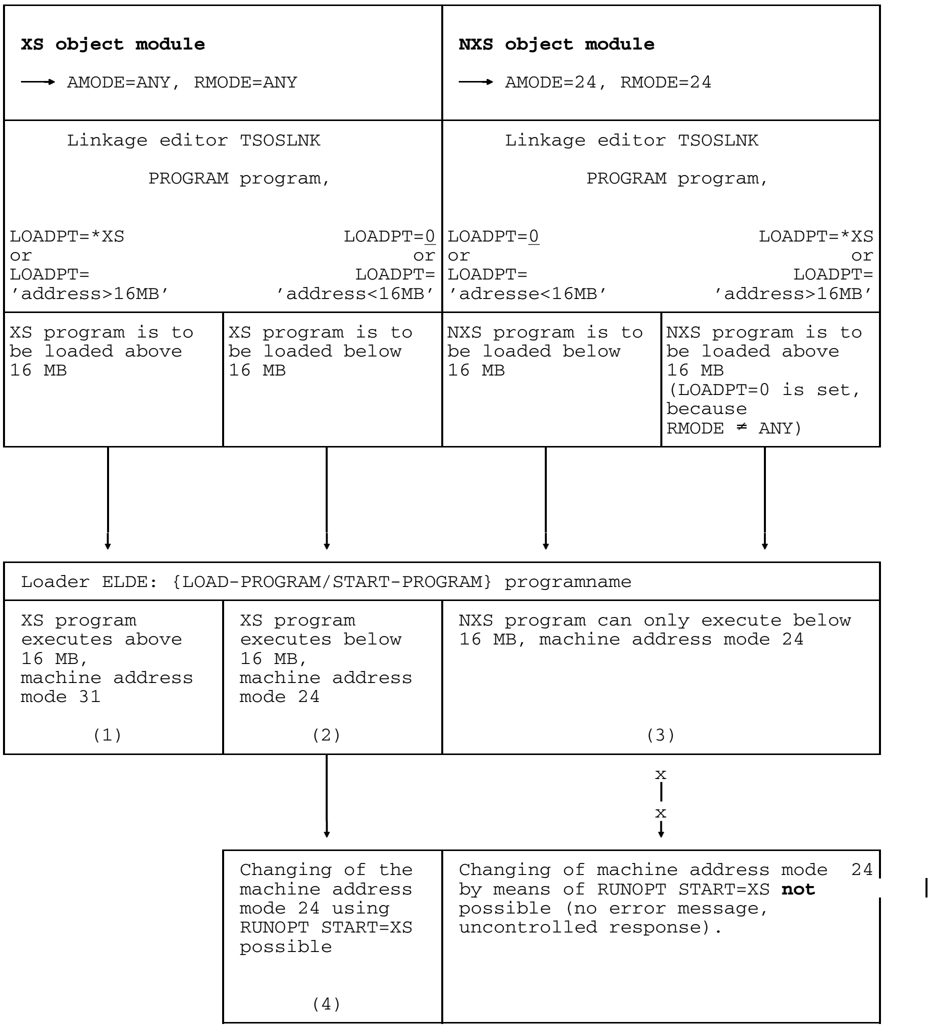
An XS object module has the attributes AMODE=ANY and RMODE=ANY, an XS object module has the attributes AMODE=24 and RMODE=24. These attributes are interpreted by the linkage/loading system or can be modified by means of the linkage/loading system. Furthermore the address mode determined by the loader can also be changed by means of the runtime option START=XS (see section 6.3.5).

Therefore the load address and the machine address mode in which a program executes are not specified until the program is started. A program can be executed in either the 24-bit or 31-bit address space.

Using the 24-bit mode, a loaded program (i.e. code as well as static and dynamic data) is run if it is within the 24-bit address space and processed in the 24-bit addressing mode by the hardware.

Using the 31 bit mode, a loaded program is run if it is within the 31-bit address space and processed in the 31-bit addressing mode by the hardware.

Figs. A.7-1 and A.7-2 show the different options according to which either an NXS program, an XS program in the address space below 16 Mbytes, or an XS program in the address space above 16 Mbytes is executed.



XS object module

⟶ AMODE=ANY, RMODE=ANY

NXS object module

⟶ AMODE=24, RMODE=24

Linkage editor TSOSLNK

PROGRAM program,

LOADPT=*XS or LOADPT= 'address>16MB'

LOADPT=0 or LOADPT= 'address<16MB'

Linkage editor TSOSLNK

PROGRAM program,

LOADPT=0 or LOADPT= 'adresse<16MB'

LOADPT=*XS or LOADPT= 'address>16MB'

XS program is to be loaded above 16 MB

XS program is to be loaded below 16 MB

NXS program is to be loaded below 16 MB

NXS program is to be loaded above 16 MB (LOADPT=0 is set, because RMODE ≠ ANY)

Loader ELDE: {LOAD-PROGRAM/START-PROGRAM} programname

XS program executes above 16 MB, machine address mode 31 (1)

XS program executes below 16 MB, machine address mode 24 (2)

NXS program can only execute below 16 MB, machine address mode 24 (3)

Changing of the machine address mode 24 using RUNOPT START=XS possible (4)

Changing of machine address mode 24 by means of RUNOPT START=XS **not** possible (no error message, uncontrolled response).

Fig. A.7-1: Load address and machine address mode for linking with TSOSLNK and loading with ELDE

Re (1),(4): Dynamic storage space is set up above 16 Mbytes with CALL ALLOC(...,'ANY'). If this is not possible, an error message is issued. With CALL ALLOC(...,'NXS'), dynamic storage space is set up below 16 Mbytes.

Re (2),(3): Dynamic storage space is set up below 16 Mbytes with CALL ALLOC(...).

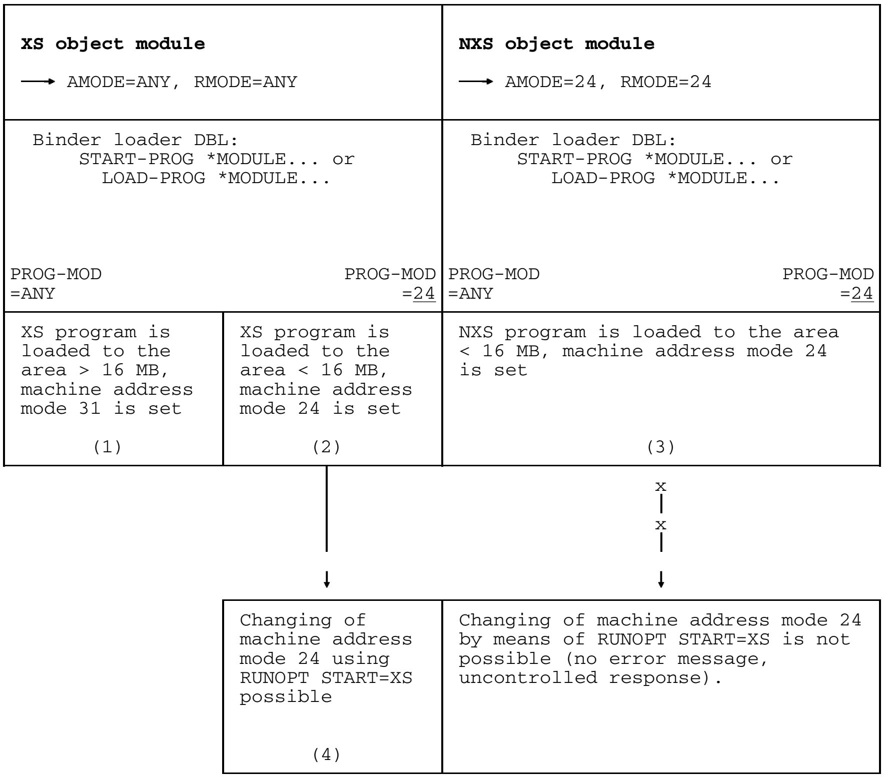

XS object module
→ AMODE=ANY, RMODE=ANY
NXS object module
→ AMODE=24, RMODE=24
Binder loader DBL:
START-PROG *MODULE... or
LOAD-PROG *MODULE...
Binder loader DBL:
START-PROG *MODULE... or
LOAD-PROG *MODULE...
PROG-MOD =ANY
PROG-MOD =24
PROG-MOD =ANY
PROG-MOD =24
XS program is loaded to the area > 16 MB, machine address mode 31 is set
(1)
XS program is loaded to the area < 16 MB, machine address mode 24 is set
(2)
NXS program is loaded to the area < 16 MB, machine address mode 24 is set
(3)
Changing of machine address mode 24 using RUNOPT START=XS possible
(4)
Changing of machine address mode 24 by means of RUNOPT START=XS is not possible (no error message, uncontrolled response).

Fig. A.7-2: Load address and machine address mode for linking and loading with DBL

Re (1),(4): Dynamic storage space is set up above 16 Mbytes with CALL ALLOC(...,’ANY’). If this is not possible, an error message is issued. With CALL ALLOC(...,’NXS’), dynamic storage space is set up below 16 Mbytes.

Re (2),(3): Dynamic storage space is set up below 16 Mbytes with CALL ALLOC(...).

A.7.2 XS, OLD and glue programs

OLD programs, i.e. programs compiled with a predecessor version of FOR1 V2.0A, can be generated as XS programs by recompiling them with a version as of FOR1 V2.0A. When recompiling with FOR1 version 2.0A or 2.1A, EXTENDED-SYSTEM=YES must be specified for the compilation. As of version 2.2A, FOR1 always generates XS modules without them having to be requested by means of an option.

In many cases, however, program interfacing between XS programs and OLD programs will still be necessary, e.g. if the source programs are no longer available. When interfacing OLD and XS programs, the user must ensure that the address mode set is correct and that the parameters are transferred using the expected format. This is accomplished by means of "glue" programs which the user must create himself. A glue program is understood to mean a program which must be inserted between two programs, between which no direct subprogram call is possible. When interfacing XS programs and OLD programs, the glue program handles the switching of the address mode and conversion of the parameter formats. The exact format of the glue programs required for execution in the same address space is described in section A.7.3, and for execution in different address spaces in section A.7.4.

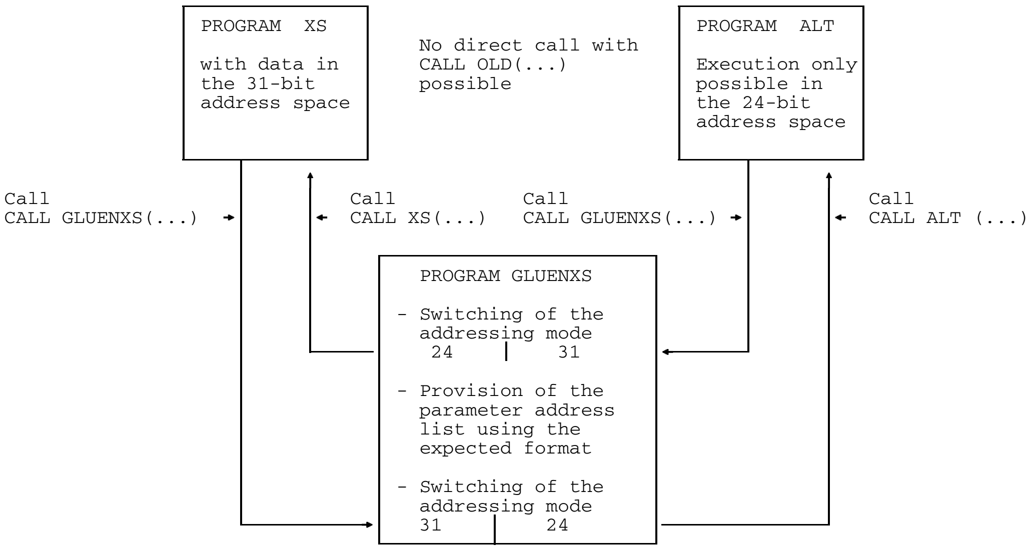

PROGRAM XS
with data in the 31-bit address space
No direct call with CALL OLD(...) possible
PROGRAM ALT
Execution only possible in the 24-bit address space
Call CALL GLUENXS(...)
Call CALL XS(...)
Call CALL GLUENXS(...)
Call CALL ALT (...)
PROGRAM GLUENXS
- Switching of the addressing mode 24 | 31
- Provision of the parameter address list using the expected format
- Switching of the addressing mode 31 | 24

Fig. A.7-3: Use of an NXS glue program for interfacing XS and OLD programs

A.7.3 Program interfacing for execution in same address space

Program interfacing in the 31-bit space

Program interfacing is possible only as follows:
XS program calls XS program.

Program interfacing in the 24-bit space without intermediate connection of a glue program

The following diagram shows permissible direct program interfacing in in the 24-bit space:

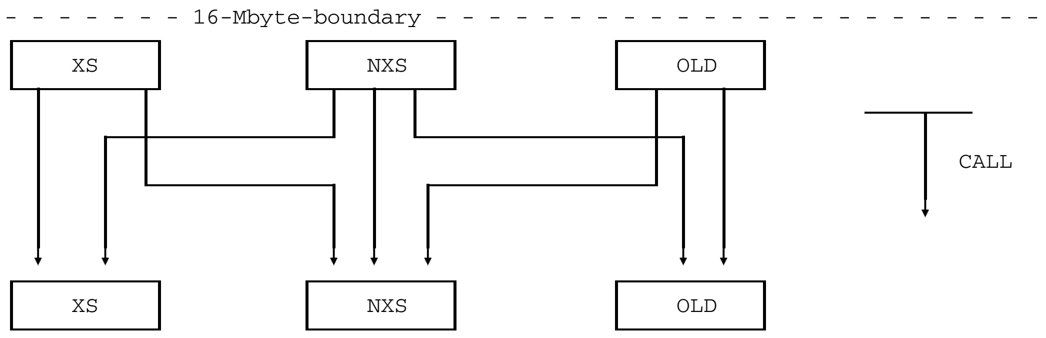


Fig. A.7-4: Permissible direct program interfacing in the 24-bit space

Since all programs execute in the 24-bit space, i.e. in the range below 16 MBytes, and are processed using the 24-bit addressing mode of the hardware, any program interfacing is permissible provided that the parameter address lists are compatible.

NXS subprograms also process OLD parameter address lists in addition to XS and NXS lists. XS subprograms only process XS and NXS parameter address lists. OLD subprograms only process OLD and NXS parameter address lists.

Program interfacing in the 24-bit space with intermediate connection of a glue program

An NXS program must always be connected between XS and OLD programs to interface them.

XS program called by an OLD program

An OLD program calls an XS program to be executed in 24-bit space. The user must write an NXS program using the name of the subprogram called in the OLD program. This NXS glue program is merely necessary for passing the parameters supplied by the OLD program in a further call to the XS program.

```
PROGRAM OLD          SUBROUTINE SUB(A,B,C)      SUBROUTINE SUBXS(A,B,C)
        .
        .             The NXS glue program      SUBXS is the original
        .             is given the name SUB     subprogram SUB re-
CALL SUB(A,B,C)       and passes the para-      compiled as an XS
        .             meters to SUBXS           program
        .
        .             CALL SUBXS(A,B,C)                   .
                                                          .
                                                          .

                     RETURN                     RETURN
END                  END                        END
```

The XS program in this configuration is always in the space below 16 MBytes and always runs in the 24-bit address mode (cf. Figs. A.7-1 and A.7-2), provided that the user has not incorrectly started the address mode by specifying RUNOPT START=XS in address mode 31.

OLD program called by an XS program

An XS program executing in the 24-bit space calls an OLD program. The user must write an NXS program whose only purpose is to forward to the OLD program the parameters passed from the XS program.

```
PROGRAM XS24         SUBROUTINE GLUENXS(A,B,C)  SUBROUTINE OLD(A,B,C)
        .
        .
        .                     .                           .
                              .                           .
  CALL OLD (A,B,C)            .                           .
  is replaced by

CALL GLUENXS(A,B,C)  CALL OLD (A,B,C)
        .
        .
        .            RETURN                     RETURN
END                  END                        END
```

A.7.4 Program interfacing for execution in different address spaces

If programs in the 31-bit address space are to be interfaced with programs in the 24-bit address space, the user must always produce an NXS glue program.

Subprograms for controlling machine address mode

For controlling machine address mode the ready-made subprograms GETMODE, NXSTOXS and XSTONXS are provided for use in NXS glue programs.

```
CALL GETMODE (mode)
```

The subprogram GETMODE (mode) returns the current machine address mode set in the INTEGER*4 variable or the INTEGER*4 array element at call time.

Possible output values for mode: 24 or 31

GETMODE returns the value 24 on NXS systems.

CALL XSTONXS (progname,$\mathtt{par}_1$,...,$\mathtt{par}_n$)

The subprogram XSTONXS converts the parameter address list of the parameters par_1,...,par_n from XS to NXS format and calls the NXS program progname.

progname	Name of an NXS subprogram declared as EXTERNAL in the calling XS program.
par_i	i-th parameter to be supplied to the NXS subprogram; $0 \le i \le n$, $0 \le n \le 408$

Subprogram XSTONXS copies the parameter address list supplied to it, removes the EXTERNAL name and converts the XS parameter address list to a corresponding NXS parameter address list. The machine address mode is changed from 31 to 24. Using the address of the parameter address list in register 1 and the parameter number in register 0, a branch is made to the NXS program.

After returning from the NXS program, the original machine address mode (31) is restored and control is again returned to the program which called the XSTONXS subprogram.

The parameters $par_1 ... par_n$ must be located in the 24-bit address space. If they are in the 31-bit address space an error message will be issued. The user must convert the parameters in the 31-bit address space to corresponding parameters in the 24-bit address space by copying them before calling XSTONXS. If required, the parameters must be copied back to the parameters in the 31-bit address space after returning from XSTONXS (see Fig. A.7-5 and the relevant example).

```
CALL NXSTOXS (progname,par1,...,parn)
```

Subprogram NXSTOXS converts the parameter address list of parameters $pa_r1,...,pa_rn$ from NXS to XS format and invokes the XS subprogram progname.

progname	Name of an XS subprogram declared as EXTERNAL in the calling NXS program
par_i	*i*-th parameter to be supplied to the XS subprogram; $0 \le i \le n$, $0 \le n \le 408$

Subprogram NXSTOXS copies the parameter address list supplied to it, removes the EXTERNAL name and converts the XS parameter address list to a corresponding NXS parameter address list. The machine address mode is changed from 24 to 31. With the address of the parameter address list in register 1 and the number of parameters in register 0, a branch is made to the XS program.

After returning from the XS program, the original machine address mode (24) is restored and control is returned again to the program which called the NXSTOXS subprogram (see Fig. A.7-6 and relevant example).

NXSTOXS must be called in machine address mode 24. If NXSTOXS is invoked in machine address mode 31, a library program error will occur.

Case 1: XS program with dynamically created data above 16 Mbytes calls the OLD subprogram

An XS program with dynamically created data above 16 Mbytes calls an OLD subprogram which contains read and write accesses to this data.

With the exception of the dynamically created data, the entire load module is below 16 Mbytes, since the XS program is used to link an OLD program.

Prerequisite:

RUNOPT START=XS

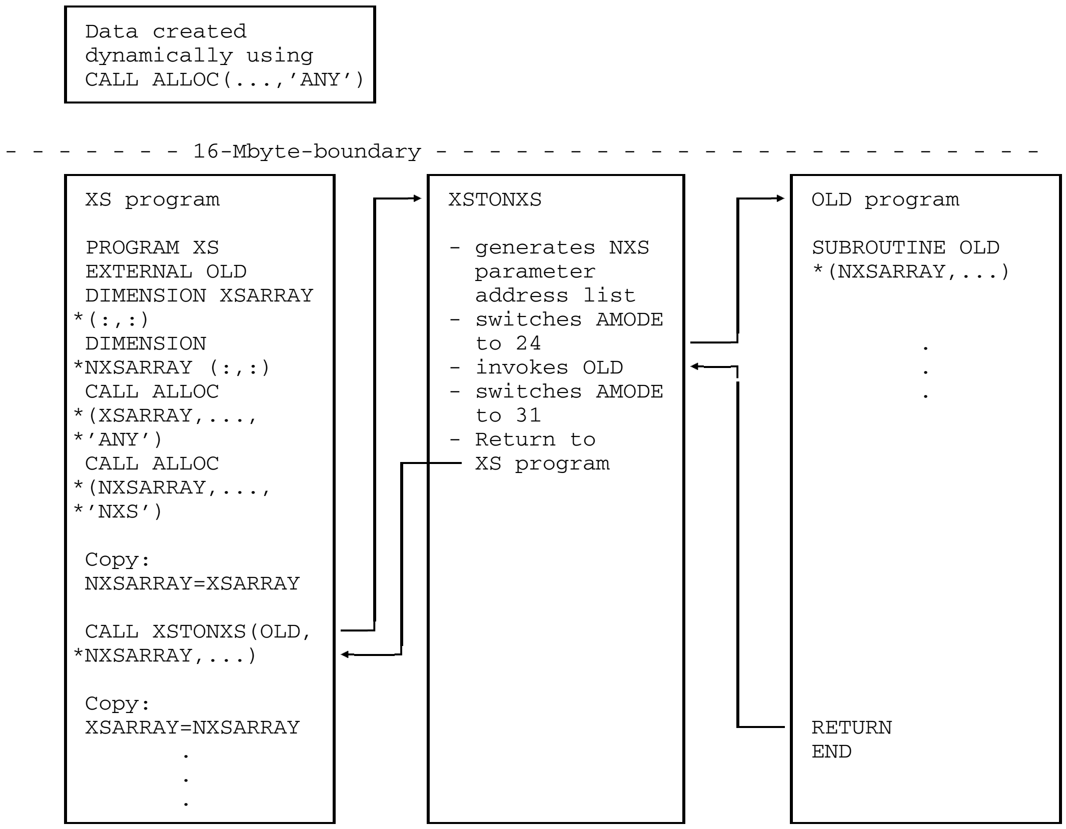


Fig. A.7-5: XS program with data above 16 Mbytes calls OLD subprogram

Example for case 1: *XS program with dynamically created data above 16 Mbytes calls OLD subprogram*

```
      PROGRAM XS
      EXTERNAL OLD

      INTEGER L1,L2,U1,U2

      DIMENSION XSARRAY (:,:)        (1)
      DIMENSION NXSARRAY (:,:)

      READ (*,*) L1,U1,L2,U2         (2)


      CALL ALLOC (XSARRAY,L1,U1,L2,U2,
     *             'ANY')            (3)
      CALL ALLOC (NXSARRAY,L1,U1,L2,U2
     *             ,'NXS')
                 .
                 .
                 .
       ┌──────────────────────┐
       │ CALL OLD (XSARRAY,   │
       │ L1,U1,L2,U2)         │
       │ is replaced by       │
       └──────────────────────┘

      DO   10   J=L2,U2      ┐
      DO   10   I=L1,U1      │      (4)
 10   NXSARRAY(I,J)=         │
     *XSARRAY(I,J)           ┘

      CALL XSTONXS (OLD, NXSARRAY, (5)
     *L1,U1,L2,U2)

      DO   20   J=L2,U2      ┐
      DO   20   I=L1,U1      │      (7)
 20   XSARRAY(I,J)=          │
     *NXSARRAY(I,J)          ┘

      CALL DEALLOC (NXSARRAY)        (8)
                .
                .
                .
      END
```

```
      SUBROUTINE OLD (NXSARRAY,L1,U1,
                     L2,U2)
      INTEGER L1,L2,U1,U2

      DIMENSION NXSARRAY (L1:U1,L2:U2)
                     .
                     .
                     .
      READ(5,*)((NXSARRAY(I,J),
     *       I=L1,U1), J=L2,U2)      (6)

                     .
                     .
                     .

                          RETURN
                          END
```

(1) The arrays XSARRAY and NXSARRAY are created as two-dimensional arrays with variable subscript bounds.

(2) The subscript bounds are not entered until runtime.

(3) Using CALL ALLOC, XSARRAY is created as a dynamic array with the subscript bounds (L1:U1,L2:U2). By specifying ANY, the array is stored above 16 Mbytes, if the current machine address mode is 31, and stored below 16 Mbytes if the current machine address mode is 24. The current machine address mode is defined

by specifying values at compile, linking and loading time and with the aid of the runtime option RUNOPT START (see Figs. A.7-1 and A.7-2). To ensure that the dynamic array XSARRAY is stored above 16 Mbytes, the runtime option RUNOPT START=XS must be specified.

(4) The array XSARRAY above 16 Mbytes is copied to NXSARRAY which is below 16 Mbytes. Copying causes the 31-bit addresses to be converted to 24-bit addresses.

(5) Runtime routine XSTONXS is called. XSTONXS has the name of the OLD program to be called as its first parameter (declared in the EXTERNAL statement in the XS program).
XSTONXS switches the machine address mode from 31 to 24 and calls the OLD program. After returning from the OLD program, the machine address mode is reset to 31 and a branch is made to the XS program.

(6) In the OLD subprogram, there is a read access to the dynamically created array NXSARRAY.

(7) After returning from XSTONXS, 31-bit addresses are generated again by copying.

(8) Using CALL DEALLOC, dynamically created memory is released again.

Case 2: OLD program calls XS subprogram with data created dynamically above 16 Mbytes

An OLD program calls an XS subprogram which has dynamically created data above 16 Mbytes. With the exception of the dynamically created data, the entire load module is below 16 Mbytes, since the XS program is used to link an OLD program. This constellation probably occurs quite rarely, e.g. whenever the source of the main program is no longer available, whereas the programs in a subprogram library can be recompiled as XS programs.

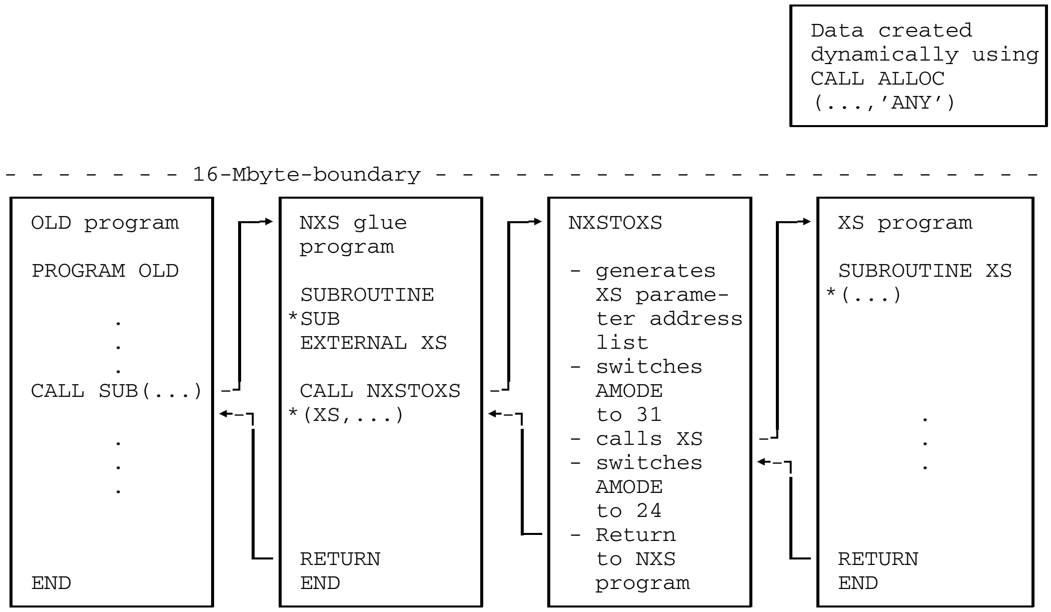
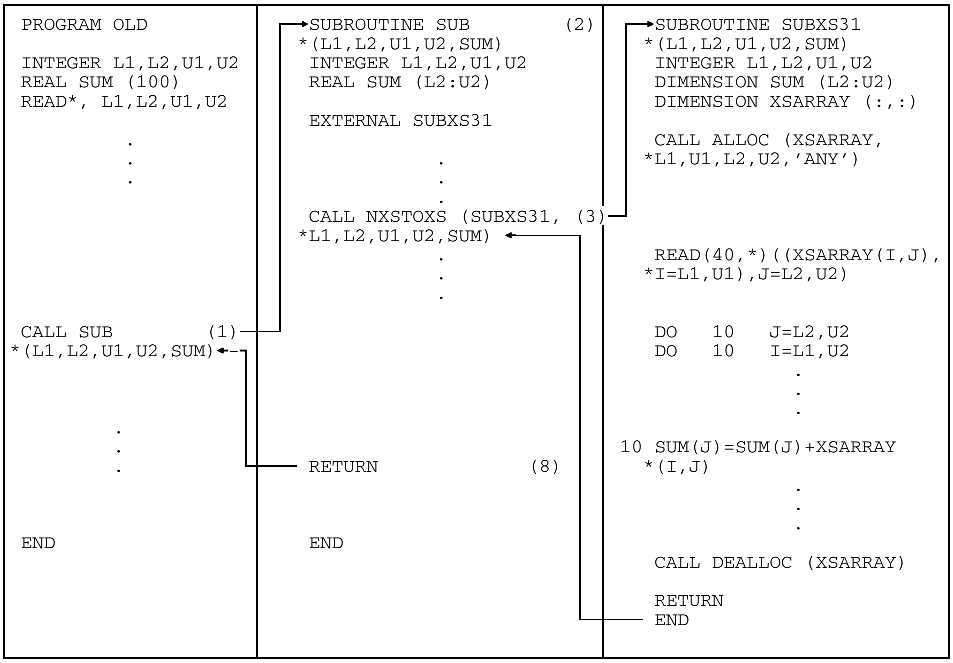


Fig. A.7-6: OLD program calls XS program with dynamically created data above 16 Mbytes

Example for case 2: *OLD program calls XS program with dynamically created data above 16 Mbytes*



```
 PROGRAM OLD

 INTEGER L1,L2,U1,U2
 REAL SUM (100)
 READ*, L1,L2,U1,U2

          .
          .
          .

 CALL SUB         (1)
*(L1,L2,U1,U2,SUM)

         .
         .
         .

 END
```

```
 SUBROUTINE SUB            (2)
*(L1,L2,U1,U2,SUM)
 INTEGER L1,L2,U1,U2
 REAL SUM (L2:U2)

 EXTERNAL SUBXS31

            .
            .
            .
 CALL NXSTOXS (SUBXS31, (3)
*L1,L2,U1,U2,SUM)
            .
            .
            .

 RETURN                 (8)

 END
```

```
 SUBROUTINE SUBXS31                (4)
*(L1,L2,U1,U2,SUM)
 INTEGER L1,L2,U1,U2
 DIMENSION SUM (L2:U2)
 DIMENSION XSARRAY (:,:)           (5)

 CALL ALLOC (XSARRAY,              (6)
*L1,U1,L2,U2,'ANY')


 READ(40,*)((XSARRAY(I,J),
*I=L1,U1),J=L2,U2)


 DO   10   J=L2,U2
 DO   10   I=L1,U2
              .
              .
              .
10 SUM(J)=SUM(J)+XSARRAY
 *(I,J)
              .
              .
              .
 CALL DEALLOC (XSARRAY)            (7)

 RETURN
 END
```

(1) The OLD program transfers its parameters to an NXS glue program which the user must generate and which contains the name of the original subprogram.

(2) In this glue program the name of the XS program to be called is declared as EXTERNAL.

(3) Runtime routine NXSTOXS is called in the glue program. In the call of the NXSTOXS routine the name of the XS subprogram to be called is specified as the first parameter; further entries specify the parameters of the XS program.

(4) NXSTOXS calls the XS program with the corresponding XS parameter address list.

(5) The original subprogram has been modified and compiled as an XS program. In the XS subprogram the array XSARRAY is declared as an array with variable subscript bounds.

(6) Using CALL ALLOC, XSARRAY is created as a dynamic array with the subscript bounds (L1:U2,L2:U2), which are supplied to the XS subprogram as the parameters. By specifying ANY the array is created above 16 Mbytes if the current machine address mode is 31, and below 16 Mbytes if the current machine address mode is 24. The current machine address mode is defined by specifying values at compile, linking and loading time and specifying the runtime option RUNOPT START (see Figs. A.7-1 and A.7-2).

(7) Using CALL DEALLOC, the dynamically created memory is released again.

(8) After returning from the NXSTOXS routine to the glue program, the control returns to the OLD program which made the original call.

A.8 Language interfacing in non-ILCS environments

A.8.1 Routines for language interfacing in non-ILCS environments

Calls for runtime system initialization routines (such as INITFOR1), runtime system termination routines (such as IF@PROT) and STXIT activation routines (such as IF@STXT) are required only for language interfacing in non-ILCS environments.

In ILCS environments, all measures required for language interfacing are taken automatically, so calls for these routines are not necessary. Programs containing calls for these routines can nevertheless also execute in ILCS environments since the calls (at least insofar as they affect the FOR1 runtime system) are ignored in ILCS environments.

Program termination routine IF@PROT

The program termination routine is required in non-ILCS environments for language interfacing (FOR1 with Assembler, PLI1) and for data base systems.

The program termination routine is invoked (in non-ILCS environments):
- implicitly at termination of the FOR1 main program;
- explicitly by the STOP or CALL EXIT statement in the source program.

In ILCS environments the IF@PROT call is ignored.

Sequence of functions

1) Outputs the dynamic program structure if the debug statement %COUNT (see section 7.4.7) is specified.
2) Calls the submitted termination procedures in the order of descending weight (15, 14, ..., 0) if these do not terminate normally.
3) Closes all opened FOR1 files in order of ascending file numbers (0, 1, ..., 99).
4) Resets (REMOVE-FILE-LINK) all SET-FILE-LINK commands implicitly initiated by the source program (if a new file was created). Sequence based on ascending file numbers (0, 1, ..., 99).
5) Releases (RELM) memory requested by REQM.
6) End message with output of the CPU time used and elapsed time.

7) On successful termination:

```
TERM UNIT=PRGR,MODE=NORMAL
```

On abnormal termination:

```
TERMJ UNIT=STEP,MODE=ABNORMAL,DUMP=N
```

If the module is activated through the entry IF@PTERM or I$PTERM (from non-FORTRAN programs only) functions 2 through 5 will be performed. Subsequently, control is returned to the calling procedure.

If the calling program is to execute with AMODE=31, the preconnection routine IF@XPTR must be called instead of IF@PTERM or I$PTERM.

IF@VAP routine for requesting termination procedures

For requesting PLI1 termination procedures in non-ILCS environments, a runtime routine with the name IF@VAP or I$VAP (same meaning) is provided. In ILCS environments, the call is ignored.

PLI1 termination procedures cannot be submitted directly from a FORTRAN program by invoking IF@VAP. IF@VAP can only be called by assembly language or PLI1 programs which the user must write himself. Requested PLI1 termination procedures are invoked by the FOR1 program termination routine IF@PROT. PLI1 termination procedures do not interface with FOR1 SUBROUTINEs or FUNCTIONs.

Parameters for calling the assembly language or PLI1 program

Three parameters containing information on the termination procedure to be called must be supplied to the IF@VAP routine:

param1	INTEGER*4 variable or INTEGER constant. Weighting of the termination procedures. The values 0 through 15 are permissible. The termination procedures are called in order of descending weight (15,14,...,0) by program termination routine IF@PROT.
param2	param2 can be: – name of the termination procedure. – zero pointer X'FFFEFFFF' (equivalent to F'-65537'). If the zero pointer is specified, the termination procedure is cancelled using the weight specified by param1.

param3	Protection code with the following possible values:
	0 The address of the termination procedure with the weight specified by param1 can be modified or deleted;
	1 The address of the termination procedure with the weight specified by param1 can neither be modified nor cancelled.

The FOR1 runtime routine IF@VAP (I$VAP) requires that the supplied parameters be contained in certain registers:

– the value of param1 (weight) must be loaded in register 1;
– the value of param2 (address of the termination procedure) must be loaded in register 2;
– the value of param3 (protection code) must be loaded in register 3.

These registers must be supplied in the assembly language or PLI1 program.

Return messages of the routine IF@VAP(I$VAP)

Possible return messages in register 1:

0	Submission of the termination procedure has been accepted.
1	An impermissible weighting was specified for param1.
2	An impermissible protection code was transferred for param3.
3	A protected address was to be changed.

Possible return messages in register 2:

Undefined	If a value not equal to 0 was returned in register 1.
Zero pointer X’FFFEFFFF’	If no termination procedure has yet been submitted for the param1 weighting (X’FFFEFFFF’ corresponds to F’-65537’).
Old address	Old address previously submitted under the param1 weighting.

The following diagram outlines requesting of termination procedures in non-ILCS environments by means of IF@VAP.

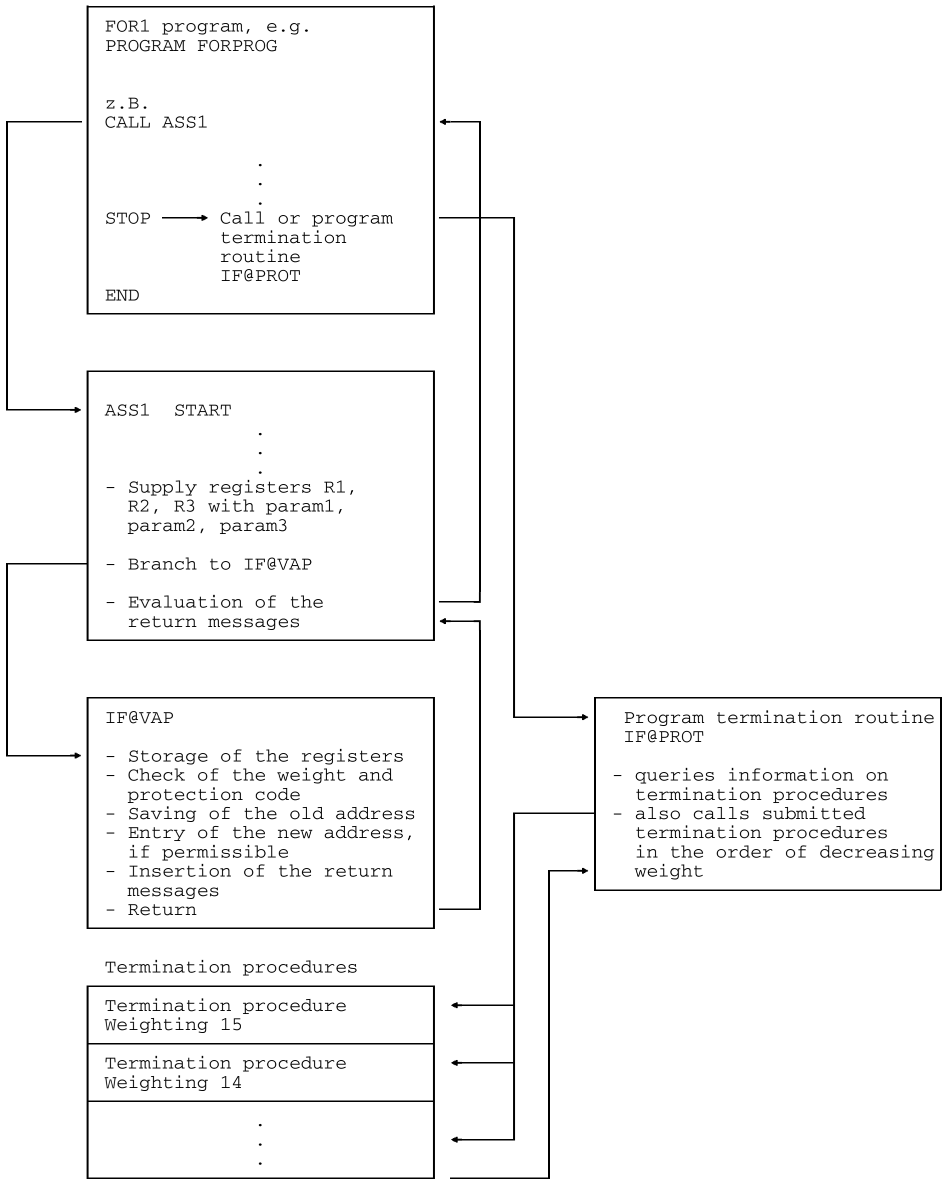

FOR1 program, e.g.
PROGRAM FORPROG
z.B.
CALL ASS1
STOP
Call or program termination routine IF@PROT
END
ASS1 START
- Supply registers R1, R2, R3 with param1, param2, param3
- Branch to IF@VAP
- Evaluation of the return messages
IF@VAP
- Storage of the registers
- Check of the weight and protection code
- Saving of the old address
- Entry of the new address, if permissible
- Insertion of the return messages
- Return
Program termination routine IF@PROT
- queries information on termination procedures
- also calls submitted termination procedures in the order of decreasing weight
Termination procedures
Termination procedure Weighting 15
Termination procedure Weighting 14

Fig. A.8-1: Requesting of termination procedures by IF@VAP

FOR1 STXIT routine

In non-ILCS environments the FOR1 program mask and the FOR1 STXIST routine are normally defined at the start of the FOR1 object module. As of FOR1 version V1.5, STXIT (contingency) is effective. The request for a FOR1 STXIT routine can be suppressed by specifying the runtime option RUNOPT STXIT=NO.

If, in a non-FORTRAN subprogram which has been called, user-own STXIT routines are requested or another program mask is set, the FOR1 STXIT routine may be deactivated as a result.

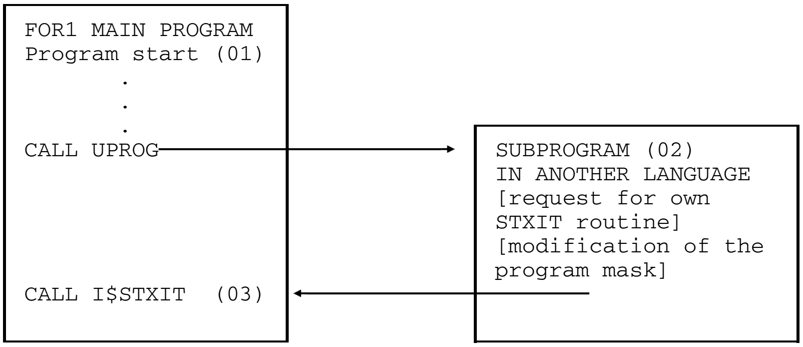
If, in the opposite case, a FORTRAN subprogram is called by a program in a different language, it may be necessary to activate the FOR1 STXIT routine in the FORTRAN subprogram.

In ILCS environments it is not necessary to call the FOR1 STXIT routine and such calls are ignored. If a program containing calls for the FOR1 STXIT routine is compiled with LINKAGE=STD and TESTOPT=(ARG), the compiler issues the following warning:

```
SA 249  ILCS-DEVIATION:  USELESS FUNCTION CALL
```

Specifying the runtime option RUNOPT=NO also has no effect in ILCS environments.

Example: Calling the FOR1 STXIT routine (non-ILCS environment)



Explanation of example:

(1) The entries IF@STXIT and I$STXIT of the STXIT routine are contained in the module IF@INIT, which is linked to each FOR1 object module as a standard procedure. At program start, the STXIT routine is activated by the module IF@INIT. This results in invocation of the FOR1 error handling routines.

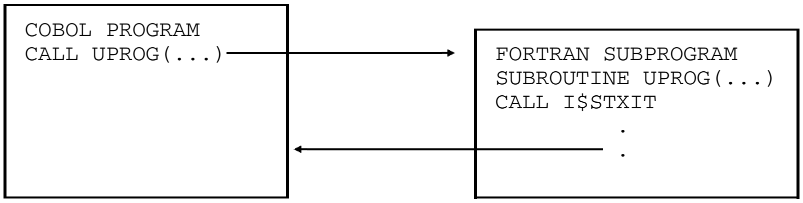
Actions of the IF@STXIT or I$STXIT entry:

- registers are stored in memory
- the program mask is set to X’0C’ if COMOPT NOEXPUNDERFLOW was used for compilation (default), or
- setting of program mask to X’0E’ if COMOPT EXPUNDERFLOW was used for compilation.
- execution of the STXIT macro with cancellation of all exits which are not interpreted by FOR1
- registers are reloaded
- return

(2) In the non-FORTRAN subprogram, a user-own STXIT routine can be invoked and the program mask modified.

(3) In this case the FOR1 STXIT routine must be reactivated by means of CALL I$STXIT in the FORTRAN main program. No parameters are required for this call.

COBOL, for example, sets the program mask to hexadecimal ’00’; an STXIT routine is not requested:



Calling the FOR1 STXIT routine is not necessary if the FOR1 STXIT routine in the calling program has been implicitly invoked by `CALL "INITFOR1"`.

A.8.2 FOR1/COBOL interfacing in non-ILCS environments

FOR1 program calls COBOL program

No special precuations are required for calling COBOL subprograms from FOR1 programs.

Call: `CALL subprog (par`$_1$`,... ,par`$_n$`)`

COBOL program calls FOR1 subprogram

In non-ILCS environments the initialization routine INITFOR1 must be called before calling the FOR1 subprogram in order to create a uniform FOR1 environment:

```
CALL "INITFOR1"
CALL "subprog" USING par1,... ,parn
```

INITFOR1 may be called only once in a program system. In ILCS environments, a call for INITFOR1 is ignored.

A.8.3 FOR1/PLI1 interfacing in non-ILCS environments

FOR1 program calls PLI1 subprogram

In non-ILCS environments the OPTIONS (FORTRAN) attribute must be specified in the PROCEDURE or ENTRY statement of the PLI1 program.

– Statement in FOR1:

```
CALL name (par1,...,parn)
```

– Statement in PLI1:

```
      ⎧PROCEDURE⎫
name: ⎨         ⎬ (par1,...,parn) OPTIONS (FORTRAN);
      ⎩ENTRY    ⎭
```

PLI1 program calls FOR1 subprogram

In non-ILCS environments a language attribute must be declared in the DEL statement in the PLI1 program :

– Statements in PLI1:

```
DCL forspro ENTRY OPTIONS (FORTRAN[INTER]);
CALL forspro (par1,...parn);
```

`forspro`	Name of FOR1 subprogram
`FORTRAN`	A FOR1 subprogram is called; STXIT is not activated in FOR1.
`FORTRAN INTER`	A FOR1 subprogram is called; STXIT is activated. Program interrupts occuring while the FOR1 subprogram is executing are processed by the FOR1 error handling routine. The PLI1 error handling facility is deactivated. In the event of abortion due to error in the FOR1 subprogram, however, the PLI1 end handling is performed in any case. Upon return to the PLI1 program, the PLI1 interrupt handling facility (STXIT) is reactivated.

– Statements in FOR1:

```
⎧SUBROUTINE forspro (par1,...,parn)⎫
⎩FUNCTION forspro (par1,...,parn)  ⎭
```

A.8.4 FOR1/C interfacing in non-ILCS environments

C program calls FOR1 program

In non-ILCS environments the FOR1 subprogram must be declared in the C source program. If the FOR1 subprogram requires the FOR1 runtime environment, this must be initialized before the subprogram is called. These steps are described in the following two sections.

The called FOR1 subprogram must not be compiled with TESTOPT=(ARG).

Declaration of FOR1 subprogram in C source program

Before a FOR1 subprogram can be called in a non-ILCS environment, it must be given the language attribute for1. This is done using the preprocessor statement #pragma, e.g.

```
#pragma for1
void forsub();   /* SUBROUTINE subprogram */

#pragma for1
<typ> forsub();  /* FUNCTION subprogram with data type <type> */
```

forsub is the entry name specified in a SUBROUTINE, FUNCTION or ENTRY statement.

The language attribute can also be specified blockwise for a number of subprograms, e.g.

```
#pragma for1 {
        forsub1();
        char forsub2();

        forsub3();
#pragma      }  [for1]
```

The #pragma statement and language attribute for1 must be in a separate line (starting at column 1).
In the case of block specifications, the language attribute for1 can be optionally specified after the closing bracket (}) in the second #pragma statement, e.g. for purposes of documentation.

In ILCS environments the C program may not contain any #pragma statements for FOR1 subprograms.

Initialization of FOR1 runtime environment

If in non-ILCS environments the called FOR1 subprogram requires the FOR1 runtime environment, this must be initialized in the C source program using the external FOR1 routine initfor1. This routine is provided in the FOR1 runtime library.

A FOR1 subprogram requires the FOR1 runtime system if the subprogram

- provides I/O operations, including PAUSE, STOP,
- calls intrinsic functions,
- executes exponentiations, complex arithmetic and comparison operations, floating point divisions and comparisons with four-fold precision,
- provides character chaining or handles character strings of varying length,
- uses debugging aids (debug options, debug statements, debugging subprograms) or
- terminates the program.

initfor1 must be declared with the language attribute for1 using a #pragma statement.

initfor1 must be called before the first call for a FOR1 subprogram. initfor1 may only be called once per program system. Even with multi-level language interfacing (C → FOR1 → C → FOR1 → etc.) this single call suffices.

In ILCS environments, a call for initfor1 is ignored.

Example:

```
#pragma for1
initfor1();
#pragma for1
forsub1();
#pragma for1
float forsub2();
   .
   .
main()
{
   float x;

   initfor1();
   forsub1();
   x = forsub2();
}
```

FOR1 program calls C program

In non-ILCS environments special measures are required for calling a C program from a FOR1 program, which are described in the following two sections.

Definition of external C function in C source program

In non-ILCS environments C functions called by FOR1 programs must be provided with the environment attribute for1 when defined. This is done with the preprocessor statement #pragma:

```
#pragma for1
<type> functname (dummy argument list)
<type> param-1;
<type> param-2;
...
<type> param-n;
{
 .
 .
}
```

The #pragma statement and the environment attribute for1 must be in a separate line (starting in column 1) and directly preceding the definition of the function.

In ILCS environments the called C function may not contain any #pragma statement defining the environment attribute.

Initialization of C runtime environment

In non-ILCS environments the FOR1 program must initialize the C runtime environment before calling the first C function. This is done by calling the C library functions cinit1 or cinit2.
As the output parameter, cinit1 provides the address of the C runtime stack. In addition, the size of the first segment of the C runtime stack can be determined using cinit2.
cinit1 or cinit2 may only be called once per program system:

```
INTEGER*4 ADR
   .
   .
CALL CINIT1(ADR)
```

Even with multi-level language interfacing (FOR1 → C → FOR1 → C → etc.) this single call suffices.

In ILCS environments, the FOR1 program may not contain any calls for cinit1 or cinit2.

A.9 Interfacing FOR1 and assembly language programs

FOR1 provides macros in the FOR1MACLIB which permit interfacing of FOR1 and assembly language programs.

Using these macros, assembly language programs can be generated which behave like non-ILCS objects generated with FOR1 in the event of language interfacing. These macros do not, however, permit the generation of ILCS assembly language programs. Therefore, although assembly language programs which use these macros can be called by FOR1 programs in ILCS environments or can themselves call FOR1 programs, it is *not* however possible to use these macros to generate assembly language main programs which are executable in ILCS environments.

However, for ILCS FOR1/assembly language interfacing, ASSEMBH (as of version 1.1A) provides suitable macros (see "ASSEMBH" Reference Manual" [10]).

Interfacing macros provided by FOR1

Assembly language program calls FOR1 subprogram		FOR1 program calls assembly language subprogram	
IFEPL	generates the parameter address list	IFAEN	generates the entry routine for the assembly language program
IFECL	calls the FOR1 program	IFART	generates the reentry routine for branching to the FOR1 program
IFESDS	generates descriptor for CHARACTER data item		
IFEADS	generates descriptor for an array		
IFEEDS	generates descriptor for an array element		
IFESAV	generates a save area		

Table A.9-1: Macros for interfacing FOR1/assembly language

Interfacing of assembly language subprograms with FOR1 ILCS programs can be achieved without problems if the parameter values FIRST=1 and LAST=12 are selected in the IFART (or IFARTO) macro.

A.9.1 Assembly language program calls FOR1 subprogram

IFEPL macro (parameter address list macro)

This macro generates the parameter address list to be supplied to the subprogram or generates the required address table when FORTRAN language element "RETURN i" is used.

Call:

```
label IFEPL operandlist

               ⎧OLD                 ⎫
      [,PARMOD=⎨                    ⎬]
               ⎩NEW[(PL=XS | NXS)]  ⎭
```

label — Symbolic address of the parameter address list.

operandlist

```
⎧symbadr   [,symbadr]...⎫
⎨                       ⎬
⎩(param)   [,(param)]...⎭
```

symbadr — Symbolic return address

```
        ⎧E[,spadr]...                      ⎫
        ⎪⎧CH ⎫                             ⎪
        ⎪⎨   ⎬     [,chadr,sdsadr]...      ⎪
param   ⎨⎩CHV⎭                             ⎬
        ⎪type [,adr]...                    ⎪
        ⎪(type1) [,arradr,adsadr]...       ⎪
        ⎩<type1> [,eladr,edsadr]...        ⎭
```

E — Indicates that the parameters to follow denote subprogram addresses.

CH — Indicates that the parameters to follow denote fixed-length CHARACTER items.

CHV — Indicates that the parameters to follow denote variable-length CHARACTER items.

type — {L1|L4|I1|I2|I4|I8|R4|R8|R16|C8|C16|C32|H}

Indicates the type of parameters to follow.

L	LOGICAL*
I	INTEGER*
R	REAL*
C	COMPLEX*
H	Hollerith data item

type 1 — {type | CH}

spadr	Symbolic address of a subprogram.
chadr	Symbolic address of a CHARACTER item.
sdsadr	Symbolic address of the descriptor of this data item.
adr	Symbolic address of a simple variable of type other than CHARACTER.
arradr	Symbolic address of an array.
adsadr	Symbolic address of the descriptor of this array.
eladr	Symbolic address of an array element.
edsadr	Symbolic address of the descriptor of this array element.

PARMOD

=OLD	Generates the same parameter address list as for FOR1 versions < V2.0A.
=<u>NEW</u> [(PL={XS|<u>NXS</u>})]	Generates the same parameter address lists as for FOR1 versions ≥ V2.0A. If (PL=XS) is specified, a parameter address list in XS format is generated; when (PL=NXS) is specified, a parameter in NXS format is generated.

For CHARACTER items, arrays and array elements, not only the address of the first byte of the item is transferred, but also a descriptor of that item. Descriptors for items of the CHARACTER type, for arrays and for array elements may be generated by the macros IFESDS (string descriptor), IFEADS (array descriptor) and IFEEDS (array element descriptor) (see below).

For CHARACTER arrays and CHARACTER array elements, the array or array element descriptor is transferred, rather than a string descriptor.

The number of parameters is limited to 255.

For INTEGER items less than 4 bytes in length, a modified address is generated; for length 1 <address-3>, for length 2 <address-2>. This modification is handled by the macro and need not be specified by the user.

Example of a valid macro call:

```
BETA IFEPL     (R8,ALPHA,SUM),
               ((I2),ARRAY,ADESCR),
               (<L1>,BOOL3,BOOL7,DBOOL7),
               (CH,CHAR,CHDESCR)
```

IFECL macro (call macro)

This macro is the interface with the FOR1 subprogram. The generated code performs the functions of saving and restoring the register contents, initializing the FOR1 "Run-time Communication Area", setting and restoring the program mask, activating the FOR1 program and returning from it. IFECL generates a code which has XS capabilities and can be executed as of runtime system version V2.0A.

Call:

```
                                  ⎧YES⎫                  ⎧NO    ⎫
[label] IFECL PROG=name [,INIT = ⎨   ⎬]     [,PARLIST = ⎨name  ⎬]
                                  ⎩NO ⎭                  ⎩(reg) ⎭

                                   ⎧YES⎫                 ⎧NO    ⎫
                        [,RESTORE =⎨   ⎬]   [,RETURNI = ⎨name  ⎬]
                                   ⎩NO ⎭                 ⎩(reg) ⎭

                                   ⎧YES  ⎫             ⎧YES ⎫
                        [,MAINSAV =⎨name ⎬] [,FTERM = ⎨    ⎬]
                                   ⎩(reg)⎭             ⎩NO  ⎭

                                   ⎧NO  ⎫               ⎧NO  ⎫
                        [,USREQM = ⎨    ⎬]  [,USRELM = ⎨    ⎬]
                                   ⎩name⎭               ⎩name⎭
```

label	Symbolic address of the macro call.
PROG=name	Name of the FOR1 subprogram to be called. If omitted, no activation takes place (BALR 14,0). This may be advisable in certain instances, e.g. if only initialization is requested.
INIT	
=<u>YES</u>	FOR1 initialization is performed.
=NO	Initialization is not performed.
PARLIST	
=<u>NO</u>	No parameters are transferred to the FOR1 subprogram.
=name	Symbolic address of the parameter address list that may be generated by the IFEPL macro.
=(reg)	Specifies the register which contains the address of the parameter address list.

RESTORE	
=YES	The code generated by the IFECL macro destroys the contents of the registers and the program mask. When RESTORE=YES is specified, the contents are saved and, following the return, restored.
=NO	No saving and restoration.
RETURNI	
=NO	The called FOR1 subprogram does not use the "RETURN i" language element.
=name	Symbolic address of an address list or implementation of the "RETURN i" language element. This address list may be generated by the IFEPL macro.
=(reg)	Specifies the register which contains the address of an address list.
MAINSAV	
=YES	A save area is implicitly generated by the IFECL macro.
=name	Symbolic address of a save area where the FOR1 subprogram saves the registers. This save area may also be generated by the IFESAV macro.
=(reg)	Specifies the register containing the address of the save area.
FTERM	
=YES	Last call of a FOR1 subprogram. All FOR1 files are closed, storage is returned to the system and the RTCA deleted before a return is made.
=NO	Not the last call for a FOR1 subprogram.
USREQM	
=NO	No memory acquisition by user-own routines is desired, i.e. the entire address space which is not occupied is available to the runtime system.
=name	Symbolic address of a user-own routine for requesting memory.
USRELM	
=NO	No memory release by user-own routines is desired. i.e. the entire address space which is not occupied is available to the runtime system.
=name	Symbolic address of a user-own routine for releasing the requested memory. FOR1 initialization must be performed using INIT=YES.

The FOR1 Runtime Communication Area (RTCA) must be initialized when performing exponentiations, input/output operations or divisions with R*16 data items in the FOR1 subprogram or when using mathematical functions or complex arithmetic. If the generated code is executed more than once, initialization takes place only the first time. However initialization must be deactivated when using this macro in an assembly langauge program which in turn is called by a FOR1 program.

IFESDS macro (string descriptor macro)

This macro generates a descriptor for a CHARACTER type data item. The address of the descriptor must be supplied in the parameter list to the called FOR1 subprogram.

Call:

```
[label] IFESDS adr, ALLOCL = a [,CURRL = c]

                               ⎡OLD                  ⎤
                  [,PARMOD =  {                       }]
                               ⎣NEW[(PL={XS|NXS})]   ⎦
```

label	Symbolic address of the macro call. This name appears as an operand in the IFEPL macro.
adr	Symbolic address of the data item to be supplied.
ALLOCL=a	Length of the occupied storage space of the data item.
CURRL=c	Currently used length of the data item. If omitted, the allocated length is assumed.
PARMOD =OLD	Generates the same descriptor as for FOR1 versions < V2.0A.
=NEW {(PL={XS|NXS})]	Generates the same descriptor as for FOR1 versions ≥ V2.0A. When (PL=XS) is specified, a descriptor is generated in XS format; when (PL=NXS) is specified, a descriptor is generated in NXS format.

IFEADS macro (array descriptor macro)

This macro generates a descriptor for an array. The address of the descriptor must be supplied in the parameter list to the called FOR1 subprogram.

Call:

```
[label] IFEADS name,ELEN = n, BOUNDS = (l:u[,l:u][,...])
```

label	Symbolic address of the macro call. This name appears as an operand in the IFEPL macro.
name	Symbolic address of the array to be supplied.
ELEN=n	Length of an array element in bytes.
BOUNDS =(l:u[,l:u] [,...])	Subscript bounds of the individual dimensions, l lower bound, u upper bound

IFEEDS macro (array element descriptor macro)

This macro generates a descriptor for an array element. The address of the descriptor must be supplied in the parameter list to the called FOR1 subprogram.

Call:

```
[label] IFEEDS iname, aend, ELEN = n

                    [,PARMOD = {OLD|NEW}]
```

label	Symbolic address of the macro call. This name appears as an operand in the IFEPL macro.
iname	Symbolic address of the data item to be supplied.
aend	Symbolic address of the end of the array (last byte+1).
ELEN=n	Length of an array element in bytes.
PARMOD =OLD	Generates the same descriptor as for FOR1 versions < V2.0A.
=NEW	Generates the same descriptor as for FOR1 versions ≥ V2.0A.

IFESAV macro (save area macro)

This macro generates a save area to which the called FOR1 subprogram saves the registers.

Call:

```
[label] IFESAV
```

label	Symbolic address of the macro call.

If this macro is called for the first time within an assembly language program, a DSECT is additionally generated. It defines the structure of the save area.

Example: Assembly language program calls FOR1 subprogram

Assembly language program ASSFOR1 calls FOR1 subprogram AFOR.

Assembly language program ASSFOR1:

```
ASSFOR1  START
         PRINT NOGEN
ANF      BALR  3,0
         USING *,3
         B     BEG
ITAB     DC    20F'0'
IERG     DC    F'0'
K        DC    F'0'
IERG1    DS    D
PARAL    IFEPL ((I4),ITAB,IND),(I4,IERG),(I4,K)                    (1)
IND      IFEADS ITAB,ELEN=4,BOUNDS=(1:20)                          (2)
DRU      DS    0CL133
SATZL    DC    X'00854040C1'
         DS    CL128
MASKE    DC    X'4020202020202120'
         DS    0F
BEG      MVI   DRU+5,X'40'
LOESCH   MVC   DRU+6(127),DRU+5
         MVC   DRU+6(27),='BRANCH TO FORTRAN PROGRAM'
         WRLST DRU,FEHL
         MVI   DRU+4,X'40'
         MVI   DRU+5,X'40'
         EX    0,LOESCH
         IFECL PROG=AFOR,PARLIST=PARAL                             (3)
RUECK    MVI   DRU+4,X'40'                                         (5)
         MVC   DRU+6(35),='RETURNED TO ASSEMBLY LANGUAGE MAIN PROGRAM'
         WRLST DRU,FEHL
         MVI   DRU+5,X'40'
         EX    0,LOESCH
         MVC   DRU+6(41),='TRANSFER RESULT FROM FORTRAN PROGRAM:'
         L     5,IERG
         CVD   5,IERG1
         MVC   DRU+60(8),MASKE
         ED    DRU+60(8),IERG1+4
         WRLST DRU,FEHL
         MVI   DRU+5,X'40'
         EX    0,LOESCH
         MVC   IERG1,=XL8'0'
         L     7,K
         XR    4,4
         LA    5,ITAB
SCHLEIFE A     4,0(5)
         LA    5,4(5)
         BCT   7,SCHLEIFE
         CVD   4,IERG1
         MVC   DRU+60(8),MASKE
         ED    DRU+60(8),IERG1+4
         MVC   DRU+6(37),='RESULT IN ASSEMBLY LANGUAGE MAIN PROGRAM:'
         WRLST DRU,FEHL
         TERM
```

```
FEHL      TERMD
         END   ANF
```

FOR1 subprogram AFOR

```
         SUBROUTINE AFOR(ITAB1,IERG1,K1)
         DIMENSION ITAB1(20)
         WRITE(99,1)
    1    FORMAT(' ',35X,'IN FORTRAN SUBPROGRAM')
         IERG1=0
         DO 10 I=1,20
         READ(1,2,END=100)ITAB1(I)                               (4)
    2    FORMAT(I8)
   10    IERG1=IERG1+ITAB1(I)
  100    WRITE(99,3)
    3    FORMAT(' ',35X,'FIXED-POINT NUMBERS READ INTO TABLE')
         K1=I-1
         WRITE(99,7)K1
    7    FORMAT(' ',35X,'NUMBER OF RECORDS FROM READ OPERATION:',I8)
         WRITE(99,4)
    4    FORMAT(' ',35X,'CALCULATION IN FORTRAN SUBPROGRAM')
         WRITE(99,5)IERG1
    5    FORMAT(' ',35X,'RESULT IN FORTRAN SUBPROGRAM:',I8)
         WRITE(99,6)
    6    FORMAT(' ',35X,'RETURN TO ASSEMBLY LANGUAGE PROGRAM'/' ')
         RETURN
         END
```

(1) Call of parameter address list macro IFEPL:

```
PARAL      IFEPL ((I4),ITAB,IND),(I4,IERG),(I4,K)
```

Macro IFEPL generates the parameter address list to be supplied to the FOR1 subprogram. The first parameter to be supplied is data area ITAB, to which the array ITAB1 in the FOR1 subprogram corresponds. In the first operand ((I4),ITAB,IND) of the IFEPL macro, the type of array (I4), the symbolic address of the array (ITAB) and the symbolic address of the descriptor of this array (IND) are specified. Symbolic address IND must be specified as the symbolic address of array descriptor macro IFEADS.

(2) Call of array descriptor macro IFEADS:

```
IND        IFEADS ITAB,ELEN=4,BOUNDS=(1:20)
```

Macro IFEADS generates a descriptor for an array. The address of this descriptor must be supplied to the called FOR1 subprogram in the parameter address list. The name of the array (ITAB) appears as the first operand. ELEN=4 denotes length 4 of an array element in bytes, BOUNDS=(1:20) denotes the upper and lower subscript bounds of the dimension.

(3) Call of macro IFECL:

```
        IFECL PROG=AFOR,PARLIST=PARAL                          (3)
```

In the call of macro IFECL, PROG=AFOR is used to specify the name of the called FOR1 program AFOR. PARLIST=PARAL is used to specify the symbolic address PARAL of the parameter address list generated by macro IFEPL. The following default values apply for the remaining IFECL operands not mentioned in the above:

INIT=YES	FOR1 runtime system initialization
RESTORE=YES	Saving and restoring all registers and the program mask
MAINSAV=YES	A save area is generated by the IFECL macro.
RETURNI=NO	FOR1 subprogram contains no RETURN i statement.
FTERM=NO	Not the last call of a FOR1 subprogram
USREQM=NO, USRELM=NO	No memory management by user-own routines.

(4) In the FORTRAN program, 20 INTEGER*4 integers are read in and the total IERG1 is calculated.

(5) Result IERG1 calculated in the FOR1 program is supplied to the assembly language program, from where it is output. The total number of array elements formed is likewise calculated in the assembly language program and then output.

In the example, the numbers 1 through 20 are read in. Output from the program to SYSLST shows the messages of the assembly language program left-justified and the messages of the FOR1 program right-justified.

```
BRANCH TO FORTRAN PROGRAM

                               IN FORTRAN SUBPROGRAM
                               FIXED-POINT NUMBERS READ INTO TABLE
                               NUMBER OF RECORDS FROM READ OPERATION: 20
                               CALCULATION IN FORTRAN SUBPROGRAM
                               RESULT IN FORTRAN SUBPROGRAM:         210
                               RETURN TO ASSEMBLY LANGUAGE PROGRAM

RETURNED TO ASSEMBLY LANGUAGE MAIN PROGRAM
TRANSFER RESULT FROM FORTRAN PROGRAM:                                210
RESULT IN ASSEMBLY LANGUAGE MAIN PROGRAM:                            210
```

A.9.2 FOR1 program calls assembly language program

Two macros are available: IFAEN, IFART

IFAEN macro (entry macro)
IFAENO macro

This macro generates the entry routine for the assembly language program.

Macro IFAENO has the same operands as the IFAEN macro, plus the operand SB=address. IFAENO calculates the address of the last parameter, sets the end bit as required, and stores the address of the end bit in the array with the address SB=address.

Call:

```
                       ⎡ENTRY⎤
[label] IFAEN  [LABEL =⎨     ⎬]        [,FIRST = reg][,LAST = reg]
                       ⎣CSECT⎦

                         ⎡NO  ⎤            ⎡STANDARD         ⎤
               [,MAINSAV = ⎨YES ⎬]   [,PMASK = ⎨                 ⎬]
                         ⎣name⎦            ⎣YES(PMSAV=name)  ⎦
-----------------------------------------------------------------
                       ⎡ENTRY⎤
[label] IFAENO [LABEL =⎨     ⎬] , ..., [SB = address]
                       ⎣CSECT⎦
```

Defaults (underlined): ENTRY, NO, STANDARD

label	Symbolic address of the macro call. This name is used for the call by the FOR1 program. If omitted, the name is defaulted to the form IFdddd, where dddd is a 4-digit number.
LABEL	
=<u>ENTRY</u>	The entry point is an entry name.
=CSECT	The entry point is a CSECT name.
FIRST=reg	Specifies the first register to be saved (default = 14).
LAST =reg	Specifies the last register to be saved (default = 12).
MAINSAV	
=<u>NO</u>	The assembly language program called needs no save area since it provides no further subprogram calls.
=YES	A save area for the assembly language program has already been generated by the IFECL macro.

=name	Symbolic address of a save area, e.g. generated by the IFESAV macro.
PMASK	
=YES	If AMODE=31 applies, the program mask cannot be saved in register 14. When PMASK=YES (PMSAV= name) is specified, the program mask is stored in an array with a length of 1 byte using the symbolic name "name".
=STANDARD	The program mask is not saved.
SB=address	Address of a 4-byte array in which the address of the end bit of the parameter address list is stored (most significant bit in the *n*-th word of the parameter address list where n is the number of parameters).

Notes

- When MAINSAV=NO is specified (default) and a runtime error occurs in the assembly language program, the call hierarchy is output only up to the calling FOR1 program unit. The names of assembly language programs in the call hierarchy are included in the output provided that MAINSAV=YES or MAINSAV=name has been specified in the IFAEN macro call.
- If the called assembly language program modifies the program mask, this may under certain circumstances violate the ILCS conventions. In such a case the program mask of the calling program should therefore be saved by specifying PMASK=YES(PMSAV=name), and reset when a return is effected through the PMASK parameter of the IFART (or IFARTO) macro,.

IFART macro (return macro)
IFARTO macro

This macro generates the routine for returning from the assembly language program to the FOR1 program.l

Macro IFARTO has the same operands as the IFART macro, plus the operand SB=address. If required, IFARTO removes the end bit whose address is stored in the array with the address SB=address.

Call:

```
                                                       ⎡NO  ⎤
[label] IFART  [FIRST = reg] [,LAST = reg]  [,MAINSAV=⎨YES ⎬]
                                                       ⎣name⎦

                        ⎡STANDARD          ⎤
               [,PMASK = ⎨YES(PMSAV=name) ⎬] [,RETURN = i]
                        ⎣NO                ⎦

                        ⎡YES⎤                          ⎡YES ⎤
               [,STXIT = ⎨   ⎬]          [,CHARFUN = ⎨NO  ⎬]
                        ⎣NO ⎦                          ⎣name⎦
-------------------------------------------------------------------
[label] IFARTO [FIRST = reg] ,..., [SB = address]
```

label	Symbolic address of the macro call.
FIRST=reg	First register to be restored (default = 14).
LAST=reg	Last register to be restored (default = 12).
MAINSAV	
=NO	No save area was needed.
=YES	A save area was generated by the IFECL macro.
=name	Symbolic address of a save area. This operand must be the same as the corresponding operand in the IFAEN macro.
PMASK	
=STANDARD	If AMODE=24 applies, the program mask of the calling program is reset. If AMODE=31 applies, the program mask of the calling program is not reset.
=YES	The program mask of the calling program which was saved by specifying PMASK=YES(PMSAV=name) in the IFAEN macro is reset. *name* in the PMASK parameter of IFART must denote the same array as in the PMASK parameter of the IFAEN macro.
=NO	The program mask is not reset. There is no saving of the program mask.

RETURN=i	Return code. The program returns to the *i*-th statement label in the parameter list of the calling FOR1 program (default = 0).
STXIT	
=YES	The FOR1 STXIT routines are reactivated. Entries are only permissible if MAINSAV=YES or MAINSAV=NO is specified.
=<u>NO</u>	No reactivation of the FOR1 STXIT routine.
CHARFUN	
=YES	R1 contains the address of the string descriptor (simulation of a CHARACTER-FUNCTION).
=<u>NO</u>	R1 contains the return code.
=name	Symbolic address of a string descriptor, such as one generated by the IFESDS macro.
SB=address	Address of a 4-byte array in which the address of the end bit of the parameter address list is stored (most significant bit in the *n*-th word of the parameter address list where n is the number of parameters).

Note

For FUNCTION subprograms of the INTEGER type, the function value is returned in register 0. In this case FIRST=1 and LAST=12 must be specified for the registers to be recovered when the IFART macro is called. If the default values are assumed, the function value in register 0 is overwritten. In addition, registers 14 and 15 in the assembly language program must also be saved.

Example: FOR1 program calls assembly language program

The FOR1 program FORASS calls the assembly language program FAASS.

FOR1 program FORASS:

```
          PROGRAM FORASS
          INTEGER ITAB(20)
          WRITE(99,1)
       1  FORMAT('1'/' ','BRANCH TO ASSEMBLY LANGUAGE PROGRAM')
          WRITE(99,2)
       2  FORMAT(' ')
          CALL EINSPR(ITAB,IERG,K)                               (1)
    C
          WRITE(99,3)

       3  FORMAT(' ','RETURNED TO FORTRAN PROGRAM')
          WRITE(99,4) IERG
       4  FORMAT(' ','TRANSFER RESULT FROM ASSEMBLY LANGUAGE PROGRAM:',I8)
          WRITE(99,5) K
       5  FORMAT(' ',I4,' TRANSFERRED VALUES ARE SUMMED')
          IERG1=0
          DO 10 I=1,K
      10  IERG1=IERG1+ITAB(I)
          WRITE(99,6) IERG1
       6  FORMAT('0'/' ','RESULT IN FORTRAN PROGRAM:',I8/'1')
          STOP
          END
```

Assembly language program FAAS:

```
FAAS     START
         PRINT NOGEN
EINSPR   IFAEN LABEL=ENTRY                                       (2)
         LM    6,8,0(1)                                          (3)
         BALR  3,0
         USING *,3
         XR    5,5
         MVI   DRU+5,X'40'
LOESCH   MVC   DRU+6(127),DRU+5
         MVC   DRU+35(20),='IN ASSEMBLY LANGUAGE PROGRAM'
         MVI   DRU+5,X'40'
         WRLST DRU,FEHL
         MVI   DRU+4,X'02'
         XR    9,9
LESEN    RDATA  EINB,LEKA,84                                     (4)
VER      PACK  DOWO,EINB+4(8)
         CVB   4,DOWO
         ST    4,0(6)
         AR    5,4
         LA    6,4(6)
         A     9,=F'1'
         B     LESEN
LEKA     STC   15,RETCO
         CLI   RETCO,X'10'
         BE    LEKA1
         CLI   RETCO,X'0C'
         B     VER
LEKA1    MVC   DRU+6(127),DRU+5                                        (5)
         MVC   DRU+35(L'TEXT),TEXT
         WRLST DRU,FEHL
         MVI   DRU+5,X'40'
         EX    0,LOESCH
         MVC   DRU+35(L'TEXTA),TEXTA
         CVD   9,ERG
         MVC   DRU+76(8),MASKE
         ED    DRU+76(8),ERG+4
         WRLST DRU,FEHL
         MVI   DRU+5,X'40'
         EX    0,LOESCH
         MVC   DRU+35(L'TEXT1),TEXT1
         WRLST DRU,FEHL
         ST    5,0(7)
         ST    9,0(8)
         CVD   5,ERG
         MVI   DRU+5,X'40'
         EX    0,LOESCH
         MVC   DRU+66(8),MASKE
         ED    DRU+66(8),ERG+4
         MVC   DRU+35(L'TEXT2),TEXT2
         WRLST DRU,FEHL
         MVI   DRU+5,X'40'
         EX    0,LOESCH
         MVC   DRU+35(L'TEXT3),TEXT3
         WRLST DRU,FEHL
         MVI   DRU+5,X'40'
         EX    0,LOESCH
         IFART                                                         (6)
         TERM
FEHL     TERMD
EINB     DS    CL84
DOWO     DS    D
DRU      DS    0CL133
SATZL    DC    X'0085404001'
         DS    CL128
RETCO    DS    C
TEXT     DC    'FIXED-POINT NUMBERS READ INTO TABLE'
TEXT1    DC    'CALCULATION IN ASSEMBLY LANGUAGE PROGRAM'
TEXT2    DC    'RESULT IN ASSEMBLY LANGUAGE PROGRAM'
TEXT3    DC    'RETURN TO FORTRAN PROGRAM'
TEXTA    DC    'NUMBER OF RECORDS FROM READ OPERATION'
ERG      DS    D
MASKE    DC    X'4020202020202120'
         END
```

(1) Call of the assembly language program with the entry name EINSPR.

(2) Call of the entry macro IFAEN:

```
EINSPR   IFAEN LABEL=ENTRY
```

EINSPR is the symbolic address of the macro call used in the FOR1 program in its CALL statement. LABEL=ENTRY defines that the entry point is an entry name.

(3) With the LOAD MULTIPLE instruction, the address of parameter ITAB is stored in register 6, the address of parameter IERG in register 7 and the address of parameter K in register 8.

(4) The assembly language program reads fixed-point numbers, stores them in area ITAB and then calculates the total from the numbers which have been read in.

(5) The number of records read and the result are output.

(6) Call of return macro IFART:

The following defaults apply for the IFART macro:

FIRST=14	Register 14 is the first register to be restored.
LAST=12	Register 12 is the last register to be restored.
MAINSAV=NO	No save area was needed.
PMASK=STANDARD	The program mask of the calling program is reset if AMODE=24 applies, or not reset if AMODE=31 applies.
RETURN=0	The parameter list contains no statement label.
STXIT=NO	No reactivation of the FOR1 STXIT routine.
CHARFUN=NO	Register 1 contains the return code.

(7) SYSDTA is assigned to a file containing the numbers 1 through 20. Output of the program to SYSLST shows the messages of the FORTRAN program (left-justified) and the messages of the assembly language program (right-justified):

```
                                                                        (7)
 BRANCH TO ASSEMBLY LANGUAGE PROGRAM
                              IN ASSEMBLY LANGUAGE PROGRAM
                              FIXED-POINT NUMBERS READ INTO TABLE
                              NUMBER OF RECORDS FROM READ OPERATION    20
                              CALCULATION IN ASSEMBLY LANGUAGE PROGRAM
                              RESULT IN ASSEMBLY LANGUAGE PROGRAM     210
                              RETURN TO FORTRAN PROGRAM

 RETURNED TO FORTRAN PROGRAM
TRANSFER RESULT FROM ASSEMBLY LANGUAGE PROGRAM                        210
  20 TRANSFERRED VALUES ARE ADDED UP

RESULT IN FORTRAN PROGRAM:                                            210
```

FOR1 XS program calls assembly language subprogram

If a FOR1 XS program calls an assembly language subprogram which determines the (variable) number of parameters by locating the so-called end bit, the user must adapt the XS parameter address list. Whereas in the case of OLD and NXS parameter address lists with *n* parameters the most significant bit in the *n*-th word of the address list is set equal to 1, this end bit is no longer set in the case of XS parameter address lists.

When adapting the XS parameter address list, the 3 following cases may be distinguished:

1. The assembly language subprogram is not to be changed.

   Solution:

   In the FORTRAN source program the user replaces the call of the assembly language subprogram "progname"

   CALL progname (par_1,...,par_n)

   with the call of the subprogram

   ```
   CALL OLDASS (progname,par1,...,parn)
   ```

progname	Name of the assembly language subprogram which must be declared in the FORTRAN program as EXTERNAL.
par_i	*i*-th parameter to be supplied to the assembly language subprogram; $0 \leq i \leq n$, $0 \leq n \leq 408$

   The subprogram OLDASS copies the XS parameter address list supplied, removes the EXTERNAL name progname and sets the most significant bit in the *n*-th word of the parameter address list. Using the address of this adapted parameter address list in register 1 and the number of parameters in register 0, a branch is made to the assembly language program. After returning from the assembly language subprogram the end bit is deleted and control is returned to the calling FORTRAN program.

2. The FOR1 XS program is not to be changed.

   Solution:

   The user adds the call of macro IFAENO with the operand SB=address to the assembly language subprogram; the macro specifies the address of an array with a length of 4 bytes. IFAENO calculates the address of the last parameter, sets the end bit if required, and stores the address of the end bit in the array with the address SB=address. Before returning to the FOR1 XS program, macro IFARTO with the operand SB=address is called, removing the end bit. Macros IFAENO and IFARTO are stored in macro library FOR1MACLIB.

3. Neither the FOR1 XS program nor the assembly language subprogram is to be changed.

   Solution:

   The user renames the NXS assembly language program and generates a glue program under the old name of the assembly language program. This assembly language glue program

   – uses the IFEANO macro to set the end bit to the address "SB=address";
   – calls the NXS assembly language program;
   – uses the IFARTO macro to remove the end bit before returning to the FOR1 XS program.

Assembly language programs to be capable of being called both by FOR1 XS programs and by COBOL programs

COBOL always notifies the number of parameters by setting an end bit in the parameter address list. An assembly language subprogram called by COBOL determines the number of parameters by locating this end bit.

If a FOR1 XS program calls an assembly language program, the FOR1 program provides the number of parameters in register 0. The end bit is not set.

An assembly language program which determines the number of parameters by locating the end bit can be called both by COBOL and FOR1 XS programs by making the following change:

The IFAEN and IFART macro calls are changed into the corresponding IFAENO and IFARTO macro calls, which contain the address of the end bit in the additional SB=address operand.

A.10 Software products for the FOR1 user

A.10.1 Utility routine FPOOLITY

Product characteristics

With the aid of the FPOOLITY utility routine, FOR1 users can subject their own subprogram interfaces to error analysis. Format errors in subprogram calls can thus be detected during compilation. Section 12.3 describes how private FPOOL files are set up using the FPOOLITY utility.

Documentation:

"FPOOLITY" Reference Manual [22]

A.10.2 Subprogram library for high-precision arithmetic ARITHMOS

Product characteristics

ARITHMOS is a subprogram library for solving the rounding error problem in scientific calculations with floating-point numbers. ARITHMOS is based on the mathematical theory of computer arithmetic by Professor Dr. Kulisch, University of Karlsruhe.

The functions of ARITHMOS can be invoked in FORTRAN programs via the CALL interface, and in PLI1 programs via the call interface to subroutines in other languages (in this case FORTRAN).

ARITHMOS provides operations for vector and matrix calculations of maximum precision. Here maximum precision means that no further floating-point number representable in FORTRAN REAL*4 or REAL*8 is located between the exact result and the result supplied by ARITHMOS. This is achieved by exact computation of the scalar product of two vectors of arbitrary length.

Additional functions support the solution of standard problems of Linear Algebra (e.g. systems of linear equations, matrix inversion, eigenvalues) with the utmost precision, providing mathematically guaranteed error bounds. Many of the functions in ARITHMOS permit interval data to be input, which serves to determine, for example, the influence of uncertain input data on the result of the computation.

The variants ARITHMOS-PC and ARITHMOS-DL, with the same range of functions as ARITHMOS, are provided for the Siemens PC-2000 in conjunction with the operating system BS2000-PC.

Documentation:

"ARITHMOS" Description [6]
"ARITHMOS" User's Guide [5]
"ARITHMOS" Tables [7]

A.10.3 Methods base library of standardized subprograms for economics and science MEB

Product characteristics

MEB is a package of program modules for the solution of econometric, administrative, statistical and scientific/technological problems.

The entire MEB library comprises 420 methods, which are grouped into 13 classes on the basis of their respective theoretical machineries:

Class 0	Input/output program and auxiliary routines
Class 1	Matrix calculation
Class 2	Differential calculus
Class 3	Integral calculus
Class 4	Equations and polynomials
Class 5	Approximation and interpolation
Class 6	Statistics
Class 7	Optimization
Class 8	Simulation
Class 9	Reporting and planning
Class A	Special functions
Class B	Time series analysis and forecasting
Class C	Finance and insurance

With respect to the major application areas the above classes are assigned to six methods packages which may be used in any combination:

MEB-MATH	Applied mathematics; classes 2, 3, 4, 5, A
MEB-OPT	Optimization; class 7
MEB-PLAN	Reporting and planning; class 9
MEB-PROG	Forecasting; classes B, 6 (part E)
MEB-STAT	Statistics; classes 6, 8
MEB-FINANZ	Finance and insurance; class C

In addition there is a base package that is required by all other methods packages:

MEB-BASIS	Basic routines; classes 0, 1

The various methods packages are tailored to practical needs and bear the name of a certain application area without being limited to it.

Application options

The MEB library is open-ended, i.e. the user may add own program modules for special applications.

In addition to the methods, the MEB library contains an extensive information module, which has a strict hierarchical, three-tiered structure and offers
- an overview of the entire library and overviews of the individual packages
- directories of all MEB classes
- uniformly arranged descriptions of the MEB methods.

This enables the user to define, on the basis of his problem-related knowledge, the program names required for a DP solution.

The method descriptions contained in the MEB information module have the following standard format and sequence:
- a description of the underlying theory
- a description of the associated DP procedure
- an example.

The MEB programs can be used in two different ways:

- via the software product MEMO (Methods Monitor) as a query language for the MEB packages or
- by linking the MEB methods into user-own main programs in FORTRAN, COBOL, ALGOL, PASCAL, PLI1 and assembly language.

In order to ensure wide applicability of the methods, all fields have variable dimensions; the field boundaries satisfying the FORTRAN conventions are passed in the parameter list.

Documentation, presently available in German only:

"MEB-BASIS" [28]
"MEB-MATH" [30]
"MEB-STAT" [34]
"MEB-PLAN" [32]
"MEB-OPT" [31]
"MEB-PROG" [33]
"MEB-FINANZ [29]
"MEB-Anwendungsbeschreibung und Bedienungsanleitung" (BS2000) [27]
(= Reference manual and operating instructions)

A.10.4 Library Maintenance System LMS

Product characteristics

LMS is the standard library system for libraries in BS2000.

Programs and sections of programs (Source, Macro, Copy, Include), object modules, link and load modules, and procedures (JCL) as well as the associated documentation can be stored, processed and managed in PLAM libraries with the aid of LMS.

PLAM libraries can be accessed by the compilers, utilities (linkage editor/loader) and the management of the system files. Procedures can be started directly from the PLAM library.

LMS provides the user with a library system directly incorporated into the BS2000 system environment. No conversion of libraries is required for replacing old utility routines, as the latter are a subset of LMS (MLU/LMR). COBLUR and FMS libraries can be read directly by LMS.

LMS supports program libraries, i.e. PAM files processed using the Program Library Access Method (PLAM).

The variants LMS-PC and LMS-DL are provided for the Siemens PC-2000 in conjunction with the operating system BS2000-PC. PLAM libraries are not yet supported in these variants.

Description of functions

LMS can be used to store, manage and update all forms of program elements.

LMS manages, stores and updates source, macro, object module, link and load module, procedure and text elements in libraries.

Source/macro, module and new program libraries can be generated and processed.

In addition, already existing MLU, LMR, COBLUR and FMS libraries can also be processed by LMS. Conversion of the previously used methods to LMS is thus considerably facilitated and simplified.

LMS and the PLAM access method permit the following:

- setting up and copying of LMS libraries and the creation of library tapes (archiving and transport function)
- reading and processing of COBLUR and FMS libraries and their elements

- common storage in one library
  - of all types of program elements, especially source programs, object modules, load modules and link and load modules, as well as listings and procedures
  - of elements with the same name but whose type designations or version designations differ
- management and maintenance of all types of library elements, including
  - addition, correction and deletion of library elements
  - renaming, transferring, numbering and identification of elements;
  - creation and management of version numbers and the respective archiving date for elements;
  - fully or partially qualified access to library elements, also taking into account the version number and date of archiving with and without the use of inclusive/exclusive strings;
  - a RUN/TEST mode for event-independent monitoring of function sequences or automatic switchover in the event of errored commands, with the purpose of avoiding consequential errors;
  - call of the editor EDT within the LMS system, for direct processing of elements at the terminal.
- access
  - by compilers, linkage editor/loader and management of system files;
  - simultaneous use of a common library by more than one user (also write access).
- space-saving storage technique through compression of elements.

Program description

Use of the BS2000 library system LMS considerably reduces the workload on the BS2000 catalog. Through compressed storage of all elements, there is an extensive reduction in the original storage space required for these elements.

System performance is enhanced through the access facilities for the compilers, utility routines and management of the system files. The processing method employed by LMS guarantees a high degree of reliability, since the status of the libraries is always kept consistent with the aid of internal mechanisms.

Documentation

"LMS" Reference Manual [25]

A.10.5 Job variables

Product characteristics

Job variables are data objects for exchanging information between users on the one hand and the operating system and users on the other hand.

The user can create and update job variables. He can instruct the operating system to set certain job variables to specified values when specific events occur.

Job variables provide a flexible tool for job monitoring under user control. They make it possible to define interdependencies between complex production sequences and they form the basis for event-driven job processing.

Description of functions

Job variables are objects which are managed by the operating system and addressed via their names; data up to a length of 256 bytes can be stored in them. They are used to exchange information between users on the one hand and between the operating system and users on the other. They can be accessed via the commmand and macro interface.
In conditional statements, job variables can be linked via Boolean operations, thus making the execution of individual actions dependent on the state of the condition.
User job variables and monitoring job variables (see below) additionally provide the facility for synchronous and asynchronous event control on command and program level.

Various job variables are available for the various application sectors:

User job variables

User job variables are the most common form of job variables offered. Their name, lifespan and the data to be stored are determined exclusively by the user. They can be given protection attributes such as passwords, write protection and expiration date. Access to them can be restricted to one user ID or generally authorized.

User job variables are especially well-suited for the exchange of information. They can, however, also be used for job control purposes.

Monitoring job variables

A monitoring job variable is a special form of user job variable. It is assigned to either a job or a program. Name, lifespan and protection attributes are defined by the user. In contrast to user job variables, however, it is supplied with fixed values specified by the operating system to reflect the status of the assigned job or program.

Monitoring job variables are especially well-suited for controlling jobs as is required, for example, within interdependent production sequences.

Documentation

"Job Variables" User Guide [24]

A.10.6 Graphical kernel system GKS-GA

Product characteristics

GKS-GA (BS2000) implements the standardized BS2000 graphical kernel system GKS (ISO 7942, DIN 66252). GKS provides the basic functions for computer-based generation and handling of two-dimensional graphics. It permits the storage and dynamic modification of graphics as well as their output to suitable output devices.

The GKS-GA functions are independent of the graphics device type, the application, and the programming language. Consequently, the application programs using GKS are independent of the device type used.

GKS-GA offers a standard interface with the application program via several programming languages as well as a uniform internal interface to the device-dependent drivers.

Description of functions

GKS-GA, via a standard application program interface, lays down a set of functions for the generation of any desired two-dimensional graphics by an application program.

All GKS functions are application independent. They are divided into the following functional areas:

– Display elements (polygon, polymarker, fill area, cell matrix, generalized display element, text).
– Display attributes (color, line thickness, line type, text alignment, etc.).
– Graphics workstation (generalization of real graphics devices for device-independent programming).
– Transformation (coordination of user (or world) coordinate system / standardized / device coordinate system; magnification, reduction.
– Picture segmentation (definition and manipulation of segments).
– Picture file (metafile: long-term storage of pictures, transfer, reentry).
– Error handling (controlled by GKS or application program).

GKS is available for applications under the transaction monitor UTM, for VTX applications and for applications in timesharing mode (TIAM).

User programming languages: COBOL, FORTRAN, assembly language.

Documentation

Graphical Kernel System [23]

References

[1] **AID** (BS2000)
Advanced Interactive Debugger
Core Manual
User Guide

Target group
Programmers in BS2000
Contents
- Overview of the AID system
- Description of facts and operands which are the same for all programming languages
- Messages
- Comparison between AID and IDA

Applications
Testing of programs in interactive or batch mode

[2] **AID** (BS2000)
Advanced Interactive Debugger
Debugging on Machine Code Level
User Guide

Target group
Programmers in BS2000
Contents
- Description of the AID commmands for debugging on machine code level
- Sample application

Applications
Testing of programs in interactive or batch mode

[3] **AID** (BS2000)
Advanced Interactive Debugger
Debugging of FORTRAN Programs
User Guide

Target group
FORTRAN programmers
Contents
– Description of the AID commands for symbolic debugging of FORTRAN programs
– Sample application
Applications
Testing of FORTRAN programs in interactive or batch mode

[4] **ARCHIVE** (BS2000)
User Guide

Target group
– BS2000 system administrators
– Operators
– End users
Contents
Description of the statements for saving and reconstructing files with ARCHIVE

[5] **ARITHMOS**
User's Guide

Target group
FORTRAN and Assembler programmers
Contents
Problems of conventional floating-point arithmetic, ARITHMOS concept, problem-solving routines, basic operations, elementary statements, error handling, and basic instructions of ARITHMOS.

[6] **ARITHMOS**
Outlinde Description

Target group
FORTRAN and Assembler programmers
Contents
Problems of conventional floating-point arithmetic, ARITHMOS concept, overview of ARITHMOS functions, practical applications, hardware and software prerequisites.

[7] **ARITHMOS**
Tables

Target group
FORTRAN and Assembler programmers
Contents
Summary overview of ARITHMOS functions, list of ARITHMOS function calls, basic instructions of ARITHMOS.

[8] **Assembler Instructions** (BS2000)
Reference Manual

Target group
BS2000 assembly-language programmers
Contents
This manual describes in alphabetical order all (nonprivileged) assembler instructions of the CPUs supported by BS2000. For each instruction the following is described:
– its function
– its assembler format, i.e. how to write it in assembly language
– its machine format, i.e. how it is represented in the CPU
– its execution sequence in detail
– any condition codes values which it sets
– possible program interrupts when it is executed
– programming notes
– one or more examples
Applications
BS2000 assembly-language application programmers

[9] **ASSEMBH** (BS2000)
User Guide

Target group
Assembly language users under BS2000
Contents
– Calling and controlling ASSEMBH
– Assembling, linking, loading, and starting programs
– Input sources and output of ASSEMBH
– Runtime system, structured programming
– Language interfacing
– Assembler Diagnostic Program ASSDIAG
– Advanced Interactive Debugger AID
– ASSEMBH messages
– Machine instruction formats

[10] **ASSEMBH** (BS2000)
Reference Manual

Target group
Assembly language users under BS2000
Contents
- Language scope of the assembler ASSEMBH
- Assembly language structure, assembler instructions
- Structure, elements and instructions of the macro language
- Structured programming with ASSEMBH-XT, predefined macros for structured programming

[11] BS2000
User Commands (ISP Format)
User Guide

Target group
BS2000 users (non-privileged)
Contents
- All BS2000 system commands in alphabetical order with detailed explanations and examples
- The following products are dealt with:
  BS2000-GA, MSCF, JV, FT, TIAM

Applications
BS2000 interactive/batch mode, procedures

[12] BS2000
User Commands (SDF Format)
User Guide

Target group
BS2000 users
Contents
BS2000 user commands in the syntax of the dialog interface SDF (System Dialog Facility)
Applications
BS2000 interactive/batch mode with SDF

[13] BS2000
Binder-Loader-Starter (BLS)
User Guide

Target group
Software developers
Contents
The binder-loader-starter (BLS) system consists of the following functional units:
– Linkage editor BINDER
– Dynamic binder loader DBL
– Static loader ELDE
The various sections contain functional descriptions and examples, plus a reference section with statements, commands and, where applicable, macros.

[14] **C** (BS2000)
C Compiler
User Guide

Target group
C users in a BS2000 environment
Contents
– Description of all activities concerned with the creation of an executable C program: compilation, linking, loading, debugging
– Programming notes and additional information on: program runtime control, file processing, event handling, locale concept, language interfacing, language features of the C compiler, messages

[15] **COBOL85 (BS2000)**
COBOL Compiler
User's Guide

Target group
COBOL users of BS2000
Contents
– Generation of the COBOL85 compiler and the software required for the linking, loading and debugging of COBOL programs
– File processing with COBOL programs
– Inter-program communication
– Structure of the COBOL85 system
– Compiler messages and runtime system messages

[16] BS2000
Introductory Guide to the SDF Dialog Interface
User Guide

Target group
BS2000 users
Contents
- The various input options offered with SDF in system operation
- Operating instructions and examples relating to optional user guidance via menus

Applications
General

[17] BS2000
Utility Routines
User Guide

Target group
BS2000 users (non-privileged)
Contents
Utility routines for non-privileged BS2000 users
Applications
BS2000 timesharing mode

[18] BS2000
DMS Introductory Guide and Command Interface
User Guide

Target group
Non-privileged BS2000 users
Contents
- Functions of DMS in BS2000
- Processing of disk and tape file
- Access methods UPAM, SAM, BTAM, EAM, ISAM
- DMS commands

[19] BS2000
DMS Assembler Interface
User Guide

Target group
Non-privileged BS2000 users/assembly-language programmers
Contents
– Functions of DMS in BS2000 (at macro level)
– Processing of disk and tape files (at macro level)
– Access methods UPAM, SAM, BTAM, EAM, ISAM (including action macros)
– File processing macros

[20] **EDT** (BS2000)
Statements
User Guide

Target group
– EDT newcomers
– End users
Contents
– Processing of SAM and ISAM files and elements from program libraries
– Introduction to the basic principles of EDT and description of the operating modes
– Creation of EDT procedures
– Descriptions of all the EDT statements. Frequent applications are illustrated with the aid of numerous examples.
Applications
File editing

[21] **FOR1** (BS2000)
FORTRAN Compiler
Reference Manual

Target group
FORTRAN users in BS2000
Contents
Description of the language range of the FOR1 compiler: basic elements of FORTRAN, control statements, input/output statements, specification statements and data initialization statements, assignment statements and formats; structure and construction of a FORTRAN program.

[22] **FPOOLITY** (BS2000)
Reference Manual

Target group
BS2000 programmers
Contents
- The FPOOL concept enables compilers to check the compatibility of interfaces to called functions
- Description of the FPOOL concept
- Operation of FPOOLITY for the generation of interface descriptions
- FPOOL handling

[23] **Graphical Kernel System** (BS2000) *
User Guide

Target group
FORTRAN, COBOL and Assembler programmers
Contents
Description of the application of the Graphical Kernel System (GKS)

[24] BS2000
Job Variables
User Guide

Target group
BS2000 users
Contents
- Applications for job variables in controlling and monitoring jobs and program runs
- Conditional job control
- All the necessary commands and macros
- Application examples

Applications
BS2000 timesharing mode

[25] **LMS** (BS2000)
ISP Format
Reference Manual

Target group
BS2000 users
Contents
Description of the LMS statements in ISP format for creating and managing PLAM libraries and the members these contain.
Frequent applications are illustrated by means of examples.

[26] BS2000
Executive Macros
User Guide

Target group
- BS2000 assembly language programmers (non-privileged)
- System administrators

Contents
- All Executive macros in alphabetical order with detailed explanations and examples; selected macros for DMS and TIAM
- Macro overview according to application areas
- Comprehensive training section dealing with eventing, serialization, inter-task communication, contingencies

Applications
BS2000 application programs

[27] **MEB** *
Reference Manual and Operating Guide

Target group
FORTRAN, COBOL, ALGOL, PASCAL, PLI1 and Assembler programmers
Contents
Reference manual for the MEB methods base library of standardized subprograms for commercial and scientific applications

[28] **MEB-BASIS**
Basic Routines *
Program Description

Target group
FORTRAN, COBOL, ALGOL, PASCAL, PLI1 and Assembler programmers
Contents
Description of program modules (input/output and other utility routines, matrix calculation) required by the MEB methods packages.

[29] **MEB-FINANZ**
Financial and Insurance Mathematics *
Program Description

Target group
FORTRAN, COBOL, ALGOL, PASCAL, PLI1 and Assembler programmers
Contents
Description of program modules for finance and insurance

[30] **MEB-MATH**
Applied Mathematics *
Program Description

Target group
FORTRAN, COBOL, ALGOL, PASCAL, PLI1 and Assembler programmers
Contents
Description of program modules for applied mathematics (differential and integral calculus, equations and polynomials, approximation and interpolation, special functions)

[31] **MEB-OPT**
Optimization *
Program Description

Target group
FORTRAN, COBOL, ALGOL, PASCAL, PLI1 and Assembler programmers
Contents
Description of program modules for optimization

[32] **MEB-PLAN**
Planning and Reporting *
Program Description

Target group
FORTRAN, COBOL, ALGOL, PASCAL, PLI1 and Assembler programmers
Contents
Description of program modules for planning and reporting

[33] **MEB-PROG**
Time Series Analysis and Forecasting *
Program Description

Target group
FORTRAN, COBOL, ALGOL, PASCAL, PLI1 and Assembler programmers
Contents
Description of program modules for forecasting (time series analysis and forecasting, statistics)

[34] **MEB-STAT**
Statistics *
Program Description

Target group
FORTRAN, COBOL, ALGOL, PASCAL, PLI1 and Assembler programmers
Contents
Description of program modules for statistics (statistics and optimization)

[35] **Pascal-XT** (BS2000)
User's Guide

Target group
Pascal-XT users in BS2000
Contents
- Operation of the programming system and of the compiler
- Description of the BS2000-specific attributes of the compiler
- Linking and executing programs
- Language interfaces
- Runtime error messages
- Description of predefined packages
- Comparison with Pascal Version 3

[36] **PERCON** (BS2000)
Reference Manual

Target group
BS2000 users
Contents
Description of the PERCON statements in ISP format for transferring and converting files with PERCON
Applications
BS2000 interactive/batch mode

[37] **PLI1** (BS2000)
PL/I Compiler
User's Guide

Target group
PL/I users in BS2000
Contents
- Invocation and control of the PLI1 compiler
- Input and compilation of source programs
- Creation and management of object and load modules
- Generation of shareable programs
- Control of program execution
- File access
- Debugging aids
- Optimization
- Internal representation of data
- Procedure interfaces
- Service procedures
- PL/I/assembly macro interface

[38] **System Installation** (BS2000)
User Guide

Target group
BS2000 system administrators
Contents
- New installation
- Version changeover
- Generation of a new public volume set
- Generation of a subsystem catalog
- Statements for SIR and UGEN

Applications
- System administration
- Computer center

[39] **Systems Standards** (BS1000, BS2000, TRANSDATA, PDN)
Reference Manual

Target group
Users of Siemens mainframes
Contents
- Operating system standards for BS1000, BS2000 and TRANSDATA PDN
- Standards for data volumes
- Codes for character representation

[40] BS2000
System Administrator's Guide
User Guide

Target group
BS2000 system administration
Contents
Description of the options and responsibilities of the system administration for the control and management of the operating system.
The manual contains the following chapters:
- System administration (user and file administration, accounting, system diagnostics, corrections to the system, parameter service)
- System control and optimization (job, task and memory management, DSSM, MPVS)
- Data security (SRPM, FACS, SAT)
- Data protection (protection strategies, software products for data protection, file reconstruction)
- Automation of system operation
- Commands in SDF format

Applications
- System administration
- Computer center

[41] BS2000
TSOSLNK
User Guide

Target group
Software developers
Contents
– Statements and macros of the linkage editor TSOSLNK for linking load modules and prelinked modules
– Commands of the static loader ELDE

* available in German only

Ordering manuals

The manuals listed above and the corresponding order numbers are to be found in the *List of Publications* issued by Siemens Nixdorf Informationssysteme AG, which also tells you how to order manuals. New publications are listed in the *Druckschriften-Neuerscheinungen (New Publications)*.

You can arrange to have both of these sent to you regularly by having your name placed on the appropriate mailing list. Your local office will help you.

Index