

C/C++ V3.2D

C/C++ Compiler

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 9001:2008

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 9001:2008.

cognitas. Gesellschaft für Technik-Dokumentation mbH

www.cognitas.de

Copyright and Trademarks

Copyright © Fujitsu Technology Solutions GmbH 2011.

All rights reserved.

Delivery subject to availability; right of technical modifications reserved.

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

Contents

1	Preface	9
1.1	Brief product description	9
1.2	Summary of contents and target group	10
1.3	Changes since the previous version	11
1.4	Notational conventions	12
1.4.1	General notational conventions	12
1.4.2	SDF notational conventions	13
2	Overview of the C/C++ development system	21
2.1	From source program to program execution	21
2.2	General requirements for compilation, linkage and program execution	23
2.3	General features of the C/C++ compiler	24
2.4	Specific CRTE components required for C/C++	25
2.4.1	Include libraries	25
2.4.2	Module libraries	26
2.5	Editing source programs	29
2.6	POSIX support	32
2.6.1	Compiler I/O in the POSIX file system	32
2.6.2	Use of POSIX library functions	35
2.7	Introductory examples	37
	Example 1: Compiling, linking, and starting a C program	37
	Example 2: Compiling, linking, and starting a C++ program (ANSI C++)	40
	Example 3: Compiling a C source program that is located in a POSIX file and uses POSIX library functions	43

3	Compilation	45
3.1	General aspects of the compiler run	45
3.1.1	Input sources and output destinations of the compiler	45
3.1.2	Construction of default names	48
	Default names for output containers	48
	Rules for constructing module names	52
3.1.3	Structure of compiler messages	55
3.2	Controlling the compiler	59
3.2.1	Calling the compiler (START-CPLUS-COMPILER)	60
3.2.2	Description of compiler statements	61
	Overview of statements	61
	Basic principles and general input rules	64
	BIND	66
	CHECK-SYNTAX	70
	COMPILE	72
	Notes on input via SYSDTA	76
	END	78
	MODIFY-BIND-PROPERTIES	79
	Interaction between the MODIFY-BIND-PROPERTIES and BIND statements	89
	MODIFY-CIF-PROPERTIES	90
	MODIFY-DIAGNOSTIC-PROPERTIES	94
	MODIFY-INCLUDE-LIBRARIES	100
	MODIFY-LISTING-PROPERTIES	105
	MODIFY-MODULE-PROPERTIES	115
	MODIFY-OPTIMIZATION-PROPERTIES	121
	The optimization process	126
	MODIFY-RUNTIME-PROPERTIES	131
	MODIFY-SOURCE-PROPERTIES	133
	MODIFY-TEST-PROPERTIES	147
	PREPROCESS	148
	RESET-TO-DEFAULT	153
	SHOW-DEFAULTS	154
	SHOW-PROPERTIES	155
3.3	Controlling the global listing generator	156
3.3.1	Calling the listing generator (START-CPLUS-LISTING-GENERATOR)	156
3.3.2	Description of statements	157
	Overview of statements and input rules	157
	END	157
	GENERATE-LISTING	158
	MODIFY-LISTING-PROPERTIES	160

4	Linkage and program execution	169
4.1	Linkage	169
4.1.1	Dynamic linking and loading with DBL	171
4.1.2	Linking with BINDER	174
4.1.3	Shareable C/C++ programs	177
4.1.4	Restriction on linking ANSI C++ programs	178
4.2	Program execution	179
4.2.1	Parameter input at program start	179
	Redirecting standard I/O files	180
	Input of parameters for the main function	181
	Definition of the main function with parameters	183
4.2.2	The advanced interactive debugger AID	184
	Requirements for symbolic debugging	186
5	Linkage to functions and languages	189
5.1	Linkage conventions specific to C and C++	189
	Parameter passing "by value"	189
5.2	Linkage between C and C++	191
5.2.1	Common types	192
5.2.2	Calling C functions in C++	193
5.2.3	Calling C++ functions in C	194
5.2.4	Problems and restrictions	195
5.3	Linkage between Cfront C++ and ANSI C++	195
5.4	Notes on linkage to ILCS programs in other languages	196

6	C language support of the compiler	199
6.1	Overview of the C language modes	200
6.2	Implementation-defined behavior based on the ANSI/ISO C standard	208
6.3	Extensions to ANSI/ISO C	219
6.4	Pragmas	223
6.4.1	aligned pragma	223
6.4.2	pack pragma	225
6.4.3	ETPND pragma	226
6.4.4	Pragmas to control the layout of listings	228
	LISTING pragma	228
	TITLE pragma	230
	PAGE pragma	230
	SPACE pragma	231
6.4.5	inline pragma	231
6.4.6	int_to_unsigned pragma	231
6.4.7	weak pragma	232
6.4.8	ident pragma	232
6.4.9	C++ specific pragmas	232
	VIRTUAL_FUNCTION_TAB pragma	232
	Pragmas to control template instantiation	233
7	C++ language support of the compiler	235
7.1	Overview of the C++ language modes	235
7.2	Implementation-defined behavior based on the ANSI/ISO C++ standard	239
7.3	Template instantiation	244
7.3.1	Fundamentals	244
7.3.2	Automatic instantiation	246
7.3.3	Generating explicit template instantiation statements (ETR files)	252
7.3.4	Implicit inclusion	259
7.4	Deviations from ANSI/ISO C++	260
7.4.1	Extensions to ANSI-/ISO-C++	260
7.4.2	extern inline vs. static inline	261
7.5	Variations in the Cfront C++ mode	267

8	The C++ libraries and C++ runtime system	275
8.1	The standard C++ library	275
8.2	The Cfront C++ library	277
8.3	The Tools.h++ library	279
8.4	The C++ runtime system	281
8.4.1	Initialization	281
8.4.2	Exception handling	282
	Additional runtime functions	282
	C signal handling and C++ exception handling	285
	longjmp support	285
	Linking old C modules with ANSI C++ modules	287
9	Appendix	289
9.1	Description of listings	289
	Source/error listing	290
	Map listing	292
	Cross-reference listing	295
	Object listing	299
9.2	Predefined preprocessor names	302
9.3	Concept of a name adapter module in the C runtime system	304
9.4	The II-UPDATE tool	305
9.5	EBCDIC table (EDF041)	313
9.6	ASCII table (ISO 8859-1)	318
	Related publications	319
	Index	323

1 Preface

1.1 Brief product description

C/C++ provides users with a versatile compiler that supports both the C and C++ programming languages.

Depending on which language mode is set, the C/C++ compiler accepts:

- C source code that conforms to the definition of C by B.W. Kernighan and D.M. Ritchie
- C source code as defined in the ANSI/ISO C standards, including the ISO C Amendment 1
- C++ code compatible with Cfront V3.0.3 *)
- C++ code that complies with the ANSI/ISO C++ Standard (Draft 1996) (see [27])

Programmers who are developing C and C++ programs are also supported by a global listing generator which, among other things, can be used to create listings for multiple modules (e.g. cross-reference listings, project listings).

Both the C/C++ compiler and the supplementary development tools can be operated and controlled via a convenient SDF user interface.

In order to generate and execute C/C++ programs, you will also need the **Common RunTime Environment CRTE**, which is supplied with the following components:

- headers and modules for the C library (ANSI C functions, POSIX functions as defined in XPG4 spec1170, BS2000-specific extensions) and
- headers and modules for various C++ libraries (i.e. the C++ V2.1 compatible C++ library, the standard C++ library and the Tools.h++ library)

*) "Cfront" refers to the C++ industry standard, as defined and implemented in the C++ translator (C++ based on C) of AT&T and USL/Novell.

1.2 Summary of contents and target group

This User Guide describes how C/C++ programs are processed with the C/C++ compiler and other components of the C/C++ Development System in a BS2000 system environment (SDF) under the BS2000 operating system.

It is intended for users who are familiar with the programming languages C and C++ as well as the BS2000 operating system.

The following topics are dealt with here:

- Preparing and compiling source programs
- Creating listings with the global listing generator
- Linking, loading and starting
- Executing C/C++ programs (parameter input, debuggers)
- Linkage to functions and languages
- C language support of the compiler (overview of C language modes, implementation defined behavior, `#pragma` directives, extensions to the ANSI/ISO C standard)
- C++ language support of the compiler (overview of C++ language modes, implementation defined behavior)
- Brief description of the C++ libraries supplied with CRTE

You can also develop programs in a POSIX environment by using the `cc`, `c89`, `CC` and `cc1istgen` commands. These commands are described in a separate manual under the title “POSIX Commands of the C/C++ Compiler” [1], which serves as the main reference source for POSIX commands.

For more detailed information on the features and functionality of the compiler (beyond the scope of POSIX control), see also the following sections and chapters in this User Guide:

- [section “The optimization process” on page 126](#)
- [section “Structure of compiler messages” on page 55](#)
- [chapter “C language support of the compiler” on page 199](#)
- [chapter “C++ language support of the compiler” on page 235](#)
- [chapter “Linkage to functions and languages” on page 189](#)
- [chapter “The C++ libraries and C++ runtime system” on page 275](#)

In the text, references to other publications are given using short titles. The full title of each publication is listed in the “References” section.

1.3 Changes since the previous version


The changes in this User Guide compared to the C/C++ compiler V3.2A are mainly due to the changes in the listings output.

1.4 Notational conventions

All notational conventions used in this manual to represent commands, statements and compiler options are explained in detail in the two subsections below.

1.4.1 General notational conventions

The following general notational conventions are used in this manual to indicate the format of BS2000 commands and program statements:

*STD	All uppercase letters, digits, and special characters that are not part of the “metalanguage” denote keywords or constants and must be entered exactly as shown.
-R msg_id	Uppercase and lowercase letters, digits, and special characters in typewritten text denote constants and must be entered exactly as shown.
<i>name</i>	Lowercase letters in <i>italics</i> denote variables that must be replaced by current values at the time of input (see also <name>).
<name>	In SDF formats, variables are additionally enclosed in angle brackets.
<u>YES</u>	Underlining denotes the default value, i.e. the value that is automatically used if no specification is made.
<u>NO</u>	
$\left. \begin{array}{l} \text{YES} \\ \text{NO} \end{array} \right\}$	Braces enclose alternatives, i.e. one of the specified values must be selected. No entry is required if the underlined default value is desired.
<u>YES</u> / <u>NO</u>	A slash between adjacent entries also indicates alternatives from which one must be selected. No entry is required if the underlined default value is desired.
[]	Square brackets enclose options that may be omitted.
()	Parentheses are constants and must be specified.
_	This symbol indicates that at least one whitespace character is necessary for the syntax.
...	Ellipses signify repetition, which means that the preceding unit can be repeated several times in succession.
	For informative texts.

1.4.2 SDF notational conventions

The following tables describe the notational conventions used to represent SDF commands and statements of the C/C++ compiler and the global listing generator.

Table 1: Special characters

The special characters and notational conventions used in SDF statement formats are explained in the table below:

Format	Meaning	Examples
UPPERCASE	Uppercase letters indicate keywords (names of commands, statements, and operands, keyword values). Some keywords start with *.	START-CPLUS-COMPILER COMPILE
=	The equals sign connects an operand name with the corresponding operand values.	ELEMENT = <u>*STD-ELEMENT</u> LISTING = <u>*NONE</u>
< >	Angle brackets indicate variables, the values of which are defined by data types and their suffixes (see the following tables).	VERSION = <text 1..24>
<u>Underline</u>	Underlining indicates default values of operands, i.e. values which are automatically used when no explicit specification is made.	SUMMARY = *YES / <u>*NO</u>
/	A slash separates alternative operand values.	TEST-SUPPORT = *YES / <u>*NO</u>
(...)	Parentheses indicate operand values which introduce a structure.	LIBRARY = *LINK(...)
[]	Square brackets indicate optional operand values that introduce structures. The structure that follows may be specified without the introductory operand value.	SOURCE = [*YES](...)
Indentation	Indentation indicates dependence on the higher- ranking operand.	LIBRARY = <filename> / *LINK(...) *LINK(...) LINK-NAME =

Format	Meaning	Examples
<pre> list-poss(n): list-poss: </pre>	<p>Vertical lines denote associated operands in a structure. The top and bottom of the line indicate the beginning and end of a structure. Further structures may occur within a structure. The number of vertical lines before an operand corresponds to the depth within the structure.</p> <p>A comma precedes further operands of the same level within the structure.</p> <p>A list can be generated from the operand values following list-poss. If (n) is specified, the list may contain at most n elements. If the list contains more than one element, it must be enclosed within parentheses. In this manual, (n) is indicated only if a compiler-specific maximum number is involved. list-poss without (n) means that the maximum SDF default value of 2000 applies.</p>	<pre> *LIBRARY-ELEMENT(...) LIBRARY = ,ELEMENT = VERSION = </pre> <pre> ,SOURCE = *NO ,SUMMARY = *NO list-poss (127): *STD / BY-SOURCE / <c-string> list-poss: <filename> / *LINK(...) </pre>

Table 2: Data types

Variable operand values are indicated in SDF by data types. Each data type represents a specific value range. The number of data types is restricted to those described in the table below.

The following descriptions of data types apply to all statements. Consequently, only the operands which deviate from the definitions given here are explained in the individual operand descriptions.

Data type	Character set	Meaning
alphanum-name	A...Z 0...9 \$, #, @	
c-string	EBCDIC characters	A sequence of EBCDIC characters enclosed in single quotes. C may precede the characters. Single quotes within the c-string must be duplicated. The data type c-string is provided with the suffix with-low, which means that a distinction is made between uppercase and lowercase letters. Examples of valid representations: 'abc' , C'_abc' , 'ABC' , 'printf("macro-text\ n")'.
composed-name	A...Z 0...9 \$, #, @ hyphen period	Name or version of a PLAM library element. composed-name is provided with the suffix with-under (underscore) (see also Table 3, page 18).
filename	A...Z 0...9 \$, #, @ hyphen period	Link name or fully-qualified name of a cataloged file or a PLAM library. The maximum length for link names is 8 characters; the maximum length for fully-qualified names including the cat-id and user-id is 54 characters (or 41 characters excluding the cat-id and user-id). Input format: link-name $ \begin{array}{l} \left. \begin{array}{l} \text{file} \\ \text{file(no)} \\ \text{group} \end{array} \right\} \\ \text{:cat:$user.} \left\{ \begin{array}{l} \\ \text{group} \left\{ \begin{array}{l} (*\text{abs}) \\) \end{array} \right\} \\ (+\text{rel}) \end{array} \right\} \end{array} $

Data type	Character set	Meaning
filename (continuation)		<p>link-name The first and last character must not be a hyphen or period; maximum of 8 characters; must contain at least A...Z.</p> <p>:cat: Optional catalog identifier; character set restricted to A...Z and 0...9; maximum of 4 characters; must be enclosed in colons; the default value is the catalog identifier assigned to the user ID in accordance with the entry in the user catalog (also called the JOIN entry).</p> <p>\$user. Optional user ID; character set is A...Z, 0...9, \$, #, @; maximum of 8 characters; must not begin with a digit; \$ and the period must be included; the default value is the own user ID.</p> <p>\$. (special case) System default identifier</p> <p>file File or job variable name; first and last character must not be a hyphen or period; maximum of 41 characters; must contain at least A...Z.</p> <p>#file @file (special case) # or @ as the first character indicates temporary files or job variables, depending on system generation.</p> <p>file(no) Tape file name no: version number; character set is A...Z, 0...9, \$, #, @. Parentheses must be included.</p>

Data type	Character set	Meaning
filename (continuation)		<p>group Name of a file generation group (see "file" for character set)</p> $\text{group} \left\{ \begin{array}{l} (*\text{abs}) \\ (+\text{rel}) \\ (-\text{rel}) \end{array} \right\}$ <p>(*abs) Absolute generation number (1-9999); * and parentheses must be included.</p> <p>(+rel) (-rel) Relative generation number (0-99); sign and parentheses are required.</p>
integer	0...9,+,-	+ or - are possible only as the first character.
name	A...Z 0...9 \$,#,@	C/C++ name for DEFINE / UNDEFINE in the MODIFY-SOURCE-PROPERTIES statement. The data type name is provided with the suffix underscore (see also Table 3, page 18). Lowercase letters cannot be represented with name; c-string must be used for this purpose (see page 15). Maximum length: 125 characters
posix-filename	A...Z 0...9 special characters	String with a maximum length of 255 characters; consists of either one or two periods, or alphanumeric characters and special characters; all special characters must be escaped with a preceding \ (backslash); the / character is not allowed. A distinction is made between uppercase and lowercase letters
posix-pathname	A...Z 0...9 special characters structural identifiers: slashes	Input format: $[/]\text{part}_1[/\dots/\text{part}_n]$ where part_i is a posix-filename; up to 1023 characters; up to 247 characters for source and header files. The data type posix-pathname is provided with the suffix mandatory-quotes and must therefore always be enclosed within single quotes.
x-string	Hexadecimal: 00...FF	Sequence of hexadecimal values enclosed within single quotes; must be preceded by the letter X.

Table 3: Data type suffixes

Data type suffixes denote further input specifications for data types. These suffixes place restrictions on the value range or extend it. The following suffixes are used in their abbreviated form in the manual:

cat-id	cat
correction-state	corr
generation	gen
lower-case	low
manual-release	man
underscore	under
user-id	user
version	vers

The following descriptions of suffixes apply to all statements. Consequently, only the operands which deviate from the definitions given here are explained in the individual operand descriptions.

Suffix	Meaning
x..y	<p>a) For data type integer: interval</p> <p>x Minimum value permitted for integer. x is an integer which may be given a sign.</p> <p>y Maximum value permitted for integer. y is an integer which may be given a sign.</p> <p>b) For other data types: length</p> <p>x Minimum length for the operand value; x is an unsigned integer.</p> <p>y Maximum length for the operand value; y is an unsigned integer.</p>
with	Extends the options for specifying a data type.
-low	A distinction is made between uppercase and lowercase letters.
-under	The underscore (_) is permitted as an additional character.
without	Restricts the options for specifying a data type.
-cat	Specifying a catalog identifier is not permitted.
-corr	Input format: [[C]'][V][m]m.na'] No correction status may be included in the product-version data type.
-gen	Specifying a file generation or file generation group is not permitted.

Suffix	Meaning
without (continuation)	
-man	Input format: [[C]'][V][m]m.n[']] Neither the release status nor the correction status may be included in the product-version data type.
-user	Specifying a user identifier is not permitted.
-vers	Specifying a user identifier is not permitted.
mandatory	Makes certain entries mandatory for a data type
-quotes	Specifications for the data types posix-filename and posix-pathname must be enclosed within single quotes.

2 Overview of the C/C++ development system

2.1 From source program to program execution

Three steps are necessary to convert a C/C++ source program into an executable program:

1. Preparation of the source program and header files

The source program may be in a cataloged BS2000 file, a PLAM library element (of type S), or a POSIX file.

Header files may be located in PLAM library elements (type S), in POSIX files, or also in cataloged files if the source program itself is located in a cataloged file.

The standard headers for the C and C++ library functions (excluding POSIX functions) are located by default in the CRTE libraries `$.SYSLIB.CRTE` and `$.SYSLIB.CRTE.CPP`, respectively; the standard headers for POSIX library functions are located in the library `$.SYSLIB.POSIX-HEADER`.

2. Compilation

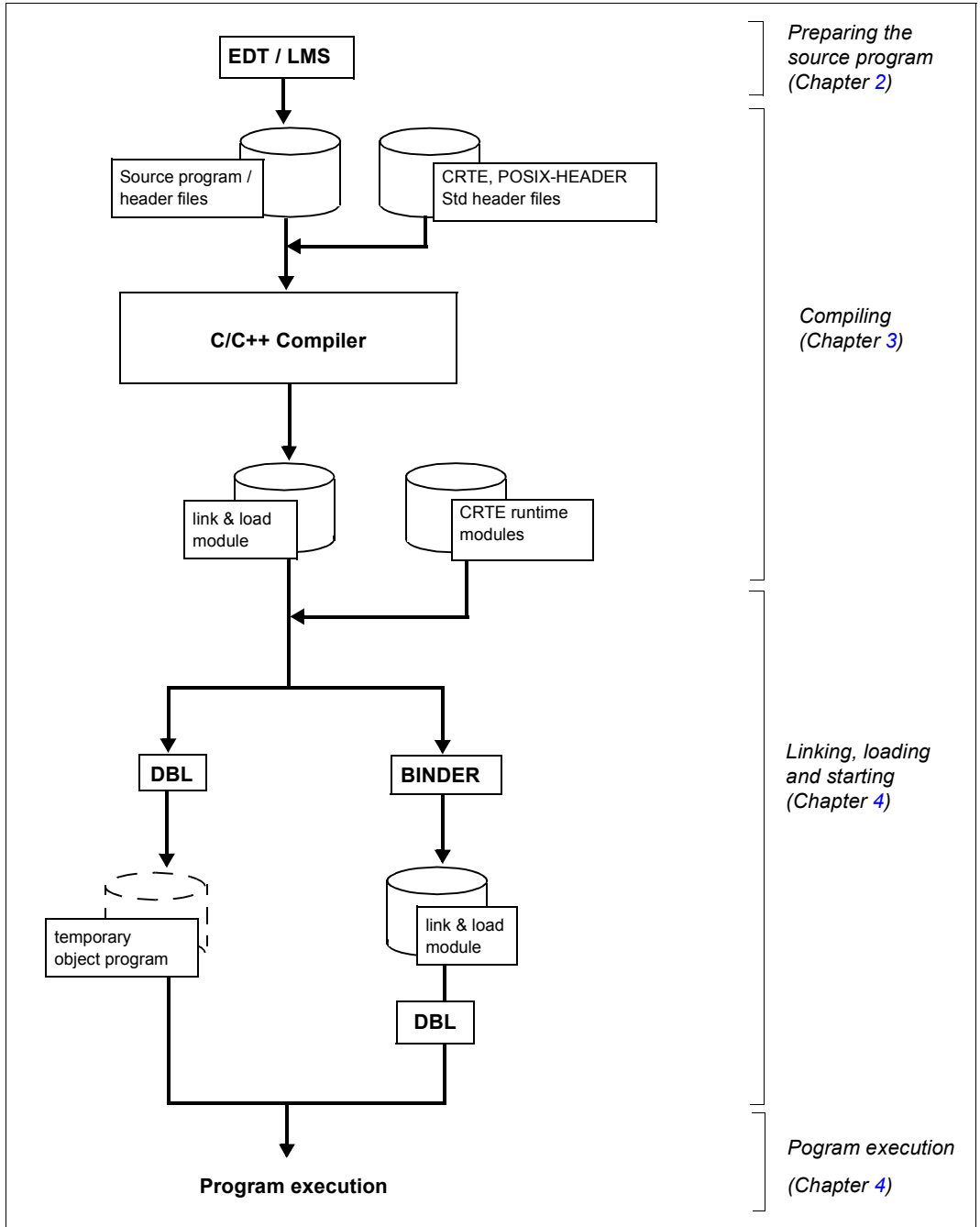
The source program must be compiled into machine language. The compiler generates all modules only in LLM format and optionally saves them in PLAM library elements (of type L) or in POSIX object files.

3. Linkage

The modules generated during compilation are linked with modules from the C runtime system and the C++ runtime system (if C++ library functions are used) to create an executable unit.

Modules created in the ANSI C++ modes should not be linked with a direct call to `BINDER`, but only with the compiler statements `MODIFY-BIND-PROPERTIES` and `BIND`.

The following overview illustrates these basic steps and indicates where other relevant information can be found in this manual.



2.2 General requirements for compilation, linkage and program execution

In order to compile, link and execute a program successfully, you will need the following components:

- Calling programs of the C/C++ compiler and global listing generator
- SDF C/C++ syntax file
- DMS/PLAM/BINDER message files
- C/C++ message file

In addition, you will also need the Common Runtime Environment CRTE V2.6, which is supplied with the components below:

- C and C++ runtime modules
- ILCS runtime modules
- Standard headers for C library functions (excluding the POSIX functions) and for C++ library functions

The preparatory steps required to enable SDF control and the message files are described in the release notice for C/C++ V3.2D.

Notes on the installation and operation of CRTE can be found in the release notice for CRTE and in the CRTE User Guide [4].

In principle, the C/C++ compiler can be used with any BS2000/OSD version as of V6.0. Note, however, that many of its features described in this manual are based on higher operating system versions and may also require some additional software products:

- Compiler I/O in the POSIX file system and the use of POSIX library functions: POSIX-HEADER as of V1.6.

For more details on software requirements, see also the release notice for C/C++ V3.2D.

2.3 General features of the C/C++ compiler

Supported C and C++ language standards

The following C and C++ standards are supported by the C/C++ compiler:

- ANSI/ISO C with the ISO C Addendum 1 (1994)
- the status of the ANSI C++ draft from early 1996 including, in particular, exception handling, templates, new-style casts, namespaces, run-time type information (RTTI), etc.

Compatibility

The following language modes are offered by the compiler to enable the migration or porting of older C and C++ applications: Kernighan&Ritchie C, Cfront C++ V3.0.3, and the preprocessor dialect based on Reiser's cpp and Johnson's pcc.

Portable software

To develop portable software, the compiler also supports the "strict" ANSI C and ANSI C++ modes. All deviations from the corresponding language standards are diagnosed in these modes.

2.4 Specific CRTE components required for C/C++

This section provides you with a quick overview of the CRTE “containers” that are relevant for programming in C++ and C. More explicit details on the functionality, scope and use of the C++ libraries provided with the CRTE are presented in the [chapter “The C++ libraries and C++ runtime system” on page 275](#).

Detailed information on the general concept of CRTE can be found in the CRTE User Guide [4].

See also the various reference manuals and publications for C and C++ library functions listed in the References section.

2.4.1 Include libraries

The SYSLIB.CRTE library

This library contains:

- Standard headers (type S) for C library functions (ANSI C and BS2000-specific extensions)
- Standard headers (type S) for classes and functions of the standard C++ library based on the ANSI C++ draft
- Standard headers (type S) for classes and functions of the Tools.h++ library

This library does not include:

- Standard headers for the use of POSIX library functions. Following the installation of POSIX-HEADER (release unit of BS2000/OSD-BC), these headers are made available in the SYSLIB.POSIX-HEADER library.
- Standard headers of the Cfront-compatible C++ library functions for complex math and stream-oriented I/O. These headers are located in the SYSLIB.CRTE.CPP library.

The SYSLIB.CRTE.CPP library

This library contains the standard headers for classes and functions of the Cfront-compatible C++ library for complex math and stream-oriented I/O. These headers are used by the compiler in the Cfront C++ mode.

See also the [section “The Cfront C++ library” on page 277](#).

2.4.2 Module libraries

The SYSLNK.CRTE library

This library contains:

- Individual modules of the C runtime system (object modules of type R)

These modules can be linked statically or dynamically. The C runtime system can also be linked dynamically, provided the SYSLNK.CRTE.PARTIAL-BIND library (for the standard partial bind linking method) or the SYSLNK.CRTE.COMPL library (for the complete partial bind linking method) is specified instead of the SYSLNK.CRTE library when linking with BINDER.

The C runtime system is always required in order to run C/C++ programs. Among other things, it contains the code for all the C library functions, the central I/O routines for the implementation of C++ I/O functions, and additional routines for the implementation of operating system interfaces.

The entry names of the C runtime system begin with “IC@”, “ICS” or “ICX”.

- Name adapter modules for new C library functions (object modules of type R and LLMs of type L)

Adapter modules belong to the non-preloadable components of the C runtime system and must consequently be linked into the application program. They are contained both in the SYSLNK.CRTE library and in the SYSLNK.CRTE.PARTIAL-BIND library.

Detailed technical information on these adapter modules can be found in the [section “Concept of a name adapter module in the C runtime system” on page 304](#).

- The C runtime system as a dynamically loadable prelinked module (LLM, L type).

This prelinked module is dynamically loaded into the user address space at runtime if the library SYSLNK.CRTE.PARTIAL-BIND is specified instead of the SYSLNK.CRTE library and if the dynamically loadable C runtime system has not been preloaded.

The SYSLNK.CRTE.PARTIAL-BIND and SYSLNK.CRTE.COMPL libraries

These libraries allow C programs to be linked without unresolved external references and enable the C runtime system to be loaded dynamically only at the time of execution. Connection modules, which are linked instead of the C runtime module and satisfy all unresolved external references of a C program to the C runtime system, are included in the libraries for this purpose. These libraries also contain all the other modules that are required in order to link a C program without any unresolved external references to the CRTE (e.g. ILCS modules and name adapter modules).

SYSLNK.CRTE.PARTIAL-BIND is needed for the standard partial bind linking method while SYSLNK.CRTE.COMPL is used for the complete partial bind linking method. For details concerning the partial bind linking method as well as for differences between standard partial bind and complete partial bind see the CRTE user guide [4].

The C runtime system itself is available in the form of a dynamically loadable prelinked module in the library SYSLNK.CRT and is dynamically loaded into the user address space at runtime if the program cannot be connected to the preloaded C runtime system.

Since only the connection modules are linked and not the entire C runtime system, the finished (i.e. fully-linked) program or module is significantly smaller in size and requires much less disk storage space than when statically linking C runtime modules from the library SYSLNK.CRTE. Furthermore, this also results in faster load times if the C runtime system is preloaded.

See also the CRTE user guide [4] for more technical information.

The SYSLNK.CRTE.CPP and SYSLNK.CRTE.CFCPP libraries

These libraries contain the modules of the Cfront-compatible C++ runtime system (i.e. LLMs of type L).

SYSLNK.CRTE.CPP contains the modules of all C++ library functions for complex math and standard I/O that are available in the Cfront C++ mode of the compiler. The entry names begin with "ICP".

SYSLNK.CRTE.CFCPP contains the modules for the internal runtime routines that are needed in the Cfront C++ mode of the compiler to implement initialization, storage management, etc. The entry names begin with "IPP".

The modules can be linked statically or dynamically with BINDER or DBL, respectively. Note that modules from the SYSLNK.CRTE.CFCPP library must be linked with precedence before the modules from the SYSLNK.CRTE.CPP library.

See also the [section "The Cfront C++ library" on page 277](#).

The SYSLNK.CRTE.POSIX library

This library contains a “linkage option” module, which must be linked whenever the POSIX functions of the C runtime system are to be used. This module must always be linked with precedence before the modules in the library SYSLNK.CRTE, SYSLNK.CRTE.PARTIAL-BIND or SYSLNK.CRTE.COMPL. When linking with BINDER, this library should preferably be linked by means of an INCLUDE statement (without specifying the module name), since the use of RESOLVE statements, by contrast, would require the linkage order to be strictly observed. This also applies when linking with the BIND statement of the compiler, which means that the INCLUDE option should be used in the MODIFY-BIND-PROPERTIES statement.

The SYSLNK.CRTE.STDCPP and SYSLNK.CRTE.RTSCPP libraries

These libraries contain the modules of the ANSI-C++ runtime system (LLMs, type L).

SYSLNK.CRTE.STDCPP contains the modules for the standard C++ library, which can be used in the ANSI C++ modes of the compiler.

SYSLNK.CRTE.RTSCPP contains the modules for the internal runtime routines that are needed in the ANSI C++ modes of the compiler to implement, among other things, the C++ exception handling and the C++ runtime type information (RTTI).

The modules can be linked statically with the BIND statement of the compiler or dynamically with DBL.

See also the [section “The standard C++ library” on page 275](#).

The SYSLNK.CRTE.TOOLS library

This library contains the modules (LLMs, type L) for the Tools.h++ library, which can be used in the ANSI C++ modes of the compiler.

The modules can be linked statically with the BIND statement of the compiler or dynamically with DBL.

See also the [section “The Tools.h++ library” on page 279](#).

The SYSLNK.CRTE.CPP-COMPL library

This library contains adapters for the SYSLNK.CRTE.STDCPP, SYSLNK.CRTE.RTSCPP and SYSLNK.CRTE.TOOLS libraries.

2.5 Editing source programs

Storage methods for source programs and header files

The C/C++ compiler processes source programs and header files which are stored as follows:

Source programs stored as

- cataloged SAM and ISAM files (with KEYPOS=5 and KEYLEN ≤ 16)
- PLAM library elements of type S
- POSIX source files

Header files stored as

- cataloged files if the source program is itself a cataloged file
- PLAM library elements of type S
- POSIX source files

The file editor EDT

The file editor EDT is available in BS2000 for editing a C/C++ source program. EDT can process cataloged SAM/ISAM files, PLAM library elements and POSIX files.

EDT converts lowercase letters to uppercase letters by default. Since source program texts in C/C++ usually contain a large number of lowercase letters, automatic conversion to uppercase must be prevented by issuing the command LOWER ON after EDT is called.

The following table provides an overview of the most important EDT statements for processing files and library elements.

The file editor EDT is described in detail in the manual “EDT (BS2000/OSD)” [15].

SAM files	
@READ'file'	Read contents of a SAM file into the current work file.
@WRITE'file'	Write contents of the current work file to a SAM file
ISAM files	
@GET'file'	Read contents of an ISAM file into the current work file
@SAVE'file'	Write contents of the current work file to an ISAM file
@OPEN'file'	Physically open ISAM file
@CLOSE	Close an ISAM file opened with @OPEN
PLAM library elements	
@OPEN LIB=lib(ELEM=elem)	Open library element in the current work files
@COPY LIB=lib(ELEM=elem)	Read contents of a library element into the current work file
@WRITE LIB=lib(ELEM=elem)	Write contents of the current work file to a library element
@CLOSE	Close a library element opened with @OPEN
POSIX files	
@XOPEN FILE=pathname,MODE=ANY/UPDATE/NEW/REPLACE	Open and read an existing POSIX file or create a new POSIX file
@CLOSE	Close a POSIX file opened with @XOPEN
@XCOPY FILE=pathname	Read contents of a POSIX file into the current work file
@XWRITE FILE=pathname,MODE=ANY/UPDATE/NEW/REPLACE	Write contents of a current work file to a POSIX file

EDT statements for processing files and library elements (extract)

By default, PLAM library elements are stored by EDT as elements of type S with the highest version number (X'FF').

Key assignments

In order to program in C and C++, the user requires a number of special characters, some of which may not be available on all keyboards. For example, keyboards with the German character set have “umlauts” assigned instead of some characters specifically required for C and C++.

The following table lists the different key assignments and their (international) hexadecimal codes. This information can be used to create a user-specific keyboard configuration if required.

Identifier / Functions in C and C++	“English” keyboard	“German” keyboard	Hexadecimal code
left angle bracket	<	<	4C
right angle bracket	>	>	6E
logical OR		ö	4F
logical exclusive OR	^	^	6A
underscore	_	_	6D
left square bracket	[Ä	BB
right square bracket]	Ü	BD
backslash	\	Ö	BC
left brace	{	ä	FB
right brace	}	ü	FD
bit complement	~	ß	FF

2.6 POSIX support

2.6.1 Compiler I/O in the POSIX file system

The C/C++ compiler supports the BS2000 file system (DMS files, PLAM libraries, etc.) as well as the POSIX file system. All inputs and outputs of the compiler in a compilation, preprocessor or syntax analysis

run can also be effected via POSIX files. This includes, in particular:

- the input of source programs (see the SOURCE option in the COMPILE, PREPROCESS and CHECK-SYNTAX statements)
- the input of header files (see the USER-INCLUDE-LIBRARY and STD-INCLUDE-LIBRARY options in the MODIFY-INCLUDE-LIBRARY statement)
- the output of LLMs (see the MODULE-OUTPUT option in the COMPILE statement)
- the output of recompilable source programs (see the OUTPUT option in the PREPROCESS statement)
- the output of compiler listings (see the OUTPUT option in the MODIFY-LISTING-PROPERTIES statement)
- the output of message lists (see the OUTPUT option in the MODIFY-DIAGNOSTIC-PROPERTIES statement)
- the output of CIF information (see the OUTPUT option in the MODIFY-CIF-PROPERTIES statement)

Arbitrary combinations, i.e. the input and output of both BS2000 as well as POSIX files in the same compilation run, are also possible.

Storage of source programs and header files

Source programs and header files may exist in EBCDIC and ASCII codes. EBCDIC is the default in the POSIX file system; ASCII is the default for file systems on remote UNIX computers or PCs. All files of a file system (POSIX file system or attached remote file systems) must, however, be in the same respective (default) codeset. The compiler checks the codeset of a file system centrally, not for each individual file. Files of an ASCII file system are internally converted to EBCDIC for the compilation.

In contrast to program development in the POSIX shell (see the manual “POSIX commands of the C/C++ Compiler [1]), the file names of source programs need not always contain one of the standard suffixes “.c”, “.C” or “.i”.

Output of LLMs

The LLMs generated by the compiler can be written to POSIX object files (“`.o`” files). These LLM object files can only be processed meaningfully in the POSIX subsystem (by using the command `cc`, `c89` or `CC`; see the manual “POSIX commands of the C/C++ Compiler [1]). The generated object file format (LLM) is not supported on UNIX systems.

Other compiler outputs

- Expanded recompilable source programs

The result of a preprocessor run can be written to a POSIX source file (“`.i`” file in C, “`.I`” file in C++). This file can be processed further in the POSIX shell with the command `cc`, `c89` or `CC` and in the BS2000 environment via the SDF interface of the C/C++ compiler.

- Compiler listings

Compiler listings can be written to a POSIX list file (“`.lst`” file). This list file can be printed in the POSIX shell with the command `bs21p` and in the BS2000 environment with the SDF command `PRINT-DOCUMENT`.

- Diagnostic messages

- Error messages of the compiler can be written to a POSIX file (“`.diag`” file). This file can be read with a file editor such as EDT in the POSIX shell (`edt` command) and in the BS2000 environment (see note on EDT on [page 29](#)).

- CIF information

The local CIF information for individual modules that is used to create global listings can be stored in a POSIX file (“`.cif`” file). These CIF files can be processed further in the POSIX shell with the command `cclistgen` (see the manual “POSIX Commands of the C/C++ Compiler [1]”) and via the SDF interface of the global listing generator in the BS2000 environment (see `START-CPLUS-LISTING-GENERATOR`, [page 156ff](#)).

Output codeset

The output codeset of the files (EBCDIC or ASCII) is determined by the codeset of the target file system. The compiler stores files in EBCDIC code in the POSIX file system and in ASCII code in file systems on UNIX computers.

Only EBCDIC execution code is generated. For example, strings and string constants are only stored in EBCDIC code.

Description of the term <posix-pathname>

When <posix-pathname> is specified as an input or output, the following can be specified:

- a file name with no path specification
- a relative path to a file
- an absolute path to a file
- a relative path to a directory
- an absolute path to a directory

If no absolute path exists, the name/path is interpreted relative to the user's home directory.

All directories in the path must already exist.

When an input file is requested, no directory may be specified.

When an input directory is requested, no file may be specified.

It only makes sense to specify an output file when a single source is processed. When multiple sources are processed, it is generally not permissible to specify an output file.

When the file name is created, it must be borne in mind that meaningful further processing of the output file in the POSIX subsystem is possible only if the name has a suitable suffix.

When a directory is specified, the compiler itself determines the name of the output file (see the [section "Default names for output containers" on page 48](#)).

2.6.2 Use of POSIX library functions

CRTE V2.0 provides a C runtime system that supports C library functions with BS2000 functionality as well as POSIX functionality.

The library functions with BS2000 functionality include all the functions and macros that were also offered earlier with the C runtime system, i.e. all ANSI-defined functions and about fifty BS2000-specific extensions.

Only these functions may be used in the following cases:

- if no POSIX subsystem is available in BS2000/OSD V6.0, or
- if no preparatory steps are taken (see below) at compile and link time in a BS2000 operating system with an available POSIX subsystem.

The library functions with BS2000 functionality are described in the manual “C Library Functions” [2].

The following functions of the C runtime system, which were introduced for the first time with CRTE V2.0, are library functions with POSIX functionality: all functions required by the XPG4 standard and about thirty UNIX-specific extensions. For more information on these functions and all functions with BS2000 functionality, see the manual “C Library Functions for POSIX Applications” [3].

Compilation and linkage of programs that use POSIX library functions

The following steps are required in order to use library functions with POSIX functionality when developing programs in the BS2000 environment (SDF):

1. The library SYSLIB.POSIX-HEADER, which contains the standard headers for POSIX functions, must be specified in addition to the CRTE library SYSLIB.CRTE in the search for standard headers at compilation.

```
//MOD-INCLUDE-LIB STD-INCLUDE-LIB=( *STANDARD-LIBRARY , &(INSTALLATION-PATH
( 'SYSLIB' , 'POSIX-HEADER' , DEFAULT='$.SYSLIB.POSIX-HEADER' )))
```

2. The `_OSD_POSIX` directive must always be set before the preprocessor encounters the first `#include` directive in the program. This can be ensured by a global setting for the entire compilation run with the `SOURCE-PROPERTIES` statement instead of a definition in the source program with the `#define` directive.

```
//MODIFY-SOURCE-PROPERTIES DEFINE=_OSD_POSIX
```

3. When linking, the link option library SYSLNK.CRTE.POSIX must be linked with precedence before the library SYSLNK.CRTE, SYSLNK.CRTE.PARTIAL-BIND or SYSLNK.CRTE.COMPL. When linking with BINDER, it is advisable to use an INCLUDE statement for the link option library (without specifying the module name), since the use of RESOLVE statements, by contrast, would require the appropriate order to be strictly observed. In the case of the BINDER statement, for example:

```
//INCLUDE-MODULES *LIB(LIB=$.SYSLNK.CRTE.POSIX,ELEM=*ALL)
```

This also applies when linking with the BIND statement of the compiler, i.e. the INCLUDE option should be used in the MODIFY-BIND-PROPERTIES statement:

```
//MOD-BIND-PROP INCLUDE=*LIB-ELEM(LIB=&(INSTALLATION-PATH('SYSLNK.POSIX',  
'CRTE',DEFAULT='$.SYSLNK.CRTE.POSIX')),ELEM=*ALL)
```

When developing programs in the POSIX environment, by contrast, no special preparatory steps are required in order to use the POSIX library functions (see also the manual “POSIX Commands of the C/C++ Compiler” [1]).

2.7 Introductory examples

Using three simple examples, this section demonstrates how C/C++ programs are compiled, linked, and executed in BS2000. The examples are based on the assumption that the C/C++ compiler and the C and C++ runtime systems were installed by default under the user ID TSOS.

Example 1: Compiling, linking, and starting a C program

The C source program is contained in the cataloged file HELLO. The compiler generates an LLM and places it in the library PLAM.TEST. This module is then linked by different methods:

Variant 1: with the BIND statement of the compiler

Variant 2: with BINDER

Variant 3: with DBL

Source program file HELLO

```
#include <stdio.h>
int main(void)
{
    printf("Hello, I am a C program\n");
    return 0;
}
```

The source program was written using EDT and saved in a file named HELLO.

Runtime listing for compilation, linkage and execution

```
(IN) indicates user inputs
(OUT) indicates system program messages

(IN) /START-CPLUS-COMPILER _____ (1)
(OUT) % BLS0523 ELEMENT 'SDFCC', VERSION '032',TYPE 'L'FROM LIBRARY
      ' :P401:$TSOS.SYSLNK.CPP.032' IN PROCESS
(OUT) % BLS0524 LLM 'SDFCC', VERSION '03.2D00' OF '2011-10-25 13:40:11'
      LOADED
(OUT) % BLS0551 COPYRIGHT (C) Fujitsu Technology Solutions 2011. ALL
      RIGHTS RESERVED
(OUT) % CDR9992 : BEGIN C/C++(BS2000/OSD) VERSION 03.2D00
(IN) //MODIFY-SOURCE-PROP LANGUAGE=*C _____ (2)
(IN) //COMPILE SOURCE=HELLO,MODULE-OUTPUT=*LIB-ELEM(LIB=PLAM.TEST) — (3)
(OUT) % CDR9907 : NOTES: 0 WARNINGS: 0 ERRORS: 0 FATALS: 0
(OUT) % CDR9937 : MODULES GENERATED, CPU TIME USED = 0.0030 SEC
```

Variant 1: Linking with the BIND statement

```
(IN) //MOD-BIND-PROP INCLUDE=*LIB(LIB=PLAM.TEST,ELEM=HELLO),- (4)
      //RUNTIME-LANG=*C,STDLIB=*STATIC _____
(IN) //BIND OUTPUT=*LIB(LIB=PLAM.TEST1,ELEM=HELLO) _____ (5)
(OUT) % BND3102 SOME WEAK EXTERNS UNRESOLVED
(OUT) % BND1501 LLM FORMAT: '1'
(OUT) % BND1101 BINDER NORMALLY TERMINATED. SEVERITY CLASS:
      'UNRESOLVED EXTERNAL'
(IN) //END _____ (6)
(OUT) % CDR9992 : END
(OUT) % CCM0998 CPU TIME USED: 1.9967 SECONDS
```

Variant 2: Linking with BINDER

```
(IN) /START-BINDER _____ (7)
(OUT) % BND0500 BINDER VERSION 'V02.6A30' STARTED
(IN) //START-LLM-CREATION INT-NAME=XY
(IN) //INCLUDE-MODULES *LIB(LIB=PLAM.TEST,ELEM=HELLO)
(IN) //RESOLVE-BY-AUTOLINK LIB=$.SYSLNK.CRTE
(IN) //SAVE-LLM LIB=PLAM.TEST1,ELEM=HELLO,MAP=*NO
(OUT) % BND3102 SOME WEAK EXTERNS UNRESOLVED
(OUT) % BND1501 LLM FORMAT: '1'
(IN) //END
(OUT) % BND1101 BINDER NORMALLY TERMINATED. SEVERITY CLASS:
      'UNRESOLVED EXTERNAL'
```

Variant 3: Linking, loading and starting with DBL

```
(IN) /ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME=$.SYSLNK.CRTE _____ (8)
(IN) /START-EXECUTABLE-PROGRAM FROM-FILE=*LIB-ELEM(LIBRARY=PLAM.TEST,-
      /ELEMENT-OR-SYMBOL=HALLO),DBL-PARAMETERS=*PARAM(-
      /RESOLUTION=*PARAM(ALTERNATE-LIBRARIES=*BLSLIB##))
(OUT) % BLS0524 LLM 'HALLO', VERSION ' ' OF '2011-11-07 13:33:56' LOADED
(OUT) Hello, I am a C program
(OUT) % CCM0998 CPU TIME USED: 0.0021 SECONDS
```

Starting programs linked with variants 1 and 2

```
(IN) /START-EXECUTABLE-PROGRAM FROM-FILE=*LIB-ELEM(LIBRARY=PLAM.TEST1,-
      /ELEMENT-OR-SYMBOL=HALLO) _____ (9)
(OUT) % BLS0524 LLM 'XY', VERSION ' ' OF '2011-11-07 13:33:50' LOADED
(OUT) Hello, I am a C program
(OUT) % CCM0998 CPU TIME USED: 0.0028 SECONDS
```

- (1) The compiler is started.
- (2) The MODIFY-SOURCE-PROPERTIES statement is used to set the ANSI C mode (the default setting is the ANSI C++ mode).
- (3) The COMPILE statement is issued with selected options to begin compilation. The SOURCE option specifies the name of the source program to be compiled, and the MODULE-OUTPUT option specifies a PLAM library as the output destination. The name of the object module to be generated is derived from that of the source program (HELLO).
- (4) The MODIFY-BIND-PROPERTIES statement specifies the modules to be linked and other conditions for the subsequent linkage run with the BIND statement of the compiler. The INCLUDE option (which corresponds to the BINDER statement INCLUDE-MODULES) specifies the LLM HELLO, which was generated earlier and placed in the library PLAM.TEST. The RUNTIME-LANGUAGE option specifies that the program to be linked is a C program (ANSI C++ is the default). Due to this option, all additional modules of the C runtime system which are required for C programs are automatically linked with the autolink mechanism. The *STATIC entry in the STDLIB option instructs the compiler to load the C runtime system from the library \$.SYSLNK.CRTE instead of the library \$.SYSLNK.CRTE.PARTIAL-BIND (the default).
- (5) The BIND statement is issued to start the linkage run. The OUTPUT option (which corresponds to the BINDER statement SAVE-LLM) causes the fully-linked LLM to be stored under the name HELLO as an element of type L in the PLAM library PLAM.TEST1. The BINDER message "SOME WEAK EXTERNS UNRESOLVED" refers to the ILCS module IT0INITS. This module contains WEAK-EXTERN references to all languages potentially required for ILCS. In this example, only the C language is involved, so the other references remain unresolved
- (6) The END statement terminates the compiler run.
- (7) The HELLO module that was generated at compilation in the PLAM library PLAM.TEST is linked with BINDER.
- (8) The HELLO module, which was generated at compilation and placed in the PLAM library PLAM.TEST, is dynamically linked, loaded and started with DBL.
- (9) The finished program HELLO, which was fully linked with the BIND statement of the compiler (see variant 1) and with BINDER (see variant 2) and placed in the library PLAM.TEST1, is loaded and started. The entry ALT-LIB=*YES (see variant 3), which is required for dynamic linking with DBL, is not needed in this case.

Example 2: Compiling, linking, and starting a C++ program (ANSI C++)

The C++ source program consists of two PLAM library elements PROG1 and PROG2. The compiler generates LLMs and writes them to the PLAM library SYS.PROG.LIB. These modules are then linked with the BIND statement of the compiler.

Source program element PROG1

```
#include <iostream.h>
extern void invoke(void);

int main(void)
{
    cout << "main(prog1)" << '\n';
    invoke();
    return 0;
}
```

Source program element PROG2

```
#include <iostream.h>
void invoke(void)
{
    cout << "invoke(prog2)" << '\n';
}
```

The source program extracts shown above were produced using EDT and then written separately to the PLAM library PLAM.EXP with explicit WRITE statements of the form: WRITE L=PLAM.EXP(E=element-name)

Runtime listing for compilation, linkage and execution

```

(IN)  indicates user inputs
(OUT) indicates system program messages

(IN)  /START-CPLUS-COMPILER _____ (1)
(OUT) % BLS0523 ELEMENT 'SDFCC', VERSION '032',TYPE 'L'FROM LIBRARY
      ' :P401:$TSOS.SYSLNK.CPP.032' IN PROCESS
(OUT) % BLS0524 LLM 'SDFCC', VERSION '03.2D00' OF '2011-10-25 13:40:11'
      LOADED
(OUT) % BLS0551 COPYRIGHT (C) Fujitsu Technology Solutions 2011. ALL
      RIGHTS RESERVED
(OUT) % CDR9992 : BEGIN C/C++(BS2000/OSD) VERSION 03.2D00
(IN)  //COMPILE SOURCE=(*LIB(LIB=PLAM.EXP,ELEM=PROG1),-
      // *LIB(LIB=PLAM.EXP,ELEM=PROG2)) _____ (2)
(OUT) % CDR9907 : NOTES: 0 WARNINGS: 0 ERRORS: 0 FATALS: 0
(OUT) % CDR9937 : MODULES GENERATED, CPU TIME USED = 0.0030 SEC
(OUT) % CDR9907 : NOTES: 0 WARNINGS: 0 ERRORS: 0 FATALS: 0
(OUT) % CDR9937 : MODULES GENERATED, CPU TIME USED = 0.0020 SEC
(IN)  //MOD-BIND-PROP INCLUDE=(*LIB(LIB=SYS.PROG.LIB,ELEM=PROG1),-
      // *LIB(LIB=SYS.PROG.LIB,ELEM=PROG2)) _____ (3)
(IN)  //BIND OUTPUT=*LIB(LIB=PLAM.EXP1,ELEM=PROG) _____ (4)
(OUT) % BND1501 LLM FORMAT: '1'
(OUT) % BND3102 SOME WEAK EXTERNS UNRESOLVED
(OUT) % BND1101 BINDER NORMALLY TERMINATED. SEVERITY CLASS:
      'UNRESOLVED EXTERNAL'
(IN)  //END _____ (5)
(OUT) % CDR9992 : END
(OUT) % CCM0998 CPU TIME USED: 1.9967 SECONDS
(IN)  /START-EXECUTABLE-PROG*L(LIB=PLAM.TEST1,E-0-S=PROG) _____ (6)
(OUT) % BLS0523 ELEMENT 'PROG', VERSION '@' FROM LIBRARY
      ':20S2:$USERIDXY.PLAM.EXP1' IN PROCESS
(OUT) % BLS0524 LLM '$LIB-ELEM$PLAM$BSP1$$PROG$$SUPPE', VERSION ' '
      OF '2011-11-07 14:22:01' LOADED
(OUT) main(prog1)
(OUT) ruf(prog2)
(OUT) % CCM0998 CPU TIME USED: 0.0018 SECONDS

```

- (1) The compiler run is started.
- (2) The COMPILE statement is issued to start compiling the C++ source program elements PROG1 and PROG2. The ANSI C++ language mode is set by default. The name off the PLAM library and that of the library element to be compiled are specified individually in the SOURCE option in the form of a list. By default, the compiler places the generated LLMs in the library SYS.PROG.LIB. The names of the LLMs are derived from the names of the source program elements (PROG1, PROG2).

- (3) The MODIFY-BIND-PROPERTIES statement specifies which modules are to be linked in the following linkage run. The INCLUDE option (which corresponds to the BINDER statement INCLUDE-MODULES) specifies the names of the LLMs to be linked, i.e. the LLM containing the `main` function (PROG1) and the additional LLM with the subroutine (PROG2). All other C and C++ runtime library modules required for ANSI C++ programs are linked in automatically (with the autolink mechanism, see [page 87](#)).
- (4) The linkage run is started with BIND statement. The OUTPUT option (which corresponds to the BINDER statement SAVE-LLM) causes the generated LLM to be saved under the name PROG as an element of type L in a PLAM library. The BINDER message "SOME WEAK EXTERNS UNRESOLVED" refers to the ILCS module IT0INITS. This module contains WEAK-EXTERN references to all potential languages for ILCS. Only the C and C++ languages are involved in this example, so the other references remain unresolved. For more information on the LLM format generated by BINDER (Format 1 in this example), please refer to the OUTPUT-FORMAT option of the BIND statement ([page 68](#)).
- (5) The compiler run is terminated with the END statement.
- (6) The fully-linked program is loaded and started with the START-EXECUTABLE-PROGRAM command.

Example 3: Compiling a C source program that is located in a POSIX file and uses POSIX library functions

The C source program exists as a POSIX source file with the name `hello.c` in the directory `/USERIDXY/source`. The program uses POSIX library functions. The compiler generates an LLM and writes it to a POSIX object file with the default name `hello.o`. This object file is stored in the directory of the source program. The object file is then processed further in the POSIX shell environment.

Source program file `hello.c`

```
#include <stdio.h>
FILE *fp;
int main(void)
{
    printf("Hello, I am a C program\n");
    fp = fopen("/USERIDXY/posixfiles/hello", "w");
    fputs("hello", fp);
    fclose(fp);
    return 0;
}
```

The source program was created with EDT and saved with the statement `@XWRITE FILE=/USERIDXY/source/hello.c`.

Runtime listing for compilation, linkage and execution

```
(IN) indicates user inputs
(OUT) indicates system program messages

(IN) /START-CPLUS-COMPILER _____ (1)
(OUT) % BLS0523 ELEMENT 'SDFCC', VERSION '032',TYPE 'L'FROM LIBRARY
      ' :P401:$TSOS.SYSLNK.CPP.032' IN PROCESS
(OUT) % BLS0524 LLM 'SDFCC', VERSION '03.2D00' OF '2011-10-25 13:40:11'
      LOADED
(OUT) % BLS0551 COPYRIGHT (C) Fujitsu Technology Solutions 2011. ALL
      RIGHTS RESERVED
(OUT) % CDR9992 : BEGIN C/C++(BS2000/OSD) VERSION 03.2D00
(IN) //MOD-SOURCE-PROP LANG=*C,DEFINE=_OSD_POSIX _____ (2)
(IN) //MOD-INCL-LIB STD-INCL=($.SYSLIB.POSIX-HEADER,*STANDARD-LIB) — (3)
(IN) //COMPILE SOURCE='/USERIDXY/source/hello.c',-
      //MODULE-OUTPUT=*SOURCE-LOC _____ (4)
(OUT) % CDR9907 : NOTES: 0 WARNINGS: 0 ERRORS: 0 FATALS: 0
(OUT) % CDR9937 : MODULES GENERATED, CPU TIME USED: 0.3829 SECONDS
(IN) //END _____ (5)
(OUT) % CDR9992 : END
(OUT) % CCM0998 CPU TIME USED: 0.3829 SECONDS
```

```

(IN)  /START-POSIX-SHELL _____ (6)
(OUT) POSIX Basisshell 09.0A41 created Feb 14 2011
      POSIX Shell 07.0A41 created Jan 27 2009
      Copyright (C) Fujitsu Technology Solutions 2009
           All Rights reserved
      Last login: Thu Nov 3 10:45:44 2011 on term/002
(IN)  cd source _____ (7)
(IN)  ls hallo* _____ (8)
(OUT) hallo.c    hallo.o
(IN)  c89 hallo.o _____ (9)
(IN)  a.out _____ (10)
(OUT) Hallo, I am a C program

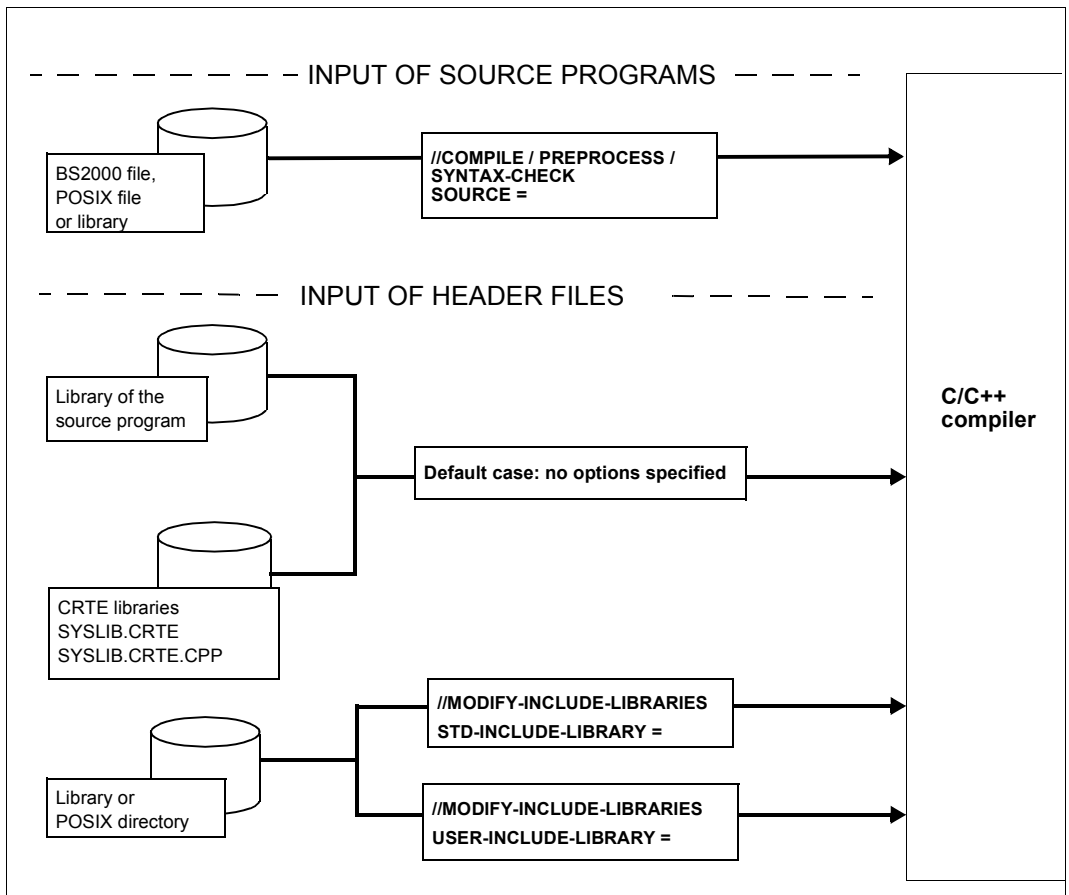
```

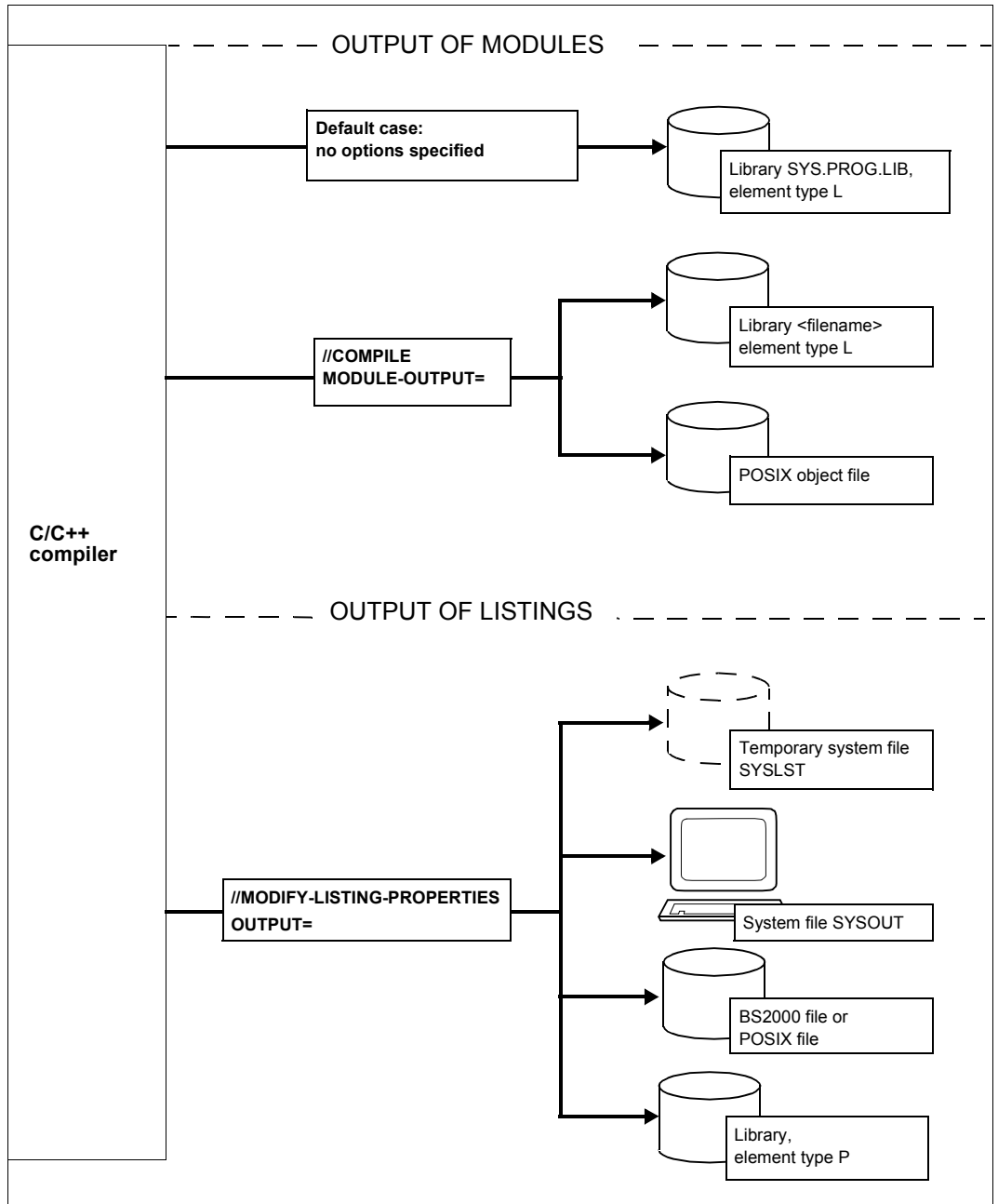
- (1) The compiler run is started.
- (2) The MODIFY-SOURCE-PROPERTIES statement activates the ANSI C language mode (the ANSI C++ mode is set by default) and sets the `_OSD_POSIX` directive required for the use of POSIX library functions.
- (3) The library containing the standard headers for POSIX library functions is assigned with the MODIFY-INCLUDE-LIBRARY statement in addition to the CRTE library `$.SYSLNK.CRTE (*STANDARD-LIBRARY)`.
- (4) The COMPILE statement is issued to start the compilation run. The absolute path name of the POSIX source file to be compiled is specified with the SOURCE option. POSIX file names must always be enclosed within single quotes. The operand value `*SOURCE-LOCATION` in the MODULE-OUTPUT option causes the compiled module to be written to a POSIX object file with the default name `hallo.o` and stored in the directory of the source program.
- (5) The compiler run is terminated with the END statement.
- (6) Since POSIX object files can only be processed further in the POSIX subsystem, the POSIX command START-POSIX-SHELL is used to switch from the BS2000 system environment (SDF) to the POSIX environment (shell). The call places the user in the home directory of the current BS2000 user ID (USERIDXY).
- (7) The POSIX command `cd` is used to change to the `source` directory in which the source program and the object file generated by the compiler are located.
- (8) On entering the POSIX command `ls`, the source file `hallo.c` and the object file `hallo.o` are listed.
- (9) The POSIX command `c89` links the object file `hallo.o` into an executable unit and places it in an executable POSIX file with the default name `a.out`. The `c89` command is described in detail in the manual "POSIX Commands of the C/ C++ Compiler" [1].
- (10) The program is executed.

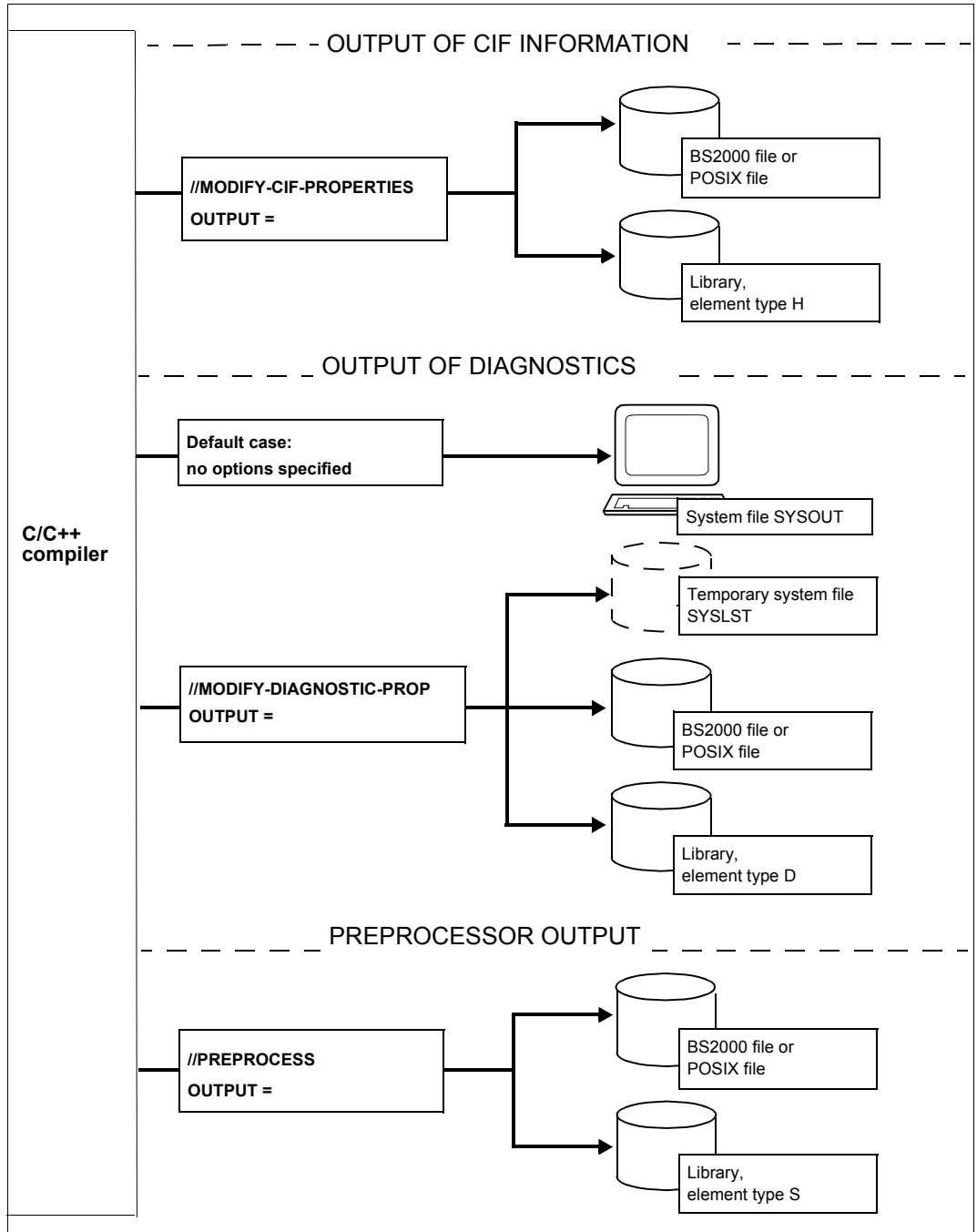
3 Compilation

3.1 General aspects of the compiler run

3.1.1 Input sources and output destinations of the compiler







3.1.2 Construction of default names

If no output files are explicitly specified for the compilation results, the compiler generates default names by deriving them from the respective source program name.

The [section “Default names for output containers”](#) below describes the rules by which the compiler constructs default names for output containers of the following compilation results: preprocessor output, messages, listings and CIF information.

The rules for constructing module names are described in the [section “Rules for constructing module names” on page 52](#)).

Default names for output containers

This section summarizes the rules by which the compiler constructs default names for output containers of the following compilation results:

- Result of a preprocessor run (PREPROCESS)
- Compiler listings (MODIFY-LISTING-PROPERTIES)
- Diagnostic messages (MODIFY-DIAGNOSTIC-PROPERTIES)
- CIF information (MODIFY-CIF-PROPERTIES)

Default names are generated, i.e. derived from the source program name, when the following entries are made in the OUTPUT options listed above:

OUTPUT=*STD-FILE

OUTPUT=*SOURCE-LOCATION

OUTPUT=*LIB-ELEM(LIB=...,ELEM=*STD-ELEMENT)

OUTPUT=<posix-pathname> (name of a POSIX file directory)

Construction of default names for cataloged BS2000 files

The output is placed in cataloged BS2000 files with default names

- whenever OUTPUT=*STD-FILE is specified;
- if OUTPUT=*SOURCE-LOCATION is specified and the source program is read from a BS2000 file or SYSDTA.

1. The following source program name components, if present, are not used to construct the default file name and are therefore dropped:
 - name components for the catid and userid in BS2000 file and library names.
Exception: If the source program entered with OUTPUT=*SOURCE-LOCATION is catalogued as a BS2000 file, then catid and userid will be kept.
 - directory names in POSIX pathnames
 - the suffixes `.C`, `.CPP`, `.CXX`, `.CC` and `.I` if the source program exists as a cataloged BS2000 file or PLAM library element
 - the suffixes `.c`, `.C`, `.cpp`, `.CPP`, `.cxx`, `.CXX`, `.cc`, `.CC`, `.c++`, `.C++`, `.i` and `.I` if the source program exists as a POSIX file
2. If the source program exists as a cataloged BS2000 file or a POSIX file, the remaining portion of the name is truncated from the right to 33 characters.
3. For OUTPUT=*STD-FILE only:

If the source program exists as a PLAM library element, the library and element names are combined with a hyphen and used in the default name as shown below:
libraryname-elementname

If this name (including the hyphen) exceeds 33 characters, the library name and, if required, also the element name, are both truncated to 16 characters from the right, starting with the library name.
4. Special characters not allowed in file names are always converted to "\$". The permitted special characters are \$, @, #, . (period) and - (hyphen).
5. Lowercase letters (in POSIX source file names) are converted to uppercase.
6. The name truncated to 33 characters is then extended with the appropriate suffix, i.e.:
`.I`, `.LST`, `.CIF` or `.DIAG`.

Summary

Contents	Source program input from			
	*SYSDTA	BS2000 file	PLAM library	POSIX file
PREPROCESS	CSTDEXP.I	file.I	lib-elem.I	file.I
LISTING	CSTDLST.LST	file.LST	lib-elem.LST	file.LST
DIAGNOSTIC	CSTDDIAG.DIAG	file.DIAG	lib-elem.DIAG	file.DIAG
CIF	CSTDCIF.CIF	file.CIF	lib-elem.CIF	file.CIF

Default names for cataloged BS2000 files, depending on input source and contents

Construction of default names for PLAM library elements

The output is placed in PLAM library elements with default names

- if OUTPUT=*LIB-ELEM(ELEMENT=*STD-ELEMENT) is specified;
- if OUTPUT=*SOURCE-LOCATION is specified and the source program exists as a PLAM library element.

If the output location is specified as OUTPUT=*LIB-ELEM(LIB=*STD-LIBRARY), the elements are written to the PLAM library with the default name SYS.PROGLIB.

1. The following source program name components, if present, are not used to construct the default element name and are therefore dropped:
 - name components for the catid and userid in BS2000 file and library names
 - directory names in POSIX path names
 - the suffixes .C , .CPP, .CXX, .CC and .I if the source program exists as a cataloged BS2000 file or PLAM library element
 - the suffixes .c , .C , .cpp, .CPP, .cxx, .CXX, .cc, .CC, .c++, .C++, .i and .I if the source program exists as a POSIX file
2. The remaining portion of the source program name is truncated from the right to 59 characters.
3. Special characters not allowed in element names are always converted to “\$”. The permitted special characters are \$, @, #, . (period), - (hyphen), and _ (underscore).
4. Lowercase letters (in POSIX source file names) are converted to uppercase.
5. The name truncated to 59 characters is then extended with the appropriate suffix .I, .LST, .CIF or .DIAG.

Summary

Contents	Source program input from			
	*SYSDTA	BS2000 file	PLAM library	POSIX file
PREPROCESS	CSTDEXP.I, S	file.I, S	elem.I, S	file.I, S
LISTING	CSTDLST.LST, P	file.LST, P	elem.LST, P	file.LST, P
DIAGNOSTIC	CSTDDIAG.DIAG, D	file.DIAG, D	elem.DIAG, D	file.DIAG, D
CIF	CSTDCIF.CIF, H	file.CIF, H	elem.CIF, H	file.CIF, H

Default names for PLAM library elements, depending on input source and contents

Construction of default names for POSIX files

The output is placed in POSIX files with default names

- if OUTPUT=*SOURCE-LOCATION is specified, and the source program exists as a POSIX file;
 - if OUTPUT=<posix-pathname> is specified, and if <posix-pathname> specifies the name of a POSIX file directory (without file name).
1. The following suffixes in source program names, if present, are not used to construct default names and are therefore dropped: .C , .CPP, .CXX, .CC or .I (for BS2000 files) and .c, .C , .cpp, .CPP, .cxx, .CXX, .cc, .CC, .c++, .C++, .i or .I (for POSIX files).
 2. The name is extended with the appropriate suffix, i.e.: .i (C compilation), .I (C++ compilation), .lst, .cif or .diag.
 3. Lowercase letters and other special characters are accepted unaltered. No name truncation occurs.

Rules for constructing module names

As far as module names are concerned, a distinction must be made between the element name (i.e. the name of the “container”) under which the module is stored as a library element and the internal module and CSECT names.

If the name of the module is not explicitly specified in the MODULE-OUTPUT option of the COMPILE statement, the element name and the internal module/CSECT names are derived from the name of the source program.

If the name of the module is explicitly specified in the MODULE-OUTPUT option of the COMPILE statement, the element name and the internal module / CSECT names are constructed from that name.

Construction of element names from LLMs in PLAM libraries

- Derivation from the source program name
 1. The following parts of the source program name, if present, are not used to construct the element name and are therefore dropped: <cat-id>, <user-id>, and the suffixes .C, .CPP, .CXX, .CC or .I (for BS2000 files) and .c, .C, .cpp, .CPP, .cxx, .CXX, .cc, .CC, .c++, .C++, .i or .I (for POSIX files).
 2. If the remaining portion of the source program name exceeds 59 characters, it is truncated from the right to 59 characters.
 3. Special characters not permitted for element names are always converted to “\$”. The permitted special characters are \$, @, #, . (period), - (hyphen), and _ (underscore). Lowercase letters in POSIX file names are converted to uppercase.
 4. If the source program is read from SYSDTA, the element name is “CSTDMOD”. This applies even if SYSDTA was assigned to a cataloged file or a library element with the ASSIGN-SYSDTA command.
- Derivation from the explicitly specified name

In this case, the specified name is accepted as the element name without changes, i.e. the suffix .C, .CPP, .CXX, .CC or .I, if present, is not removed. Similarly, no truncation of the name occurs, i.e. the name can have a length of up to 64 characters.

Special Note

To enable further processing with DBL, the maximum permissible length for the element names of LLMs is currently 32 characters. This applies to the LLM specified in the START-EXECUTABLE-PROGRAM command (module containing the `main` function) as well as all LLMs to be dynamically linked.

LLMs that have element names which exceed 32 characters must therefore be linked with BINDER.

Construction of LLM object file names in the POSIX file system

- Derivation from the source program name

The name of the LLM object file is derived from the name of the POSIX source file as follows: only the suffix `.c`, `.C`, `.cpp`, `.CPP`, `.cxx`, `.CXX`, `.cc`, `.CC`, `.c++`, `.C++`, `.i` or `.I`, if present, is removed from the name of the POSIX source file, and the suffix `.o` is appended to the remaining portion. No truncation of names or conversion of special characters to the dollar sign “\$” and of uppercase to lowercase letters occur.

- Derivation from the explicitly specified name

The specified name is accepted without changes. Note, however, that LLM object files must be provided with the suffix `.o` to enable further processing with the appropriate link editors in the POSIX subsystem.

Construction of the internal module and CSECT names of LLMs

- Derivation from the source program name

1. The following parts of the source program name, if present, are not used to construct the module/CSECT name and are therefore dropped: `<cat-id>`, `<user-id>` and the suffixes `.C`, `.CPP`, `.CXX`, `.CC` or `.I` (for BS2000 files), and `.c`, `.C`, `.cpp`, `.CPP`, `.cxx`, `.CXX`, `.cc`, `.CC`, `.c++`, `.C++`, `.i` or `.I` (for POSIX files).
2. If the remaining portion of the source program name exceeds 30 characters, it is truncated from the right to 30 characters.
3. Special characters that are not permitted for internal LLM names are always converted to “\$”. The permitted special characters are \$, @, #, - (hyphen), and _ (underscore). Lowercase letters in POSIX file names are converted to uppercase.
4. If the source program is read from SYSDDTA, “CSTDMOD” is used as a basis for constructing the name. This also applies if SYSDDTA is assigned to a cataloged file or a library element by means of an ASSIGN-SYSDDTA command.
5. The core name obtained by applying rules 1 through 4 is then used to construct the final module and CSECT names by appending a 2-character suffix, i.e. “&@” or “&#”:

Module name	<code><core-name>&@</code>
Code CSECT	<code><core-name>&@</code>
Data CSECT	<code><core-name>&#</code>

- Derivation from the explicitly specified name
 1. The complete specified name is used to construct the module/CSECT name, i.e. the suffix `.c`, `.C`, `.cpp`, `.CPP`, `.cxx`, `.CXX`, `.cc`, `.CC`, `.c++`, `.C++`, `.i` or `.I`, if present, is not removed.
 2. The truncation of the name from the right to 30 characters, the conversion of invalid special characters to the dollar sign “\$”, and the appending of suffixes occurs as described for the derivation from the source program name.

Construction of the names of ii files

An instantiation information file (ii file) is created for each of the source files used by a template, as long as one does not already exist.

The name of this ii file is derived from the associated object and is formed by adding the suffix `.ii`.

For example, the compiled version of `hugo.C` would have the ii file `hugo.o.ii`, unless an object name other than `hugo.o` is specified.

The ii file is located in that library in which the object file itself is stored.

3.1.3 Structure of compiler messages

The compiler issues the following types of messages:

- Information messages that notify the user about the general compilation process (e.g. the start and termination messages from the compiler and messages such as 'MODULES GENERATED').
- Error messages (of the frontend compiler) that directly refer to the compiled source program (e.g. syntax and semantic errors).

These consist of

1. the actual error message line
 2. the invalid source program line (optional)
 3. a flag ^ to indicate the error location (optional)
- Error messages not related to the source program (e.g. disk full, error on opening files, etc.).

Information and error messages are output to one or more of the following destinations during compilation:

- at the terminal and/or
- optionally to files or libraries.

The output destination for these messages can be controlled with the MODIFY-DIAGNOSTIC-PROPERTIES statement ([page 94](#)).

Error messages that refer to the source program are also documented in the source/error listing (controlled by the MODIFY-LISTING-PROPERTIES statement; see [page 105](#)).

General format of the compiler messages:

Error message line:

message-number [*message-weight*]: *filename* / *line-number*: *message-text*

Optional for error messages referring to the source program:

Invalid source program line

^ *marker to indicate the error location*

The file name and line number are omitted in error messages (with a message weight of FATAL or ERROR) which do not refer to the source program. In information messages, the message weight is also omitted.

message-number

The 7-digit message number consists of a 3-digit message class, which identifies the compiler component, and a 4-digit message number.

Message class	Components
CFE	Frontend of the compiler: scanner, preprocessor, parser, listing generator, message facility
CDR	Compiler driver, II-UPDATE
UMP	Intermediate language module (ULS): optimizer, inliner
BEM	Backend
SIS	Compiler I/O interface (PROSOS), Object format (module) generator (OFG), Diagnostic information generator for AID (DIG)
BND	BINDER (when linking with the BIND statement of the compiler)
CCM	C runtime system

[message weight]

Indicates the message weight, i.e. the severity of the error that has occurred:

[NOTE]	Inconsequential errors, e.g. "ugly" or superfluous constructs which usually have no impact on subsequent program behavior. Notes are not issued automatically unless the options <code>-R minweight,notes</code> or <code>MINIMAL-MSG-WEIGHT=*NOTE</code> are specified.
[WARNING]	Errors for which the compiler will generate a module, but which could lead to situations resulting in deviant program behavior.

[ERROR], [*ERROR] Errors for which no module is generated. The compiler will attempt to continue the compilation until the number of errors specified with the `-R limit, n` or `MAX-ERROR-NUMBER=n` option is reached. Errors of the frontend compiler that can either be reduced to the severity of a WARNING/NOTE with the `-R warning/note` or `CHANGE-MSG-WEIGHT=*WARNING()/*NOTE()` option and warnings that were upgraded to the severity of an ERROR with the `-R error` or `CHANGE-MSG-WEIGHT=*ERROR()` option are flagged with an asterisk.

[FATAL] Fatal errors leading to abortion of the compilation

[INTERNAL] Compiler errors, also leading to abortion of the compilation

filename

Name of the file or library element with the invalid source program code

line-number

Indicates the source program line in which the error occurred.

message-text

Text of the error message; can be output in English or German.

Invalid source program line with a marker to indicate the error location

`-R show_column` or `SHOW-COLUMN=*YES` is the default setting, which means that the original source program line is shown with the error location marked (with `^`) in addition to the diagnostic message itself. If `-R no_show_column` or `SHOW-COLUMN=*NO` is specified, the marked source program line is not output.

Example

```
//MODIFY-SOURCE-PROP LANG=*C
//MODIFY-DIAGNOSTIC-PROP CHANGE-MSG-WEIGHT=*ERROR(CFE1064)
//COMPILE SOURCE=TEST.C

% CFE1064 [*ERROR]: TEST.C / 1: declaration does not declare anything
  struct {};
  • ^

% CFE1077 [ERROR]: TEST.C / 3: this declaration has no storage class or
  type specifier
  xxxxx;
  ^

% CFE1054 [*ERROR]: USER.H / 3: too few arguments in macro invocation
  int i = A(3);
          ^
```

The error number CFE1064 was originally a warning, but was upgraded to the ERROR class with the option CHANGE-MSG-WEIGHT=*ERROR(CFE1064). The class of errors that are not originally output with an asterisk (e.g. CFE1077 in this case) cannot be changed.

Error number CFE1054 is an example of an error that is generated in strict ANSI C mode, but can be downgraded to the message weight of a warning with CHANGE-MSG-WEIGHT=*WARNING(CFE1054). The same error number is assigned the error class WARNING in extended ANSI C mode.

The HELP-MSG command can be used to obtain a more detailed explanation of all the error messages:

```
HELP-MSG MSG-ID=msgid[, LANGUAGE=D / E]
```

The compiler messages can be output in English or German. The default setting depends on system generation. The following command can be used to change the task-specific default:

```
MODIFY-MSG-ATTRIBUTES TASK-LANGUAGE = D / E
```

3.2 Controlling the compiler

As in the case of many other BS2000 programs (BINDER, LMS, etc.), the C/C++ compiler can be called as an independent program and controlled via a convenient SDF command interface. This provides the user with all the facilities offered by the product SDF (System Dialog Facility) for the input of program statements, i.e.: various levels of guided and unguided dialogs, input from procedure or command files, etc.

A detailed explanation of the SDF dialog interface with examples on the various forms of SDF control can be found in the manual "SDF Dialog Interface, User Guide" [12].

3.2.1 Calling the compiler (START-CPLUS-COMPILER)

/START-CPLUS-COMPILER

Abbreviations: SRCPC, CPLUS-COMPILER, CPC

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

,CPU-LIMIT = *JOB-REST / <integer 1..32767>

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

The compiler run can be monitored by means of a BS2000 job variable. <filename> assigns a monitoring job variable in which the compiler indicates runtime errors that may occur.

Two values are entered into a job variable by the operating system:

a status indicator with a length of 3 bytes, and
a return code with a length of 4 bytes.

The table below shows how job variables are supplied in various termination states.

Error class	Compiler termination	Status indicator	Return code	Procedure behavior	Exit status
no error	normal	\$T	0000	no branch	-
[NOTE]			1001		
[WARNING]			1002		
[ERROR]			2003	branch to next STEP, etc.	1
[ERROR]			2004		
[FATAL]	abnormal	\$A	3005	EXIT_FAILURE 9990888	
[INTERNAL]			3006		

CPU-LIMIT = *JOB-REST / <integer 1..32767>

This option can be used to define the maximum CPU time for the compiler run. The value entered here corresponds to the CPU-LIMIT operand of the START-EXECUTABLE-PROGRAM command.

3.2.2 Description of compiler statements

Overview of statements

For more information on the principle of “executing” and “modifying” compiler statements, see the notes in the [section “Basic principles and general input rules” on page 64](#).

- Executing statements

BIND

Start the linkage and/or prelinker run

CHECK-SYNTAX

Start syntax analysis

COMPILE

Start the compilation run (including module generation)

PREPROCESS

Start the preprocessor run

- Modifying and defining statements

The following list shows the various MODIFY statements together with the executing statement(s) for which they are evaluated (in parentheses).

MODIFY-BIND-PROPERTIES

Options to control the linkage and/or prelinker run (BIND)

MODIFY-CIF-PROPERTIES

Options to output CIF information (CHECK-SYNTAX, COMPILE, PREPROCESS)

MODIFY-DIAGNOSTIC-PROPERTIES

Options to control message output (CHECK-SYNTAX, COMPILE, PREPROCESS)

MODIFY-INCLUDE-LIBRARIES

Options to specify header files (CHECK-SYNTAX, COMPILE, PREPROCESS)

MODIFY-LISTING-PROPERTIES

Options to output listings (CHECK-SYNTAX, COMPILE, PREPROCESS)

MODIFY-MODULE-PROPERTIES

Options to control object and module attributes (COMPILE)

MODIFY-OPTIMIZATION-PROPERTIES

Optimization options (COMPILE)

MODIFY-RUNTIME-PROPERTIES

Runtime options (COMPILE)

MODIFY-SOURCE-PROPERTIES

Frontend options (CHECK-SYNTAX, COMPILE, PREPROCESS)

MODIFY-TEST-PROPERTIES

Debugging options (COMPILE)

- Informative statements

SHOW-DEFAULTS

Show default settings of the compiler

SHOW-PROPERTIES

Show current option values of MODIFY statements

- RESET-TO-DEFAULT

Reset option values of MODIFY statements to the default settings of the compiler

- END

Terminate the compiler run

- Standard SDF statements

Besides the C/C++-specific statements listed above, the following standard SDF statements may also be used during a compiler run. A detailed description of these SDF statements can be found in the manual "SDF Dialog Interface, User Guide" [12].

EXECUTE-SYSTEM-CMD

Execute a system command during the compiler run

HELP-MSG-INFORMATION

Write text of a system or compiler message to SYSOUT

HOLD-PROGRAM

Stop the compiler run, e.g. to enter system commands. Control is returned to the statement mode of the compiler with the command RESUME-PROGRAM

MODIFY-SDF-OPTIONS

Enable/disable user syntax file and change SDF settings

REMARK

Enter comments in a sequence of statements

RESET-INPUT-DEFAULTS

Reset task-specific default values

RESTORE-SDF-INPUT

Show previously entered statements of commands

SHOW-INPUT-DEFAULTS

Show task-specific default values

SHOW-INPUT-HISTORY

Show contents of input buffer

SHOW-SDF-OPTIONS

Show information on all active syntax files and SDF options for the current task

SHOW-STMT

Show SDF syntax for a statement

STEP

Define restart point within a command/procedure file

WRITE-TEXT

Display text

Basic principles and general input rules

- Executing and modifying compiler statements

The compiler run is started with the command START-CPLUS-COMPILER and is terminated with END statement. During each compiler run, a number of different compilation and/or linkage runs may be started with appropriate "executing" statements (see [page 61](#)). These compilation and linkage runs can be controlled by means of MODIFY statements, which must always precede the executing statements to be used. These MODIFY statements remain in effect even after the executing statements have been completed (see the example on [page 65](#) for details).

- The operand value *UNCHANGED and default settings of the compiler

When using the compiler in SDF interactive mode with guidance, the operand value *UNCHANGED is frequently displayed for many operands in the SDF statement menus of the compiler.

The value *UNCHANGED is an SDF default value that is always used in the current compiler statement (and displayed on the screen) whenever no explicit specification for an operand is made.

*UNCHANGED simply means that the operand value that was defined with an earlier compiler statement still applies to the operand involved. If no such value was specified in the entire compiler run, e.g. immediately after starting the compiler, *UNCHANGED refers to the default value of the compiler, and this default value remains in effect until some other operand value is specified with a compiler statement. In this manual, all default values of the compiler are underlined in the individual statement descriptions. The currently applicable operand values for compiler statements can be listed with the SHOW-PROPERTIES statement. In this case, the actual operand value is displayed instead of *UNCHANGED.

- Aliases

Most of the compiler statements have aliases, which may be used instead of the names displayed in the guided dialog. These aliases are listed for each statement at the start of the corresponding statement description.

- Spin-off mechanism

Executing statements that do not succeed trigger the spin-off mechanism. All statements up to the next STEP are ignored in such cases.

- Input/Output library elements

The version specification `VERSION=@` is rejected, since this is not a valid value for the LMS version.

Example

```

/START-CPLUS-COMPILER
//MODIFY-SOURCE-PROPERTIES DEFINE=_OSD_POSIX _____ (1)
//MODIFY-LISTING-PROPERTIES SOURCE=*YES
//MODIFY-INCLUDE-LIBRARIES STD-INCLUDE-LIBRARY=
  (*STANDARD-LIBRARY, $.SYSLIB.POSIX-HEADER) _____ (2)
//COMPILE SOURCE=HELLO1.CC _____ (3)
...
//MODIFY-SOURCE-PROPERTIES LANG=*C
//COMPILE SOURCE=HELLO2.C _____ (4)
...
//END
/

```

The `MODIFY-SOURCE-PROPERTIES` statement (1) only issues a directive, so the default settings of the compiler, e.g. extended ANSI C++ language mode (`LANGUAGE=*CPLUSPLUS(MODE=*ANSI)`), apply to all other operands.

On compiling (2) the C++ source program `HELLO1.CC`, the preceding `MODIFY` statements are evaluated, and the following steps are performed as a result:

1. The `_OSD_POSIX` directive is issued.
2. The header libraries `$.SYSLIB.CRTE` and `$.SYSLIB.POSIX-HEADER` are searched.
3. The source program is compiled in the ANSI C++ language mode and a source/error listing is output to `SYSLST`.
4. The C source program is compiled in the ANSI C language mode and a source/error listing is output to `SYSLST`.

Before the second compilation (4), the ANSI C language mode is set in a new `MODIFY-SOURCE-PROPERTIES` statement (3). Apart from for this change, all other entries in the earlier `MODIFY` statements remain in effect when compiling the C source program `HELLO2.C`. In other words, steps 1., 2., and 3. are also performed.

BIND

Alias: LINK

This statement starts a linkage run and, in the case of ANSI C++ objects, also activates the prelinker for automatic template instantiation. The input sources and other conditions for the linkage and prelinker runs are defined in the preceding MODIFY-BIND-PROPERTIES statements.

BIND

```

ACTION = list-poss: *PRELINK / *MODULE-GENERATION
,OUTPUT = *NONE / *LIBRARY-ELEMENT(...)
  *LIBRARY-ELEMENT(...)
    | LIBRARY = <filename 1..54> / *LINK(...)
    | *LINK(...)
    | | LINK-NAME = <filename 1..8>
    | ,ELEMENT = <composed-name 1..64 with-under>(…)
    | | <composed-name 1..64 with-under>(…)
    | | VERSION = *UPPER-LIMIT / *INCREMENT / <composed-name 1..24 with-under>
,OUTPUT-FORMAT = *UNCHANGED / *LLM(...)
  *LLM(...)
    | EXTERNAL-NAMES = *UNCHANGED / *STD / *SHORT / *EXTENDED
,ADD-OPTION = *UNCHANGED / *NONE / <c-string 1..1800 with-low>

```

ACTION = list-poss: *PRELINK / *MODULE-GENERATION

The *PRELINK specification is only relevant in the ANSI C++ modes and affects the automatic template instantiation by the prelinker.

If the ACTION option is not specified, the following default applies:

```
ACTION = (*PRELINK,*MODULE-GENERATION)
```

This causes both a prelinker and a linkage run to be performed in the ANSI C++ modes. The result comprises individual modules in which all templates are instantiated and a linked module. Note that the *PRELINK specification is ignored in the C modes and in the Cfront C++ mode, so only a linkage run is performed in these modes.

If ACTION=*PRELINK is specified, only a prelinker run is performed. The result consists of individual modules in which all templates are instantiated. Specifications in the OUTPUT and OUTPUT-FORMAT options, if any, are ignored.

If ACTION=*MODULE-GENERATION is specified, only a linkage run is performed.

OUTPUT = *NONE / *LIBRARY-ELEMENT(...)

This option specifies the PLAM library (LIBRARY=) and the element name (ELEMENT=) under which the linked module is to be stored. These values are passed to BINDER (as a MODULE-CONTAINER operand) in a SAVE-LLM statement. If no further specifications are made with the ADD-OPTION option, the remaining operands of the SAVE-LLM statement are set to the corresponding defaults for BINDER.

The *NONE option is the default. But because this does not create a link object, this option can only sensibly be used with (see above, ACTION =).

LIBRARY =<filename 1..54>

The module is written to a PLAM library with the specified name.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>

<filename> can be used (instead of a cataloged library name) to specify a valid link name for the library. The link name must already have been assigned to the PLAM library with the ADD-FILE-LINK command before the compiler is called.

ELEMENT = <composed-name 1..64 with-under>(...)

The module is written to the element with the assigned name under the PLAM library specified with LIBRARY=.

VERSION = *UPPER-LIMIT

If the element entry does not contain a version ID, the highest possible version is used by the compiler.

VERSION = *INCREMENT

The element is assigned a version number that is obtained by incrementing the highest existing version number by 1, assuming that the highest existing version ID ends with a digit that can be incremented. If the version ID cannot be incremented, the compiler run is aborted with an error message.

See the COMPILE statement ([page 75](#)) for an example.

VERSION = <composed-name 1..24 with-under>

The element is assigned the specified version.

OUTPUT-FORMAT = *LLM(EXTERNAL-NAMES = *UNCHANGED / *STD / *SHORT / *EXTENDED)

This option controls how symbol names in EEN (Extended External Name) format are handled by BINDER. The entries made here are passed on to BINDER as the FOR-BS2000-VERSION operand of the SAVE-LLM statement.

EENs, i.e. external C++ symbols with untruncated names, are generally contained in modules that were generated with the compiler in the ANSI C++ mode.

Untruncated external C symbols are generated only if the following option is specified at compilation:

MODIFY-MODULE-PROPERTIES C-NAMES=*UNLIMITED (see [page 118](#)).

In this case, even longer external C symbols are not truncated by the compiler to 32 bytes. Modules with EENs are stored by the compiler in LLM Format 4. The modules of the C++ libraries and of the CRTE runtime systems used in ANSI C++ mode are also provided in LLM Format 4.

If the modules generated by the compiler do not include any EENs, i.e. are in LLM Format 1, this option has no effect, since the BINDER generates an input format corresponding to LLM Format 1.

EXTERNAL-NAMES = *UNCHANGED

The specification in the last BIND statement applies.

EXTERNAL-NAMES = *STD

By default, BINDER generates LLM Format 4. The EENs remain in the result module without being truncated. LLMs in Format 4 can be partially linked, i.e. first linked with unresolved external references to EENs and then processed further as desired by means of BINDER or DBL.

EXTERNAL-NAMES = *SHORT

This entry is needed if BINDER is to generate LLM Format 1. By default, LLM Format 4 is generated.

EXTERNAL-NAMES = *EXTENDED

This entry is supported for compatibility reasons only.

Summary of generated LLM formats

Input format	SDF option EXTERNAL-NAMES =	Output format
LLM 1	No entry / *STD / *SHORT / *EXTENDED	LLM 1
LLM 4 (EEN)	No entry / *STD / *EXTENDED	LLM 4
	*SHORT	LLM 1

ADD-OPTION = *UNCHANGED / *NONE / <c-string 1..1800 with-low>

<c-string> can be used to specify further operands of the BINDER statement SAVE-LLM in addition to the MODULE-CONTAINER operand (see OUTPUT option) and the FOR-BS2000-VERSION operand (see OUTPUT-FORMAT option). If more than one operand is specified, they must be separated by commas as follows:

'operand1, operand2, ...'

The operands are directly passed through to BINDER without SDF analysis.

The following operands of the BINDER statement SAVE-LLM are obtained from the MODIFY-BIND-PROPERTIES statement and must therefore not be specified with ADD-OPTION: TEST-SUPPORT (to save LSD information) and MAP (to generate a map listing).

Automatic template instantiation

The operands specified with ADD-OPTION are currently not considered during the automatic template instantiation by the prelinker, but only during subsequent linkage with BINDER. The operands should not affect the type, number or order of the modules to be linked, since different preconditions during the prelinker and linkage runs could result in duplicates or unresolved external references.

Example

```
//BIND OUTPUT=*LIB(LIB=PLAM.BSP,ELEM=HELLO),ADD-OPT='OVERWRITE=*NO,
NAME-COLLISION=*WARNING'
```

CHECK-SYNTAX

Alias: DO-SYNTAX-CHECK

The compilation run is terminated after checking the syntax of one or more source programs. No object code is generated.

CHECK-SYNTAX

SOURCE = *SYSDTA / list-poss: <filename 1..54> / <posix-pathname> / *LIBRARY-ELEMENT(...)

*LIBRARY-ELEMENT(...)

LIBRARY = <filename 1..54> / *LINK(...)

 *LINK(...)

LINK-NAME = <filename 1..8>

,ELEMENT = <composed-name 1..64 with-under>(...)

 <composed-name 1..64 with-under>(...)

VERSION = HIGHEST-EXISTING / <composed-name 1..24 with-under>

SOURCE =

This option specifies one or more source programs for which a syntax check is to be performed.

A source program can be read from the system file SYSDTA, a cataloged BS2000 file, a PLAM library or a POSIX file.

Note that if the source program is entered from SYSDTA, only one source program can be read per CHECK-SYNTAX statement.

SOURCE = *SYSDTA

Input is accepted from the system file SYSDTA. SYSDTA is assigned to the terminal in interactive mode but can be reassigned to a cataloged file or a PLAM library element with the ASSIGN-SYSDTA command (see also [page 76](#)).

SOURCE = <filename 1..54>

<filename> is used to specify the name of a cataloged BS2000 file.

SOURCE = <posix-pathname>

Only a file name is permitted as <posix-pathname>. See [page 34](#) for a description of the term <posix-pathname>.

SOURCE = *LIBRARY-ELEMENT(...)

This option is used to specify a PLAM library and an element in it.

LIBRARY = <filename 1..54>

<filename> assigns the name of a PLAM library.

LIBRARY = *LINK(...)**LINK-NAME = <filename 1..8>**

<filename> is the link name for a PLAM library. The link name must be assigned to the library name by means of the ADD-FILE-LINK command before the compiler is called.

ELEMENT = <composed-name 1..64 with-under>(...)

<composed-name> identifies the fully-qualified name of an element from the PLAM library specified earlier. The element must be of type S.

VERSION = *HIGHEST-EXISTING

If the element specification contains no version ID, the compiler uses the element with the highest existing version.

VERSION = <composed-name 1..24 with-under>

The compiler uses the element with the specified version.

Note

Templates in ANSI-C++ sources (in contrast to the COMPILE statement) are **not** implicitly included.

COMPILE

This statement compiles one or more source programs and generates a module for each compilation unit.

COMPILE
<pre> SOURCE = *SYSDTA / list-poss: <filename 1..54> / <posix-pathname> / *LIBRARY-ELEMENT(...) *LIBRARY-ELEMENT(...) LIBRARY = <filename 1..54> / *LINK(...) *LINK(...) LINK-NAME = <filename 1..8> ,ELEMENT = <composed-name 1..64 with-under>(…) <composed-name 1..64 with-under>(…) VERSION = *HIGHEST-EXISTING / <composed-name 1..24 with-under> ,MODULE-OUTPUT = *SOURCE-LOCATION / <posix-pathname> / *LIBRARY-ELEMENT(...) *LIBRARY-ELEMENT(...) LIBRARY = *STD-LIBRARY / *SOURCE-LIBRARY / <filename 1..54> / *LINK(...) *LINK(...) LINK-NAME = <filename 1..8> ,ELEMENT = *STD-ELEMENT(...) / <composed-name 1..64 with-under>(…) *STD-ELEMENT(...) VERSION = *UPPER-LIMIT / *INCREMENT / <composed-name 1..24 with-under> <composed-name 1..64 with-under>(…) VERSION = *UPPER-LIMIT / *INCREMENT / <composed-name 1..24 with-under> </pre>

SOURCE =

This option specifies one or more source programs to be compiled.

A source program can be read from the system file SYSDTA, a cataloged BS2000 file, a PLAM library or a POSIX file.

Note that if the source program is entered from SYSDTA, only one source program can be read per COMPILE statement.

SOURCE = *SYSDTA

Input from the system file SYSDTA is only possible in the C modes and in the Cfront C++ mode of the compiler. In the ANSI C++ modes, the value *SYSDTA is rejected with a corresponding error message. SYSDTA is assigned to the terminal in interactive mode but can be reassigned to a cataloged file or a PLAM library element with the ASSIGN-SYSDTA command (see also [page 76](#)).

SOURCE = <filename 1..54>

<filename> is the name of a cataloged BS2000 file.

SOURCE = <posix-pathname>

Only a file name is permitted as <posix-pathname>. See [page 34](#) for a description of the term <posix-pathname>.

SOURCE = *LIBRARY-ELEMENT(...)

This option is used to specify a PLAM library and an element in it.

LIBRARY = <filename 1..54>

<filename> assigns the name of a PLAM library.

LIBRARY = *LINK(...)**LINK-NAME = <filename 1..8>**

<filename> is used to specify a link name for a PLAM library. The link name must already have been assigned to the library name with the ADD-FILE-LINK command before the compiler is called.

ELEMENT = <composed-name 1..64 with-under>(...)

<composed-name> identifies the fully-qualified name of an element from the PLAM library specified earlier. The element must be of type S.

VERSION = *HIGHEST-EXISTING

If the element specification contains no version ID, the compiler uses the element with the highest existing version.

VERSION = <composed-name 1..24 with-under>

The compiler uses the element with the specified version.

MODULE-OUTPUT =

This option allows the user to control the library and/or name under which the generated modules are stored.

MODULE-OUTPUT = *SOURCE-LOCATION

The module is written to the same location as the source program. If the source program is a PLAM library element, the module is placed in the library of the source program. If the source program is a POSIX file, the module is written as an object file (".o" file) into the directory of the source program.

The element or object file name is derived from the name of the source program (see the [section “Rules for constructing module names” on page 52](#)).

The module cannot be written to a `write-only` file. In Posix, it must always be possible to read the library or object file.

The `*SOURCE-LOCATION` specification is invalid if the source program is read from a cataloged BS2000 file or via SYSDTA (see [page 76](#)).

MODULE-OUTPUT = <posix-pathname>

The module is written as an LLM object file in the POSIX file system.

Both a file name and a directory are permitted as `<posix-pathname>`. See [page 34](#) for a description of the term `<posix-pathname>`.

When a file name is specified, the object file is stored under this name. Specification of a file name is illegal when compiling multiple source programs with one `COMPILE` statement.

When a directory name *dir* is specified, an object file for each compiled source program is written under the default name *sourcefile.o* to the directory *dir* (see also the [section “Rules for constructing module names” on page 52](#)).

The directories specified with `<posix-pathname>` must already exist.

When constructing file names, it must be noted that object files can only be meaningfully processed further (i.e. linked) in the POSIX subsystem if the name contains the suffix `.o` or a suffix defined with the `-Y F` option of the `cc/c89/CC` commands (see also the manual “POSIX Commands of the C/C++ Compiler” [1]).

MODULE-OUTPUT = *LIBRARY-ELEMENT(...)

This option specifies the PLAM library (`LIBRARY=`) and the element name (`ELEMENT=`) under which the module is to be stored.

LIBRARY = *STD-LIBRARY

Modules are stored in the library `SYS.PROG.LIB` by default.

LIBRARY = *SOURCE-LIBRARY

The module is written to the PLAM library which contains the source program.

The `*SOURCE-LIBRARY` specification is invalid if the source program is read from a cataloged BS2000 file, a POSIX file or via SYSDTA.

LIBRARY =<filename 1..54>

The module is written to a PLAM library with the specified name.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>

`<filename>` can be used (instead of a cataloged library name) to specify a valid link name for the library. The link name must already have been assigned to the library name with the `ADD-FILE-LINK` command before the compiler is called.

ELEMENT = *STD-ELEMENT(...)

The element name is derived from the name of the source program (see [section “Rules for constructing module names” on page 52](#)).

VERSION = *UPPER-LIMIT

If the element entry does not contain a version ID, the highest possible version is used by the compiler.

VERSION = *INCREMENT

The element is assigned a version number that is obtained by incrementing the highest existing version number by 1, assuming that the highest existing version ID ends with a digit that can be incremented. If the version ID cannot be incremented, the compiler run is aborted with an error message.

Warning: You may not specify ***INCREMENT** in the ANSI-C++ mode.

Example

Highest existing version	Version generated by *INCREMENT
ABC1	ABC2
ABC9	Error
ABC09	ABC10
003	004
None	001

VERSION = <composed-name 1..24 with-underscore>

The element is assigned the specified version.

ELEMENT = <composed-name 1..64 with-underscore>(…)

The module is written under the assigned name to the PLAM library specified with LIBRARY= . This specification is invalid when compiling multiple source programs with one COMPILE statement.

To enable further processing with DBL, element names of LLMs must not exceed a maximum of 32 characters (see [section “Rules for constructing module names” on page 52](#)).

VERSION = *UPPER-LIMIT / *INCREMENT / <composed-name 1..24 with-underscore>

The version can be specified as described above for ELEMENT=*STD-ELEMENT(…).

Warning: You may not specify ***INCREMENT** in the ANSI-C++ mode.

Notes on input via SYSDTA

For compatibility reasons, input of a source program via the system file SYSDTA can be enabled in the C language modes and in the Cfront C++ language mode of the compiler. The following option must be specified for this purpose:

```
SOURCE=*SYSDTA
```

Input via SYSDTA is not possible in the ANSI C++ modes.

In interactive mode, SYSDTA is assigned to the terminal by default. Although source programs could basically be entered from the terminal, it would not be very practical, since the source would not be available later.

This means that if *SYSDTA is specified with the SOURCE option, SYSDTA must be assigned to a cataloged file or a PLAM library element. The appropriate command for this purpose is as follows:

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

The assignment can be made by the following methods:

1. Assigning SYSDTA before calling the compiler

Example

```
/ASSIGN-SYSDTA TO-FILE=source
/START-CPLUS-COMPILER
```

The assigned source must include all required MODIFY statements and the COMPILE statement before the actual source code:

```
//MODIFY-SOURCE-PROP LANG=*C
//MODIFY-...
//COMPILE SOURCE=*SYSDTA,...
#include ...
main() {
...
}
```

2. Assigning SYSDDTA after calling the compiler

Example

```
/START-CPLUS-COMPILER
//MODIFY-SOURCE-PROP LANG=*C
//MODIFY-...
//EXECUTE-SYSTEM-CMD (ASSIGN-SYSDTA TO-FILE=source)
```

The required MODIFY statements are specified before reassigning SYSDDTA and are hence not needed in the assigned source. Following the EXECUTE statement, no further SDF statements may be specified. The assigned source must include at least the COMPILE statement before the actual source code:

```
//COMPILE SOURCE=*SYSDTA,...
#include ...
main() {
...
}
```

When the end of the source file is reached, the compiler run terminates. SYSDDTA cannot be reassigned again within a compiler run.

Note that the input source continues to be SYSDDTA even though the source program has been assigned with the ASSIGN-SYSDTA command. Consequently, the source program name "CSTD" is used as the basis for generating default names (see also the output parameter *SOURCE-LOCATION or ELEMENT=*STD-ELEMENT). Furthermore, when a library element is assigned with the ASSIGN-SYSDTA command instead of the SOURCE option, the *SOURCE-LIBRARY specification has no effect.

END

This statement ends the compiler run.

END

MODIFY-BIND-PROPERTIES

Aliases: SET-BIND-PROPERTIES
 MODIFY-LINK-PROPERTIES
 SET-LINK-PROPERTIES

This statement specifies the input sources and other conditions for the prelinker and linkage run that is started thereafter with a BIND statement.

MODIFY-BIND-PROPERTIES

```

START-LLM-CREATION = *YES / *NO
,INCLUDE = *UNCHANGED / list-poss: *LIBRARY-ELEMENT(...) / *NONE
  *LIBRARY-ELEMENT(...)
    | LIBRARY = <filename 1..54> / *LINK(...)
      *LINK(...)
        | LINK-NAME = <filename 1..8>
          ,ELEMENT = *ALL(...) / <composed-name 1..64 with-under>(…)
            *ALL(...)
              | VERSION = *HIGHEST-EXISTING / <composed-name 1..24 with-under>
                <composed-name 1..64 with-under>(…)
                  | VERSION = *HIGHEST-EXISTING / <composed-name 1..24 with-under>
                    ,ADD-OPTION = *UNCHANGED / *NONE / <c-string 1..1800 with-low>
,RESOLVE = *UNCHANGED / [*AUTOLINK](…) / *NONE
  *AUTOLINK(...)
    | LIBRARY = list-poss(40) : <filename 1..54> / *LINK(...)
      *LINK(...)
        | LINK-NAME = <filename 1..8>
          ,INSTANTIATE = *NO / *YES / *IGNORE
        ,ADD-OPTION = *UNCHANGED / *NONE / <c-string 1..1800 with-low>

```

```

,ADD-PRELINK-FILES = *UNCHANGED / list-poss: *LIBRARY-ELEMENT(...) / *LIBRARY(...) / *NONE
  *LIBRARY-ELEMENT(...)
    |
    | LIBRARY = <filename 1..54> / *LINK(...)
    |
    | *LINK(...)
    | | LINK-NAME = <filename 1..8>
    |
    | ,ELEMENT = *ALL(...) / <composed-name 1..64 with-under>(…)
    | *ALL(...)
    | | VERSION = *HIGHEST-EXISTING / <composed-name 1..24 with-under>
    | | <composed-name 1..64 with-under>(…)
    | | VERSION = *HIGHEST-EXISTING / <composed-name 1..24 with-under>
  *LIBRARY(...)
    |
    | LIBRARY-NAME = <filename 1..54> / *LINK(...)
    |
    | *LINK(...)
    | | LINK-NAME = <filename 1..8>
,MAX-INSTANTIATE-ITER = 30 / *UNCHANGED / <integer 0..100>
,TEMPLATE-DEF-LIST = *UNCHANGED / *YES / *NO
,ADD-STATEMENT = *UNCHANGED / *NONE / list-poss: <c-string 1..1800 with-low>
,RUNTIME-LANGUAGE = *UNCHANGED / *C / *CPLUSPLUS(…)
  *CPLUSPLUS(…)
    |
    | MODE = *ANSI / *CPP
,STDLIB = *UNCHANGED / *DYNAMIC / *DYNAMIC-COMPLETE / *STATIC / *NONE
,TOOLS_LIB = *UNCHANGED / *YES / *NO
,TEST-SUPPORT = *UNCHANGED / *YES / *NO
,LISTING = *UNCHANGED / *NONE / *SYSLST / <filename 1..54>

```

START-LLM-CREATION = *YES / *NO

The option START-LLM-CREATION=*YES is analogous to the BINDER statement START-LLM-CREATION, which begins a new link process and creates a new LLM in the work area. *YES is the default setting of the compiler and is automatically used if the MODIFY-BIND-PROPERTIES statement involved is the first such statement issued since calling the compiler.

Starting with the second MODIFY-BIND-PROPERTIES statement, START-LLM-CREATION is automatically set to the SDF default value *NO. This value remains in effect for all subsequent MODIFY-BIND-PROPERTIES statements (and even across multiple BIND statements) until it is explicitly changed to *YES.

As long as this value is set to *NO, the specifications made for INCLUDE and RESOLVE libraries remain in effect for all subsequent linkage runs.

For more information on the interaction between the MODIFY-BIND-PROPERTIES and BIND statements, see also the [section “Interaction between the MODIFY-BIND-PROPERTIES and BIND statements” on page 89](#).

INCLUDE =

This option, which is analogous to the MODULE-CONTAINER operand of the BINDER statement INCLUDE-MODULES, specifies the modules to be linked.

Automatic template instantiation

The modules specified with the INCLUDE option are always instantiated by the prelinker. In cases where the same library is also specified in the RESOLVE option, it is important to ensure that the template instantiation is not disabled there (see RESOLVE option, INSTANTIATE = *NO / *IGNORE, [page 83](#)).

INCLUDE = *UNCHANGED

The values specified in the MODIFY-BIND-PROPERTIES statements since the last START-LLM-CREATION=*YES apply.

INCLUDE = list-poss: *LIBRARY-ELEMENT(...)

This option specifies the PLAM library (LIBRARY=) and the element name (ELEMENT=) under which the module to be linked is stored.

Multiple *LIBRARY-ELEMENT entries may be specified here in a list. These entries are converted internally into multiple INCLUDE-MODULES statements of the BINDER in the same order as the *LIBRARY-ELEMENT entries in the list.

LIBRARY = <filename 1..54>

<filename> assigns the name of a PLAM library.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>

<filename> is used to specify a link name for a PLAM library. The link name must already have been assigned to the library name with the ADD-FILE-LINK command before the compiler is called.

ELEMENT = *ALL(...)

All modules from the PLAM library specified with LIBRARY= are linked.

VERSION = *HIGHEST-EXISTING

If the element specification contains no version ID, the compiler uses the element with the highest existing version.

VERSION = <composed-name 1..24 with-underscore>

The compiler uses the element with the specified version.

ELEMENT = <composed-name 1..64 with-underscore>(…)

<composed-name> identifies the fully-qualified name of a module from the PLAM library specified with LIBRARY=.

VERSION = *HIGHEST-EXISTING

If the element specification contains no version ID, the compiler uses the element with the highest existing version.

VERSION = <composed-name 1..24 with-underscore>

The compiler uses the element with the specified version.

ADD-OPTION = *UNCHANGED / *NONE / <c-string 1..1800 with-low>

<c-string> can be used to specify further operands of the BINDER statement INCLUDE-MODULES in addition to the MODULE-CONTAINER operand. If more than one operand is specified, they must be separated by commas as follows:

'operand1 , operand2 , ...'

The operands are directly passed through to BINDER without SDF analysis. See the BIND statement for an input example ([page 69](#)).

Automatic template instantiation

The operands specified with ADD-OPTION are currently not considered during the automatic template instantiation by the prelinker, but only during subsequent linkage with BINDER. The operands should not affect the type, number or order of the modules to be linked, since different preconditions during the prelinker and linkage runs could result in duplicates or unresolved external references.

INCLUDE = *NONE

The operand *NONE only deletes the corresponding INCLUDE chain, but does not trigger the complete action like for START-LLM-CREATION=YES. NONE is the default value of the compiler, i.e. the value after starting the compiler or after a RESET-TO-DEFAULT.

RESOLVE =

This option, which is analogous to the BINDER statement RESOLVE-BY-AUTOLINK, specifies the libraries from which unresolved external references are to be satisfied by BINDER (when using the autolink mechanism).

The C/C++ runtime libraries of the CRTE must not be specified with the RESOLVE option. These libraries are implicitly linked in the correct order and with the correct versions in accordance with the entries in the RUNTIME-LANGUAGE-MODE, STDLIB, TOOLSLIB and RUNTIME-ENVIRONMENT options.

Automatic template instantiation

In contrast to the INCLUDE option, the template instantiation can be controlled with the RESOLVE option. By default, modules within RESOLVE libraries are not instantiated by the prelinker. See also the INSTANTIATE specification ([page 83](#)).

RESOLVE = *UNCHANGED

The values specified in the MODIFY-BIND-PROPERTIES statements since the last START-LLM-CREATION=*YES apply.

RESOLVE = [*AUTOLINK](...)

LIBRARY = list-poss(40): <filename 1..54> / *LINK(LINK-NAME = <filename 1..8>)
This option can be used to specify the cataloged file name or link name of a PLAM library to be searched when using the autolink mechanism. If multiple libraries are specified in a list, this is equivalent to a RESOLVE-BY-AUTOLINK statement with a list of RESOLVE libraries. The list may contain a maximum of 40 libraries (binder restriction).

INSTANTIATE = *NO / *YES / *IGNORE

This option is only relevant for automatic template instantiation by the prelinker. It controls whether the libraries specified with the RESOLVE option are to be instantiated (*YES), not instantiated (*NO by default), or completely ignored (*IGNORE) by the prelinker. Whereas libraries for which *NO or *YES is specified are taken into account at instantiation, i.e., are searched for existing definitions, libraries for which *IGNORE is set are not even looked at by the prelinker and are only taken into account in the subsequent linkage process.



If the same library is specified in both the RESOLVE and the INCLUDE option, the following must be observed:

As soon as INSTANTIATE=*NO (the default setting) or *IGNORE is specified for a library in a RESOLVE option, the prelinker will not perform any template instantiations for all elements of that library, even if the elements have been explicitly been linked with the INCLUDE option.

ADD-OPTION = *UNCHANGED / *NONE / <c-string 1..1800 with-low>

<c-string> can be used to specify further operands of the BINDER statement RESOLVE-BY-AUTOLINK in addition to the LIBRARY operand. If more than one operand is specified, they must be separated by commas as follows:

'operand1 , operand2 , ...'

The operands are directly passed through to BINDER without SDF analysis. See the BIND statement for an input example ([page 69](#)).

Automatic template instantiation

The operands specified with ADD-OPTION are currently not considered during the automatic template instantiation by the prelinker, but only during subsequent linkage with BINDER. The operands should not affect the type, number or order of the modules to be linked, since different preconditions during the prelinker and linkage runs could result in duplicates or unresolved external references.

RESOLVE = *NONE

The operand *NONE only deletes the corresponding RESOLVE chain, but does not trigger the complete action like for START-LLM-CREATION=YES. NONE is the default value of the compiler, i.e. the value after starting the compiler or after a RESET-TO-DEFAULT.

ADD-PRELINK-FILES =

This option is only relevant for automatic template instantiation by the prelinker. This option can be used to specify entire PLAM libraries or individual library modules, which are taken into account by the prelinker when determining the instances to be generated as follows:

- If the specified library contains the definition of a template entity (function or static data element), no instance that is a duplicate of that entity is generated.
- No instantiations are performed in the specified library itself. In other words, if the library requires instances for template entities (external references), these are not generated.

If the BIND statement is used to start both a prelinker and a linkage run, the libraries or modules specified with ADD-PRELINK-FILES are not considered for linkage.

Problem

The modules in the libraries PLAM.X and PLAM.Y contain references to the same template instances. If the modules of these two libraries are both preinstantiated in the BIND statement with the ACTION=*PRELINK option, this will result in duplicates.

In such cases, the prelinker must be given a hint that symbols are defined elsewhere and that no instances should hence be generated. This can be done by using the ADD-PRELINK-FILES option.

Solution

To begin with, the modules of the library PLAM.X are preinstantiated:

```
//MODIFY-BIND-PROPERTIES INCLUDE=*LIB-ELEM(LIB=PLAM.X,ELEM=*ALL)  
//BIND ACTION=*PRELINK
```

This is followed by the preinstantiation of the modules in the library PLAM.Y, but the option ADD-PRELINK-FILES is used here to inform the prelinker that the library PLAM.X needs to be considered and that no duplicates for PLAM.X should be generated.

```
//MODIFY-BIND-PROPERTIES INCLUDE=*LIB-ELEM(LIB=PLAM.Y,ELEM=*ALL),
//ADD-PRELINK-FILES=*LIBRARY(LIB=PLAM.X)
//BIND ACTION=*PRELINK
```

ADD-PRELINK-FILES = *UNCHANGED

The values specified in the MODIFY-BIND-PROPERTIES statements since the last START-LLM-CREATION=*YES apply.

ADD-PRELINK-FILES = list-poss: *LIBRARY-ELEMENT(...)

Like the INCLUDE option (or the INCLUDE-MODULES statement of BINDER), this option specifies library elements to be considered by the prelinker at preinstantiation.

LIBRARY = <filename 1..54>

<filename> assigns the name of a PLAM library.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>

<filename> is used to specify a link name for a PLAM library. The link name must already have been assigned to the library name with the ADD-FILE-LINK command before the compiler is called.

ELEMENT = *ALL(...)

All modules from the PLAM library specified with LIBRARY= are considered at preinstantiation.

VERSION = *HIGHEST-EXISTING

If the element entry does not contain a version ID, the highest possible version is used by the compiler.

VERSION = <composed-name 1..24 with-under>

The compiler uses the element with the specified version.

ELEMENT = <composed-name 1..64 with-under>(...)

<composed-name> identifies the fully-qualified name of a module from the PLAM library specified with LIBRARY=.

VERSION = *HIGHEST-EXISTING

If the element specification contains no version ID, the compiler uses the element with the highest existing version.

VERSION = <composed-name 1..24 with-under>

The compiler uses the element with the specified version.

ADD-PRELINK-FILES = list-poss: *LIBRARY(...)

The specified libraries are treated by the prelinker in the same way as RESOLVE libraries, i.e. only those library modules which could potentially be used to satisfy unresolved external references when actually linking the program are considered at preinstantiation.

LIBRARY-NAME = <filename 1..54>

<filename> assigns the name of a PLAM library.

LIBRARY = *LINK(...)**LINK-NAME = <filename 1..8>**

<filename> is used to specify a link name for a PLAM library. The link name must already have been assigned to the library name with the ADD-FILE-LINK command before the compiler is called.

ADD-PRELINK-FILES = *NONE

The operand *NONE only deletes the corresponding ADD-PRELINK-FILES chain, but does not trigger the complete action like for START-LLM-CREATION=YES. NONE is the default value of the compiler, i.e. the value after starting the compiler or after a RESET-TO-DEFAULT.

MAX-INSTANTIATE-ITER = 30 / *UNCHANGED / <integer 0..100>

This option is only relevant for automatic template instantiation by the prelinker. It defines the maximum number of prelinker iterations. The default value is 30. If a value of 0 is specified for <integer>, the number of prelinker iterations is unlimited.

TEMPLATE-DEF-LIST = *UNCHANGED / *YES / *NO

This option is used to switch a communication technology on and off between the front end and prelinker during the internal post-compilation phase via a definition list (see also [“First instantiation with the help of the definition list \(temporary repository\)” on page 248](#)).

ADD-STATEMENT = *UNCHANGED / *NONE / list-poss: <c-string 1..1800 with-low>

<c-string> can be used to specify an additional BINDER statement. Multiple BINDER statements can be specified as a list of c-strings as follows:

('statement1' , 'statement2' , '...').

The operands are directly passed through to BINDER without SDF analysis.

Automatic template instantiation

The BINDER statements specified with ADD-STATEMENT are currently not considered during the automatic template instantiation by the prelinker, but only during subsequent linkage with BINDER. The operands should not affect the type, number or order of the modules to be linked, since different preconditions during the prelinker and linkage runs could result in duplicates or unresolved external references.

RUNTIME-LANGUAGE =

This option causes the appropriate C/C++ runtime system of the CRTE to be automatically linked in accordance with the defined C or C++ language mode.

RUNTIME-LANGUAGE = *UNCHANGED

The value specified in the last MODIFY-BIND-PROPERTIES statement applies.

RUNTIME-LANGUAGE = *C

The C runtime system is linked.

The STDLIB option defines the method by which the C runtime system is linked. The RUNTIME-ENVIRONMENT option determines whether the RISC version (“SRULNK”) of the C runtime system is to be used instead of the /390 version (“SYSLNK”).

RUNTIME-LANGUAGE = *CPLUSPLUS(...)

Apart from the C runtime system, which is always required, additional C++ libraries and runtime systems of the CRTE are linked.

The STDLIB option specifies the mode in which these CRTE libraries are linked.

MODE = *ANSI / *CPP

Depending on whether the user modules were created in the extended or strict ANSI C++ mode (*ANSI) or in the Cfront C++ mode (*CPP) of the compiler, the appropriate CRTE libraries will be required when linking.

*ANSI: ANSI C++ runtime system (SYSLNK.CRTE.RTSCPP) and standard C++ library (SYSLNK.CRTE.STDCPP). The linkage of the Tools.h++ library, which can be used in the ANSI C++ modes, is controlled by means of a separate TOOLSLIB option (see [page 88](#)).

*CPP: Cfront C++ runtime system (SYSLNK.CRTE.CFCPP) and Cfront C++ library for complex math and stream-oriented I/O (SYSLNK.CRTE.CPP).

STDLIB = *UNCHANGED / *DYNAMIC / *DYNAMIC-COMPLETE / *STATIC / *NONE

This option defines how external references to the C/C++ libraries of the CRTE corresponding to the RUNTIME-LANGUAGE options are resolved.

The *DYNAMIC and *STATIC entries currently have a different effect only when linking the C runtime system.

The C++ libraries are treated identically for both the *DYNAMIC and *STATIC specifications, i.e. are always linked fully (and “statically”).

*DYNAMIC (default): All external references to the CRTE libraries are resolved. In the case of the C runtime system only an adapter module from the SYSLNK.CRTE.PARTIAL-BIND library is linked in permanently. The preloaded C runtime system is connected to the program only at runtime.

***DYNAMIC-COMPLETE:** This option represents a variant of dynamic linkage that uses the complete partial bind libraries of CRTE. The external references are resolved at runtime from the SYSLNK.CRTE.COMPL library. In the ANSI-C++ mode, i.e. when `RUNTIME-LANGUAGE=*CPLUSPLUS(MODE=*ANSI)` is specified, the SYSLNK.CRTE.CPP-COMPL library is used instead of the SYSLNK.CRTE.RTSCPP and SYSLNK.CRTE.STDCPP libraries.



In the CFRONT-C++ mode the `STDLIB=DYNAMIC-COMPLETE` option is reset to `STDLIB=DYNAMIC`. In particular, the complete partial bind method is not supported in the CFRONT-C++ mode. You will find a complete description of the complete partial bind libraries in the "CRTE" manual [4].

***STATIC:** All external references to the CRTE libraries are resolved. In the case of the C runtime system, all individual modules from the SYSLNK.CRTE library are linked in permanently.

***NONE:** The external references to the CRTE libraries remain open and are subsequently resolved either in a later linkage run (i.e. permanently with `BINDER` or the `BIND` statement) or directly at runtime via dynamic linkage with `DBL`.



The following must be observed when external references (`STDLIB=*NONE`) are left unresolved:

The standard C++ library (SYSLNK.CRTE.STDCPP) is not taken into account during the automatic template instantiation by the prelinker, which means that duplicate definitions could potentially exist. You will find more detailed information on this problem in the [section "Automatic instantiation" on page 246](#).

TOOLS_LIB = *UNCHANGED / *YES / *NO

If `*YES` is specified, the Tools.h++ library (SYSLNK.CRTE.TOOLS). When the `STDLIB=DYNAMIC-COMPLETE` option is specified, the SYSLNK.CRTE.CPP-COMPL library is used instead of the SYSLNK.CRTE.TOOLS library. If `*NO` is specified, the external references to the Tools.h++ library remain unresolved.

TEST-SUPPORT = *UNCHANGED / *YES / *NO

This option controls whether the LSD information generated at compilation is to be saved in the linked module for subsequent use with the Advanced Interactive Debugger AID. The value is passed as the `TEST-SUPPORT` operand of the `BINDER` statement `SAVE-LLM`.

LISTING = *UNCHANGED / *NONE / *SYSLST / <filename 1..54>

This option, which is analogous to the `MAP` operand of the `BINDER` statement `SAVE-LLM`, can be used to request the standard listings of `BINDER`. These listings are output to the system file `SYSLST` or to a cataloged file specified with `<filename>`.

Interaction between the MODIFY-BIND-PROPERTIES and BIND statements

Several MODIFY-BIND-PROPERTIES statements may be collectively involved in a linkage run. The libraries specified with the RESOLVE, INCLUDE and ADD-PRELINK-FILES options, for example, are collected and remain in effect even for subsequent BIND statements so long as the option START-LLM-CREATION is set to *YES.

START-LLM-CREATION is automatically set to *YES only on the first call to the MODIFY-BIND-PROPERTIES statement after calling the compiler or after a RESET-TO-DEFAULT statement. Starting with the second MODIFY statement, the default value is *NO.

A list of libraries within a RESOLVE option has the same effect as a list of libraries within a RESOLVE-BY-AUTOLINK statement of BINDER. Additional MODIFY-BIND-PROPERTIES statements with a RESOLVE option are issued as independent RESOLVE-BY-AUTOLINK statements of BINDER and are processed separately.

MODIFY-CIF-PROPERTIES

Aliases: SET-CIF-PROPERTIES
 MODIFY-INFO-FILE-PROPERTIES
 SET-INFO-FILE-PROPERTIES

This statement can be used to instruct the compiler to create a CIF (Compiler Information File), which may include information on some or all compiler listings. For each compiled source program, the CIF is written to a separate file, which can then be processed further with the global listing generator (see [page 156ff](#) for details).

MODIFY-CIF-PROPERTIES

```

CONSUMER = *UNCHANGED / *NONE / *ALL / list-poss(9): *BY-LISTING-PROPERTIES / *OPTIONS /
  *SOURCE / *PREPROCESSING-RESULT / *DATA-ALLOCATION-MAP /
  *CROSS-REFERENCE / *PROJECT-INFORMATION / *ASSEMBLER-CODE /
  *SUMMARY

,INCLUDE-INFORMATION = *UNCHANGED / *NONE / *ALL / *USER-INCLUDES-ONLY

,OUTPUT = *UNCHANGED / *NONE / *STD-FILE / *SOURCE-LOCATION / <filename 1..54> /
  <posix-pathname> / *LIBRARY-ELEMENT(...)

*LIBRARY-ELEMENT(...)
  | LIBRARY = *STD-LIBRARY / *SOURCE-LIBRARY / <filename 1..54>
  | ELEMENT = *STD-ELEMENT(...) / <composed-name 1..64 with-under>(…)
  | *STD-ELEMENT(...)
  | | VERSION = *UPPER-LIMIT / <composed-name 1..24 with-under>
  | <composed-name 1..64 with-under>(…)
  | | VERSION = *UPPER-LIMIT / <composed-name 1..24 with-under>

```

CONSUMER = *UNCHANGED

The value specified in the last MODIFY-CIF-PROPERTIES statement applies.

CONSUMER = *NONE

No permanent CIF is created.

When local listings are requested (with MODIFY-LISTING-PROPERTIES), a temporary CIF is created (with prefix #T) to generate these listings. This CIF is deleted at TASK end.

CONSUMER = *ALL

The created CIF contains information on all listings that can be generated, depending on which compiler components (PREPROCESS, CHECK-SYNTAX, COMPILE) are run.

CONSUMER = list-poss(9): *BY-LISTING-PROPERTIES / *OPTIONS / *SOURCE / *PREPROCESSING-RESULT / *DATA-ALLOCATION-MAP / *CROSS-REFERENCE / *PROJECT-INFORMATION / *ASSEMBLER-CODE / *SUMMARY

A CIF containing information on the specified listings is created.

*BY-LISTING-PROPERTIES: CIF information is created for all listings requested with the MODIFY-LISTING-PROPERTIES statement.

INCLUDE-INFORMATION = *UNCHANGED / *NONE / *ALL / *USER-INCLUDES-ONLY

This option can be used to control whether and from which header files CIF information is to be generated for the source, preprocessor and cross-reference listings. By default, only the user-defined headers and not the standard headers are taken into account.

OUTPUT = *UNCHANGED

The value specified in the last MODIFY-CIF-PROPERTIES statement applies.

OUTPUT = *NONE

No permanent CIF is created.

OUTPUT = *STD-FILE

The CIF is written to a cataloged BS2000 file. The name of this file is derived from the name of the source program as follows:

Source	*SYSDTA	BS2000 file	PLAM library	POSIX file
Default name	CSTDCIF.CIF	file.CIF	lib-elem.CIF	file.CIF

If the source program is located in a PLAM library, the library and element name of the source are combined with a hyphen (lib-elem) and used in the default file name. The rules by which the compiler constructs default names are described in detail in the [section "Default names for output containers" on page 48](#).

OUTPUT = *SOURCE-LOCATION

The output destination and name of the CIF are derived from the location and name of the source program as follows:

Source	*SYSDTA	BS2000 file	PLAM library	POSIX file
Output destination	BS2000 file	BS2000 file	Library of source	Directory of source
Default name	CSTDCIF.CIF	file.CIF	elem.CIF (type H)	file.cif

The rules by which the compiler constructs default names are described in detail in the [section "Default names for output containers" on page 48](#).

OUTPUT = <filename 1..54>

The CIF is written to a cataloged BS2000 file with the specified name. This entry is invalid when compiling multiple source programs.

OUTPUT = <posix-pathname>

The CIF is written to a POSIX file.

Both a file name and a directory are permitted as <posix-pathname>. See [page 34](#) for a description of the term <posix-pathname>.

When a file name is specified, the CIF is stored under this name. Specification of a file name is invalid when compiling multiple source programs with one statement.

When a directory name *dir* is specified, the CIF for each compiled source program is written under the default name *sourcefile.cif* to the directory *dir* (see also [section “Default names for output containers” on page 48](#)).

The directories specified with <posix-pathname> must already exist.

CIF files can be processed further in the POSIX subsystem with the global listing generator `cclistgen`.

OUTPUT = *LIBRARY-ELEMENT(...)

This option specifies the PLAM library (LIBRARY=) and the element name (ELEMENT=) under which the CIF is to be stored. The elements are stored as elements of type H.

LIBRARY = *STD-LIBRARY

The CIF is written to the library SYS.PROG.LIB by default.

LIBRARY = *SOURCE-LIBRARY

The CIF is written to the PLAM library which contains the source program.

The *SOURCE-LIBRARY specification is invalid if the source program is read from a cataloged BS2000 file, a POSIX file or via SYSDTA.

LIBRARY = <filename 1..54>

The CIF is written to a PLAM library with the specified name.

ELEMENT = *STD-ELEMENT(...)

By default, the element name of the CIF is derived from the name of the source program as follows:

Source	*SYSDTA	BS2000 file	PLAM library	POSIX file
Default name	CSTDCIF.CIF	file.CIF	elem.CIF	file.CIF

The rules by which the compiler constructs default names are described in detail in the [section “Default names for output containers” on page 48](#).

VERSION = *UPPER-LIMIT

If the element entry does not contain a version ID, the highest possible version is used by the compiler.

VERSION = <composed-name 1..24 with-under>

The CIF is written to the element with the specified version ID.

ELEMENT = <composed-name 1..64 with-under>(…)

The CIF is written to a library element (type H) with the specified name. This specification is invalid compiling multiple source programs.

**VERSION = *UPPER-LIMIT /
<composed-name 1..24 with-under>**

The version can be specified as described above for
ELEMENT=*STD-ELEMENT(…).

MODIFY-DIAGNOSTIC-PROPERTIES

Alias: SET-DIAGNOSTIC-PROPERTIES

This statement can be used to create a user-defined list of compiler messages and to specify various output destinations for it. In addition, it can be used to define error message weights and error conditions for aborting compilation.

MODIFY-DIAGNOSTIC-PROPERTIES

```

MINIMAL-MSG-WEIGHT = *UNCHANGED / *NOTE / *WARNING / *ERROR / *FATAL
,CHANGE-MSG-WEIGHT = *UNCHANGED / list-poss: *NOTE(...) / *WARNING(...) / *ERROR(...)
  *NOTE(...)
    | MSGID = list-poss: <alphanum-name 7..7>
  *WARNING(...)
    | MSGID = list-poss: <alphanum-name 7..7>
  *ERROR(...)
    | MSGID = list-poss: <alphanum-name 7..7>
,SUPPRESS-MSG = *UNCHANGED / *NONE / list-poss: *USE-BEFORE-SET / <alphanum-name 7..7>
,MAX-ERROR-NUMBER = *UNCHANGED / 50 / <integer 1..255>
,ANSI-VIOLATIONS = *UNCHANGED / *WARNING / *ERROR
,SHOW-COLUMN = *UNCHANGED / *YES / *NO
,SHOW-INCLUDES = *UNCHANGED / *YES / *NO
,VERBOSE = *UNCHANGED / *NO / list-poss: *VERSION / *MESSAGES
,GENERATE-ETR-FILE = *UNCHANGED / *NO / *ALL-INSTANTIATIONS / *ASSIGNED-INSTANTIATIONS
,OUTPUT = *UNCHANGED / list-poss(10): *SYSOUT / *SYSLST / *STD-FILE / *SOURCE-LOCATION /
  *TO-LISTING / <filename 1..54> / <posix-pathname> / *LIBRARY-ELEMENT(...)
  *LIBRARY-ELEMENT(...)
    | LIBRARY = *STD-LIBRARY / *SOURCE-LIBRARY / <filename 1..54> / *LINK(...)
      *LINK(...)
        | LINK-NAME = <filename 1..8>
      ,ELEMENT = *STD-ELEMENT(...) / <composed-name 1..64 with-under>(…)
        *STD-ELEMENT(...)
          | VERSION = *UPPER-LIMIT / <composed-name 1..24 with-under>
          <composed-name 1..64 with-under>(…)
          | VERSION = *UPPER-LIMIT / <composed-name 1..24 with-under>

```

MINIMAL-MSG-WEIGHT = *UNCHANGED / *NOTE / *WARNING / *ERROR / *FATAL

This option can be used to specify the minimum message weight for which compiler messages are to be included in the message listing.

CHANGE-MSG-WEIGHT = *UNCHANGED / list-poss: *NOTE(...) / *WARNING(...) / *ERROR(...)

These options can be used to change the default message weights for diagnostic messages that are output by the frontend compiler (beginning with CFE). Notes can be upgraded to the message weight ERROR, and warnings can be downgraded to NOTE or upgraded to ERROR. Errors can be downgraded to the weight of a NOTE, but only if they were flagged with an asterisk in the original message: [**ERROR*]. The message weight for fatal errors cannot be changed.

MSGID = list-poss: <alphanum-name 7..7>

<alphanum-name> is the 7-digit message key of the compiler message for which the message weight is to be changed.

SUPPRESS-MSG = *UNCHANGED / *NONE / list-poss: *USE-BEFORE-SET / <alphanum-name 7..7>

This option can be used to suppress compiler messages with the message weights NOTE and WARNING, thus restricting the listed messages to the ones most important to the user.

**USE-BEFORE-SET*: The output of warnings is suppressed if local `auto` variables are used in the program before being assigned a value.

<alphanum-name 7..7>: is the message key of a compiler message (NOTE or WARNING) to be suppressed.

Example

```
SUPP-MSG=(CFE2802,CFE9095)
```

MAX-ERROR-NUMBER = *UNCHANGED / 50 / <integer 1..255>

This option can be used to specify after how many errors the compiler run is to be aborted (notes and warnings are counted separately).

50: The compiler aborts any compilation in which more than 49 errors occur.

<integer 1..255>: specifies after how many errors the compiler run is to be aborted.

ANSI-VIOLATIONS = *UNCHANGED / *WARNING / *ERROR

This option can be meaningfully used only in the strict ANSIC/C++ modes.

*WARNING is the default setting, which means that warnings are issued on detecting the use of language constructs which deviate from the ANSI/ISO C or C++ standards, but which are not a serious violation of the language rules defined therein (e.g. implementation-specific language extensions; see also [section “Extensions to ANSI/ISO C” on page 219](#) and [section “Extensions to ANSI/ISO-C++” on page 260](#)).

If *ERROR is specified here, errors are reported in such cases.
Serious violations automatically result in errors.

SHOW-COLUMN = *UNCHANGED / *YES / *NO

This option determines whether the diagnostic messages of the compiler are generated in short or long form.

*YES: The original source program line is shown with the error location marked (with ^) in addition to the diagnostic message itself.

*NO: The marked source program line is not output.

SHOW-INCLUDES = *UNCHANGED / *YES / *NO

If *YES is specified, the names of header files used by the preprocessor are output.

VERBOSE = *UNCHANGED / *NO / list-poss: *VERSION / *MESSAGES

*VERSION: Details on each active compiler component (component code, version, copyright) are output. In link processes, the version of the CRTE used and the names of the CRTE libraries are also output.

*MESSAGES: This value is currently meaningful only in the ANSI C++ modes. It causes additional information on automatic template instantiation by the prelinker to be written to SYSOUT.

GENERATE-ETR-FILE = *UNCHANGED / *NO / *ALL-INSTANTIATIONS / *ASSIGNED-INSTANTIATIONS

This option can be used to create an ETR file (ETR=Explicit Template Request) which contains the instantiation statements for the templates used (see [section “Generating explicit template instantiation statements \(ETR files\)” on page 252](#)). The file name is derived from the name of the object file and has the suffix .etr.

The default for this option is *NO. This does not create an ETR file. If *ALL-INSTANTIATIONS is specified, all used instances are recorded. If *ASSIGNED-INSTANTIATIONS is specified only those instances assigned to this file by the prelinker, and are thus contained in the ii file.

OUTPUT = *UNCHANGED / list-poss(10): *SYSOUT / *SYSLST / *STD-FILE / *SOURCE-LOCATION / <filename 1..54> / <posix-pathname> / *LIBRARY-ELEMENT(...)

The OUTPUT option can be used to request the concurrent output of the message listing at different output destinations.



In addition to the output destinations specified in this option information messages of the compiler (messages without message weight) are also output to SYSOUT. All other messages are only output to the specified output destinations.

OUTPUT = *SYSOUT

Compilation messages are written to the terminal (system file SYSOUT) by default.

OUTPUT = *SYSLST

The message listing is written to the temporary system file SYSLST and is sent from there to the printer at the end of the task (at LOGOFF).

OUTPUT = *STD-FILE

The message listing is written to a cataloged BS2000 file. The name of this file is derived from the name of the source program as follows:

Source	*SYSDTA	BS2000 file	PLAM library	POSIX file
Default name	CSTDDIAG.DIAG	file.DIAG	lib-elem.DIAG	file.DIAG

If the source program is located in a PLAM library, the library and element name of the source are combined with a hyphen (lib-elem) and used in the default file name. The rules by which the compiler constructs default names are described in detail in the [section "Default names for output containers" on page 48](#).

OUTPUT = *SOURCE-LOCATION

The output destination and name of the message listing are derived from the location and name of the source program as follows:

Source	*SYSDTA	BS2000 file	PLAM library	POSIX file
Output destination	BS2000 file	BS2000 file	Library of source	Directory of source
Default name	CSTDDIAG.DIAG	file.DIAG	elem.DIAG (type D)	file.diag

The rules by which the compiler constructs default names are described in detail in the [section "Default names for output containers" on page 48](#).

OUTPUT = *TO-LISTING

The diagnostic output is appended to the end of the listing file as an own sublisting. The message list is sorted according to the message weight but information messages (messages without message weight) are not included in the message list. The name of the output file is defined by the value of the option OUTPUT in MODIFY-LISTING-PROPERTIES. The restrictions given there for ANSI C++ must be taken into account.

OUTPUT = <filename 1..54>

The message listing is written to a cataloged BS2000 file with the specified name. This specification is not meaningful when compiling multiple source programs, since the file is overwritten in each case.

OUTPUT = <posix-pathname>

The message listing is written to the POSIX file system.

Both a file name and a directory are permitted as <posix-pathname>. See [page 34](#) for a description of the term <posix-pathname>.

When a file name is specified, the message listing is stored under this name.

When a directory name *dir* is specified, the message listing for each compiled source program is written under the default name *sourcefile.diag* to the directory *dir* (see also [section "Default names for output containers" on page 48](#)).

The directories specified with <posix-pathname> must already exist. Note that the specification of a file name is not meaningful when compiling multiple source programs, since the file is overwritten in each case.

OUTPUT = *LIBRARY-ELEMENT(...)

This option specifies the PLAM library (LIBRARY=) and the element name (ELEMENT=) under which the message listing is to be stored. The elements are saved as elements of type D.

LIBRARY = *STD-LIBRARY

The message listing is stored in the library SYS.PROG.LIB by default.

LIBRARY = *SOURCE-LIBRARY

The message listing is written to the PLAM library which contains the source program. The *SOURCE-LIBRARY specification is invalid if the source program is read from a cataloged BS2000 file, a POSIX file or via SYSDDTA.

LIBRARY = <filename 1..54>

The message listing is stored in a PLAM library with the specified name.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>

A link name can also be specified instead of the cataloged library name. This link name must already have been assigned to the PLAM library (with the ADD-FILE-LINK command) before the compiler is called.

ELEMENT = *STD-ELEMENT(...)

By default, the element name of the message listing is derived from the name of the source program as follows:

Source	*SYSDDTA	BS2000 file	PLAM library	POSIX file
Default name	CSTDDIAG.DIAG	file.DIAG	elem.DIAG	file.DIAG

The rules by which the compiler constructs default names are described in detail in the [section “Default names for output containers” on page 48](#).

VERSION = *UPPER-LIMIT

If the element entry does not contain a version ID, the highest possible version is used by the compiler.

VERSION = <composed-name 1..24 with-under>

The message listing is stored in the element with the specified version ID.

ELEMENT = <composed-name 1..64 with-under>(…)

The message listing is written to a library element (type D) with the specified name. Note that the specification of an element name is not meaningful when compiling multiple source programs, since the element is overwritten in each case.

VERSION = *UPPER-LIMIT / <composed-name 1..24 with-under>

The version can be specified as described above for
ELEMENT=*STD-ELEMENT(…).

MODIFY-INCLUDE-LIBRARIES

Aliases: MODIFY-INCLUDE-SEARCH
 SET-INCLUDE-LIBRARIES
 SET-INCLUDE-SEARCH

This statement specifies which include (or header) libraries and file directories are to be searched by the compiler. It also defines the order in which these libraries and directories are searched.

MODIFY-INCLUDE-LIBRARIES

```

USER-INCLUDE-LIBRARY = *UNCHANGED / list-poss: *SOURCE-LIBRARY / *STANDARD-LIBRARY /
    <filename 1..54> / <posix-pathname> / *LINK(...)

    *LINK(...)
    | LINK-NAME = <filename 1..8>

,STD-INCLUDE-LIBRARY = *UNCHANGED / list-poss: *USER-INCLUDE-LIBRARY /
    *STANDARD-LIBRARY / *SOURCE-LIBRARY / <filename 1..54> /
    <posix-pathname> / *LINK(...)

    *LINK(...)
    | LINK-NAME = <filename 1..8>

,CURRENT-LIBRARY = *UNCHANGED / *YES / *NO
  
```

USER-INCLUDE-LIBRARY =

This option can be used to assign PLAM libraries or POSIX directories which contain user-defined header files (requested with `#include "..."`). Depending on the setting of the **CURRENT-LIBRARY** (see [page 102](#)) option, the library or directory containing the source or header file listed in the include directive `#include "..."` is searched first, i.e., before the libraries and directories specified with **USER-INCLUDE-LIBRARY**.

If the **USER-INCLUDE-LIBRARY** option is not specified, the following default is used:

```
USER-INCLUDE-LIBRARY = (*SOURCE-LIBRARY, *STANDARD-LIBRARY)
```

If header files are enclosed in quotes in the `#include` directive (`#include "name"`), the compiler will search for these elements in the library or directory of the source program first, and then in the CRTE libraries containing the standard header files. The CRTE library `$.SYSLIB.CRTE` is assigned for the search in all C language modes and in the ANSI C++ modes; in Cfront C++ mode, the CRTE libraries `$.SYSLIB.CRTE.CPP` and `$.SYSLIB.CRTE` are also searched.

If the option is specified, the above default is deactivated, and the compiler searches only in the libraries/directories explicitly specified. In other words, if the source program library or directory and the CRTE libraries are to be searched for header files, the *SOURCE-LIBRARY or *STANDARD-LIBRARY options must be explicitly specified.

The libraries/directories are searched in the order in which they are specified.

USER-INCLUDE-LIBRARY = *UNCHANGED

The values specified in the last MODIFY-INCLUDE-LIBRARY statement apply. If no values have been changed since starting the compiler, the default settings of the compiler (*SOURCE-LIBRARY, *STANDARD-LIBRARY) apply.

USER-INCLUDE-LIBRARY = *SOURCE-LIBRARY

The library or directory containing the source program is searched for the required header file. The *SOURCE-LIBRARY specification has no effect if the source program is in a cataloged BS2000 file or, in general, when source programs are entered via SYSDTA (see also [page 76](#)).

USER-INCLUDE-LIBRARY = *STANDARD-LIBRARY

The CRTE library \$.SYSLIB.CRTE is assigned for the search in all C language modes and in the ANSI C++ modes; in the Cfront C++ mode, the CRTE library \$.SYSLIB.CRTE.CPP is searched first, followed by the library \$.SYSLIB.CRTE.

USER-INCLUDE-LIBRARY = <filename 1..54>

<filename> identifies the name of the PLAM library in which the required user-defined header files are to be searched.

USER-INCLUDE-LIBRARY = <posix-pathname>

<posix-pathname> designates the name of the POSIX directory in which the required user-defined header files are to be searched. See [page 34](#) for a description of the term <posix-pathname>.

USER-INCLUDE-LIBRARY = *LINK(...)**LINK-NAME = <filename 1..8>**

A link name can also be specified instead of a cataloged PLAM library name.

<filename> designates the link name of the assigned header library. This link name must already have been assigned to the PLAM library (with the ADD-FILE-LINK command) before the compiler is called.

STD-INCLUDE-LIBRARY =

This option can be used to specify the PLAM libraries and POSIX directories which contain the required standard headers (requested with `#include <...>`).

STD-INCLUDE-LIBRARY = *UNCHANGED

The values specified in the last MODIFY-INCLUDE-LIBRARY statement apply. If no values have been changed since starting the compiler, the default settings of the compiler (*USER-INCLUDE-LIBRARY, *STANDARD-LIBRARY) apply.

STD-INCLUDE-LIBRARY = *USER-INCLUDE-LIBRARY

The search order for standard header files is derived from specifications in the USER-INCLUDE-LIBRARY option. Only the libraries/directories which are explicitly specified in that option are taken into account, not the entries for *SOURCE-LIBRARY and *STANDARD-LIBRARY.

STD-INCLUDE-LIBRARY = *STANDARD-LIBRARY

The CRTE library \$.SYSLIB.CRTE is assigned for the search in all C language modes and in the ANSI C++ modes; in the Cfront C++ mode, the CRTE library \$.SYSLIB.CRTE.CPP is searched first, followed by the library \$.SYSLIB.CRTE.

See also the notes on standard header files for POSIX library functions ([page 103](#)).

STD-INCLUDE-LIBRARY = <filename 1..54>

<filename> designates the name of the PLAM library that is to be searched for the required standard header files.

STD-INCLUDE-LIBRARY = <posix-pathname>

<posix-pathname> designates the name of the POSIX directory in which the required header files are to be searched. See [page 34](#) for a description of the term <posix-pathname>.

This specification is not meaningful for standard headers of the CRTE, since they are obtained from the CRTE libraries.

STD-INCLUDE-LIBRARY = *LINK(...)**LINK-NAME = <filename 1..8>**

A link name can also be specified instead of the cataloged PALM library name.

<filename> designates the link name of the assigned header library. This link name must already have been assigned to the PLAM library (with the ADD-FILE-LINK command) before the compiler is called.

CURRENT-LIBRARY = *UNCHANGED / *YES / *NO

This option determines the search for header files requested with the `#include "..."` directive.

*YES: By default, when searching for headers, the library or directory of the source or header containing the `#include` directive is searched first, followed by the directories/libraries specified with the USER-INCLUDE-LIBRARY option. This corresponds to the behavior of the compiler in POSIX.

If the source program is contained in a cataloged BS2000 file, header files are searched for in the cataloged BS2000 files.

*NO: If *NO is set, only the directories/libraries specified in the USER-INCLUDE-LIBRARY option are searched. This corresponds to the behavior of the earlier C and C++ V2.2 compilers.

Standard header files for POSIX library functions

The standard header files required for the use of the POSIX library functions are not contained in the library \$.SYSLNK.CRTE. They are located in the library \$.SYSLIB.POSIX-HEADER, which is supplied with the product BS2000/OSD BC. This library must always be specified in addition to the library \$.SYSLNK.CRTE if the program uses POSIX functions. Furthermore, the `_OSD_POSIX` directive must be set before the occurrence of the first `#include` directive in the program. This is guaranteed, for example, if the `DEFINE` option of the `MODIFY-SOURCE-PROPERTIES` statement is used for the definition at compilation.

Examples of the MODIFY-INCLUDE-LIBRARY statement

Example 1

The source program is located in a PLAM library named PLAM.SOURCE, and extended ANSI C is set as the language mode.

The source program contains the directive

```
#include "incl.h"
```

The user makes the following entries:

```
//MODIFY-INCLUDE-LIBRARIES USER-INCL-LIB=(*STANDARD-LIBRARY,LIB1,-
//*LINK(LIB2),*SOURCE-LIBRARY,'/home/user-incl')
```

Since the `CURRENT-LIBRARY` option is not specified, `CURRENT-LIBRARY=*YES` applies.

Search procedure:

The following libraries/directories are searched in sequence for the "incl.h" header:

PLAM.SOURCE	(Due to the option <code>CURRENT-LIBRARY=*YES</code> , the library of the source program containing the <code>#include</code> directive)
\$.SYSLIB.CRTE	(Standard header library)
LIB1	(Library with the name LIB1)
LIB2	(Library with the link name LIB2)
PLAM.SOURCE	(Source program library)
/home/user-incl	(POSIX directory with the name /home/user-incl)

Example 2

The source program is located in a PLAM library named PLAM.SRC, and Cfront C++ is set as the language mode.

The user makes the following entries:

```
//MODIFY-INCLUDE-LIBRARIES USER-INCLUDE-LIBRARY = ($XYZ.LIB,-
//*SOURCE-LIBRARY, LIB1),STD-INCLUDE-LIBRARY = (*USER-INCLUDE-LIBRARY,-
//*STANDARD-LIBRARY),CURRENT-LIBRARY = *NO
```

In this case, the search for header files will proceed as follows:

<code>#include "..."</code>	<code>#include <...></code>
<code>\$XYZ.LIB</code>	<code>\$XYZ.LIB</code>
<code>PLAM.SRC</code>	<code>LIB1</code>
<code>LIB1</code>	<code>\$.SYSLIB.CRTE.CPP</code>
	<code>\$.SYSLIB.CRTE</code>

Example 3

The source program is located in the POSIX directory /home/xy/src, and extended ANSI C is set as the language mode.

The user makes the following entries:

```
//MODIFY-INCLUDE-LIBRARIES -
//USER-INCLUDE-LIBRARY=(*SOURCE-LIBRARY, '/home/xy/inc11', *STANDARD-LIBRARY),-
//STD-INCLUDE-LIBRARY=(*STANDARD-LIBRARY, $.SYSLIB.POSIX-HEADER,-
//*USER-INCLUDE-LIBRARY, '/home/xy/inc12),CURRENT-LIBRARY=*NO
```

In this case, the search for header files will proceed as follows:

<code>#include "..."</code>	<code>#include <...></code>
<code>/home/xy/src</code>	<code>\$.SYSLIB.CRTE</code>
<code>/home/xy/inc1</code>	<code>\$.SYSLIB.POSIX-HEADER</code>
<code>\$.SYSLIB.CRTE</code>	<code>/home/xy/inc1</code>
	<code>/home/xy/inc2</code>

MODIFY-LISTING-PROPERTIES

Alias: SET-LISTING-PROPERTIES

This statement can be used to select which compiler listings are to be generated by the compiler. It can also be used to define the layout of the listings and their output destinations. The structure of the compiler listing is explained in the [section “Description of listings” on page 289ff](#) with the help of examples.

MODIFY-LISTING-PROPERTIES

```

OPTIONS = *UNCHANGED / *YES / *NO
,SOURCE = *UNCHANGED / *NO / [*YES](...)
    *YES(...)
        | MINIMAL-MSG-WEIGHT = *NOTE / *WARNING / *ERROR / *FATAL
,PREPROCESSING-RESULT = *UNCHANGED / *NO / [*YES](...)
    *YES(...)
        | COMMENTS = *YES / *NO
,DATA-ALLOCATION-MAP = *UNCHANGED / *NO / [*YES](...)
    *YES(...)
        | STRUCTURE-LEVEL = *UNCHANGED / *NONE / *MAX / <integer 0..256>
,CROSS-REFERENCE = *UNCHANGED / *NO / [*YES](...)
    *YES(...)
        | PREPROCESSING-INFO = *YES / *NO
        | ,TYPES = *YES / *NO
        | ,VARIABLES = *YES / *NO
        | ,FUNCTIONS = *YES / *NO
        | ,LABELS = *YES / *NO
        | ,TEMPLATES = *YES / *NO
        | ,ORDER = *STD / list-poss(6): *PREPROCESSING-INFO / *TYPES / *VARIABLES /
            *FUNCTIONS / *LABELS / *TEMPLATES
,PROJECT-INFORMATION = *UNCHANGED / *YES / *NO
,ASSEMBLER-CODE = *UNCHANGED / *YES / *NO
,SUMMARY = *UNCHANGED / *YES / *NO

```

```

,LAYOUT = *UNCHANGED / *FOR-NORMAL-PRINT(...) / *FOR-ROTATION-PRINT(...)
  *FOR-NORMAL-PRINT(...)
    | LINE-SIZE = *UNCHANGED / *STD / <integer 120..255>
    | ,LINES-PER-PAGE = *UNCHANGED / *STD / <integer 11..255>
  *FOR-ROTATION-PRINT(...)
    | LINE-SIZE = *UNCHANGED / *STD / <integer 120..255>
    | ,LINES-PER-PAGE = *UNCHANGED / *STD / <integer 11..255>
,INCLUDE-INFORMATION = *UNCHANGED / *NONE / *ALL / *USER-INCLUDES-ONLY
,LISTING-PRAGMAS = *UNCHANGED / *IGNORED / *INTERPRETED / *SELECT(...)
  *SELECT(...)
    | PAGE = *YES / *NO
    | ,TITLE = *YES / *NO
    | ,SPACE = *YES / *NO
    | ,LIST = *YES / *NO
,INITIAL-TITLE-TEXT = *UNCHANGED / *NONE / <c-string 1..256 with-low>
,OUTPUT = *UNCHANGED / *SYSLST / *SYSOUT / *STD-FILE / *SOURCE-LOCATION /
  <filename 1..54> / <posix-pathname> / *LIBRARY-ELEMENT(...)
  *LIBRARY-ELEMENT(...)
    | LIBRARY = *STD-LIBRARY / *SOURCE-LIBRARY / <filename 1..54> / *LINK(...)
    | *LINK(...)
    | | LINK-NAME = <filename 1..8>
    | ,ELEMENT = *STD-ELEMENT(...) / <composed-name 1..64 with-under>(…)
    | *STD-ELEMENT(...)
    | | VERSION = *UPPER-LIMIT / *INCREMENT / <composed-name 1..24 with-under>
    | | <composed-name 1..64 with-under>(…)
    | | VERSION = *UPPER-LIMIT / *INCREMENT / <composed-name 1..24 with-under>

```

OPTIONS = *UNCHANGED / *YES / *NO

*YES: The compiler creates a comprehensive listing of all predefined and user-specified compiler options.

SOURCE = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

SOURCE = *NO

No source/error listing is generated.

SOURCE = *YES(...)

A source/error listing is generated (not for PREPROCESS).

MINIMAL-MSG-WEIGHT = *NOTE / *WARNING / *ERROR / *FATAL

This operand can be used to specify the minimum message weight for which error messages are to be included in the source/error listing.

Warning:

This suboption is used to limit the number of messages output as compared to the similar MODIFY-DIAGNOSTIC-PROPERTIES suboption, e.g. from WARNING to ERROR.

If MINIMAL-MSG-WEIGHT = *WARNING (default setting) was specified for MODIFY-DIAGNOSTIC-PROPERTIES, you will not be able to output notes with MODIFY-LISTING-PROPERTIES.

Examples

1. To output error messages in the source listing with a message weight of NOTE:

```
MODIFY-DIAGNOSTIC-PROPERTIES MINIMAL-MSG-WEIGHT=*NOTE
MODIFY-LISTING-PROPERTIES SOURCE=*YES(MINIMAL-MSG-WEIGHT=*NOTE)
```

2. Write all error messages with a minimum message weight of NOTE to the system file SYSOUT, and error messages with a minimum message weight of WARNING to the source listing:

```
MODIFY-DIAGNOSTIC-PROPERTIES MINIMAL-MSG-WEIGHT=*NOTE
MODIFY-LISTING-PROPERTIES SOURCE=*YES(MINIMAL-MSG-WEIGHT=*WARNING)
[Default]
```



When the value given in MAX-ERROR-NUMBER is reached, no further source program information will be output in the source/error listing. In this case the listing can no longer be used as a reliable guide to current error status.

PREPROCESSING-RESULT = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

PREPROCESSING-RESULT = *NO

The compiler does not generate a preprocessor listing.

PREPROCESSING-RESULT = *YES(...)

The compiler generates a preprocessor listing.

COMMENTS = *YES / *NO

Comments from the source file are included in the preprocessor listing (can be suppressed with *NO).

DATA-ALLOCATION-MAP = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

DATA-ALLOCATION-MAP = *NO

The compiler does not generate a map listing.

DATA-ALLOCATION-MAP = *YES(...)

The compiler generates a map listing (only for COMPILE).

STRUCTURE-LEVEL = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

STRUCTURE-LEVEL = *NONE

Structure elements are not included in the map listing.

STRUCTURE-LEVEL = *MAX

Only the structure elements up to a maximum nesting level (256) are included in the map listing.

STRUCTURE-LEVEL = <integer 0..256>

Only the structure elements up to the nesting level specified by <integer> are included in the map listing. If a nesting level of 0 is specified, no structure elements are output (equivalent to STRUCTURE-LEVEL=*NONE).

Structure elements are represented with indentation and bracketing {}. Elements with a nesting level of 16 or higher are indented no further.

CROSS-REFERENCE = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

CROSS-REFERENCE = *NO

The compiler does not create a cross-reference listing.

CROSS-REFERENCE = *YES(...)

The compiler creates a cross-reference listing (not for PREPROCESS). This listing always contains a FILETABLE section with the names of all files, libraries, and elements that are used as sources by the compiler.

The cross-reference listing is created for each compilation unit, i.e. for the local module. If a global (multi-module) cross-reference listing is required, the MODIFY-CIF-PROPERTIES statement can be used to create CIF information for subsequent processing with the global listing generator.

PREPROCESSING-INFO = *YES / *NO

The cross-reference listing may optionally include a list of names processed by the preprocessor.

TYPES = *YES / *NO

The cross-reference listing may optionally include a list of user-defined types (typedefs, structure, union, class and enumeration types).

VARIABLES = *YES / *NO

The cross-reference listing contains a list of variables (can be suppressed with *NO).

FUNCTIONS = *YES / *NO

The cross-reference listing contains a list of functions (can be suppressed with *NO).

LABELS = *YES / *NO

The cross-reference listing contains a list of labels (can be suppressed with *NO).

TEMPLATES = *YES / *NO

The cross-reference listing optionally contains a list of templates (only for ANSI C++ compilations).

ORDER = *STD / list-poss(6): *PREPROCESSING-INFO / *TYPES / *VARIABLES / *FUNCTIONS / *LABELS / *TEMPLATES

This option specifies the order in which the individual parts of the cross-reference listing are shown.

*STD: The default is in the order shown after list-poss above.

PROJECT-INFORMATION = *UNCHANGED / *YES / *NO

*YES: The compiler creates a project listing (only for COMPILE) showing the names originally used in the source program and corresponding names internally generated by the compiler for the linkage editor. This is important especially in the case of C++ compilations because the generated names of functions also contain the types of their parameters in code form.

The project listing is created for each compilation unit, i.e. for the local module. If a global (multi-module) project listing is required, the MODIFY-CIF-PROPERTIES statement can be used to create CIF information for subsequent processing with the global listing generator.

ASSEMBLER-CODE = *UNCHANGED / *YES / *NO

*YES: The compiler generates an object code listing (only for COMPILE).

SUMMARY = *UNCHANGED / *YES / *NO

*YES: The compiler creates a listing containing statistical data on the compiler run.

LAYOUT =

This option can be used to define the page length (number of lines per page) and the page width (number of characters per line) for the compiler listings.

If a line width of 120 characters is selected, all the listings will have narrower headers and footers. Text lines are wrapped only in the table listings (option, cross-reference and map listings). Overlong text lines in the source, preprocessor and object code listings may be truncated when the listings are printed.

When a BS2000 output file is specified, the first column of every line is reserved to control the line feed.

When the output is sent to a POSIX file, the appropriate POSIX control characters for line and page feeds are generated. The result is that the line length in the POSIX output file is up to 3 characters larger than the selected line width specification.

LAYOUT = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

LAYOUT = *FOR-NORMAL-PRINT(...)**LINE-SIZE = *UNCHANGED**

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

LINE-SIZE = *STD

132 characters per line are output.

LINE-SIZE = <integer 120..255>

120 to 255 characters per line are output.

LINES-PER-PAGE = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

LINES-PER-PAGE = *STD

64 lines per page are printed.

LINES-PER-PAGE = <integer 11..255>

11 to 255 lines are printed per page.

The lower limit is fixed at 11 lines so that at least the listing header and footer and one line of text can be printed.

LAYOUT = *FOR-ROTATION-PRINT(...)

In order to print such listings, the ROTATION parameter must be specified in the PRINT-FILE command.

LINE-SIZE = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

LINE-SIZE = *STD

120 characters per line are output.

LINE-SIZE = <integer 120..255>

120 to 255 characters per line are output.

LINES-PER-PAGE = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

LINES-PER-PAGE = *STD

84 lines per page are printed.

LINES-PER-PAGE = <integer 11..255>

11 to 255 lines are printed per page.

The lower limit is fixed at 11 lines so that at least the listing header and footer and one line of text can be printed.

INCLUDE-INFORMATION = *UNCHANGED / *NONE* / ALL / *USER-INCLUDES-ONLY

This option is used to specify which header files (if any) are to be shown in the source, preprocessor and cross-reference listings. By default, only the user-defined header files are shown, not the standard headers.

LISTING-PRAGMAS =

This operand controls which existing `#pragma` directives (if any) in the source text are to be interpreted when creating source and preprocessor listings.

A description of `#pragma` directives can be found in the [section "Pragmas to control the layout of listings" on page 228](#).

LISTING-PRAGMAS = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

LISTING-PRAGMAS = *INTERPRETED / *IGNORED

All `#pragma` directives are interpreted (*INTERPRETED) or ignored (*IGNORED).

LISTING-PRAGMAS = *SELECT(...)

One or more of the following `#pragma` directives to control listings are interpreted (*YES) or ignored (*NO).

PAGE = *YES / *NO

Directive `#pragma PAGE [text]`:

for a page feed and optional line in the listing header

TITLE = *YES / *NO

Directive `#pragma TITLE text`:

for a line in the listing header

SPACE = *YES / *NO

Directive `#pragma SPACE [n]`:

to insert blank lines

LIST = *YES / *NO

Directive `#pragma LIST[ING] ON` or `#pragma LIST[ING] OFF`:
to suppress the output of source text lines

INITIAL-TITLE-TEXT = *UNCHANGED / *NONE / <c-string 1..256>

This operand can be used to specify if an additional line is to appear in the header of the listing and the text that is to be entered in it. In contrast to pragmas, which only apply to source and preprocessor listings, the INITIAL-TITLE-TEXT specification applies to all compiler listings.

In the case of source and preprocessor listings, TITLE and PAGE pragmas (if any) override the INITIAL-TITLE-TEXT specification.

OUTPUT = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

OUTPUT = *SYSLST

The listings are written to the temporary system file SYSLST by default and are sent from there to the printer at the end of the task (at LOGOFF). Output to SYSLST is not supported in the ANSI C++ modes and is rejected with a corresponding error message.

OUTPUT = *SYSOUT

The listings are written to the system file SYSOUT, which is assigned to the terminal in interactive mode. Output to SYSLST is not supported in the ANSI C++ modes and is rejected with a corresponding error message.

OUTPUT = *STD-FILE

The listings are written to a cataloged BS2000 file. The name of this file is derived from the name of the source program as follows:

Source	*SYSDTA	BS2000 file	PLAM library	POSIX file
Default name	CSTDLST.LST	file.LST	lib-elem.LST	file.LST

If the source program is located in a PLAM library, the library and element name of the source are combined with a hyphen (lib-elem) and used in the default file name. The rules by which the compiler constructs default names are described in detail in the [section "Default names for output containers" on page 48](#).

OUTPUT = *SOURCE-LOCATION

The output destination and name are derived from the location and name of the source program as follows:

Source	*SYSDTA	BS2000 file	PLAM library	POSIX file
Output destination	BS2000 file	BS2000 file	Library of source	Directory of source
Default name	CSTDLST.LST	file.LST	elem.LST (type P)	file.lst

The rules by which the compiler constructs default names are described in detail in the [section “Default names for output containers” on page 48](#).

OUTPUT = <filename 1..54>

The listings are written to a cataloged BS2000 file with the specified name. Note that this specification is not meaningful when compiling multiple source programs, since the file is overwritten in each case.

OUTPUT = <posix-pathname>

The listings are written to the POSIX file system.

Both a file name and a directory are permitted as <posix-pathname>. See [page 34](#) for a description of the term <posix-pathname>.

When a file name is specified, the listings are stored under this name.

When a directory name *dir* is specified, the listings for each compiled source program are written under the default name *sourcefile.lst* to the directory *dir* (see also [section “Default names for output containers” on page 48](#)).

The directories specified with <posix-pathname> must already exist.

The specification of a file name is not meaningful when compiling multiple source programs, since the file is overwritten in each case.

OUTPUT = *LIBRARY-ELEMENT(...)

This option specifies the PLAM library (LIBRARY=) and the element name (ELEMENT=) under which the expanded program is to be stored. The elements are stored as elements of type P.

LIBRARY = *STD-LIBRARY

The listings are written to the library SYS.PROG.LIB by default.

LIBRARY = *SOURCE-LIBRARY

The listings are written to the PLAM library which contains the source program.

The *SOURCE-LIBRARY specification is invalid if the source program is read from a cataloged BS2000 file, a POSIX file or via SYSDTA.

LIBRARY = <filename 1..54>

The listings are written to a PLAM library with the specified name.

LIBRARY = *LINK(...)**LINK-NAME = <filename 1..8>**

A link name can also be specified instead of the library name. <filename> designates the link name of the assigned library. This link name must already have been assigned to the PLAM library (with the ADD-FILE-LINK command) before the compiler is called.

ELEMENT = *STD-ELEMENT(...)

By default, the element name of the listing is derived from the name of the source program as follows:

Source	*SYSDTA	BS2000 file	PLAM library	POSIX file
Default name	CSTDLST.LST	file.LST	elem.LST	file.LST

The rules by which the compiler constructs default names are described in detail in the [section “Default names for output containers” on page 48](#).

VERSION = *UPPER-LIMIT

If the element entry does not contain a version ID, the highest possible version is used by the compiler.

VERSION = *INCREMENT

The element is assigned a version number that is obtained by incrementing the highest existing version number by 1, assuming that the highest existing version ID ends with a digit that can be incremented. If the version ID cannot be incremented, the compiler run is aborted with an error message.

See the COMPILE statement ([page 75](#)) for an example.

Warning: You may not specify *INCREMENT in the ANSI-C++ mode.

VERSION = <composed-name 1..24 with-underscore>

The element is assigned the specified version.

ELEMENT = <composed-name 1..64 with-underscore>(...)

The listings are written to a library element (type P) with the specified name. Note that the specification of an element name is not meaningful when compiling multiple source programs, since the element is overwritten in each case.

VERSION = *UPPER-LIMIT / *INCREMENT / <composed-name 1..24 with-underscore>

The version can be specified as described above for ELEMENT=*STD-ELEMENT(...).

Warning: You may not specify *INCREMENT in the ANSI-C++ mode.

MODIFY-MODULE-PROPERTIES

Alias: SET-MODULE-PROPERTIES

This statement is used to define the properties of the module to be generated.

MODIFY-MODULE-PROPERTIES

```

SHAREABLE-CODE = *UNCHANGED / *NO / [*YES] (...)
    *YES(...)
        | PUBLIC-SLICING=*UNCHANGED / *YES / *NO
,LINKAGE = *UNCHANGED / *ILCS-OUT / *ILCS-INLINE
,WORKSPACE = *UNCHANGED / *TO-STATIC-AREA / *TO-STACK
,SUBROUTINE-CALL = *UNCHANGED / *BASR / *LAB
,ETPND-GENERATION = *UNCHANGED / *NO / [*YES] (...)
    *YES(...)
        | DATE-FORMAT = *UNCHANGED / *CALENDAR-DATE-ONLY / *WITH-JULIAN-DATE
,LOWER-CASE-NAMES = *UNCHANGED / *YES / *NO
,SPECIAL-CHARACTERS = *UNCHANGED / *CONVERT-TO-DOLLAR / *KEEP
,STRING-LITERALS = *UNCHANGED / *WRITEABLE / *READ-ONLY
,CONSTANTS = *UNCHANGED / *WRITEABLE / *READ-ONLY
,C-NAMES = *UNCHANGED / *STD / *UNLIMITED / *SHORT
,FP-ARITHMETICS = *UNCHANGED / *390-FORMAT / *IEEE-FORMAT

```

SHAREABLE-CODE = *UNCHANGED / *NO / [*YES] (...)

This option controls whether or not the generated code is shareable.

SHAREABLE-CODE = *UNCHANGED

The settings of the last MODIFY-MODULE-PROPERTIES statement are used.

SHAREABLE-CODE = *NO

The compiler does not generate shareable code.

*NO is the default setting.

SHAREABLE-CODE = [*YES] (PUBLIC-SLICING = ...)

The compiler creates shareable code in the form of an LLM with a shareable code CSECT and a non-shareable data CSECT.

PUBLIC-SLICING =

PUBLIC SLICING defines whether or not the object generated immediately after the attribute PUBLIC is distributed in slices.

PUBLIC-SLICING = *UNCHANGED

The settings of the last MODIFY-MODULE-PROPERTIES statement are used.

PUBLIC-SLICING = *YES

The object will be distributed in slices immediately after the attribute PUBLIC.

*YES is the default setting.

PUBLIC-SLICING = *NO

The object is placed in a slice.



POSIX objects are always generated with PUBLIC-SLICING = *NO.

With very large programs it is a good idea to set PUBLIC-SLICING=*NO. This will avoid the use of the long run times needed to read objects with slices.

LINKAGE = *UNCHANGED / *ILCS-OUT / *ILCS-INLINE

Function calls in the generated module are handled via ILCS by default. This option can be used to specify whether the ILCS entry code for function calls is to be directly inserted at the calling point ("inline") or whether it is to be accessed "out-of-line" in the runtime system.

*ILCS-OUT: The ILCS entry code for function calls is accessed "out-of-line" in the runtime system. This reduces the volume of code in the module.

*ILCS-INLINE: By default, the ILCS entry code is generated inline. This causes the generated object to execute faster.

WORKSPACE =

This option affects some optimization steps of the compiler.

WORKSPACE =*UNCHANGED

The values specified in the last MODIFY-MODULE-PROPERTIES statement apply.

WORKSPACE = *TO-STATIC-AREA

The following optimizations are performed by the compiler:

- An auxiliary storage area is created in the static data area and supplied with constant values.
- In the case of innermost functions (i.e. those without further calls), data that is only valid within the function (auto variables) is not placed on the stack, but is stored together with the static data in the data module.

These optimization measures eliminate the need to maintain a separate stack for innermost functions and thus reduce the function entry code, and consequently the runtime of the generated object.

WORKSPACE = *TO-STACK

The above optimizations are suppressed.

All data (doublewords for conversions as well as auto variables and tempos of innermost functions) is mapped on the stack.

If a function includes conversions, some of the conversions may require a doubleword to be placed on the stack and supplied dynamically (requiring an additional instruction) before each such conversion.

SUBROUTINE-CALL = *UNCHANGED / *BASR / *LAB

This option controls the implementation of subroutine entries via Assembler instructions.

*BASR: The BASR instruction is generated by default.

*LAB: The LAB entry generates the machine-independent Assembler instructions LA and B. Programs with this instruction sequence are executable on all 7500 systems.

Warning: This option is not permitted in the ANSI-C++ mode.

ETPND-GENERATION =

This option serves to delete the `#pragma` directive used to generate an ETPND area (see [page 226](#)) or to specify the date format of the ETPND area.

ETPND-GENERATION = *UNCHANGED

The values specified in the last MODIFY-MODULE-PROPERTIES statement apply.

ETPND-GENERATION = *NO

By default, no ETPND area is created.

ETPND-GENERATION = *YES(...)

DATE-FORMAT = *UNCHANGED / *CALENDAR-DATE-ONLY / *WITH-JULIAN-DATE

*CALENDAR-DATE-ONLY: The date format in the ETPND area is assigned the form: 8-byte calendar date - 4-byte load address.

*WITH-JULIAN-DATE: The following date format is generated in the ETPND area: 6-byte calendar date - 3-byte Julian date - 4-byte load address.

LOWER-CASE-NAMES = *UNCHANGED / *NO / *YES

This option for converting lowercase letters to uppercase affects all external symbols in the C language modes and in the Cfront C++ mode, but only the symbols declared with `extern "C"` in the ANSI C++ modes. When external C++ symbols are coded in the ANSI C++ modes, lowercase letters are always retained.

*NO: By default, lowercase letters are converted to uppercase when generating entry names.

*YES: Lowercase letters are retained when generating entry names.

SPECIAL-CHARACTERS = *UNCHANGED / *CONVERT-TO-DOLLAR / *KEEP

This option for converting the underscore affects all external symbols in the C language modes, but only the symbols declared with `extern "C"` directives (not the entry names of the C library functions) in the C++ language modes. When external C++ symbols are coded, underscores are always retained.

*CONVERT-TO-DOLLAR: By default, underscores are converted to dollar signs when generating entry names.

*KEEP: Underscores are retained when generating entry names.

Notes on LOWER-CASE-NAMES and SPECIAL-CHARACTERS

1. The (default) conversion of lowercase to uppercase and of underscores to dollar signs is required whenever the generated LLM is to be linked with objects in which the entry names have been converted accordingly. These are:
 - Object modules.
 - LLMs generated with the C V2.0 compiler.
 - LLMs generated with the C/C++ compiler, in cases where the entry names were converted accordingly.
 - Objects created with other language translators (e.g. COBOL, ASSEMBLER).
2. The C library functions are only available in full when the options LOWER-CASE-NAMES and SPECIAL-CHARACTERS are present in one of the following combinations:
 - SPECIAL-CHARACTERS=*CONVERT-TO-DOLLAR and LOWER-CASE-NAMES=*NO
 - SPECIAL-CHARACTERS=*KEEP and LOWER-CASE-NAMES=*YES

STRING-LITERALS = *UNCHANGED / *WRITEABLE / *READ-ONLY

This option determines whether string literals (e.g. "abc") are placed in the data module (*WRITEABLE) or in the code module (*READ-ONLY).

CONSTANTS = *UNCHANGED / *WRITEABLE / *READ-ONLY

This option determines whether constants (i.e. objects declared with the `const` qualifier) are stored in the data module (*WRITEABLE) or the code module (*READ-ONLY).

C-NAMES = *UNCHANGED / *STD / *UNLIMITED / *SHORT

This option determines the length of external C names and affects all external symbols in the C language modes and only the symbols declared `extern "C"` in the C++ language modes (not the entry names of C library functions).

This option also works with static functions.

*STD: By default, external C names can have a maximum length of 32 characters. Longer names are truncated by the compiler to 32 characters. When generating shareable code, only 30 characters may be used.

*UNLIMITED: No name truncation occurs. In this case, the compiler generates entry names in EEN format. EEN names can have a maximum length of 32000 characters. Modules containing EEN names are stored by the compiler in LLM Format 4. More details on the subsequent processing of LLMs in Format 4 can be found under the OUTPUT-FORMAT option of the BIND statement ([page 68](#)).

The *UNLIMITED value is not supported in the Cfront C++ mode.

*SHORT: External C names are truncated by the compiler to 8 characters, which corresponds to the handling of external names within object modules. This option is required if external names exceeding 8 characters in length are used in the program and if the module generated by the C/C++ compiler is to be linked with object modules that were created with the earlier C/C++ compilers or with compilers for other ILCS languages (e.g. COBOL85).

FP-ARITHMETICS = *UNCHANGED / *390-FORMAT / *IEEE-FORMAT

This option determines whether the C/C++ compiler generates floating-point numbers and operation codes in /390 format or IEEE format. This option effects all variables and constants of the float, double, and long double data types in C/C++ programs.

*390-FORMAT: The C/C++ compiler creates code for constants and operations in the /390 format (/390 floating-point arithmetics).

*390-FORMAT is the default setting.

*IEEE-FORMAT: The C/C++ compiler creates code for constants and operations in IEEE format (IEEE floating-point arithmetics).

Important!

- It is not possible to undo links with objects which have been compiled with different floating-point arithmetics. This characteristic can lead to unexpected results when these programs are run.
- The same C/C++ program can produce different results depending on whether the IEEE format or the /390 format is used for floating-point data types and operations. The reasons for this are as follows:
 - IEEE floating-point numbers use a different internal notation from /390 floating-point numbers.
 - IEEE floating-point operations use different semantics from /390 floating-point operations even on the same type of operation. This is the case for example in rounding. IEEE format uses "Round to Nearest" as default whereas /390 format uses "Round to Zero" as default.

Requirements:

- If you are using IEEE floating-point arithmetics, you must not declare the C library functions explicitly in your source program. C library functions should be declared indirectly by including the corresponding CRTE headers (see the "C Library Functions" manual [2]). Otherwise compilation error 'CFE1079[ERROR]...: Typangabe erwartet / expected a type specifier' can occur.
- For *each and every* CRTE function that works with floating-point numbers in your program, you must use the corresponding or matching include file. If you do not do this, the CRTE functions will not be able to process the floating-point numbers correctly. You should ensure that you include the include file `<stdio.h>` for the function `printf()` with `#include <stdio.h>`.

Important!

C++ library functions do not support the IEEE format and must therefore be replaced with C functions where necessary.

- In the CRTE runtime environment, some C library functions use the IEEE format for floating-point arithmetics. If you are using the IEEE floating-point arithmetics, you should specify the `MODIFY-MODULE-PROPERTIES` statement as follows:

```
MODIFY-MODULE-PROPERTIES      -
...
FP-ARITHMETICS=*IEEE-FORMAT,  -
LOWER-CASE-NAMES=*YES,        -
SPECIAL-CHARACTERS=*KEEP,     -
...
```


MODIFY-OPTIMIZATION-PROPERTIES

Alias: SET-OPTIMIZATION-PROPERTIES

This option can be used to activate or deactivate some or all of the optimizations performed by the compiler.

For more details on the effects of optimization, see the section starting on [page 126](#).

```
MODIFY-OPTIMIZATION-PROPERTIES
```

```
LEVEL = *UNCHANGED / *LOW / *HIGH(...) / *VERY-HIGH(...)
```

```
*HIGH(...) / *VERY-HIGH(...)
```

```
    | ,LOOP-UNROLLING = *UNCHANGED / *NO / [*YES](...)
```

```
        | *YES(...)
```

```
            | FACTOR = 4 / <integer 1..100>
```

```
,INLINING = *UNCHANGED / *NO / [*YES](...)
```

```
*YES(...)
```

```
    | USER-FUNCTIONS = *UNCHANGED / list-poss(127): *STD / *BY-SOURCE /
```

```
        <c-string 1..255 with-low>
```

```
,BUILTIN-FUNCTIONS = *UNCHANGED / *NONE / *ALL / list-poss(11): <c-string 1..125 with-low>
```

LEVEL = *UNCHANGED

The value specified in the last MODIFY-OPTIMIZATION-PROPERTIES statement applies.

LEVEL = *LOW

With this optimization level, no standard optimizations are performed, so debugging with AID is possible.

LEVEL=*LOW is automatically set instead of the *HIGH or *VERY-HIGH specification when the TEST-SUPPORT=*YES option has also been set.

LEVEL = *HIGH(...) / *VERY-HIGH(...)

If *HIGH or *VERY-HIGH is specified, all standard optimizations are performed (see “[Standard optimizations](#)” on [page 126](#)). The only difference between these two levels is the fact that every optimization strategy is internally executed only once for *HIGH, but several times for *VERY-HIGH. Consequently, if the “highly-optimized” level *VERY-HIGH is set, the overall compile time is much greater than the compile time for the *HIGH optimization level.

The parameters of the *HIGH or *VERY-HIGH structure can be used to individually control the expansion of loops. Debugging with AID is not possible with this optimization level.

LOOP-UNROLLING = *UNCHANGED / *NO / *YES(FACTOR = 4 / <integer 1..100>...)

This option controls the expansion of loops. Multiple expansion of the body of a loop reduces the execution time for the iterations of the loop. Expanding the body of the loop provides an opportunity for further optimization; however, the repetition of code also implies an increase in the size of the generated object.

By default, the optimizer expands the body of a loop 4 times.

- Where required, a separate expansion factor can be selected with <integer>. Specifying an expansion factor does not, however, ensure that the loop expansion will be carried out in all cases. In order to ensure that the expansion is carried out correctly, you should concentrate on optimizing the loop structure and the specified expansion factor.
- Loop expansion can be suppressed with *NO.

For further details, see [“Expansion of loops” on page 130](#).

INLINING =

This option controls the inline substitution of user-defined functions. As in the case of the inline substitution of some C library functions from the standard library (see BUILTIN-FUNCTIONS), each call to an inline function is replaced by the corresponding function code. Consequently, no call and return code sequence is required, and better execution time is achieved. This optimization measure is, however, associated with an increase in the size of the generated module due to the repetition of code.

For further details, see the section on [“Inline substitution of user-defined functions” on page 129](#).

Standard settings of the compiler

If the INLINING option is not specified, the following default settings apply for the C and C++ language modes, respectively:

1. C modes (MODIFY-SOURCE-PROPERTIES LANGUAGE=*C):

INLINING=*NO

No inline substitution is performed.

2. C++ modes (MODIFY-SOURCE-PROPERTIES LANGUAGE=*CPLUSPLUS):

INLINING=*YES(USER-FUNCTIONS=*BY-SOURCE)

Inline substitution is performed for all C++-specific inline functions (i.e. functions with the `inline` attribute and member functions defined within classes).

INLINING = *UNCHANGED

The value specified in the last MODIFY-OPTIMIZATION-PROPERTIES statement applies.

INLINING = *NO

No user-defined functions are generated inline by the optimizer. *NO is the default setting in the C language modes if no INLINING option has been specified.

In addition, *NO is also automatically assumed by the compiler instead of *YES(...) values if the option TEST-SUPPORT=*YES has also been set.

INLINING = *YES(USER-FUNCTIONS = *UNCHANGED / list-poss: *STD / *BY-SOURCE / <c-string 1..255 with-low>)

***STD:**

If only *STD is specified, the optimizer selects functions for inline substitution on the basis of its own criteria. *STD implies *BY-SOURCE, which means that even inline pragmas and C++-specific inline functions are considered by the optimizer when searching for suitable candidates to be inlined (see also *BY-SOURCE).

*STD is the default when INLINING=*YES is specified.

***BY-SOURCE:**

If only *BY-SOURCE is specified, only the following user-defined functions are substituted inline:

- In the C modes: all C functions specified with the following `#pragma` directive:
`#pragma inline function-name`
 The inline pragma is not supported in the C++ language modes.
 See also the [section “inline pragma” on page 231](#).
- In the C++ modes: all C++ functions with the `inline` attribute and all C++ functions defined within classes.

*BY-SOURCE is the default in the C++ language modes if no INLINING option is specified.

<c-string>:

<c-string> can be used to specify the name of a user-defined function to be inlined by the optimizer. The specification of user-selected functions with <c-string> is only supported in the C language modes, since C++ has its own language elements for the inline substitution of functions. <c-string> implies the specification of *BY-SOURCE, which means that even inline pragmas are considered by the optimizer (see also *BY-SOURCE in the C modes).

list-poss:

*STD, <c-string> and - for compatibility reasons - even *BY-SOURCE may also be specified together, in which case the optimizer will first attempt the inline substitution of the function(s) specified with *BY-SOURCE and/or <c-string> and then select other functions for inline

substitution (provided *STD is also specified) according to its own criteria. Note that *BY-SOURCE need not be specified here, since it is implicitly assumed whenever *STD or <c-string> is specified at the same time.

The meaningful combinations are therefore:

*STD, <c-string>
 <-c-string>, <c-string>, ...

Example of the INLINING option

```
//MODIFY-OPTIMIZATION-PROP -
//LEVEL=*HIGH, INLINING=*YES(USER-FUNCT=(*STD, 'funct1', 'funct2'))
```

See the [“Inline substitution of user-defined functions”](#) on page 129 for further details.

BUILTIN-FUNCTIONS = *UNCHANGED / *NONE / *ALL /

list-poss: <c-string 1..125 with-low>

This option can be used to specify the C library functions for which the implementation in the CRTE can be assumed. This permits better optimization of the program.

*NONE

No library function call is specially optimized.

*ALL

All calls for known library functions are handled separately.

<c-string>

Calls for this function are handled separately.

The compiler achieves the greatest effect through inline substitution of a function. In this case the function code is directly inserted at the point of call. This eliminates some of the time-consuming administrative tasks of the runtime system (e.g. saving and restoring registers, return from the function, etc.) and thus reduces the overall execution time of the program.

The following C library functions can be expanded inline.:

strcpy	memcpy
strcmp	memset
strncmp	abs
strlen	fabs
strcat	labs
memcpy	

Notes

- Inlined functions cannot be replaced by other functions at link time and cannot be used as checkpoints when debugging with AID.
- Non-inlined functions are retained as a call. However, optimizations are possible which cannot be achieved in the user functions. For example, the compiler can use the information that the `isdigit()` function has no side effects.

- If a function is defined by the user with a name which the compiler knows, conflicts can occur with this option. The function written by the user will generally have a different implementation from the function in the CRTE. Warning CFE2067 is issued instead of the definition in order to indicate the conflict.
- Note that the features of the CRTE implementation are used in every compilation unit. However, the warning is only output in the compilation unit containing the private definition.

The default settings of the MODIFY-OPTIMIZATION-PROPERTIES statement and possible modifications are repeated in summarized form in the table below.

	*HIGH(...) / *VERY-HIGH(...)	*LOW
Standard optimizations	*YES	*NO
LOOP-UNROLLING	*YES (controllable)	*NO
BUILTIN-FUNCTIONS	*NONE (controllable)	
INLINING (in C)	*NO (controllable)	
INLINING (in C++)	*YES(USER-FUNCTIONS=*BY-SOURCE) (controllable)	

The optimization process

In the description that follows, the term “standard optimizations” is used to collectively refer to all optimization measures that can only be activated or deactivated globally, i.e. which cannot be controlled individually.

In contrast to the standard optimizations, some optimization steps such as loop expansion, and inlining can be individually controlled. Selective control is provided in these cases because, among other things, the benefits of such optimization (e.g. quicker execution times) may need to be weighed against other disadvantages (e.g. an increase in the size of the generated module or longer compilation times).

In some cases, setting the LOOP-UNROLLING and INLINING options may actually degrade performance. It is therefore advisable to determine the best settings for each application by running some tests.

Standard optimizations

The C++ compiler performs the following optimizations:

- evaluation of constant expressions at compile time
- optimization of subscript computation in loops
- elimination of superfluous assignments
- propagation of constant expressions
- elimination of redundant expressions
- optimization of branches to unconditional branch instructions

The term “base block” is fundamental to the concept of optimization and refers to a maximum, unbranched command sequence. Such a command sequence has precisely one entry point and one exit.

Base blocks are the units via which most optimization steps with the C++ compiler are implemented.

1. Evaluation of constant expressions at compile time

By evaluating expressions for which operand values are known during compilation, the execution of commands is relocated from the program run to the compiler run. Consequently, the program executes faster.

Evaluations at compile time involve integer arithmetics and relational operations.

Example

Before optimization	After optimization
---------------------	--------------------

<code>I = 1 + 2;</code>	<code>I = 3;</code>
<code>I = 2 * 4;</code>	<code>I = 8;</code>
<code>1 <= 5;</code>	<code><TRUE></code>

2. Elimination of superfluous assignments

If an assignment to a variable v is followed by a change in the value of v without the original value ever being used, this assignment is eliminated. Assignments to variables whose values are of no further consequence to the program run are also eliminated.

Example

Before optimization	After optimization
<pre>i = 5; i = 3;</pre>	<pre>i = 3;</pre>

3. Propagation of constant expressions

If a variable whose value is already known at the time of compilation is used in an expression, this variable will be replaced by the appropriate value.

Example

Before optimization	After optimization
<pre>a = 3; i = a;</pre>	<pre>a = 3; i = 3;</pre>

This optimization also has a bearing on other optimization techniques. After the successful propagation of a variable, the original assignment may be deleted, or a new constant expression that has already been evaluated at compile time may appear.

Example

Before optimization	After optimization
<pre>a = 3; i = a + 4; a = 5;</pre>	<pre>i = 7; a = 5;</pre>

4. Elimination of redundant expressions

If the value of an expression occurring within a basic block is already known at the time of compilation as a result of an earlier calculation, then this expression is redundant. To avoid a repeat calculation, the expression is assigned to a new variable and replaced by this new variable wherever it occurs.

Example

Before optimization	After optimization
a = b * c + 20;	h = b * c;
e = b * c - 10;	a = h + 20;
	e = h - 10;

5. Optimization of subscript computation in loops

If an array element is subscripted in a loop via an iteration variable, the multiplication required in order to compute the address of the array element is reduced to additions. As a rule, the address of the array element is calculated as

$$\text{base address} + \text{index} * \text{length of an array element}$$

Before entering a loop, the optimizer supplies an address variable with the address of the array element that was referenced during the first iteration. With every following iteration, this address variable is then incremented with a fixed length of one array element.

This optimization technique is especially rewarding in the case of multidimensional arrays, since one multiplication step can be eliminated per dimension under optimum conditions.

6. Optimization of branches to unconditional branch instructions

In branch instructions that have an unconditional branch as their destination, the branch address of the unconditional branch is substituted for the original branch address. This also helps eliminate superfluous code, i.e. code which specifies addresses that cannot be accessed.

Example

Before optimization	After optimization
goto lab1;	goto lab2;
...	...
lab1:	ab1:
goto lab2;	goto lab2;
...	...
lab2:	lab2:

Inline substitution of user-defined functions

The inline substitution of a function eliminates the need to call the function at runtime, since the function code is integrated into the source code at the point of call. This reduces the administrative overhead and code required for function calls and returns (e.g. saving and restoring registers, allocating stacks, writing parameters to the transfer area, etc.) and can thus produce substantial savings in execution time. In addition, the inlining of functions enhances the effect of the standard optimizations due to the larger context.

Inlined `static` functions are deleted.

Inline substitution does, however, increase the size of the generated modules. It may produce extremely large functions with an increasing number of variables as potential recipients of registers that cannot all be supplied due to the limited number of registers available. This fact must be weighed against the advantages of this method of optimization.

Inline substitution is particularly suitable for small functions where the overhead for the function call and return constitutes a substantial part of the actual function code.

Functions with the following attributes can never be expanded inline:

- functions with a variable number of parameters (cf. `va_...` macros in `<stdarg.h>`)
- functions containing `setjmp` calls
- recursive functions

Expansion of loops

Loop expansion reduces the number of iterations in a loop by repeating, i.e. “expanding” the body of the loop (statement block) one or more times. Since each iteration of a loop requires loop control statements to test the value of the current iteration and to branch accordingly, a reduction in the number of iterations also improves execution time.

For example, if the body of a loop is doubled (expansion factor 2), the overhead for loop control statements is reduced by half. In general, the following rule applies: an expansion factor of n reduces the overhead for loop control statements to $1/n$.

The size of the generated module is, however, increased by the repetition of code. By default, the optimizer uses an expansion factor of 4.

Expanding the body of a loop also creates the potential for new optimization. For example, the extended base blocks can be made more efficient by the propagation of constant expressions or the elimination of redundant expressions.

Example of a loop expansion with expansion factor 4

Before expansion:

```
i = 0;
while(i < 80) {
    a[i] = b[i+1];
    i++;
}
```

After expansion:

```
i = 0;
while(i < 80) {
    a[i] = b[i+1];
    i++;
    if(!(i < 80)) goto end;
    a[i] = b[i+1];
    i++;
    if(!(i < 80)) goto end;
    a[i] = b[i+1];
    i++;
    if(!(i < 80)) goto end;
    a[i] = b[i+1];
    i++;
}
end:
```

Optimizations in the body of the loop are also possible here.

MODIFY-RUNTIME-PROPERTIES

Alias: SET-RUNTIME-PROPERTIES

This statement can be used to influence runtime behavior when compiling programs containing the `main` function.

MODIFY-RUNTIME-PROPERTIES

```

PARAMETER-PROMPTING = *UNCHANGED / *YES / *NO
,STACK-SIZE = 64 / *UNCHANGED / <integer 8..99999> / <x-string 1..8>
,STATISTIC-MESSAGES = *UNCHANGED / *CPU-TIME / *NONE
,PROGRAM-INTERRUPT = *UNCHANGED / *INTEGER-OVERFLOW / *NONE
,ENVIRONMENT-ENCODING = *UNCHANGED / *STD / *EBCDIC

```

PARAMETER-PROMPTING = *UNCHANGED / *YES / *NO

*YES: The executable program is to simulate the UNIX environment, in other words:

- the program with parameters can be called using `START-EXECUTABLE-PROGRAM`, or
- a parameter line should appear after the program is started to allow the input of parameters for the `main` function or redirection of `stdin`, `stdout` or `stderr` (see also [section “Parameter input at program start” on page 179](#)).

*NO: The executable program is started without the parameter line being requested.

If the program is started from the POSIX shell, the entries in the `PARAMETER-PROMPTING` option will be meaningless, since parameters are always entered from the command line in this case.

STACK-SIZE = 64 / *UNCHANGED / <integer 8..99999> / <x-string 1..8>

This option can be used to determine the amount of space to be reserved for the first segment of the C runtime stack.

64: The default value is 64 kilobytes.

<integer>: Between 8 and 99999 kilobytes of storage space can be reserved.

<x-string>: 8 ..1869F

STATISTIC-MESSAGES = *UNCHANGED / *CPU-TIME / *NONE

When a program is terminated, the CPU time used is reported by default. This message can be suppressed by specifying *NONE.

PROGRAM-INTERRUPT = *UNCHANGED / *INTEGER-OVERFLOW / *NONE

This operand can be used to set the program mask when compiling programs that contain the `main` function

*INTEGER-OVERFLOW corresponds to the ILCS program mask X'0C'.



This option does not affect the selection of generated commands. The result is that permitting INTEGER-OVERFLOWS does not necessarily mean that an overflow is triggered in all cases.

*NONE corresponds to the program mask X'00'.

The effects of these two program masks are as follows:

	INTEGER-OVERFLOW	NONE
fixed-point overflow	permitted	suppressed
decimal overflow	permitted	suppressed
exponent overflow	suppressed	suppressed
null mantissa	suppressed	suppressed



No changes in the ILCS program mask are permitted if mixed languages are used!

ENVIRONMENT-ENCODING = *UNCHANGED / *STD / *EBCDIC

These options enable the manner in which external strings (arguments of `main` and environment variables) are handled to be controlled in the case of programs which contain the `main` function.

*STD is the default value. This causes the external strings to be coded in the manner specified in the MODIFY-SOURCE-PROPERTIES LITERAL-ENCODING option.

The *EBCDIC option is offered for reasons of compatibility and causes external strings to be coded in EBCDIC even when MODIFY-SOURCE-PROPERTIES=*ASCII or *ASCII-FULL is specified.

MODIFY-SOURCE-PROPERTIES

Alias: SET-SOURCE-PROPERTIES

This statement can be used to define the properties of a source program and to control the behavior of the preprocessor and the C and C++ frontends.

MODIFY-SOURCE-PROPERTIES

/ Options to select the language mode */*

LANGUAGE = ***UNCHANGED** / ***C(...)** / ***CPLUSPLUS(...)**

***C(...)**

MODE = ***UNCHANGED** / ***ANSI** / ***STRICT-ANSI** / ***KERNIGHAN-RITCHIE**

***CPLUSPLUS(...)**

MODE = ***UNCHANGED** / ***ANSI** / ***STRICT-ANSI** / ***CPP**

/ Preprocessor options */*

,DEFINE = ***NONE** / ***UNCHANGED** / list-poss: <c-string 1..125 with-low> / <name 1..125 with-under> / ***SUBSTITUTE(...)**

***SUBSTITUTE(...)**

IDENTIFIER = <c-string 1..125 with-low> / <name 1..125 with-under>

,TOKEN-STRING = <c-string 1..125 with-low> / <name 1..125 with-under>

,UNDEFINE = ***NONE** / ***UNCHANGED** / ***ALL** / list-poss: <c-string 1..125 with-low> / <name 1..125 with-under> /

,ASSERT = ***NONE** / ***UNCHANGED** / list-poss: ***SUBSTITUTE(...)**

***SUBSTITUTE(...)**

IDENTIFIER = <c-string 1..125 with-low> / <name 1..125 with-under>

,TOKEN-STRING = <c-string 1..125 with-low> / <name 1..125 with-under>

,PREINCLUDE = ***UNCHANGED** / ***NONE** / <c-string 1..1024 with-low>

,COMMENTS = ***UNCHANGED** / ***YES** / ***NO**

,PREPROCESSING-MODE = ***UNCHANGED** / ***ANSI** / ***KR**

,IMPLICIT-INCLUDE = ***UNCHANGED** / ***YES** / ***NO**

```

/* Common frontend options in C and C++ */
,SIGNED-CHARACTER = *UNCHANGED / *YES / *NO
,AT-ALLOWED = *UNCHANGED / *YES / *NO
,DOLLAR-ALLOWED = *UNCHANGED / *YES / *NO
,ENUM-TYPE = *UNCHANGED / *VALUE-DEPENDENT / *LONG
,SIGNED-FIELDS = *UNCHANGED / *SIGNED / *UNSIGNED
,PLAIN-FIELDS = *UNCHANGED / *SIGNED / *UNSIGNED
,PRESERVING = *UNCHANGED / *UNSIGNED / *LONG
,ALTERNATIVE-TOKENS = *UNCHANGED / *YES / *NO
,EXTERNAL-DEFINITION = *UNCHANGED / *BY-SOURCE-LANGUAGE / *UNIQUE /
    *MULTIPLY-ALLOWED
,LOGLONG = *UNCHANGED / *YES / *NO
,END-OF-LINE-COMMENTS = *UNCHANGED / *YES / *NO
,LITERAL-ENCODING = *UNCHANGED / *NATIVE / *ASCII / *ASCII-FULL / *EBCDIC / *EBCDIC-FULL

/* C++ specific options */
,INSTANTIATION = *UNCHANGED / *NONE / *AUTO / *LOCAL / *ALL
,VIRTUAL-FUNCTION-TAB = *UNCHANGED / *INTERNALLY-DEFINED / *GLOBALLY-DEFINED /
    *EXTERNALLY-DECLARED
,USE-STD-NAMESPACE = *UNCHANGED / *YES / *NO
,KEYWORD-BOOL = *UNCHANGED / *YES / *NO
,KEYWORD-WCHAR = *UNCHANGED / *YES / *NO
,LOOP-INIT = *UNCHANGED / *OLD / *NEW
,SPECIALIZATION = *UNCHANGED / *OLD / *NEW

```

Options to select the language mode

LANGUAGE =

This option specifies in which programming language (C or C++) the sources to be compiled were written. The value specified in the suboption MODE= determines in which of the available C language modes (K&R C, extended or strict ANSI C) or C++ language modes (Cfront C++, extended or strict ANSI C++) the source programs are compiled.

The default setting of the compiler is extended ANSI C++:

LANGUAGE=*CPLUSPLUS(MODE=*ANSI)

LANGUAGE = *UNCHANGED

The value specified in the last MODIFY-SOURCE-PROPERTIES statement applies.

LANGUAGE = *C(...)

The source program is a C program.

MODE = *UNCHANGED

The value specified in the last MODIFY-SOURCE-PROPERTIES statement with LANGUAGE=*C applies.

MODE = *ANSI

Extended ANSI C mode (default setting)

The compiler supports C code, as defined in the ANSI/ISO C standard, including the ISO C Amendment 1. In addition, various other language extensions are also supported (see the [chapter “C language support of the compiler” on page 199ff](#)). Note that the namespace is not restricted to names specified by the standard. All C library functions of the CRTE (ANSI functions, POSIX and X/OPEN functions, and UNIX extensions) may be used.

`__STDC__` has a value of 0, and `__STDC_VERSION__` a value of 199409L.

MODE = *STRICT-ANSI

Strict ANSI C mode

This mode can be used to test a program for ANSI/ISO conformance.

As in the extended ANSI C mode, the compiler supports C code in accordance with the ANSI/ISO C standard.

However, in contrast to the extended ANSI C mode, the namespace is restricted to the names defined in the standard, and only the C library functions defined in the ANSI/ISO standard are available. This is technically accomplished as follows:

When *STRICT-ANSI is specified, the `_STRICT_STDC` directive is set internally. The `_STRICT_STDC` setting instructs the compiler to deactivate or bypass the prototype declarations for all non-ANSI/ISO C library functions in the standard headers (`stdio.h`, `stdlib.h`, etc.). The `_STRICT_STDC` directive is, however, only effective for the prototype declarations in the standard headers defined by ANSI/ISO; the BS2000 and POSIX-specific headers do not include a check for this directive.

Deviations from the standard result in compiler messages (mostly warnings). If desired, the output of errors can be forced in such cases by specifying the option `ANSI-VIOLATIONS=*ERROR` (see [page 96](#)).

`__STDC__` has a value of 1, and `__STDC_VERSION__` a value of 199409L.

MODE = *KERNIGHAN-RITCHIE

K&R C mode

This mode should not be used for new developments. It is typically intended for porting "old" K&R C sources and/or systematic conversions to ANSI C.

The compiler accepts C code, as defined by Kernighan&Ritchie in the reference manual ("The C Programming Language", First Edition). It also supports C language elements of the ANSI C standard that are semantically identical to the Kernighan&Ritchie "definition" of the C language (e.g. function prototypes, `const`, `volatile`). This simplifies the conversion of a K&R C source to ANSI C. All C library functions of the CRTE (i.e. ANSI functions, POSIX and X/OPEN functions, UNIX extensions) are available for use.

As far as the preprocessor behavior is concerned, ANSI/ISO C is the default. If desired, the option `PREPROCESSING-MODE=*KR` (see [page 139](#)) can be specified to convert the preprocessor behavior to K&R C (as required when porting old C sources from a UNIX system, for example).

`__STDC__` has a value of 0, and `__STDC_VERSION__` is not defined.

LANGUAGE = *CPLUSPLUS(...)

The source program is a C++ program. This is also the default setting of the compiler before the programming language is defined for the first time with the LANGUAGE option.

MODE = *UNCHANGED

The value specified in the last MODIFY-SOURCE-PROPERTIES statement with `LANGUAGE=*CPLUSPLUS` applies.

MODE = *ANSI

Extended ANSI C++ mode (default setting)

The compiler supports C++ code in accordance with the definition proposed in the ANSI C++ draft for the future ANSI/ISO C++ standard. In this case, the namespace is not restricted to names specified in the standard.

The following C++ libraries are available:

- the standard C++ library (strings, containers, iterators, algorithms, and numerics), including the Cfront-compatible I/O classes
- the Tools.h++ library

For more information on C++ libraries, see also the [chapter "The C++ libraries and C++ runtime system" on page 275ff.](#)

As in the extended ANSI C mode, various language extensions as well as all C library functions of the CRTE are available for use.

`__STDC__` has a value of 0, `__cplusplus` a value of 2, and `__STDC_VERSION__` a value of 199409L.

MODE = *STRICT-ANSI

Strict ANSI C++ mode

In terms of the C++ language support (based on the ANSI/ISO C++) and the available C++ libraries, this mode corresponds to the extended ANSI C++ mode.

However, in contrast to the extended ANSI C++ mode, only the C library functions defined in the ANSI/ISO standard are available (as in the case of the strict ANSI C mode).

Deviations from the standard result in compiler messages (mostly warnings), but the output of errors can be forced in such cases by specifying the option `ANSI-VIOLATIONS=*ERROR` (see [page 96](#)).

`__STDC__` has a value of 1, `__cplusplus` a value of 199612L (which may change in future versions; see the Release Notice for details), and `__STDC_VERSION` a value of 199409L.

MODE = *CPP

Cfront C++ mode

This mode is only offered for compatibility reasons and should not be used for new developments. It supports the C++ language elements of Cfront V3.0.3.

Cfront V3.0.3 was first released with the C++ compiler V2.1.

The Cfront compatible C++ library with complex math and stream-oriented I/O is available.

More information on the Cfront C++ library can be found in the [section “The Cfront C++ library” on page 277](#).

C++ sources must be compiled with `MODE=*CPP` if their modules are to be linkable with C++ V2.1/V2.2 modules.

`__STDC__` has a value of 0, `__cplusplus` a value of 1, and `__STDC_VERSION` a value of 199409L.

*Preprocessor options***DEFINE = *NONE**

Only the names and values that are specified by `#define` directives in the program or are predefined by the compiler are considered valid for the current compiler run.

DEFINE = *UNCHANGED

The values specified in the last `MODIFY-SOURCE-PROPERTIES` statement apply.

DEFINE = <c-string 1..125 with-low> / <name 1..125 with-under>

`<c-string> / <name>` is used to define a name. This definition has the same effect as the following statement in a program:

```
#define name 1
```

Such names are queried in the program with, for example, the preprocessor directives `#ifdef`, `#ifndef` or `#if defined()`, `#if !defined()`. See also the example below. When POSIX library functions are used, the `_OSD_POSIX` directive must be set before the occurrence of the first `#include` directive in the program. The easiest way to do this is by means of a definition at the time of compilation.

DEFINE = *SUBSTITUTE(...)

This substructure is used to define macros and symbolic constants (analogous to a `#define` directive for replacing text). See also the example below.

IDENTIFIER = <c-string 1..125 with-low> / <name 1..125 with-under>

<c-string> / <name> designates the name to be replaced in the source program by the value or text specified with `TOKEN-STRING`.

TOKEN-STRING = <c-string 1..125 with-low> / <name 1..125 with-under>

<c-string> / <name> specifies the value or text to be substituted in the source program for the name indicated by `IDENTIFIER`.

Note

If data in the `DEFINE` option is inconsistent with any `#define` directives in the source program, the entries in the source program will always be given precedence!

Example: DEFINE option

```
MODIFY-SOURCE-PROP DEFINE=('mch_file',DEBUG,_OSD_POSIX,*SUB('host',BS2000),-
*SUB(LAN,'C++'))
```

Values set with `DEFINE` must be enclosed in single quotes if they contain characters other than uppercase `A` to `Z`, the digits `0` to `9`, or the special characters `$`, `#`, `@`, and `_` (see also the table on [page 15](#)).

The above entries in the `DEFINE` option correspond to the following `#define` directives in the source program:

```
#define mch_file 1
#define DEBUG 1
#define _OSD_POSIX 1
#define host BS2000
#define LAN C++
```

UNDEFINE = *NONE

The `DEFINE` entries (see above) remain unmodified by default.

UNDEFINE = *UNCHANGED

The values specified in the last `MODIFY-SOURCE-PROPERTIES` statement apply.

UNDEFINE = *ALL

All `DEFINE` entries are deleted.

UNDEFINE = <c-string 1..125 with-low> / <name 1..125 with-under>

The names specified with DEFINE <c-string> / <name> are deleted.

ASSERT = *NONE / *UNCHANGED / list-poss: *SUBSTITUTE(...)

This option can be used to define an assertion, as if by a preprocessor `#assert` directive (see [page 221](#)).

ASSERT= *SUBSTITUTE(...)

IDENTIFIER = <c-string 1..125 with-low> / <name 1..125 with-under>

<c-string> / <name> designates the name of the assertion.

TOKEN-STRING = <c-string 1..125 with-low> / <name 1..125 with-under>

<c-string> / <name> specifies the value or text to be substituted for the assertion designated by IDENTIFIER.

PREINCLUDE = *UNCHANGED / *NONE / <c-string 1..1024 with-low>

The pre-include option specifies an include file which is to be included at the start of the source program via an imaginary `#include` statement. The preprocessor searches for this include file in the USER-INCLUDE paths.

The include file specified via the PREINCLUDE option will be handled like an include file which is specified inside an `#include` statement at the beginning of the source program.

If several include files are to be pre-included then the corresponding `#include` statements should be collected together in a single include file and this include file should then be specified via the PREINCLUDE option.

COMMENTS = *UNCHANGED / *YES / *NO

This option can be used to specify whether the expanded and recompilable source program created by the preprocessor may also contain comments.

PREPROCESSING-MODE = *UNCHANGED / *ANSI / *KR

*ANSI: This is the default setting in all C and C++ language modes of the compiler. In other words, preprocessor behavior in accordance with the ANSI/ISO C standard is also supported in the K&R C mode by default.

*KR: The obsolete preprocessor behavior based on Reiser's `cpp` and Johnson's `pcc` can be turned on with *KR.

IMPLICIT-INCLUDE = *UNCHANGED / *YES / *NO

This option only affects C++ templates. It determines whether or not the definition of a template is included implicitly (see the [section "Implicit inclusion" on page 259](#)).

Common frontend options in C and C++

SIGNED-CHARACTER = *UNCHANGED / *YES / *NO

*NO: The data type `char` is unsigned by default.

*YES: `char` is treated as a signed `char` in expressions and conversions.

Note that the use of this option may result in portability problems!

AT-ALLOWED = *UNCHANGED / *YES / *NO

Determines whether the "at" sign '@' is allowed (*YES) or not allowed (*NO) in names.



The Cfront-C++ library contains declarations with the "at" sign (@) (see [page 277](#)).

DOLLAR-ALLOWED = *UNCHANGED / *YES / *NO

Determines whether the "dollar" sign '\$' is allowed (*YES) or not allowed (*NO) in names.

ENUM-TYPE = *UNCHANGED / *VALUE-DEPENDENT / *LONG

This option controls the handling of `enum` data.

*VALUE-DEPENDENT: depending on the range of values, `enum` data is represented as `char`, `short`, or `long`.

*LONG: `enum` data is always treated as objects of type `long`.

SIGNED-FIELDS = *UNCHANGED / *SIGNED / *UNSIGNED

*SIGNED: By default, signed bit fields are treated as signed.

*UNSIGNED: signed bit fields are always interpreted as unsigned. This option is only offered for compatibility reasons with older C versions and is only meaningful in K&R C mode.

PLAIN-FIELDS = *UNCHANGED / *SIGNED / *UNSIGNED

This option controls whether integer bit fields (`short`, `int`, `long`) are treated as signed (*SIGNED) or unsigned (*UNSIGNED) types. signed is the default.

PRESERVING = *UNCHANGED / *UNSIGNED / *LONG

This option controls whether arithmetic operations with operands of type `long` and unsigned `int` return a result of type `long` (*LONG) in accordance with K&R mode (first edition; see section 6.6 in the appendix) or of type unsigned `long` (*UNSIGNED) in accordance with ANSI/ISO C.

ALTERNATIVE-TOKENS = *UNCHANGED / *YES / *NO

This option controls whether alternative tokens are to be recognized by the compiler.

This includes:

- digraph sequences (e.g. <: for `␣`) in the C and C++ modes and
- additional keywords for operators (e.g. `and` for `&&`, `bitand` for `&`), which are only valid in the C++ language mode.

*YES is the default for the ANSI C++ modes.

*NO is the default for all other modes.

EXTERNAL-DEFINITION =

This option controls how the compiler reserves memory for the externally visible variables of a module. This is important if the program consists of several modules that are to be subsequently linked into an object program.

EXTERNAL-DEFINITION = *UNCHANGED

The values specified in the last MODIFY-SOURCE-PROPERTIES statement apply.

EXTERNAL-DEFINITION = *BY-SOURCE-LANGUAGE

The value of the EXTERNAL-DEFINITION option depends on the information in the language mode options:

```
LANGUAGE=*C(MODE=*KERNIGHAN-RITCHIE): MULTIPLY-ALLOWED
```

```
LANGUAGE=*C(MODE=*ANSI/*STRICT-ANSI): UNIQUE
```

```
LANGUAGE=*CPLUSPLUS(MODE=*ANSI/*STRICT-ANSI/*CPP): UNIQUE
```

EXTERNAL-DEFINITION = *UNIQUE

Externally visible variables may be defined in precisely one module only and must be declared in all other modules as `extern`. The memory space for such variables is set up in the data module of the object in which the variable has been defined. If the variable is defined in more than one module, an appropriate error message will be issued at the time of linking.

EXTERNAL-DEFINITION = *MULTIPLY-ALLOWED

This option is used for programs in which an externally visible variable is defined in more than one module, but is to be assigned to precisely one memory area. To achieve this, the variable must not be statically initialized in any definition. The compiler sets aside the memory for this variable in the COMMON area so that later, after linkage, only one memory area is assigned to the multiply defined variable.

If the variable is initialized statically in the definition, the memory area is created not in the COMMON area but in the data area. Assignment to precisely one memory area is then not possible!

LONGLONG = *UNCHANGED / *YES / *NO

This option determines whether the data type `long long` is recognized by the compiler.

*YES: By default, the data type `long long` is not recognized. In this case, the preprocessor define `_LONGLONG` is set. The data type `long long` is an extension to the ANSI C and C++ Standard.

*NO: The use of the data type `long long` results in an error.

END-OF-LINE-COMMENTS = *UNCHANGED / *YES / *NO

This option determines whether the compiler accepts C++ comments (`//...`) in C programs as well. C++ comments can only be allowed in the extended ANSI C mode. They are not allowed in the other C modes (STRICT-ANSI and KERNIGHAN-RITCHIE) and are always valid in the C++ modes.

*YES: The compiler accepts C++ comments in extended ANSI C mode.

*NO: The compiler does not accept C++ comments in extended ANSI C mode (default setting).

LITERAL-ENCODING = *UNCHANGED / *NATIVE / *ASCII / *ASCII-FULL / *EBCDIC / *EBCDIC-FULL

This option determines whether the C/C++ compiler object code for EBCDIC characters and EBCDIC literal strings creates characters and literal strings in EBCDIC or ASCII format (ISO 8859-1).

In C/C++, literal strings can contain binary coded characters as octal or hexadecimal escape sequences with the following syntax:

- octal escape sequences: `^[0-7] [0-7] [0-7]^`
- hexadecimal escape sequences: `^[x0-9A-F] [0-9A-F]^`

Whether or not the C/C++ compiler escape sequences are converted into ASCII format depends on the value specified for the option `LITERAL-ENCODING = ...`.

LITERAL-ENCODING = *UNCHANGED

The settings of the last MODIFY-SOURCE-PROPERTIES statement apply.

LITERAL-ENCODING = *NATIVE

The C/C++ compiler leaves the character and literal string code in the EBCDIC format, i.e. it transfers the characters and strings into the object code without converting them.

*NATIVE is the default setting.

LITERAL-ENCODING = *ASCII

The C/C++ compiler encodes the characters and literal strings in ASCII format. Strings containing escape sequences *will not be* converted into ASCII format.

LITERAL-ENCODING =*ASCII-FULL

The C/C++ compiler encodes the characters and literal strings in ASCII format. Strings containing escape sequences *will be* converted into ASCII format.

LITERAL-ENCODING =*EBCDIC

The C/C++ compiler leaves the character and literal string code in the EBCDIC format, i.e. it transfers the characters and strings into the object code without converting them.

LITERAL-ENCODING=*EBCDIC has the same effect as
LITERAL-ENCODING=*EBCDIC-FULL or LITERAL-ENCODING=*NATIVE

LITERAL-ENCODING =*EBCDIC-FULL

The C/C++ compiler leaves the character and literal string code in the EBCDIC format, it transfers the characters and strings into the object code without converting them.

LITERAL-ENCODING=*EBCDIC-FULL has the same effect as
LITERAL-ENCODING=*EBCDIC or LITERAL-ENCODING=*NATIVE

Requirements:

- If you are using ASCII notation for characters and literal strings, you must not declare C library functions explicitly in your source program. C library functions should be declared indirectly by including the corresponding CRTE headers. Otherwise the compilation error 'CFE1079[ERROR]...: Typangabe erwartet / expected a type specifier' can occur.
- If you select the ASCII or ASCII_FULL options, you should note the following. For *each and every* CRTE function (C library function) in your program that works with characters or strings, you must use the corresponding or matching include file. If you do not do this, the CRTE functions will not be able to process the character strings correctly. You should ensure that you include the include file <stdio.h> for the function *printf()* with <stdio.h>.
- In the CRTE runtime environment, some C library functions work with ASCII character strings.

If you are using ASCII character strings, you should specify the MODIFY-SOURCE-PROPERTIES statement as follows:

```
MODIFY-SOURCE-PROPERTIES      -
...
LITERAL-ENCODING=ASCII[-FULL] -
...
```

You must also specify the `MODIFY-MODULE-PROPERTIES` statement with the following entries:

```
MODIFY-MODULE-PROPERTIES      -
...
LOWER-CASE-NAMES=*YES,      -
SPECIAL-CHARACTERS=*KEEP,   -
...
```

Caution:

C++ library functions do not support ASCII format and must therefore possibly be replaced by C functions.

C++ specific frontend options

INSTANTIATION = *UNCHANGED / *NONE / *AUTO / *LOCAL / *ALL

This option is only relevant in the ANSI C++ modes. It controls how templates with external linkage are instantiated. This includes function templates as well as (non-static and non-inline) functions and static variables that are members of template classes. These templates types are combined under the generic term "template entity" below.

All instantiations requested explicitly with the instantiation directive `template declaration` or with the instantiation pragma `#pragma instantiate template-entity` are always created by the compiler for each compilation unit in all instantiation modes.

The remaining template entities are instantiated as follows:

***NONE:**

No instantiations other than those requested explicitly are created.

***AUTO (default setting):**

Instantiation is performed across all compilation units by means of a prelinker. This prelinker is activated with the `BIND` statement (see [page 66](#)). The principle of automatic instantiation is discussed in detail in the [section "Automatic instantiation" on page 246](#).

***LOCAL:**

Instantiations are created per compilation unit, i.e. all template entities that are used in a compilation unit are instantiated. The generated functions are given internal linkage. This mode provides a very simple mechanism for getting started with template programming. The compiler will instantiate the functions that are used in each compilation unit as local functions, and the program will link and run correctly. This method does, however, result in multiple copies of instantiated functions and is therefore not suitable for production use. Note that there may also be problems due to multiple copies of local static variables. For the same reasons, this method is also not suitable if one of the templates contains a `static` variable.

Warning:

The `basic_string` template contains a `static` variable in order to represent an empty string. If you use the `*LOCAL` option and the type `string` from the library, this empty string is no longer recognized. Try to avoid using this combination as it can lead to serious problems.

***ALL:**

Instantiations are created per compilation unit, i.e. all template entities that are declared or referenced in a compilation unit are instantiated. All member functions and static variables of a template class are instantiated, regardless of whether or not they are used. Template functions are instantiated even if they have only been declared.

VIRTUAL-FUNCTION-TAB = *UNCHANGED / *INTERNALLY-DEFINED / *GLOBALLY-DEFINED / *EXTERNALLY-DECLARED

This option can be used to specify how the virtual functions table is to be generated by the compiler.

***INTERNALLY-DEFINED:** The virtual functions table is declared `static` by default, i.e., a copy of the table is created for each module.

GLOBALLY-DEFINED / *EXTERNALLY-DECLARED:** This option can be used to reduce the memory requirement for modules by defining the table as global in just one module (GLOBALLY-DEFINED**) and then declaring it as `extern` (***EXTERNALLY-DECLARED**) in other modules.

Note

The **VIRTUAL-FUNCTION-TAB** option only has an effect on classes in which the normal heuristics for positioning the table of virtual functions do not work. It therefore only applies for classes which contain no “non-inline non-pure virtual function”.

USE-STD-NAMESPACE = *UNCHANGED / *YES / *NO

This option determines the use of ANSI C++ library functions for which names have been defined in the standard `std` namespace.

***YES** is the default setting in extended ANSI C++ mode. The compiler behaves as if the following lines were entered at the start of a compilation unit:

```
namespace std{}
using namespace std;
```

***NO** is the default setting in strict ANSI C++ mode and the only possible behavior in Cfront C++ mode.

If **USE-STD-NAMESPACE=*NO** is set in the extended or strict ANSI C++ mode, the source program must contain the statement `using namespace std;` otherwise, the names must be qualified appropriately before the first call to an ANSI C++ library function.

KEYWORD-BOOL = *UNCHANGED / *YES / *NO

This option can be used to define whether `bool` is recognized as a keyword.

*YES is the default setting in the ANSI C++ modes. In this case, the preprocessor macro `_BOOL` is defined.

*NO is the default setting and the only possible behavior in the Cfront C++ mode.

KEYWORD-WCHAR = *UNCHANGED / *YES / *NO

This option can be used to define whether `wchar_t` is recognized as a keyword.

*YES is the default setting in the ANSI C++ modes. In this case, the preprocessor macro `_WCHAR_T` is defined.

*NO is the default setting and the only possible behavior in the Cfront C++ mode.

LOOP-INIT = *UNCHANGED / *OLD / *NEW

This option defines how an initialization statement in `for` and `while` loops is to be treated.

*OLD is the default setting in the C++ mode and specifies that an initialization statement has the same scope as the entire loop.

*NEW is the default setting in the ANSI C++ modes and specifies the new ANSI C++-conformant scope rule, which surrounds the entire loop in its own implicitly generated scope.

SPECIALIZATION = *UNCHANGED / *OLD / *NEW

This option is only relevant in the ANSI C++ modes. It is used to enable or disable acceptance of the new `template<>` syntax for template specializations.

*NEW is the default setting. In this case, the compiler implicitly defines the macro `__OLD_SPECIALIZATION_SYNTAX` with the value 0.

If *OLD is specified, the compiler implicitly defines the macro `__OLD_SPECIALIZATION_SYNTAX` with the value 1.

MODIFY-TEST-PROPERTIES

Alias: SET-TEST-PROPERTIES

This statement controls whether information is generated for the AID debugger.

MODIFY-TEST-PROPERTIES
TEST-SUPPORT = *UNCHANGED / *YES / <u>*NO</u>

TEST-SUPPORT = *YES

Debugging information is generated for AID.

In order to debug without any restrictions with AID, the optimization and inline substitution of user-defined functions must be suppressed (see MODIFY-OPTIMIZATION-PROPERTIES). The compiler assumes INLINING=*NO for all configurations. The optimization level is reset to *LOW.

TEST-SUPPORT = NO

No debugging information is generated for AID.

However, call hierarchies can be traced back (e.g. by specifying %SDUMP %NEST after the program terminates).

PREPROCESS

Alias: DO-PREPROCESSING

This statement can be used to end the compilation of one or more source programs on completion of the preprocessor phase. During the preprocessing, an expanded and recompilable source program can be generated for each compilation unit.

PREPROCESS

SOURCE = *SYSDTA / list-poss: <filename 1..54>/ <posix-pathname> / *LIBRARY-ELEMENT(...)

*LIBRARY-ELEMENT(...)

LIBRARY = <filename 1..54> / *LINK(...)

 *LINK(...)

LINK-NAME = <filename 1..8>

,ELEMENT = <composed-name 1..64 with-under>(...)

 <composed-name 1..64 with-under>(...)

VERSION = *HIGHEST-EXISTING / <composed-name 1..24 with-under>

,OUTPUT = *NONE / *STD-FILE / *SOURCE-LOCATION / <filename 1..54>/

 <posix-pathname> / *LIBRARY-ELEMENT(...)

*LIBRARY-ELEMENT(...)

LIBRARY = *STD-LIBRARY / *SOURCE-LIBRARY / <filename 1..54> / *LINK(...)

 *LINK(...)

LINK-NAME = <filename 1..8>

,ELEMENT = *STD-ELEMENT(...) / <composed-name 1..64 with-under>(...)

 *STD-ELEMENT(...)

VERSION = *UPPER-LIMIT / *INCREMENT / <composed-name 1..24 with-under>

 <composed-name 1..64 with-under>(...)

VERSION = *UPPER-LIMIT / *INCREMENT / <composed-name 1..24 with-under>

SOURCE =

This option specifies one or more source programs to be compiled.

A source program can be read from the system file SYSDTA, a cataloged BS2000 file, a PLAM library or a POSIX file.

Note that if the source program is entered from SYSDTA, only one source program can be read per PREPROCESS statement.

SOURCE = *SYSDTA

Input is accepted from the system file SYSDTA. SYSDTA is assigned to the terminal in interactive mode but can be reassigned to a cataloged file or a PLAM library element with the ASSIGN-SYSDTA command (see also [page 76](#)).

SOURCE = <filename 1..54>

<filename> is the name of a cataloged BS2000 file.

SOURCE = <posix-pathname>

Only a file name is permitted as <posix-pathname>. See [page 34](#) for a description of the term <posix-pathname>.

SOURCE = *LIBRARY-ELEMENT(...)

This option is used to specify a PLAM library and an element in it.

LIBRARY = <filename 1..54>

<filename> assigns the name of a PLAM library.

LIBRARY = *LINK(...)**LINK-NAME = <filename 1..8>**

<filename> is used to specify a link name for a PLAM library. The link name must already have been assigned to the library name with the ADD-FILE-LINK command before the compiler is called.

ELEMENT = <composed-name 1..64 with-under>(...)

<composed-name> identifies the fully-qualified name of an element from the PLAM library specified earlier. The element must be of type S.

VERSION = *HIGHEST-EXISTING

If the element specification contains no version ID, the compiler uses the element with the highest existing version.

VERSION = <composed-name 1..24 with-under>

The compiler uses the element with the specified version.

OUTPUT =

This option can be used to specify if and where the result of the preprocessor run is to be stored.

OUTPUT = *NONE

No expanded and recompilable source program is generated. The result of the preprocessor run (expansions and error messages, if any) can only be checked by requesting a preprocessor listing.

OUTPUT = *STD-FILE

The expanded program is written by default to a cataloged BS2000 file. The name of this file is derived from the name of the source program as follows:

Output	*SYSDTA	BS2000 file	PLAM library	POSIX file
Default name	CSTDEXP.I	file.I	lib-elem.I	file.I

If the source program is located in a PLAM library, the library and element name of the source are combined with a hyphen (lib-elem) and used in the default file name. The rules by which the compiler constructs default names are described in detail in the [section “Default names for output containers” on page 48](#).

OUTPUT = *SOURCE-LOCATION

The output destination and name of the expanded program are derived from the location and name of the source program as follows:

Source	*SYSDTA	BS2000 file	PLAM library	POSIX file
Output destination	BS2000 file	BS2000 file	Library of source	Directory of source
Default name	CSTDEXP.I	file.I	elem.I (type S)	file.i (C source) file.I (C++ source)

The rules by which the compiler constructs default names are described in detail in the [section “Default names for output containers” on page 48](#).

OUTPUT = <filename 1..54>

The expanded program is written to a cataloged BS2000 file with the specified name. This specification is invalid when compiling multiple source programs.

OUTPUT = <posix-pathname>

The expanded program is written to a POSIX file system.

Both a file name and a directory are permitted as <posix-pathname>. See [page 34](#) for a description of the term <posix-pathname>.

When a file name is specified, the expanded program is stored under this name. Specification of a file name is invalid when compiling multiple source programs with one statement.

When a directory name *dir* is specified, the expanded program for each source program is written under the default name *sourcefile.i* (C source) or *sourcefile.I* (C++ source) to the directory *dir* (see also [section “Default names for output containers” on page 48](#)).

The directories specified with <posix-pathname> must already exist.

When constructing file names, it must be noted that expanded source programs can only be meaningfully processed further in the POSIX subsystem if the name contains the suffix *.i* or *.I* or a suffix defined with the `-Y F` option of the `cc/c89/CC` commands (see also the manual “POSIX Commands of the C/C++ Compiler” [1]).

OUTPUT = *LIBRARY-ELEMENT(...)

This option specifies the PLAM library (LIBRARY=) and the element name (ELEMENT=) under which the expanded program is to be stored. The elements are stored as elements of type S.

LIBRARY = *STD-LIBRARY

The expanded program is written by default to the library SYS.PROG.LIB.

LIBRARY = *SOURCE-LIBRARY

The expanded program is written to the PLAM library which contains the source program.

The *SOURCE-LIBRARY specification is invalid if the source program is read from a cataloged BS2000 file, a POSIX file or via SYSDTA.

LIBRARY = <filename 1..54>

The expanded program is written to a PLAM library with the specified name.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>

<filename> can be used to specify a valid link name for the PLAM library.

The link name must already have been assigned to the library name with the ADD-FILE-LINK command before the compiler is called.

ELEMENT = *STD-ELEMENT(...)

By default, the element name of the expanded program is derived from the name of the source program as follows:

Output	*SYSDTA	BS2000 file	PLAM library	POSIX file
Default name	CSTDEXP.I	file.I	elem.I	file.I

The rules by which the compiler constructs default names are described in detail in the [section “Default names for output containers” on page 48](#).

VERSION = *UPPER-LIMIT

If the element entry does not contain a version ID, the highest possible version is used by the compiler.

VERSION = *INCREMENT

The element is assigned a version number that is obtained by incrementing the highest existing version number by 1, assuming that the highest existing version ID ends with a digit that can be incremented. If the version ID cannot be incremented, the compiler run is aborted with an error message.

See the COMPILE statement ([page 75](#)) for an example.

VERSION = <composed-name 1..24 with-under>

The compiler uses the version specified.

ELEMENT = <composed-name 1..64 with-under>(…)

<composed-name> designates the fully-qualified element name of the expanded program. This specification is invalid when compiling multiple source programs.

VERSION = *UPPER-LIMIT / *INCREMENT /

<composed-name 1..24 with-under>

The version can be specified as described above for
ELEMENT=*STD-ELEMENT(…).

RESET-TO-DEFAULT

This statement resets the current option values in the corresponding MODIFY statements to the default values of the compiler, i.e., the values that take effect immediately after the START-CPLUS-COMPILER command.

RESET-TO-DEFAULT
<pre>SELECT = *UNCHANGED / <u>*ALL</u> / list-poss(10): *INCLUDE / *SOURCE / *MODULE / *OPTIMIZATION / *RUNTIME / *TEST / *DIAGNOSTIC / *LISTING / *CIF / *BIND</pre>

SELECT = *UNCHANGED

The value specified in the last RESET-TO-DEFAULT statement applies.

SELECT = *ALL

The option values of all MODIFY statements are reset to the default values of the compiler.

SELECT = list-poss(10): *INCLUDE / *SOURCE / *MODULE / *OPTIMIZATION / *RUNTIME / *TEST / *DIAGNOSTIC / *LISTING / *CIF / *BIND

Only the option values of the specified MODIFY statements are reset to the default values of the compiler. For example, *INCLUDE stands for MODIFY-INCLUDE-LIBRARIES, *SOURCE for MODIFY-SOURCE-PROPERTIES, etc.

In addition to the options of MODIFY-BIND-PROPERTIES statement, *BIND also resets the ACTION and OUTPUT-FORMAT options of the BIND statement to the default values of the compiler.

Information on default values of the compiler can also be obtained by means of the SHOW-DEFAULTS statement (see [page 154](#)). In this manual, the default values of the compiler are underlined.

SHOW-DEFAULTS

This statement can be used to show the default values of the compiler, i.e., the values which apply to the corresponding MODIFY statements immediately after calling the START-CPLUS-COMPILER command and which remain in effect until a value is explicitly changed. These default values can also be set with the RESET-TO-DEFAULT statement.

```
SHOW-DEFAULTS
```

```
SELECT = *UNCHANGED / *ALL / list-poss(10):*INCLUDE / *SOURCE / *MODULE / *OPTIMIZATION /
      *RUNTIME / *TEST / *DIAGNOSTIC / *LISTING / *CIF / *BIND
,OUTPUT = *UNCHANGED / *SYSOUT / *SYSLST
```

SELECT = *UNCHANGED

The value specified in the last SHOW-DEFAULTS statement applies.

SELECT = *ALL

Information on all MODIFY statements is shown.

SELECT = list-poss(10): *INCLUDE / *SOURCE / *MODULE / *OPTIMIZATION / *RUNTIME / *TEST / *DIAGNOSTIC / *LISTING / *CIF / *BIND

Information is shown only for the specified MODIFY statements.

For example, *INCLUDE stands for MODIFY-INCLUDE-LIBRARIES, *SOURCE for MODIFY-SOURCE-PROPERTIES, etc.

OUTPUT = *UNCHANGED / *SYSOUT / *SYSLST

The information is output via SYSOUT (default setting) or SYSLST.

Note

Some options of the MODIFY-SOURCE-PROPERTIES statement are displayed with the option value `__unset__`. These are options that can have different default values (so-called “alternative defaults”), depending on the language mode.

SHOW-PROPERTIES

This statement can be used to view the currently set option values in the selected or all MODIFY statements. In the output indirect values are replaced by real ones if possible.

```
SHOW-PROPERTIES
```

```
SELECT = *UNCHANGED / *ALL / list-poss(10): *INCLUDE / *SOURCE / *MODULE / *OPTIMIZATION /
        *RUNTIME / *TEST / *DIAGNOSTIC / *LISTING / *CIF / *BIND
,OUTPUT = *UNCHANGED / *SYSOUT / *SYSLST
```

SELECT = *UNCHANGED

The value specified in the last SHOW-PROPERTIES statement applies.

SELECT = *ALL

Information on all MODIFY statements is shown.

SELECT = list-poss(10): *INCLUDE / *SOURCE / *MODULE / *OPTIMIZATION / *RUNTIME / *TEST / *DIAGNOSTIC / *LISTING / *CIF / *BIND

Information is shown only for the specified MODIFY statements.

For example, *INCLUDE stands for MODIFY-INCLUDE-LIBRARIES, *SOURCE for MODIFY-SOURCE-PROPERTIES, etc.

OUTPUT = *UNCHANGED / *SYSOUT / *SYSLST

The information is output via SYSOUT (default setting) or SYSLST.

Note

The outputs of the SHOW-PROPERTIES and SHOW-DEFAULTS statements are virtually identical on starting the compiler and just after a RESET-TO-DEFAULT statement, except for the fact that SHOW-PROPERTIES displays the actual value (in accordance with the predefined ANSI C++ mode) instead of the option value `__unset__`.

3.3 Controlling the global listing generator

The global listing generator is called with the START-CPLUS-LISTING-GENERATOR command. The input source for the listing generator consists of CIF information, which is created by the compiler for each compilation unit and stored in PLAM library elements (type H), in cataloged BS2000 files or in POSIX files (see the MODIFY-CIF-PROPERTIES statement, [page 90](#)). All listings generated by the global listing generator are written to a single output file, which may be the system file SYSLST, by default, or the output file specified with the OUTPUT option. The listing generator uses the CIF information stored for local cross-reference and project listings to create global (i.e. multi-module) cross-reference and project listings. The remaining listings are generated per source file.

3.3.1 Calling the listing generator (START-CPLUS-LISTING-GENERATOR)

```
/START-CPLUS-LISTING-GENERATOR Abbreviations: S-CP-L-G, CPLUS-LISTING-GENERATOR, CPLG
```

```
MONJV = *NONE / <filename 1..54>  
,CPU-LIMIT = *JOB-REST / <integer 1..32767>
```

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

<filename> is used to assign a monitoring job variable in which the listing generator can indicate runtime errors that may occur. The status indicators and return codes used here are identical to those of the C/C++ compiler (see START-CPLUS-COMPILER, [page 60](#)).

3.3.2 Description of statements

Overview of statements and input rules

The following statements can be used to control the global listing generator:

- The modification statement MODIFY-LISTING-PROPERTIES defines the type, layout and output destination of the listing to be generated.
- The execution statement GENERATE-LISTING defines the input sources and starts the actual generation of the listing.
- The END statement terminates the global listing generator.
- The SDF standard statements (analogous to controlling the compilers; see [page 62](#))

The MODIFY-LISTING-PROPERTIES statement must precede the GENERATE-LISTING statement. If no MODIFY-LISTING-PROPERTIES statement is specified, no listings are generated (in accordance with the default setting). The general rules for the SDF statement interface of the compiler are also applicable to the global listing generator (see [“Basic principles and general input rules” on page 64](#)).

END

This statement ends the listing generator run.

```
END
```

GENERATE-LISTING

Aliases: DO-LISTING-GENERATION, LIST

This statement defines the input sources for the global listing generator and starts the generation of the listing. The scope, layout and output destination of listings depend on the specifications in a preceding MODIFY-LISTING-PROPERTIES statement.

GENERATE-LISTING

```

CIF-FILE = list-poss: <filename 1..54> / <posix-pathname> / *LIBRARY-ELEMENT(...)
*LIBRARY-ELEMENT(...)
    LIBRARY = *STD-LIBRARY / <filename 1..54> / *LINK(...)
        *LINK(...)
            LINK-NAME = <filename 1..8>
,ELEMENT = <composed-name 1..64 with-under>(…)
    <composed-name 1..64 with-under>(…)
        VERSION = *HIGHEST-EXISTING / <composed-name 1..24 with-under>

```

CIF-FILE = <filename 1..54>

<filename> specifies the name of a cataloged BS2000 file containing the CIF information.

CIF-FILE = <posix-pathname>

<posix-pathname> specifies the name of a POSIX file containing the CIF information.

See [page 34](#) for a description of the term <posix-pathname>.

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

This option is used to specify a PLAM library and one or more of its (type H) elements containing the CIF information.

LIBRARY = *STD-LIBRARY

The listing generator processes CIF elements contained in the library SYS.PROG.LIB by default.

LIBRARY = <filename 1..54>

<filename> assigns the name of a PLAM library.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>

<filename> is used to specify a link name for a PLAM library. The link name must be assigned to the library name by means of the ADD-FILE-LINK command before the listing generator is called.

ELEMENT = <composed-name 1..64 with-under>(…)

<composed-name> identifies the fully-qualified name of a CIF element from the PLAM library specified earlier. The element must be of type H.

VERSION = *HIGHEST-EXISTING

If the element specification contains no version ID, the listing generator uses the element with the highest existing version.

VERSION = <composed-name 1..24 with-under>

The listing generator uses the element with the specified version.

MODIFY-LISTING-PROPERTIES

Alias: SET-LISTING-PROPERTIES

This statement can be used to select which listings are to be generated by the listing generator. It can also be used to define the layout and the output destinations for these listings. Apart from a few deviations in the OUTPUT option, the syntax of this statement is identical to the compiler statement of the same name.

MODIFY-LISTING-PROPERTIES

```

OPTIONS = *UNCHANGED / *YES / *NO
,SOURCE = *UNCHANGED / *NO / [*YES](...)
  *YES(...)
    | MINIMAL-MSG-WEIGHT = *NOTE / *WARNING / *ERROR / *FATAL
,PREPROCESSING-RESULT = *UNCHANGED / *NO / [*YES](...)
  *YES(...)
    | COMMENTS = *YES / *NO
,DATA-ALLOCATION-MAP = *UNCHANGED / *NO / [*YES](...)
  *YES(...)
    | STRUCTURE-LEVEL = *UNCHANGED / *NONE / *MAX / <integer 0..256>
,CROSS-REFERENCE = *UNCHANGED / *NO / [*YES](...)
  *YES(...)
    | PREPROCESSING-INFO = *YES / *NO
    | ,TYPES = *YES / *NO
    | ,VARIABLES = *YES / *NO
    | ,FUNCTIONS = *YES / *NO
    | ,LABELS = *YES / *NO
    | ,TEMPLATES = *YES / *NO
    | ,ORDER = *STD / list-poss(6): *PREPROCESSING-INFO / *TYPES / *VARIABLES /
      *FUNCTIONS / *LABELS / *TEMPLATES
,PROJECT-INFORMATION = *UNCHANGED / *YES / *NO
,ASSEMBLER-CODE = *UNCHANGED / *YES / *NO
,SUMMARY = *UNCHANGED / *YES / *NO

```



```

,LAYOUT = *UNCHANGED / *FOR-NORMAL-PRINT(...) / *FOR-ROTATION-PRINT(...)
  *FOR-NORMAL-PRINT(...)
    | LINE-SIZE = *UNCHANGED / *STD / <integer 120..255>
    | ,LINES-PER-PAGE = *UNCHANGED / *STD / <integer 11..255>
  *FOR-ROTATION-PRINT(...)
    | LINE-SIZE = *UNCHANGED / *STD / <integer 120..255>
    | ,LINES-PER-PAGE = *UNCHANGED / *STD / <integer 11..255>
,INCLUDE-INFORMATION = *UNCHANGED / *NONE / *ALL / *USER-INCLUDES-ONLY
,LISTING-PRAGMAS = *UNCHANGED / *IGNORED / *INTERPRETED / *SELECT(...)
  *SELECT(...)
    | PAGE = *YES / *NO
    | ,TITLE = *YES / *NO
    | ,SPACE = *YES / *NO
    | ,LIST = *YES / *NO
,INITIAL-TITLE-TEXT = *UNCHANGED / *NONE / <c-string 1..256 with-low>
,OUTPUT = *UNCHANGED / *SYSLST / *SYSOUT / <filename 1..54> / <posix-pathname> /
  *LIBRARY-ELEMENT(...)
  *LIBRARY-ELEMENT(...)
    | LIBRARY = *STD-LIBRARY / <filename 1..54> / *LINK(...)
    *LINK(...)
      | LINK-NAME = <filename 1..8>
    ,ELEMENT = <composed-name 1..64 with-under>(…)
      <composed-name 1..64 with-under>(…)
      | VERSION = *UPPER-LIMIT / *INCREMENT / <composed-name 1..24 with-under>

```

OPTIONS = *UNCHANGED / *YES / *NO

*YES: The listing generator creates a listing of all default and user-defined compiler options.

SOURCE = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

SOURCE = *NO

No source/error listing is generated.

SOURCE = *YES(...)

A source/error listing is generated.

MINIMAL-MSG-WEIGHT = *NOTE / *WARNING / *ERROR / *FATAL

This option can be used to specify the minimum message weight for which error messages are to be included in the source listing. Note that the minimum message weight must be exactly the same as specified using the operand of the same name in the MODIFY-DIAGNOSTIC-PROPERTIES statement.

Examples

1. In the list of source programs, error messages as that are weighted higher than NOTE are to be output:

```
MODIFY-DIAGNOSTIC-PROPERTIES MINIMAL-MSG-WEIGHT=*NOTE
MODIFY-LISTING-PROPERTIES MINIMAL-MSG-WEIGHT=*NOTE
```

2. In the system file SYSOUT, error messages weighted higher than NOTE are to be output, and in the list of source programs, error messages weighted higher than WARNING:

```
MODIFY-DIAGNOSTIC-PROPERTIES MINIMAL-MSG-WEIGHT=*NOTE
MODIFY-LISTING-PROPERTIES MINIMAL-MSG-WEIGHT=*WARNING
```



When the value given in MAX-ERROR-NUMBER (controlled via MODIFY-DIAGNOSTIC-PROPERTIES) is reached, no further source program information will be output in the source/error listing. In this case the listing can no longer be used as a reliable guide to current error status.

PREPROCESSING-RESULT = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

PREPROCESSING-RESULT = *NO

The listing generator does not create a preprocessor listing.

PREPROCESSING-RESULT = *YES(...)

The listing generator creates a preprocessor listing.

COMMENTS = *YES / *NO

Comments from the source file are included in the preprocessor listing (can be suppressed with *NO).

DATA-ALLOCATION-MAP = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

DATA-ALLOCATION-MAP = *NO

The listing generator does not create a map listing.

DATA-ALLOCATION-MAP = *YES(...)

The listing generator creates a preprocessor listing.

STRUCTURE-LEVEL = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

STRUCTURE-LEVEL = *NONE

Structure elements are not included in the map listing.

STRUCTURE-LEVEL = *MAX

Structure elements up to the maximum nesting level (256) are included in the map listing.

STRUCTURE-LEVEL = <integer 0..256>

Only the structure elements up to the nesting level specified by <integer> are included in the map listing. If the specified as the nesting level is 0, no structure elements are included (corresponds to STRUCTURE-LEVEL=*NONE).

CROSS-REFERENCE = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

CROSS-REFERENCE = *NO

The listing generator does not create a cross-reference listing.

CROSS-REFERENCE = *YES(...)

The listing generator creates a "global" cross-reference listing, i.e. a cross-reference listing for multiple modules. This listing always contains a FILETABLE section with the names of all files, libraries, and elements that are used as sources by the compiler.

PREPROCESSING-INFO = *YES / *NO

The cross-reference listing may optionally include a list of names processed by the preprocessor.

TYPES = *YES / *NO

The cross-reference listing may optionally include a list of user-defined types (typedefs, structure, union, class and enumeration types).

VARIABLES = *YES / *NO

The cross-reference listing contains a list of variables (can be suppressed with *NO).

FUNCTIONS = *YES / *NO

The cross-reference listing contains a list of functions (can be suppressed with *NO).

LABELS = *YES / *NO

The cross-reference listing contains a list of labels (can be suppressed with *NO).

TEMPLATES = *YES / *NO

The cross-reference listing may optionally contain a list of templates.

ORDER = *STD / list-poss(6): *PREPROCESSING-INFO / *TYPES / *VARIABLES / *FUNCTIONS / *LABELS / *TEMPLATES

This option specifies the order in which the individual parts of the cross-reference listing are shown.

*STD: The default is in the order shown after list-poss above.

PROJECT-INFORMATION = *UNCHANGED / *YES / *NO

*YES: The listing generator creates a project listing showing a comparison of all external names originally used in the source program and the names internally generated by the compiler for the linkage editor.

ASSEMBLER-CODE = *UNCHANGED / *YES / *NO

*YES: The listing generator creates an object code listing.

SUMMARY = *UNCHANGED / *YES / *NO

*YES: The listing generator creates a listing containing statistical data on the compiler run.

LAYOUT =

This option can be used to define the page width (number of characters per line) and the page length (number of lines per page) for the listings.

If a line width of 120 characters is selected, all the listings will have narrower headers and footers. Text lines are wrapped only in the table listings (option, cross-reference and map listings). Overlong text lines in the source, preprocessor and object code listings may be truncated when the listings are printed.

When a BS2000 output file is specified, the first column of every line is reserved to control the line feed.

When the output is sent to a POSIX file, the appropriate POSIX control characters for line and page feeds are generated. The result is that the line length in the POSIX output file is up to 3 characters larger than the selected line width specification.

LAYOUT = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

LAYOUT = *FOR-NORMAL-PRINT(...)

LINE-SIZE = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

LINE-SIZE = *STD

132 characters per line are output.

LINE-SIZE = <integer 120..255>

120 to 255 characters per line are output.

LINES-PER-PAGE = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

LINES-PER-PAGE = *STD

64 lines per page are output.

LINES-PER-PAGE = <integer 11..255>

11 to 255 lines are printed per page.

The lower limit is fixed at 11 lines so that at least the listing header and footer and one line of text can be printed.

LAYOUT = *FOR-ROTATION-PRINT(...)

In order to print such listings, the ROTATION parameter must be specified in the PRINT-FILE command.

LINE-SIZE = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

LINE-SIZE = *STD

120 characters per line are output.

LINE-SIZE = <integer 120..255>

120 to 255 characters per line are output.

LINES-PER-PAGE = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

LINES-PER-PAGE = *STD

84 lines per page are output.

LINES-PER-PAGE = <integer 11..255>

11 to 255 lines are printed per page.

The lower limit is fixed at 11 lines so that at least the listing header and footer and one line of text can be printed.

INCLUDE-INFORMATION = *UNCHANGED / *ALL / *NONE / *USER-INCLUDES-ONLY

This option is used to specify which header files (if any) are to be shown in the source, preprocessor and cross-reference listings. By default, only the user-defined header files are shown, not the standard headers. Note that this option can only restrict the information created on the include files at compilation, include file that are explicitly or implicitly determined using the operand of the same name in the MODIFY-CIF-PROPERTIES statement.

LISTING-PRAGMAS =

This option controls which existing #pragma directives (if any) in the source text are to be interpreted when creating source and preprocessor listings.

A description of the #pragma directives can be found in the [section “Pragmas to control the layout of listings” on page 228](#).

LISTING-PRAGMAS = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

LISTING-PRAGMAS = *INTERPRETED / *IGNORED

All #pragma directives are interpreted (*INTERPRETED) or ignored (*IGNORED).

LISTING-PRAGMAS = *SELECT(...)

One or more of the following #pragma directives to control listings are interpreted (*YES) or ignored (*NO).

PAGE = *YES / *NO

Directive #pragma PAGE [*text*]:
for a page feed and optional line in the listing header

TITLE = *YES / *NO

Directive #pragma TITLE *text*:
for a line in the listing header

SPACE = *YES / *NO

Directive #pragma SPACE [*n*]:
to insert blank lines

LIST = *YES / *NO

Directive #pragma LIST[ING] ON or #pragma LIST[ING] OFF:
to suppress the output of source text lines

INITIAL-TITLE-TEXT = *UNCHANGED / *NONE / <c-string 1..256>

This option can be used to specify if an additional line is to appear in the header of the listing and the text that is to be entered in it. In contrast to pragmas, which only apply to source and preprocessor listings, the INITIAL-TITLE-TEXT option applies to all compiler listings. If the text is longer than the line length defined with the LINE-SIZE option (see [page 110ff](#)), it is split into multiple lines of appropriate length.

In the case of source and preprocessor listings, TITLE and PAGE pragmas (if any) override the INITIAL-TITLE-TEXT specification.

OUTPUT = *UNCHANGED

The value specified in the last MODIFY-LISTING-PROPERTIES statement applies.

OUTPUT = *SYSLST

The listings are written to the temporary system file SYSLST by default and are sent from there to the printer at the end of the task (at LOGOFF).

OUTPUT = *SYSOUT

The listings are written to the system file SYSOUT, which is assigned to the terminal in interactive mode.

OUTPUT = <filename 1..54>

The listings are written to a cataloged BS2000 file with the specified name.

OUTPUT = <posix-pathname>

The listings are written to a POSIX file.

Only a file name is permitted as <posix-pathname>. See [page 34](#) for a description of the term <posix-pathname>.

OUTPUT = *LIBRARY-ELEMENT(...)

This option specifies the PLAM library (LIBRARY=) and the element name (ELEMENT=) under which the listings are to be stored. The elements are saved as elements of type P.

LIBRARY = *STD-LIBRARY

The listings are stored in the library SYS.PROG.LIB by default.

LIBRARY = <filename 1..54>

The listings are written to a PLAM library with the specified name.

LIBRARY = *LINK(...)**LINK-NAME = <filename 1..8>**

A link name can also be specified instead of the library name.

<filename> designates the link name of the assigned library. This link name must already have been assigned to the PLAM library (with the ADD-FILE-LINK command) before the listing generator is called.

ELEMENT = <composed-name 1..64 with-under>(...)

The listings are written to a PLAM library element (type P) with the specified name.

VERSION = *UPPER-LIMIT

If the element entry does not contain a version ID, the highest possible version is used by the listing generator.

VERSION = *INCREMENT

The element is assigned a version number that is obtained by incrementing the highest existing version number by 1, assuming that the highest existing version ID ends with a digit that can be incremented. If the version ID cannot be incremented, the generation of the listing is aborted with an error message.

See the COMPILE statement ([page 75](#)) for an example.

4 Linkage and program execution

4.1 Linkage

The result produced by the C/C++ compiler on compiling a source program consists of a link-and-load module or LLM. This module already consists of machine code, but cannot be executed on the computer until all generated modules have been combined (i.e. linked) with other modules to create an executable unit.

The additionally required modules are usually modules of the runtime system, but other modules (e.g. modules of subroutines in other languages or separately compiled C/C++ program segments) can also be linked in, if required. These additional modules may include modules that were compiled by different compilers at various times.

The main function of the **linkage editor** is to select the modules required for the loadable unit from various sources (files, libraries) and link them to one another. Linkage basically means that the linkage editor supplements each module with the addresses that refer to areas external to the module (external references).

Before the unit generated during the linkage phase can be run, a **loader** must load it into memory so that the processor can access the code and execute it.

The following functional units are available in the **Binder-Loader-Starter** system of BS2000 for performing the linking and loading tasks:

- Dynamic binder loader DBL

The dynamic binder loader DBL combines three steps into a single operation by linking modules (object modules, LLMs) into a temporary loadable unit, loading it immediately into memory and then starting it.

- Linkage editor BINDER

BINDER links modules (object modules, LLMs) to form a logically and physically structured loadable unit. This unit is called a link-and-load module (Link and Load Module, LLM). BINDER stores an LLM as a type-L element in a PLAM library.

There are two ways of linking programs with BINDER:

1. directly, i.e. by calling BINDER with the START-BINDER command, and
2. implicitly, i.e. with the compiler statements BIND and MODIFY-BIND-PROPERTIES

The above compiler statements are not dealt with here, since they are described in detail in chapter 3 (see BIND on [page 66ff](#) and MODIFY-BIND-PROPERTIES, [page 79 ff](#)).

Modules generated in the ANSI C++ modes of the compiler contain symbol names in the EEN format (EEN = Extended External Name), which is supported by DBL only as of BLSSERV V2.0. The dynamic loading of ANSI C++ modules with DBL is therefore only possible with BLSSERV V2.0 or later. Static linkage of ANSI C++ modules must be performed with the BIND statement of the compiler.

See also the [section "Restriction on linking ANSI C++ programs" on page 178](#).

The C and C++ runtime modules needed for linking C/C++ programs are a component of the CRTE runtime system. An overview of all C/C++-specific CRTE libraries is provided in the [section "Specific CRTE components required for C/C++" on page 25ff](#). More detailed information, with appropriate notes on linking the C++ runtime libraries, in particular, can be found in the [chapter "The C++ libraries and C++ runtime system" on page 275](#).

4.1.1 Dynamic linking and loading with DBL

When the dynamic binder loader DBL is used, modules are temporarily linked into a loadable unit, which is then loaded into memory and executed, all in one operation. The generated load unit is automatically deleted after the program run.

The mode of operation of DBL is described in detail in the “Dynamic Binder Loader / Starter” manual [13].

Since the C/C++ compiler always generates modules only in LLM format, the START-EXECUTABLE-PROGRAM always must be used when linking and loading with DBL.

With this command, object modules and LLMs can be processed. Alternative libraries to be searched (runtime libraries and possibly other libraries) are assigned with the link name BLSLIBnn ($00 \leq nn \leq 99$). This is done with the ADD-FILE-LINK command before the linkage editor is called. For example:

```
/ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME=PLAM.USER  
/ADD-FILE-LINK LINK-NAME=BLSLIB02,FILE-NAME=$.SYSLNK.CRTE
```

To ensure that DBL searches these alternative libraries, the following entry must be made in the RESOLUTION parameter of the START-EXECUTABLE-PROGRAM command:

```
..ALTERNATE-LIBRARIES=*BLSLIB##
```

The linkage run with DBL is initiated by means of the START-EXECUTABLE-PROGRAM or LOAD-EXECUTABLE-PROGRAM command.

If the START-EXECUTABLE-PROGRAM command is used, the program is executed immediately. With the LOAD-EXECUTABLE-PROGRAM command, however, you have the option of entering additional commands (e.g. debugging commands).

```

/ { START-EXECUTABLE-PROGRAM
  LOAD-EXECUTABLE-PROGRAM } FROM-FILE=*LIBRARY-ELEMENT(LIBRARY=library,
  ELEMENT-OR-SYMBOL=mainmod),
  DBL-PARAMETERS=*PARAMETERS(RESOLUTION=
  *PARAMETERS(ALTERNATE-LIBRARIES=*BLSLIB##))

```

LIB=*library*,ELEM=*mainmod*

DBL accesses the specified PLAM library. The name of the module containing the main function must be specified as the element name.

RESOLUTION=*PARAMETERS (ALTERNATE-LIBRARIES=*BLSLIB##)

This entry is always required when LLMs are to be linked dynamically.

Load and start function of DBL

Any LLM that has been fully linked with BINDER (i.e. with all external references resolved) can be loaded and started with DBL without assigning alternative libraries as follows:

```
START-EXECUTABLE-PROGRAM FROM-FILE=*LIBRARY-ELEMENT(*LIBRARY=library,
  ELEMENT-OR-SYMBOL=module)
```

Recommended BLSLIBnn order for assigning CRTE libraries

1. C programs

```
/ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME=user-library
/ADD-FILE-LINK LINK-NAME=BLSLIB02,FILE-NAME=$.SYSLNK.CRTE
```

2. Cfront C++ programs

```
/ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME=user-library
/ADD-FILE-LINK LINK-NAME=BLSLIB02,FILE-NAME=$.SYSLNK.CRTE.CFCPP
/ADD-FILE-LINK LINK-NAME=BLSLIB03,FILE-NAME=$.SYSLNK.CRTE.CPP
/ADD-FILE-LINK LINK-NAME=BLSLIB04,FILE-NAME=$.SYSLNK.CRTE
```

3. ANSI C++ programs (using the Tools.h++ library)

```
/ADD-FILE-LINK LINK-NAME=BLSLIB01, FILE-NAME=user-library  
/ADD-FILE-LINK LINK-NAME=BLSLIB02, FILE-NAME=$.SYSLNK.CRTE.TOOLS  
/ADD-FILE-LINK LINK-NAME=BLSLIB03, FILE-NAME=$.SYSLNK.CRTE.STDCPP  
/ADD-FILE-LINK LINK-NAME=BLSLIB04, FILE-NAME=$.SYSLNK.CRTE.RTSCPP  
/ADD-FILE-LINK LINK-NAME=BLSLIB05, FILE-NAME=$.SYSLNK.CRTE
```

4.1.2 Linking with BINDER

BINDER enables object modules and LLMs to be linked into an LLM and stored as a type-L element in a PLAM library. BINDER is described in detail in the “BINDER” manual [14].

Note that modules generated in the ANSI C++ modes of the compiler cannot be linked with a direct call to BINDER. These modules can only be linked with the compiler statement BIND (see also the [section “Restriction on linking ANSI C++ programs” on page 178](#)).

Control statements for BINDER (selection)

```

/START-BINDER _____ (1)
//START-LLM-CREATION INT-NAME=name _____ (2)
//INCLUDE-MODULES MOD-CONTAINER=*LIB(LIB=bibliothek, ELEM=  $\left. \begin{array}{l} \textit{mainmod} \\ *ALL \end{array} \right\}$  ) _____ (3)
[//INCLUDE-MODULES MOD-CONTAINER=*LIB(LIB=..., ELEM=...)] _____ (4)
[//INCLUDE-MODULES MOD-CONTAINER=*LIB(LIB=$.SYSLNK.CRTE.POSIX, ELEM=*ALL)] _____ (5)
[//RESOLVE-BY-AUTOLINK LIB=..., [SYMBOL-NAME=externverweis]] _____ (6)
 $\left\{ \begin{array}{l} ([\$.SYSLNK.CRTE.CFCPP, \$.SYSLNK.CRTE.CPP,] \\ \$.SYSLNK.CRTE) \\ ([\$.SYSLNK.CRTE.CFCPP, \$.SYSLNK.CRTE.CPP,] \\ \$.SYSLNK.CRTE.PARTIAL-BIND) \end{array} \right\} ]$  _____ (7)
[//MODIFY-SYMBOL-VISIBILITY ..., VISIBLE=*NO] _____ (8)
//SAVE-LLM MOD-CONTAINER=*LIB(LIB=library, ELEM=element) _____ (9)
//END _____ (10)

```

- (1) BINDER is invoked.
- (2) This statement generates a new LLM with the internal name *name* in the work area. The generated LLM is saved with the SAVE-LLM statement (see 9) as a type-L element in a PLAM library.

- (3) *mainmod* is the name of the LLM that contains the `main` function.
library is the name of the PLAM library in which the object modules are stored. If *ALL is specified, all the modules from the specified input source are linked.
- (4) Further INCLUDE-MODULE statements can be used to link in additional modules from various libraries.
- (5) The library SYSLNK.CRTE.POSIX must always be linked if POSIX library functions are used. Since this “linkage option” library must be linked with precedence before the C runtime system, the INCLUDE-MODULES statement should always be used for linkage.
- (6) The RESOLVE-BY-AUTOLINK statements inform BINDER of the external references (=module names) and the corresponding libraries (or only the libraries) that are to be searched with the autolink mechanism for external references that are still unresolved. RESOLVE-BY-AUTOLINK statements for user-defined libraries and modules must always be specified before those for runtime libraries (see 7).
- (7) The respective CRTE runtime libraries to be linked are specified in a list.
All modules of the C runtime system that are required by the program are linked in permanently with a RESOLVE on the SYSLNK.CRTE library.
The RESOLVE on the library SYSLNK.CRTE.PARTIAL-BIND links in a connection module instead of the C runtime system. All external references to the C runtime system are satisfied from this module. The C runtime system itself is loaded dynamically at runtime. The fully linked module requires far less disk storage space than when statically linking C runtime modules from the library SYSLNK.CRTE. Furthermore, the program executes faster.
If no RESOLVE-BY-AUTOLINK statement is specified, the external references to the runtime system remain unresolved. The runtime modules are then dynamically linked and loaded at runtime (see the [section “Dynamic linking and loading with DBL” on page 171](#)).
- (8) The MODIFY-SYMBOL-VISIBILITY statement can be used to mask external symbols for subsequent linkage runs. The symbols remain visible by default. See also [“Masking symbols” on page 176](#).
- (9) This statement stores the current LLM generated with START-LLM-CREATION as a type-L element in a PLAM library.
- (10) The END statement terminates the BINDER run.

In the INCLUDE-MODULES and RESOLVE-BY-AUTOLINK statements, LIB=*BLS-LINK may also be specified instead of the library name (LIB=*library*). In this case, the libraries to be searched must be assigned with the link name BLSLIB nn ($00 \leq nn \leq 99$). This is done with the ADD-FILE-LINK command before BINDER is called. For example:

```
/ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME=PLAM.USER1  
/ADD-FILE-LINK LINK-NAME=BLSLIB02,FILE-NAME=PLAM.USER2
```

If all external references have been resolved, any LLM generated with BINDER can be loaded and started with DBL without assigning alternative libraries as follows:

```
START-EXECUTABLE-PROGRAM *LIBRARY-ELEMENT(LIB=library,ELEM=modul)
```

Masking symbols

In contrast to TSOSLNK, symbols (CSECTs, ENTRYs) are not masked by default when linking with BINDER and thus remain visible for subsequent linkage runs with BINDER or DBL.

This affects dynamic linking with DBL and can, among other things, have the following consequences:

If a PLAM library contains individual modules generated by the compiler as well as LLMs with a linked-in runtime system, then external references to the runtime system will be resolved from any prelinked module and not from the runtime library when the individual modules are linked dynamically. This results in several “DUPLICATES” warnings from DBL. Due to the autolink mechanism, the library in which the individual module resides is searched first, before the runtime libraries assigned with the link name BLSLIB nn .

To ensure that external references are always resolved from the current runtime library at link time rather than some arbitrary module,

- the individual modules and prelinked modules must be maintained in different libraries,
- or the symbols must be masked with the MODIFY-SYMBOL-VISIBILITY statement when linking with BINDER.

4.1.3 Shareable C/C++ programs

In the case of large programs, it is often advantageous to ensure that all individual program sections to be accessed concurrently by several users (or tasks) are made shareable.

The following compiler option must be specified at compilation in order to generate shareable programs:

```
//MODIFY-MODULE-PROP SHAREABLE-CODE=*YES
```

For each compilation unit, the compiler generates one LLM containing a non-shareable data CSECT and a shareable code CSECT. The code CSECT is marked with the attribute PUBLIC.

A subsequent linkage run creates a PUBLIC slice from the code CSECT and a PRIVATE slice from the data CSECT.

The PUBLIC slice is declared shareable by the system administrator with the ADD-SHARED command, and only the PRIVATE slice is subsequently loaded with the START-EXECUTABLE-PROGRAM command.

Example

```
/START-CPLUS-COMPILER
//MOD-SOURCE-PROP LANG=*C
//MOD-MODULE-PROP SHAREABLE-CODE=*YES
//COMPILE SOURCE=MODUL1.C,MODULE-OUTPUT=*LIB-ELEM(LIB=A.TEST)
//COMPILE SOURCE=MODUL2.C,MODULE-OUTPUT=*LIB-ELEM(LIB=A.TEST)
//END

/START-BINDER
//START-LLM-CREATION INT-NAME=TEST,-
//          SLICE-DEFINITION=BY-ATTRIBUTE(PUBLIC=*YES)
//INCLUDE-MODULES LIB=A.TEST,ELEM=(MODUL1)
//INCLUDE-MODULES LIB=A.TEST,ELEM=(MODUL2)
//SAVE-LLM LIB=B.TEST,ELEM=TEST
//END

/ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME=$.SYSLNK.CRTE
/START-EXECUTABLE-PROGRAM FROM-FILE=*LIBRARY-ELEMENT(-
/          LIB=B.TEST,ELEM=TEST),-
/          DBL-PAR=*PAR(RESOLUTION=*PAR(ALTERNATE-LIBRARIES=*BLSLIB##))
```

4.1.4 Restriction on linking ANSI C++ programs

It is only when using the BIND statement of the compiler that the correct linkage of an ANSI C++ program can be guaranteed. This is because, among other things, the automatic template instantiation by the prelinker is performed with the BIND statement (see also [page 244ff](#)).

4.2 Program execution

4.2.1 Parameter input at program start

By default, a C/C++ program is executed immediately as soon as it is invoked with the START-EXECUTABLE-PROGRAM command.

The PARAMETER-PROMPTING option of the MODIFY-RUNTIME-PROPERTIES statement (see [page 131](#)) can be used to control whether the standard I/O files are to be redirected before program execution and whether parameters can be passed to the `main` function.

PARAMETER-PROMPTING = *NO

By default, the program is executed immediately after it is started with the START-EXECUTABLE-PROGRAM command, i.e. without a preceding dialog step for the input of parameters. Only the name of the program is passed to the `main` function.

PARAMETER-PROMPTING = *YES

After it is started with the START-EXECUTABLE-PROGRAM command, the program issues the message:

```
CCM0001 enter options :  
*
```

The START-EXECUTABLE-PROGRAM or SRX command also enables the parameters to be transferred at the same time:

```
/START-EXECUTABLE-PROGRAM...,PROGRAM-PARAMETERS='...'
```

As in the command line of the UNIX operating system, one or more parameter lines can now be used

- to redirect the standard I/O files for C (`stdin`, `stdout` and `stderr`) and C++ (`cin`, `cout`, `cerr` and `clog`; see [page 180](#)), and
- to pass parameters to the `main` function (see [page 181](#)).

To ensure that these parameters can be addressed in the program, the `main` function must include two formal parameters, which are usually named `argc` (argument count) and `argv` (argument vector). See [page 183](#).

The individual entries (for redirection and `main` parameters) must be separated in the parameter line by whitespace characters.

If one parameter line is not sufficient for the input, the line can be terminated with a backslash (`\`), and the next parameter can be entered at the start of the following line. As with the whitespace character, the backslash (`\`) is interpreted here as a delimiter between two parameters (see also [page 182](#) for further meanings of `\`).

Example

```
*par1 par2 par3\  
*par4
```

Redirecting standard I/O files

At the start of program execution, the standard I/O files are assigned to the following BS2000 system files:

stdin/cin	SYSDTA
stdout/cout	SYSOUT
stderr/cerr/clog	SYSOUT

These default values can be modified in the parameter line. The dummy parameters *input* and *output* must be replaced by currently applicable values.

<input

The standard input (*stdin*, *cin*) is to be read in from *input*.

>output

The standard output (*stdout*, *cout*) is to be written to *output*.

If the file already exists, it is overwritten; if not, it is created as a new file.

>>output

The standard output (*stdout*, *cout*) is to be appended to *output*.

If the file does not exist, it is created as a new file.

2>output

The standard error output (*stderr*, *cerr*, *clog*) is to be written to *output*.

If the file already exists, it is overwritten; if not, it is created as a new file.

2>>output

The standard error output (*stderr*, *cerr*, *clog*) is to be appended to *output*.

If the file does not exist, it is created as a new file.

The following current values can be inserted for *input* or *output*:

(SYSDTA)

designates the system file SYSDTA. This specification can only be used for *input*.

(SYSOUT)

(SYSLST)

designate the system files SYSOUT and SYSLST. These entries are only valid for *output*.

filename

designates the name of a cataloged BS2000 file. This specification is valid for both *input* and *output*.

**POSIX(filename)*

filename designates the name of a POSIX file. This redirection specification is valid for both *input* and *output* and is only possible if the POSIX link option library is linked (see item 3 on [page 36](#)).

Files to be used as input files must already exist.

If an output file does not exist, it is created as a new file. If it already exists, it will either be overwritten (> or 2>) or extended by the addition of new output (>> or 2>>).

Note

The redirection of standard I/O files affects all I/O functions that read from standard input or write to standard output by default, as well as all functions that use the file pointers `stdin/stdout` or the file descriptors 0/1 as arguments. No redirection takes place for I/O operations on files that were explicitly opened with the names “(SYSDTA)”, “(SYSOUT)”, or “(SYSTEM)”.

Input of parameters for the main function

Input data that is in the parameter line but does not serve to redirect the standard I/O files (see [page 180](#)) is passed to the `main` function as parameters (i.e. actual arguments). In the program, these parameters can be processed as strings that are terminated with the null byte (`\0`).

In the explanation given below, the dummy parameters in italics must be replaced by currently applicable values:

%pattern

All file names that correspond to the specified pattern are passed as parameters.

pattern is a fully or partially qualified file name with wildcard syntax.

For compatibility reasons, further parameters can also be specified in order to control which files are selected, e.g.:

- file and catalog attributes (FCBTYPE, SHARE, etc.)
- creation and access date (CRDATE, EXDATE, etc.)

These parameters must be specified in the syntax of the ISP command `FSTAT`.

For example, the following specification supplies the names of all files that were created today with the suffix `.C`.

```
%*.c,cr=t
```

string' or "string"

string may contain any characters, including whitespace characters. These characters are passed to the program in an unchanged state and without the delimiting quotes (' or "). If the single or double quote that is used as a delimiter appears in the string, it must be preceded by a backslash (\). For example: '\quotation\' or "\"quotation\"".

The end-of-line character (\n) can also be passed to the program by terminating the line (with a backslash) within the string and entering any further characters as well as the closing quote in the continuation line. For example:

```
'Part 1\  
Part 2'
```

other-parameters

The character or characters are passed to the program directly. Whitespace characters and the backslash at the end of the line are treated as delimiters between two parameters.

If characters with a special meaning when passing parameters (e.g. % or >) are to be transferred to the program, they must be preceded by a backslash. In such a case, the backslash is removed, and the character itself is transferred. This method can also be used to cancel the special meaning of the backslash. Thus, \\ stands for a backslash without special meaning.

To enable the transfer of lowercase letters via procedure parameters, any uppercase letter that is preceded by a backslash is converted to the corresponding lowercase letter.

Effect of the backslash (summary):

\letter

The letter is passed as a lowercase letter. This makes it possible to pass lowercase letters even in cases where the BS2000 command interpreter automatically converts lowercase into uppercase (e.g. in procedure parameters).

\end-of-line

Backslash immediately followed by end-of-line.

Outside quoted strings, the backslash is interpreted as a delimiter between two parameters. Consequently, additional parameters can be entered in continuation lines.

Within quoted strings, the end-of-line loses its special meaning and is passed to the program as an end-of-line character (\n).

\other-character

The backslash is removed, and the *other-character* is passed. This method can be used to cancel the special meaning of specific characters.

Examples

Input:	Passed parameters:
\\AB\\C	"aBc"
%exp.	All file names with the prefix EXP.
'string with \ newline'	"string with \ \\nnewline"

Definition of the main function with parameters

Two formal parameters are required in the `main` function in order to enable the program to address data that has been entered in the parameter line (see [page 181](#)):

```
int main(int argc, char *argv[])
```

The names of the parameters (*argc*, *argv*) may be freely selected, but these are the names commonly used in the UNIX operating system.

The first parameter *argc* indicates the number of parameters that have been passed (including the program name).

The second parameter *argv* is a pointer to an array of char pointers (strings). The program name (in `argv[0]`) and all entered parameters are stored in it as strings that are terminated with the null byte (`\0`).

Example

The following example outputs the program name and all entered parameters.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    printf("Program name: %s\n", argv[0]);
    for (i=1; i<argc; ++i)
        printf("%d. parameter is: %s\n", i, argv[i]);
    return 0;
}
```

4.2.2 The advanced interactive debugger AID

C and C++ programs can be tested using the Advanced Interactive Debugger AID.

This User Guide only provides a brief description of AID. A complete description of the debugger can be found in the manual “AID, Debugging of C/C++ programs” [9].

AID offers the following features:

- It enables “symbolic” debugging, i.e. symbolic names from the source program may be specified in commands instead of hexadecimal addresses, provided LSD information is generated during compilation and passed to the loaded program.

It is not always necessary to load this information for the complete program together with that program. AID allows the LSD information to be dynamically loaded for each translation unit if the associated object modules are stored with the LSD information in a PLAM library. This means that resources can be used more efficiently:

- Program memory is freed, since LSD information needs to be loaded only when required for debugging
 - A program which is found to be error-free in the debugging session does not need to be recompiled or relinked (without LSD information) before being used.
 - If a program needs to be debugged when in productive use, the required LSD information will already be available without having to recompile and relink the program.
- It offers functions which, in particular, enable:
 - program execution to be traced and logged on a symbolic level (TRACE function)
 - program execution to be interrupted at fixed points or on the occurrence of defined events so that AID or BS2000 commands (so-called “subcommands”) can be executed
 - the contents of variables to be output in a format that takes data definitions of the source program into account
 - the contents of variables to be changed
 - call hierarchies to be traced even without LSD informations (%SDUMP %NEST).
 - In addition to the diagnosis of loaded programs, it supports the analysis of memory dumps in disk files.
 - It can be used in batch mode and in interactive mode. Note, however, that the interactive dialog mode is recommended for debugging programs, since the sequence of commands need not be defined in advance but can be adapted to suit the particular situation.

The following types of symbols can be addressed in C:

- simple (scalar) types
- arrays and elements of arrays
- structures/unions and their components
- enumeration constants (enum)
- bit fields
- pointers
- functions
- labels

Note, however, that preprocessor constants and macros (#DEFINES), typedef names, enum, structure and union types (labels), and inline substituted functions cannot be referenced.

In addition, the following types of symbols can be addressed in C++:

- functions and data elements within classes
- overloaded functions and operators
- references
- templates
- namespaces

Source references are used to refer to individual statement lines and block references to the start of blocks:

```
S' [<UNIQUE-no.> -] <line-no.> [: <rel.statement-no.>]' or
```

```
BLK = '<UNIQUE-no.> -] <line-no.> [: <rel.block-no.>]'
```

The UNIQUE number and line number also appear in the source/error listing.

Requirements for symbolic debugging

For debugging at symbolic level, AID enables variables to be addressed with the names defined in the source program and source references to be used to refer to individual statement lines. LSD information must be made available to AID for this purpose.

The generation and transfer of this information is controlled in each of the following steps by specifying appropriate operands in the control statements or commands:

- compiling with the C++ compiler
- linking and loading with the dynamic binder loader
- linking with BINDER
- linking with the compiler statement BIND

Compiling with the C/C++ compiler

The generation of LSD information is controlled with the following option:

```
//MODIFY-TEST-PROPERTIES TEST-SUPPORT = *NO / *YES
```

***NO** *NO is the default value, which means that the compiler does not generate LSD information. Even without LSD information, it is possible to trace the call hierarchies. In other words, %SDUMP %NEST can always be used if the program is aborted.

***YES** The compiler generates LSD information.
However, this is only possible for non-optimized programs. If optimization happens to be set (see the MODIFY-OPTIMIZATION-PROPERTIES statement), the compiler treats the request for debugging support as the higher priority, switches the optimization level to *LOW, and issues an appropriate message.

Linking, loading and starting

Once the LSD information has been generated during compilation, it is possible to

- load it together with the overall program or
- dynamically load it on request for each translation unit if the associated modules are maintained in a PLAM library.

The table below shows which operands have to be specified in both cases in order to transfer LSD information:

Method of linking/loading	Loading LSD information with the overall program:	Dynamic loading of LSD information by AID (%SYMLIB):
Linking, loading and starting with DBL	LOAD-EXECUTABLE-PROGRAM ..., TEST-OPTIONS=*AID or START-EXECUTABLE-PROGRAM ..., TEST-OPTIONS=*AID	LOAD-EXECUTABLE-PROGRAM ..., [TEST-OPTIONS=*NONE] or START-EXECUTABLE-PROGRAM ..., [TEST-OPTIONS=*NONE]
Linking with BINDER	START-LLM-CREATION ..., INC-DEF=*PAR(..., TEST-SUPPORT=*YES) or INCLUDE-MODULES ..., TEST-SUPPORT=*YES	START-LLM-CREATION ..., [INC-DEF=*PAR(..., TEST-SUPPORT=*NO)] or INCLUDE-MODULES ..., [TEST-SUPPORT=*NO]
Linking with the BIND statement	MODIFY-BIND-PROP ..., TEST-SUPPORT=*YES	MODIFY-BIND-PROP ..., [TEST-SUPPORT=*NO]
Loading and starting with DBL	LOAD-EXECUTABLE-PROGRAM ..., TEST-OPTIONS=*AID or START-EXECUTABLE-PROGRAM ..., TEST-OPTIONS=*AID	LOAD-EXECUTABLE-PROGRAM ..., [TEST-OPTIONS=*NONE] or START-EXECUTABLE-PROGRAM ..., [TEST-OPTIONS=*NONE]

A program to be tested should be loaded with the LOAD-EXECUTABLE-PROGRAM command so that AID commands can be subsequently entered.

A program that is to be processed with AID only if an error occurs can be loaded and started with the START-EXECUTABLE-PROGRAM command.

Note on linking the C runtime systems

If the modules from the CRTE library SYSLNK.CRTE.PARTIAL-BIND are linked, and the C runtime system itself is loaded dynamically only at runtime (which reflects the default setting when linking with the BIND statement of the compiler), the following restriction applies when debugging with AID: in the case of some program errors, e.g. due to invalid parameter passing to C library functions, AID cannot display the call hierarchy fully, since the last function before the error occurred may be missing. This restriction can be eliminated by linking the C runtime system statically. The following entry is required for this purpose when linking with the BIND statement of the compiler: MODIFY-BIND-PROPERTIES STDLIB=*STATIC.

Name of the translation unit in the S qualification

For LLMS, the so-called “source module name” is specified. This name is derived from the name of the source program by the C/C++ compiler as follows:

1. The <cat-id> and <user-id> name components, if present, are not used.
2. If the file or element name of the source program exceeds 32 characters, it is truncated from the right to 32 characters.

If the source module name contains periods, it must be specified in the S qualification using the n'name' notation.

Example

The file name of the source program is HELLO.C.

The S qualification in the %TRACE command, for example, would be:

```
/%t 1 in s=n'hello.c'
```

5 Linkage to functions and languages

The C++ compiler complies with the conventions of the program communications interface ILCS (Inter-Language Communication Services). In other words, it normally creates ILCS objects at compilation.

These objects can be linked to objects in other languages, provided such objects also comply with ILCS conventions.

A detailed description of the inter-language communication services (ILCS) and the general conventions for linkage to other languages can be found in the CRTE User Guide [4].

This chapter contains supplementary details with respect to special conventions that must be observed for linkage between C and C++ programs.

5.1 Linkage conventions specific to C and C++

Some of the specific conventions to be observed in C/C++ in addition to the ILCS conventions described in the CRTE User Guide are outlined below.

Parameter passing “by value”

In C/C++, parameters are passed “by value” (as defined in the respective language standards). The parameter list therefore contains the values of the individual parameters. This form of parameter passing is possible only if all objects to be linked are C and C++ objects.

The internal structure of the parameter list is described in the [section “Implementation-defined behavior based on the ANSI/ISO C standard” on page 216](#).

The following points must be noted with respect to the passing of floating-point numbers when specifying linkage from K&R C to ANSI C or C++ objects:

In K&R mode, `float` values are always passed as `double`. In ANSI C mode, `float` values are only passed as `double` if no prototype declaration exists; otherwise, they are actually passed as `float` values.

In C++ mode, `float` values are always passed unconverted.

The proper transfer of floating-point numbers between K&R C and ANSI C or C++ can only be guaranteed if floating-point numbers are declared `double` in the ANSI C or C++ program.

The following measures are necessary for linkage to objects in other languages:

- When a routine in some other language is called, the address of the data element to be transferred must be specified (e.g. `&par`). Technically speaking, the address of the data element is then passed “by value”.
- If a C/C++ routine is called from a routine in another language, the formal parameters in C and C++ must be declared as pointers to the data elements to be transferred, e.g. `f(T *par)`.

Passing function return values

In the case of a structure function, the result structure is copied to the parameter list, and a pointer to the parameter list is supplied in R1.

5.2 Linkage between C and C++

The C programming language is the foundation for C++. One of the main advantages of C++ is that libraries written in C can also be easily used.

External names, however, are handled differently in C and C++ implementations. In contrast to C, external names in C++ are coded for the linkage editor.

To enable the common use of functions and data in C++ and C (or other languages), the coding of external names must therefore be suppressed. The appropriate language element in C++ for this purpose is as follows:

```
extern "C" declaration;
```

or

```
extern "C" {  
    declarations  
}
```

The above `extern "C"` declarations are required in the C++ program for all functions and data that are defined in C++ as well as those which are defined in C (or other languages).

When standard C library functions are called from C++ source programs, only the corresponding standard header elements (e.g. `stdio.h`) need to be specified, since they already contain the `extern "C"` declarations for all C library functions.

To provide linkage between C and C++, only ANSI C objects should be used.

Linkage between C++ objects and K&R C objects is also technically possible, but may create problems in some cases (different restrictions apply to function declarations; `float` parameters are passed differently, etc.).

5.2.1 Common types

In order to use data and functions in C and C++, they must have the same or comparable types. C++ contains the same set of basic data types as C; however, since C++ is a true superset of C, it also contains more types than C.

The following C++ types may be used for linkage between C++ and other languages:

- **fundamental types:** char, short, int, long, float, double, long double, void
- **qualifiers:** unsigned, const, volatile
- pointers to common types
- arrays of common types (passed as pointers)
- structures and unions
These must not contain the following components in C++: member functions, constructors, destructors, base classes, and access specifiers. The structure and union components must have common types, i.e. must be syntactically identical.
- enumerations
- functions with common argument and result types
Ellipses (...) are permitted.

The following C++ types must not be used if linkage to other languages is required:

- reference types
- pointers to members of classes, unions, or structures
- classes
- structures and unions containing member functions, constructors, destructors, base classes, or access specifiers
- types defined locally in a class
- member functions
- templates

5.2.2 Calling C functions in C++

C functions can be used in C++ without any problems, provided each C function is declared in C++ with an `extern "C"` directive and its complete prototype.

Example:

C source:

```
/* file = C_file.c */

int error_level;

void error(int number, char *text)
{
    printf("Error %d, Reason: %s\n", number, text);
}
```

C++ source:

```
// file = C++_file.C

extern "C" int error_level;
extern "C" void error(int, char *);

int main(void)
{
    error_level = 100;
    error(error_level, "TEST");
    return 0;
}
```

The C++ source contains `extern "C"` declarations for all the C identifiers used in it.

5.2.3 Calling C++ functions in C

C++ functions can be used in C only if they have a type that can also be represented in C.

Depending on whether or not the C++ source is available, there are different ways to achieve linkage between functions.

If the C++ source code is available, the C++ function can be declared there with an `extern "C"` directive; however, it must be declared with `extern "C"` in every C++ source fragment that calls the function!

In the calling C source code, a "plain" `extern` declaration is required for the C++ function.

If the C++ source is not available or a general new declaration and recompilation is too cumbersome, an additional function level can be inserted. This is done by writing a so-called "wrapper" function in C++ and defining it as `extern "C"`. The wrapper function can then be called in C without restrictions.

Example

C++ function to be called:

```
int hidden(int i)
{
    // ...
}
```

Wrapper function:

```
extern int hidden(int);

extern "C" int wrapper(int i)
{
    return hidden(i);
}
```

C source fragment that calls the wrapper function:

```
extern int wrapper(int);

int main(void)
{
    printf("%d\n", wrapper(100));
    return 0;
}
```

This technique can also be used if a C++ function contains a parameter with a type that does not directly match a C type. In the case of a C++ function with a reference type parameter, for example, a wrapper function with a parameter of type pointer could be written and called instead.

5.2.4 Problems and restrictions

The combined usage of C and C++ in a program system is subject to the following restrictions:

- Free store management

The `new` and `delete` operators in C++ provide a separate facility for free store management. Storage space management in C, by contrast, is achieved by the functions `malloc` and `free`. These two methods must not be combined.

The effect of using `delete` to free storage that was requested with `malloc` (or using `free` on storage that was requested with `new`) is undefined.

- Standard I/O files

C++ offers a new interface for file processing. The C++ standard files `cin`, `cout`, `cerr` and `clog` correspond to the standard I/O files `stdin`, `stdout` and `stderr` in C.

Note that if the corresponding standard files are used in both C and C++, the behavior of the program cannot be guaranteed.

If the standard I/O files for C and C++ are to be used in combination in the same C++ program, the C++ function `ios::syn_with_stdio()` must be called (see the manual “C++ Library Functions” [5]).

5.3 Linkage between Cfront C++ and ANSI C++

It is not possible to combine Cfront C++ modules with ANSI C++ modules.

5.4 Notes on linkage to ILCS programs in other languages

Linkage to C++

In C++, external names are encoded for the link editor (BINDER).

If a combination of functions and data in C++ and some other ILCS language is being used, the encoding of external names must be suppressed. The C++ language construct for this purpose is:

```
extern "C" declaration;
```

or

```
extern "C" {  
    declarations  
}
```

Note that `extern "C"` declarations are needed in the C++ program not only for the data and functions defined in C++, but also for those defined in the other ILCS language.

Shared file processing

Shared files must be opened both in the C/C++ section and in the other language section. Control over their processing is implemented internally via different FCBs.

Since processing of a shared file is accomplished using different FCBs, alternate reading of the file into the C section and the other language section is not possible. All characters of the file are supplied both to the C section and to the other language section.

Calling the main function

Calling a `main` function is possible. `MAIN` must be then used as the entry address.

If there are several modules with `main` functions in a library, the desired `main` function can be selected by explicitly linking in the appropriate module (by using an `INCLUDE` statement instead of `RESOLVE`).

It is not possible to redirect the standard I/O files or pass parameters to the `main` function.

Shared STXIT event handling

ILCS distinguishes between implicit language-specific event handling and event handling that can be enabled and disabled explicitly.

Implicit language-specific event handling is restricted to the STXIT events ERROR and PROCHK, while explicit enabling and disabling is possible for all STXIT events.

STXIT routines explicitly enabled by routines in different languages are managed in parallel, i.e. when an event occurs, the enabled routines of all languages involved are called (in the order in which they were enabled).

Implicit event handling is rendered ineffective by an explicitly enabled routine.

C/C++ has no implicit language-specific event handling. Explicit enabling and disabling is possible using the C library functions `signal` and `cstxit`.

Enabling event handling routines for the STXIT event classes ERROR and PROCHK causes implicit event handling by other languages to be deactivated. On exiting from the C/C++ language environment, the event handling routines for ERROR and PROCHK should therefore be explicitly disabled in order to reactivate implicit event handling by other languages.

With the C library function `signal`, the routines are disabled by assigning `SIG_DFL`.

6 C language support of the compiler

The compiler supports the C language scope as defined by Kernighan & Ritchie as well as the ANSI/ISO standard (including the ISO C Amendment 1).

The Kernighan & Ritchie definition is documented in:

“The C Programming Language” by B.W. Kernighan and D.M. Ritchie, First Edition, Prentice Hall, 1978

The ANSI/ISO definition is documented in:

“The C Programming Language” by B.W. Kernighan and D.M. Ritchie, Second Edition, Based on Draft-Proposed ANSI C, 1988, Prentice Hall, ISBN 0-13-110362-8

“American National Standard for Information Systems - Programming Language - C” X3.159-1989, Doc. No. X3J11/90-013, February 14, 1990 and

“International Standard ISO/IEC 9899 : 1990, Programming languages - C”

The ISO C Amendment is documented in:

“International Standard ISO/IEC 9899 : 1990, Programming languages - C / Amendment 1 : 1994”

The following sections, which are intended as a supplement to the vendor-independent literature listed above, describe the implementation and machine-specific characteristics of this compiler and the various extensions to the standard C language definitions above.

Section 6.1 compares the C language modes of the compiler and points out the most important differences between them.

Section 6.2 describes implementation-defined behavior based on the ANSI/ISO standard.

Section 6.3 describes the C language extensions to the definition in the ANSI/ISO standard.

Section 6.4 describes the `#pragma` directives supported by this compiler.

6.1 Overview of the C language modes

In accordance with the different language standards defined for C, the compiler supports three C compilation modes:

K&R C mode (POSIX option `-X t`, SDF option `MODE=KERNIGHAN-RITCHIE`)

The compiler accepts C code based on the language definition by Kernighan & Ritchie as well as some ANSI-specific extensions.

Extended ANSI C mode (POSIX option `-X a`, SDF option `MODE=ANSI`) or strict ANSI C mode (POSIX option `-X c`, SDF option `MODE=STRICT-ANSI`)

The compiler accepts C code based on the ANSI/ISO definition.

The following table contains an overview of the language elements defined in the ANSI/ISO C standard and indicates which of those elements are supported in the K&R and ANSI C modes.

Key to the entries in the table:

X	Fully supported
XE	Extension to ANSI/ISO C that is fully supported in strict ANSI C mode, but results in a warning
o	Supported syntactically, but not semantically
–	Not supported
1) to 9)	Notes at the end of the table

C language elements	Language definition		Compilation mode		
	ANSI / ISO	K&R	extended ANSI	strict ANSI	K&R
Lexical elements					
Multibyte characters	X	–	X	X	X
Trigraph sequences	X	–	X	X	–
??= #					
??([
??/ \					
??)]					
??' ^					
??< {					
??!					
??> }					
??- ~					
Digraph sequences ¹⁾	X	–	X	X	X
<: [
:>]					
<% {					
%> }					
#: #					
%%: ##					
Escape sequences					
\a	X	–	X	X	–
\b	X	X	X	X	X
\f	X	X	X	X	X
\n	X	X	X	X	X
\r	X	X	X	X	X
\t	X	X	X	X	X
\v	X	–	X	X	X
\'	X	X	X	X	X
\"	X	–	X	X	X
\?	X	–	X	X	X
\\	X	X	X	X	X
\ <i>octdigits</i>	X	X	X	X	X
\ <i>x hexdigits</i>	X	–	X	X	X
Lengths of identifiers					
internal	31	8	all characters are significant		
external	6	<8	30/32		
Keywords ²⁾					

Overview of the C language modes

C language elements	Language definition		Compilation mode		
	ANSI / ISO	K&R	extended ANSI	strict ANSI	K&R
Constants					
integer	X	X	X	X	X
float	X	X	X	X	X
character	X	X	X	X	X
L' character'	X	–	X	X	X
string	X	X	X	X	X
L"string"	X	–	X	X	X
enum	X	X	X	X	X
Suffixes					
integer L, l	X	X	X	X	X
integer U, u	X	–	X	X	X
integer LL, ll ³⁾	–	–	X	XE	X
float F, f	X	–	X	X	X
float L, l	X	–	X	X	X
Data type declarations					
Type specifiers					
void ⁴⁾	X	–	X	X	X
void * ⁴⁾	X	–	X	X	X
char	X	X	X	X	X
short	X	X	X	X	X
int	X	X	X	X	X
long	X	X	X	X	X
long long ⁵⁾	–	–	X	XE	X
float	X	X	X	X	X
double	X	X	X	X	X
long double	X	–	X	X	X
signed	X	–	X	X	X
unsigned	X	X	X	X	X
array []	X	X	X	X	X
structure ⁶⁾	X	X	X	X	X
union ⁶⁾	X	X	X	X	X
(*)	X	X	X	X	X
enum	X	X	X	X	X
()	X	X	X	X	X
Type qualifiers					
const	X	–	X	X	o
volatile	X	–	X	X	o
Initialization					
auto aggregate	X	–	X	X	X

Overview of the C language modes

C language elements	Language definition		Compilation mode		
	ANSI / ISO	K&R	extended ANSI	strict ANSI	K&R
Storage classes					
typedef	X	X	X	X	X
extern	X	X	X	X	X
static	X	X	X	X	X
auto	X	X	X	X	X
register	X	X	X	X	X
Bitfield types					
int	X	X	X	X	X
signed int	X	X	X	X	X
all integral	–	–	X	XE	X
Conversion rules 7)					
value preserving	X	–	X	X	–
sign preserving	–	X	–	–	X
Functions					
Definition “old” 8)	X	X	X	X	X
Definition “new”	X	–	X	X	X
Prototyping 9)	X	–	X	X	o
Parameter type matching 10)	X	–	X	X	–
Preprocessor directives					
# (stringizing)	X	–	X	X	X
## (token pasting)	X	–	X	X	X
#assert / #unassert	–	–	X	XE	X
#define	X	X	X	X	X
defined	X	–	X	X	X
#elif	X	–	X	X	X
#else	X	X	X	X	X
#endif	X	X	X	X	X
#error	X	–	X	X	X
#include	X	X	X	X	X
#if	X	X	X	X	X
#ifdef	X	X	X	X	X
#ifndef	X	X	X	X	X
#ident	–	–	o	o	o

C language elements	Language definition		Compilation mode		
	ANSI / ISO	K&R	extended ANSI	strict ANSI	K&R
#line	X	X	X	X	X
#line (old style)	–	X	X	XE	X
#pragma	X	–	X	X	X
#undef	X	X	X	X	X
# (null directive)	X	–	X	X	X
Predefined macro names					
__LINE__	X	–	X	X	X
__FILE__	X	–	X	X	X
__DATE__	X	–	X	X	X
__TIME__	X	–	X	X	X
__STDC__	X	–	X	X	X
__STDC_VERSION__ 11)	X	–	X	X	–

Overview of the C language modes

*Notes***1) Digraph sequences**

Digraph sequences are defined in the ISO C Amendment 1 and are recognized in the C compilation modes only if the POSIX option `-K alternative_tokens` or the SDF option `ALTERNATIVE-TOKENS=*YES` is set.

2) Reserved keywords

<code>asm</code>	<code>continue</code>	<code>extern</code>	<code>long</code>	<code>static</code>	<code>void</code>
<code>auto</code>	<code>default</code>	<code>float</code>	<code>register</code>	<code>struct</code>	<code>volatile</code>
<code>break</code>	<code>do</code>	<code>for</code>	<code>return</code>	<code>switch</code>	<code>while</code>
<code>case</code>	<code>double</code>	<code>goto</code>	<code>short</code>	<code>typedef</code>	
<code>char</code>	<code>else</code>	<code>if</code>	<code>signed</code>	<code>union</code>	
<code>const</code>	<code>enum</code>	<code>int</code>	<code>sizeof</code>	<code>unsigned</code>	

The `asm` keyword is reserved in K&R mode and in extended ANSI mode. However, since the inline substitution of Assembler code is not yet supported, the use of this keyword will result in an error. The `asm` keyword is not reserved in the strict ANSI C mode.

3) Suffixes: LL, ll

These suffixes are an extension to ANSI/ISO C Standard. They identify integer constants of type `long long` (see [page 219](#)).

4) void

The type `void` signifies an empty set of values. It can be used in the following three ways:

1. Result type of functions which do not return a value.
2. A pointer to void points to an object of any data type.
3. The number of parameters and the data types of the parameters can be specified in a function declaration (see Prototyping). If `void` is used instead of the parameter list, then no parameters are defined.

5) Data type long long

This data type is an extension to the ANSI/ISO C Standard (see [page 219](#)).

`long long` is also supported in K&R mode.

6) structure, union

In accordance with ANSI C, structures and unions may be mutually assigned (if of the same type), passed to functions as parameters, and returned as exit values of functions.

These options are also supported in K&R mode.

7) Implicit arithmetic conversions

One important difference between ANSI and K&R lies in the area of implicit arithmetic conversions.

In K&R mode, operands of an expression are converted using the “unsigned-preserving” rule, i.e. extending an operand of type `unsigned char` or `unsigned short` produces a result of type `unsigned int`. If unsigned types appear in an expression together with other types, the result is always `unsigned`.

In ANSI mode, by contrast, the “value-preserving” rule applies, i.e. the result type depends on the size of the operand type. Extending an operand of type `unsigned char` or `unsigned short` thus produces a result of type `int` if `int` is large enough to represent all values of the smaller type. Otherwise, the result is an `unsigned int`.

Due to this difference, the results of arithmetic expressions could differ in some cases and thus lead to erratic program behavior. This must be taken into account when moving from K&R C to ANSI C.

8) Definition of functions

In contrast to K&R, ANSI has introduced a new syntax for the definition of formal function parameters, but also allows the “old-style” (K&R) syntax.

Both definition types are also supported in K&R mode.

9) Prototyping

In contrast to K&R, ANSI defines function prototypes. These are function declarations in which the number and types of individual parameters are also specified. This enables the compiler to compare the types of current parameters with those of formal parameters in the declaration and to adapt them to the formal parameters as required.

Prototype declarations are syntactically allowed in K&R mode, but have no semantic significance.

10) Parameter type matching

The advantage of prototyping is that the parameters specified in the function declaration are not subject to standard conversion rules. A parameter that is declared there as `float` will also be passed as `float`, without first being converted to `double`. If K&R and ANSI objects are to be combined, floating-point parameters should always be declared `double`.

The automatic matching of parameter types is only supported in ANSI modes.

11) `__STDC_VERSION__`

This preprocessor macro is defined in the ISO C Amendment 1 and has the value 199409L in the extended and strict ANSI modes. It is not defined in K&R mode.

6.2 Implementation-defined behavior based on the ANSI/ISO C standard

Identifiers

In principle, both internal and external names can have any length. In the case of internal names, all characters are significant. For external names, the compiler evaluates a maximum of 32 characters by default (see the rules below). In accordance with ANSI/ISO C, the following characters are allowed when constructing names: the digits 0 to 9, the uppercase letters A to Z, the lowercase letters a to z, and the underscore `_`. Furthermore, as an extension to ANSI/ISO C, the “dollar” character `$` and the “at” sign `@` are also allowed in names by default, but this can be turned off with the appropriate options (`-K no_dollar`, `-K no_at` and `DOLLAR-ALLOWED=*NO`, `AT-ALLOWED=*NO`).

Multibyte characters are not supported in identifiers.

The following applies to external names:

- By default, i.e. when the options `-K c_names_std` and `C-NAMES=*STD` are set, external names can have a maximum length of 32 characters. Longer names are truncated by the compiler to 32 characters.
Up to 30 characters may be used when generating shared code (with the `-K share` and `SHAREABLE-CODE=*YES` options).

If the options `-K c_names_unlimited` and `C-NAMES=*UNLIMITED` are set, no name truncation occurs. The compiler generates entry names in the EEN format. EEN names can have a maximum length of 32000 characters.
- By default, lowercase letters are converted to uppercase, and underscores (`_`) are converted to the dollar character (`$`). These conversions can be suppressed by specifying the appropriate options (`-K llm_case_lower`, `-K llm_keep` or `LOWER-CASE-NAMES=*YES`, `SPECIAL-CHARACTERS=*KEEP`) so that lowercase letters and underscores are retained in external names.
- External names must not begin with “l”.

The above rules also apply to external names that are declared as `extern "C"` in C++ and also for static functions.

main function

The compiler allows the return types `int` and `void` for the `main` function.

Two formal parameters are provided for the `main` function to allow arguments to be passed to a program in a call:

```
int main(int argc, char *argv[])
```

The first parameter *argc* shows the number of passed arguments. Since the first argument `argv[0]` is conventionally the program name, the number of arguments is at least 1.

The second parameter *argv* is a pointer to an array of strings. It holds the program name (in `argv[0]`) and all arguments entered in the program call in the form of strings terminated with the null byte (`\0`).

As an extension to ANSI/ISO C, it is also possible to declare a third parameter `char *envp[]` for the `main` function (see [page 219](#)).

More details on passing parameters to `main` functions can be found in the [section “Input of parameters for the main function” on page 181](#).

Characters

By default, the data type `char` is treated as `unsigned` by the compiler (see also the `-K uchar`, `-K schar` and `SIGNED-CHARACTER=*NO/*YES` options).

The value of an EBCDIC character is always positive.

The value of `'\377'` (octal) or `'\xFF'` (hexadecimal) is thus 255.

If a character constant contains a numeric value that is not included in the EBCDIC character set, behavior is undefined.

The value of a character constant that contains more than one character (e.g. `'ab'`) is computed from the EBCDIC value of the character as a number to the base 256. The first (right) character is multiplied by 1, the second character by 256, the third character by $256 * 256$, the fourth character by $256 * 256 * 256$.

For example, `'abcd'` produces the value $'a' * 256^3 + 'b' * 256^2 + 'c' * 256 + 'd'$ (= 2172814212).

The value of a multibyte character constant in the form `L'ab'` is identical to the value of a character constant in the form `'ab'` in this implementation.

If a character constant contains five or more characters, an error occurs, and no code is generated.

The assignment of `int` to `char` occurs modulo 256.

Multibyte characters

In this implementation, multibyte characters always have a length of 1 byte, and `wchar_t` values are always 32-bit integer values.

Pointers

A pointer is represented in 4 bytes and is aligned on a word boundary. The difference between two pointers is of type `int` (`ptrdiff_t`).

Arrays

In C, arrays always have fixed limits; the size of an array is thus already known at compile time. An array name in C is always treated as a pointer that points to the first element of the array.

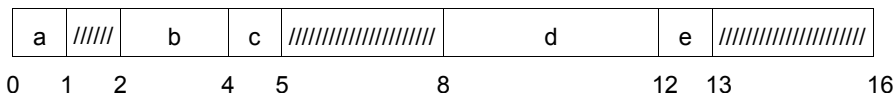
The elements are sequentially stored in memory; the first element has the index zero. In the case of multi-dimensional arrays, the elements are stored in memory in such a way that the last index is the first to vary. Like the array itself, each element is aligned in accordance with the element type.

Structures

In structures, components occupy space in the order of their declaration. Each component is aligned in accordance with its type. The structure itself is aligned on the maximum alignment size required for a component. The size of the structure is a multiple of this alignment so that arrays can be constructed from these structures. See also [“Internal representation of data types \(alignment and representation in registers\)” on page 215](#) and the preprocessor directive `#pragma aligned` on [page 223](#).

Example

		Size:	Alignment:	Offset:
struct {	char a;	1 byte	byte boundary	0 (word boundary)
	short b;	2 bytes	half-word boundary	2
	char c;	1 byte	byte boundary	4
	long d;	4 bytes	word boundary	8
	char e;	1 byte	byte boundary	12
}				16 (structure end)



Bitfields

Bitfields are stored from left to right in a maximum of 32 bits (one word).

Bitfields can be defined as follows:

```
int          unsigned int      signed int
long        unsigned long      signed long
short       unsigned short     signed short
char        unsigned char      signed char
```

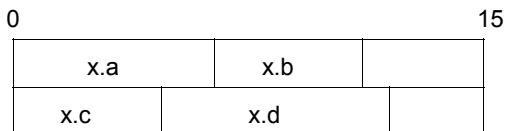
Bitfields without the `unsigned` or `signed` keyword are represented in accordance with the base type, i.e. `char` as unsigned and `int`, `long` and `short` as signed. If `signed` or `unsigned` is specified explicitly, the bitfields are represented accordingly. This default behavior can be modified by means of the following options:

`-K schar`, `-K signed_fields_unsigned` and `-K plain_fields_unsigned` or `SIGNED-FIELDS=*UNSIGNED`, `PLAIN-FIELDS=*UNSIGNED`, `SIGNED-CHARACTER=*YES`.

If the bitfield fits in the current byte, half-word, word or double-word, the specified number of bits are placed in it without being aligned; otherwise, the bitfield is aligned on a byte, half-word, word or double-word boundary in accordance with its base type (see example below).

Example

```
struct
{
    unsigned short  a : 7;
    unsigned short  b : 5;
    unsigned short  c : 5;
    unsigned short  d : 8;
} x;
```



Enumerations (enum)

Without an explicit value assignment, the numbers 0, 1, etc. are sequentially assigned to the constants when an enumeration type is defined. If a value is explicitly assigned to a constant, the following constants automatically receive a correspondingly higher value.

By default, an enumeration type is represented as `char`, `short` or `long`, depending on the threshold limits (highest and lowest values). Regardless of the actual storage space requirements, `enum` data can always be represented as `long` by using the `-K enum_long` or `ENUM-TYPE=*LONG` compiler options .

Type qualifier volatile

`volatile` prevents optimization on accessing a variable. This means that instead of using the old contents, new values are always read from storage. For all assignments, including redundant ones, the appropriate value is directly written to storage. In contrast to non-`volatile` objects, which are subject to extensive optimization and are typically held in registers, the implementation guarantees that all references to `volatile` objects will always point to values in storage.

`volatile` is only accepted syntactically in K&R mode.

size_t

In this implementation, `size_t` corresponds to `unsigned int`.

ptrdiff_t

In this implementation, `ptrdiff_t` corresponds to `int`.

Conversion of data types

– integer --> integer

When an unsigned integer value is converted to a signed integer type of the same size, the bit pattern is retained. If the value cannot be accommodated, the result corresponds to the subtraction of the largest possible number + 1 from the given size.

If a conversion of an integer value to a smaller integer type is involved, and the value cannot be accommodated, the bit pattern is retained and the higher-valued bits are truncated.

- floating-point number --> integer

When a floating-point number is converted to an integer, the number is truncated toward zero.

Example

`(int)(-1.5)` is -1

`(int)(1.5)` is 1

The result is undefined if the floating-point number to be converted is too large to be represented as an integer value.

- integer --> floating-point number

The conversion of an integer to a floating-point type that cannot accept the correct value is accomplished by rounding.

- floating-point number --> floating-point number

The conversion of a floating-point number to a smaller floating-point number (e.g. `double` to `float`) is accomplished by rounding.

- integer <--> pointer

When an integer is converted to a pointer, and vice versa, the bit pattern is not changed (simple reinterpretation).

Sign of division remainder

The remainder of an integral division always has the same sign as the dividend.

Example

`(-5) / 2` is -2, `(-5) % 2` is -1

`5 / (-2)` is -2, `5 % (-2)` is 1

Logical and arithmetic right shift

If the left operand is unsigned, the right shift is logical (padding of 0 bits); otherwise, arithmetic (padding of signed bits).

Example

`(-8) >> 1` is -4

Bitwise operations on signed integer values

Bitwise operations (operators `~`, `<<`, `&`, `^`, and `|`) are executed as unsigned integers on interpretation; however, the result is signed.

Declarators

Any number of declarators may be used to declare a type.

switch statement

Any number of `case` branches may be used per `switch` statement.

Preprocessor directives

– `#include`

`#include` directives cannot be specified with a sequence of `<name>` or “name” headers. Only the first name is accepted.

The compiler accepts `#include` directives in which the names of headers contain slashes (`/`) for directories even in the case of PLAM library elements. Every slash in the names of user-defined and standard headers is internally converted to a period for the search in PLAM libraries.

Consequently, in source programs which are ported out of POSIX or UNIX system, for example, the slashes need not be converted to periods.

Example

```
#include <sys/types.h>
```

The compiler looks for the standard header `SYS.TYPES.H` in the CRTE library `$.SYSLIB.CRTE`.

There are no restrictions with respect to the nesting of header files.

– `#pragma`

See the [section “Pragmas” on page 223](#).

– `__DATE`, `__TIME__`

If the date and time of compilation are not available, these macros are defined as follows:

```
__DATE__      "Jan 1 1970"  
__TIME__      "01:00:00"
```

Size and value ranges for elementary data types

Type	Bit	Value ranges
char	8	0 .. 255
signed char	8	-128 .. 127
short	16	-32768 .. 32767
unsigned short	16	0 .. 65535
int	32	-2147483648 .. 2147483647 ($-2^{31} .. 2^{31}-1$)
unsigned int	32	0 .. 4294967295 ($0 .. 2^{32}-1$)
long	32	same as int
unsigned long	32	same as unsigned int
long long *)	64	-9223372036854775808 .. 9223372036854775807 ($-2^{63} .. 2^{63}-1$)
unsigned long long *)	64	0 .. 18446744073709551615 ($0 .. 2^{64}-1$)
float	32	$10^{-75} .. 0.79 \cdot 10^{76}$
double	64	same as float
long double	124	same as float

*) The data type `long long` is an extension to the ANSI/ISO C Standard; see also [page 219](#).

Internal representation of data types (alignment and representation in registers)

This section illustrates how individual C data types are internally represented in memory.

For scalar types, additional details are provided on their representation in registers. On the one hand, this defines how the variables are represented with the `register` storage class; on the other, it illustrates how the value of such a variable is interpreted in expressions.

Data type	Size	Alignment	Representation in registers
char, unsigned char, signed char	1 byte	byte boundary	right aligned
short, unsigned short	2 bytes	half-word boundary	right aligned
int, unsigned int	4 bytes	word boundary	as in memory
long, unsigned long	4 bytes	word boundary	as in memory
long long, unsigned long long	8 Byte	double-word boundary	no representation in registers
pointer	4 bytes	word boundary	as in memory
float	4 bytes	word boundary	left aligned
double	8 bytes	double-word boundary	no conversion is required for representation
long double	16 bytes	double-word boundary	represented by a pair of floating-point registers

Data type	Size and alignment
Enumerations	Represented as char, short or long with corresponding alignment, depending on limits.
Arrays	Size and alignment correspond to element type.
Structures	Size and alignment for individual components based on above rules; overall alignment based on maximum alignment for components.
Bitfields	If the alignment boundary for the base type is not exceeded, the specified number of bits is created without alignment; otherwise, the bits are aligned in accordance with the base type.

Implementation-defined limits

Most limits depend on the available system resources (e.g. on virtual memory). Only the following limits are implementation-defined:

Characteristic	Maximum value
Number of parameters in a macro definition	$2^{24}-1$
Number of arguments in a macro call	$2^{24}-1$
sizeof limit	2^{31}

Storage classes

This section summarizes how storage space is assigned to variables, depending on their storage class.

Storage class register

Variables can be declared as register variables with `register`. This is a hint to the compiler that the variables are used relatively often and should therefore be held in registers. This saves the high overhead of accessing storage when reading and writing such variables. Note, however, that the optimization mechanism of the compiler may ignore such hints and implement variables as register variables in accordance with its own algorithm.

Storage class auto (default)

Storage space is reserved in an Automatic Data Area for local variables with the (predefined) storage class `auto`.

Parameters in the parameter list

Function parameters are passed in the order of their appearance in a parameter list.

All `unsigned...` parameters are represented as `unsigned`; all other integer parameters (`char`, `short`) as `int`: right-justified in one word each, aligned on a word boundary, and padded on the left with sign bits (`int`) or zeros (`unsigned...`) where necessary. Pointers occupy one word.

Depending on the language mode, floating-point numbers are passed differently.

In K&R mode, floating-point numbers (`float`, `double`) are always passed in double precision, i.e. as a double-word aligned on a double-word boundary.

In ANSI mode, `float` values are passed in double precision only if no prototype declaration is present. Otherwise, `float` values are passed in single precision, i.e. as a word aligned on a word boundary.

In C++ mode, `float` values are always passed in single precision, since prototype declarations must be present.

`long double` is passed in two double-words, aligned on a double-word boundary.

Structure parameters are aligned on a word or double-word boundary as required. The size of a structure is padded in accordance with the maximum alignment requirement for a component. For example, if a structure contains only `short` and `char` components, the size will be a multiple of 2 bytes.

Arrays cannot be passed as values. A pointer to the first array element is passed.

Static variables

The compiler reserves storage space for the following types of static variables already at the time of compilation:

- local static variables
- global static variables
- global external variables

The difference between these storage classes lies in their scope:

- Local static variables are variables that are defined with the `static` storage class specifier within a function. They are only recognized in the function in which they are defined.
- Global static variables are variables that are defined with the `static` storage class specifier outside a function. They are only recognized within a compilation unit.
- Global external variables are variables that are defined outside a function without the `static` storage class specifier. These variables can also be accessed in other compilation units, provided they are declared there with the `extern` storage class specifier.

Functions without a prototype

If a function without a prototype is called and there is parameter information present, then an error may be output in some cases. An error is output when an “old style” definition or a prototype in the K&R mode is found.

If the argument and the parameter are of different types (according to their customary type extensions) and one of the following arises, then an error is output:

- Parameter and argument are of different sizes
- Parameter and argument have different alignments
- The parameter is of type float, double or long double
- The argument is of type float, double or long double

The error can be downgraded to a warning. If this is done, the call made at runtime will generally fail.

6.3 Extensions to ANSI/ISO C

The extensions described below do not conform to the language features described in the ANSI/ISO standard and could thus result in potentially non-portable source programs if used.

Special character \$ and @ in identifiers

By default, the “dollar” symbol \$ and the “at” sign @ are permitted in internal and external names. This can be suppressed by means of options (see [“Identifiers” on page 208](#)).

main function with three parameters

In addition to the two parameters *argc* and *argv* (see [page 209](#)), a third parameter `char *envp[]` may be declared, where *envp* is a pointer to an array of strings that is supplied with information on the system environment. See the manual “C Library Functions for POSIX Applications” [3] for details.

Scope of functions

`extern` declarations of functions within blocks apply to the entire compilation unit. If multiple `extern` declarations are available for the same function, they are tested for a match.

Write access to string literals

In this implementation, string literals can be overwritten by default. This ensures that the literals do not overlap. Identical literals are stored in separate areas.

The `-K rostr` and `STRING-LITERALS=*READ-ONLY` options can be used to specify read-only access for string literals.

Data type long long

The data type `long long` (together with `unsigned long long`) is represented in 8 bytes, aligned on a double-word boundary. Constants of type `long long` are identified with the suffixes `LL` or `ll` following the number. When a constant can no longer be represented as an `unsigned long`, it is treated as a constant of type `long long`. A `long long` constant is `unsigned` if it includes the additional suffix `U` or `u` or if it is too large to be represented as a `signed long long`.

The following restrictions apply when using the data type `long long`:

Bitfields, array subscripts and expressions in `switch` statements of type `long long` are not supported.

Conversion of function pointers

The cast operator can be used to convert pointers to objects to pointers to functions, and vice versa. In the case of implicit conversions, warnings are issued by the compiler.

Non-integer bitfields

All integral types (except `long long`) can be used as bitfields (see also [“Bitfields” on page 211](#)). The standard only defines the types `int`, `unsigned int` and `signed int`.

The `asm` keyword

The `asm` keyword is reserved in K&R mode and in extended ANSI mode. However, since the inline substitution of Assembler code is not yet supported, the use of this keyword will result in an error. The `asm` keyword is not reserved in the strict ANSI C mode.

Multiple definitions of external variables

If there are so-called “tentative” definitions for the same object (i.e. external declarations of variables without the attribute `extern` or `static`) in several compilation units, these must always be of the same type. Different type declarations for the same external object are not recognized by the compiler. Multiple initializations of external variables will result in errors on linkage. This behavior can be controlled with options (`-K external_multiple`, `-K external_unique` and `EXTERNAL-DEFINITION=*UNIQUE/*MULTIPLY-ALLOWED`).

Empty macro arguments

When calling macros, it is also possible to pass empty arguments.

Example

```
#define F(a,b)  f(a)+f(b);  
  
F(1)    /* produces f(1)+f(); */  
F(,1)   /* produces f()+f(1); */
```

Predefined macros

For compatibility reasons, some predefined macros do not begin with the underscore (`_`) character (see the section [“Predefined preprocessor names” on page 302](#)).

Additional preprocessor directives

The following additional preprocessor directives are accepted by the compiler:

`#line` (old format), `#ident`, `#assert` and `#unassert`.

`#line` directive (old format):

```
#_digit-sequence_[header-file]
```

This directive is identical to the `#line` directive, except for the fact that the keyword `line` is omitted.

`#ident` directive:

```
#ident_"string"
```

The `#ident` directive, which is equivalent to the `#pragma ident`, is accepted syntactically, but does cause any changes in the generated object. The compiler does not issue any notes or warnings, since these directives are used internally in system headers.

`#assert` directive:

```
#assert_name[(token-sequence)]
```

The `#assert` directive can be used to define an assertion. Assertions are independent of macro definitions.

name is the name of the assertion, and *token-sequence* is the value to which the assertion applies.

A single *token* may be one of the following lexical units: name, keyword, constant, string, operator, separator/punctuation character. If no *token-sequence* is specified, the assertion is considered defined as in the case of a symbolic constant, but no value is assigned.

The `#if` statement can be used to test whether an assertion applies to a value:

```
#if #name(non-empty-token-sequence)
```

name is the name of the assertion, and *non-empty-token-sequence* is the value to be tested. For example, the following test for the predefined assertion `compiler` would return the value "true":

```
#if #system(bs2000)
```

The predefined assertions are described in the appendix under [“Predefined preprocessor assertions \(#assert\)” on page 303](#).

`#unassert` directive:

`#unassert`_{*name*}`[` (*token-sequence*) `]`

The `#unassert` directive has the same syntax as `#assert` and can be used to remove any assertion.

If a *token-sequence* is specified, only the assertion for that value is removed; otherwise, i.e. if no *token-sequence* is specified, the entire assertion is deleted.

6.4 Pragmas

This section describes the `#pragma` directives which are defined as implementation-dependent in the ANSI/ISO standard and which are accepted by the compiler.

6.4.1 aligned pragma

The `aligned` pragma can be used to align data elements within classes, structures and unions on a larger number of bytes than the default minimum alignment set for the compiler.

This pragma can be used in all language modes of the compiler.

```
#pragma aligned n
```

n is a number of bytes, which is specified in steps to the power of 2 up to a maximum of 8, and optionally entered in decimal, octal or hexadecimal notation. The permitted values (in decimal notation) are 1, 2, 4, 8 and 16 (still permitted by the syntax).

Notes

For the sake of simplicity, the following notes only refer to structures, but are also analogously applicable to classes and unions.

- Pragmas that specify fewer bytes than the minimum alignment intended for the corresponding data type (see table below) are not permitted and are ignored.

Data type	Minimum alignment (number of bytes)
char	1
short	2
int	4
long	4
long long	8
float	4
pointer	4
double	8
long double	8

- The data element to be aligned may also be a bitfield, in which case the bitfield is created in a new base field with the required alignment.
- In the case of `static` elements, the pragma is ignored.
- The pragma must be placed in the structure definition immediately before the data element to be aligned. Otherwise, it will be ignored.

- If multiple pragmas precede a data element, the one with the largest alignment specification is considered.
- If a pragma precedes the declaration of several structure elements, it is applied to only the first declared element.
- The alignment of a structure is based on the maximum alignment of its elements.
- If a structure appears as an element in another structure, the pragma preceding it applies to the alignment of the entire structure, and not the alignment of its elements.
- A pragma that precedes a structure element that represents an array applies to the alignment of the entire array (i.e. the first array element), and not the remaining array elements.
- Aligned pragmas and pack pragmas (see [page 225](#)) can be active at the same time for a specified structure element. In this case, the aligned pragmas take precedence.

Example

```

...
class bsp1
{
    int a;                // Aligned on 4 bytes.
#pragma aligned 8
    int b,c;              // b is aligned on 8 bytes.
                        // c is aligned on 4 bytes!
                        // The maximum alignment of an element
                        // of class bsp1 is thus equal to 8 bytes.
                        // Class bsp1 is therefore aligned on
                        // 8 bytes.
    int d;                // Aligned on 4 bytes.
};
class bsp2
{
public:
double dens;             // Aligned on 8 bytes.
#pragma aligned 4
                        // Is ignored. Since the maximum alignment
                        // of an element of the structure stru1 is
                        // 8 bytes, stru1 is also aligned on 8 bytes.
                        // A corresponding warning is issued.

    struct
    {
        int istru1;       // Aligned on 4 bytes.
        double dstru12;   // Aligned on 8 bytes.
    } stru1;              // Aligned on 8 bytes (see above).

    struct
    {

```



```

        short s1;           // Aligned on 2 bytes.
        short s2;           // Aligned on 2 bytes.
    } stru2;                 // Aligned on 2 bytes, since the maximum
                            // alignment of an element equals 2 bytes.
#pragma aligned 4
    char c;                 // Aligned on 4 bytes.
    char c1;                // Aligned on 1 byte.
#pragma aligned 8
    short ar1[16];         // The array is aligned on 8 bytes,
                            // but not the individual array elements.
    ...
}
...

```

6.4.2 pack pragma

This pragma controls structure layout. The alignment of the structural elements in all structures is reduced to the number specified in the pragma. This has the effect of reducing the size of the structure. The scope of a pack pragma extends as far as the next pack pragma..

```
#pragma_pack ([n])
```

n can be specified as a decimal, octal or hexadecimal number. The permitted values (in decimal notation) are 1, 2, 4 and 8.

Default value: 8

Important notes

Using pack pragmas has the effect of increasing run time since accessing unaligned structure elements takes longer than accessing aligned structure elements. This effect is not so noticeable on S servers as on other servers (e.g. SX servers).

You should note that using the address of a structure element is not without risks. Attempting to access a structure element by using its address can cause a dump.

6.4.3 ETPND pragma

This pragma can be used to create a documentation area in each module (code and data CSECT) of the generated LLM. This area contains general information on the translation unit (e.g. the version number or creation date), but no data on the function of the module. If desired, a patch area can be reserved for corrections that may be required at a later stage.

Format:

```
#pragma _ETPND_ { CODE }
                  { DATA }

                  [ , VER=version ]
                  [ , DATE=date ]
                  [ , COMPNR=compno ]
                  [ , PATCH=number ]
                  [ , MODULLENGTH=length ]
```

CODE	The ETPND area is created in the code or data module.
DATA	
<i>version</i>	Module version as a decimal value in the range [0..999]. If this entry is omitted, 0 is assumed.
<i>date</i>	Date of creation, in the form <i>yyyymmdd</i> , <i>yyyy-mm-dd</i> or <i>yymmdd</i> . If the six-digit format is specified, the year must be between 1960 and 2059. The two missing digits for the year (19 or 20) are added as appropriate. If the eight-digit is specified, the date must lie between 1.1.1880 (18800101) and 1.1.2021 (20210101). Overflows in months or days, if present, are converted to the canonical form. An entry of 19961335 (35.13.1996), for example, would thus correspond to 19970204 (4.2.1997). If no creation date is specified, the compiler uses the date of compilation.
<i>compno</i>	Component number as a decimal value in the range [0..99999999]. If this entry is omitted, 0 is assumed.
<i>number</i>	Size of the patch area in bytes. The value must not exceed 4294967295 (0xFFFFFFFF). If this entry is omitted, an area of 200 bytes is reserved in the code module, and no area (0) is reserved in the data module. If PATCH=0 is specified, no patch area is reserved in the code module either. The value may be specified as a decimal, octal or hexadecimal number.

length Length of the module including that of the ETPND area (24+7 bytes due to alignment on a double-word boundary) in bytes; this operand can be used to terminate a module on a page boundary. The value must not exceed 4294967295 (0xFFFFFFFF). If the value specified for *length* is less than the actual length of the module; the operand will be ignored. The value may be specified as a decimal, octal or hexadecimal number.

Notes

- An ETPND pragma cannot be used to specify values for both MODULLENGTH and PATCH at the same time. If the ETPND pragma contains both entries, MODULLENTGH is ignored.
- Only one ETPND pragma may be specified per module (i.e. per code and data CSECT). If multiple ETPND pragmas are specified for the same module, only the last pragma specified is used.

6.4.4 Pragmas to control the layout of listings

The layout of source, error, and preprocessor listings can be controlled from within the source text by means of `#pragma` directives. Other listings generated by the compiler cannot be controlled by pragmas.

The `-K pragmas_interpreted`, `-K pragmas_ignored` and `LISTING-PRAGMAS=...` options can be used to globally specify which of the pragmas that appear in the source text are to be interpreted or ignored.

LISTING pragma

The LISTING pragma can be used to suppress the output of source text lines.

$$\#pragma_LIST[ING]_ \left\{ \begin{array}{l} \text{OFF} \\ \text{ON} \end{array} \right\}$$

If LIST OFF is specified, all source text lines that follow this directive are suppressed in the listing.

LIST ON cancels the effect of LIST OFF, i.e. all subsequent source text lines are shown in the listing.

Source text lines in a header file are not shown in the listing if the corresponding `#include` directives are enclosed within `#pragma LIST OFF` and `#pragma LIST ON` directives.

Error messages of the compiler that are related to source text lines for which output has been suppressed are inserted at the point at which the source text in question would normally appear. In such cases, the appropriate line number and the file or element name are also shown.

*Example***Source text:**

```
98     ...
99     ...           ← Message related to line 99
100    ...
101    #pragma LIST OFF
102    #include "msg.h"
103    1     ...
104    ...           ← Messages related to suppressed lines
105    45    ...
106    #pragma LIST ON
107    ...
```

Source/Error listing:

```
98     ...
99     ...

****> CFE2234 [ERROR]   : ...

100    ...
101    #pragma LIST OFF

****> CFE1004 [NOTE]    : msg.h / 13: ...
****> CFE1386 [WARNING] : msg.h / 37: ...

103    #pragma LIST ON
104    ...
```

TITLE pragma

The TITLE pragma can be used to define additional text to be included in the default header generated for the listing.

```
#pragma TITLE text
```

text Any string that is enclosed within quotes and consists of printable characters. Control characters such as `\n`, `\t`, etc. are not permitted.

If the text to be added exceeds the maximum line length defined with the `-N output` or `LINE-SIZE=...` options, it is split into multiple lines of appropriate length.

The text in the TITLE pragma appears only as of the second page in the header of the listing and overwrites any text defined with the INITIAL-TITLE-TEXT option. The first page of the listing contains either a blank line or the text defined with the INITIAL-TITLE-TEXT option.

The defined text remains in effect until the next TITLE or PAGE pragma or until the end of the file.

Invalid characters (e.g. control characters) also result in a warning and are replaced by blanks.

If the generation of multiple additional text lines in the header of a listing makes the minimum prescribed number of 11 lines per page (see the options `-N output` and `LINES-PER-PAGE`) insufficient for the output of at least one line of source text in addition to the header and footer, the listing generator issues a warning and selects a correspondingly higher number of lines.

PAGE pragma

The PAGE pragma generates a page feed and can optionally be used like the TITLE pragma to define an additional line of text to be entered in the header of the listing.

```
#pragma PAGE [text]
```

text Any string that is enclosed within quotes and consists of printable characters. Control characters such as `\n`, `\t`, etc. are not permitted.

The `#pragma PAGE` directive without *text* generates a page feed.

The directive `#pragma PAGE text` generates a page feed with the line specified by *text* appearing as an additional line in the header of the listing. All other conditions for the output of text lines are the same as those for the TITLE pragma.

SPACE pragma

The SPACE pragma can be used to generate blank lines in the listing.

```
#pragma SPACE[n]
```

n is a non-negative integer. The above directive inserts *n* blank lines into the listing. *n* can be specified as a decimal, octal or hexadecimal number.

If *n* is omitted, one blank line is inserted.

6.4.5 inline pragma

The inline pragma is used to specify the names of the user-defined C functions to be substituted inline by the compiler.

```
#pragma inline name
```

name Name of a C function to be substituted inline.

The given C functions are substituted inline only if the following options are specified: `-F inline_by_source` or `-F i` in POSIX and `INLINING=*YES` in SDF. The inline pragma is not supported in the C++ modes, since C++ has built-in mechanisms for inlining functions.

6.4.6 int_to_unsigned pragma

```
#pragma int_to_unsigned name
```

The pragma is supported because it may be contained in older C sources, typically those ported from a UNIX system. It only works in K&R mode and instructs the compiler to treat a function *name* with the result type `unsigned` as if its result type were still `int`.

The declaration of the function *name* with the result type `unsigned` must appear before the `#pragma` directive, e.g.:

```
unsigned int strlen(const char*);
#pragma int_to_unsigned strlen
```

6.4.7 weak pragma

```
#pragma weak name
```

This pragma declares *name* as a global symbol with the linkage attribute “weak external reference” (WXTRN). For more information on WXTRNs, see the manual “BINDER” [14]. Symbols that are declared as `weak` may remain unresolved at linkage. This means that if a reference to *name* cannot be resolved, BINDER will only issue an information message. Weak external references are resolved only on explicit inclusion (INCLUDE-MODULES). They are not resolved when modules are included with the autolink mechanism (RESOLVE-BY-AUTOLINK).

Note that external C++ names cannot be declared as `weak`.

6.4.8 ident pragma

```
#pragma ident "string"
```

This pragma is only accepted syntactically by the compiler, since it may be contained in older C sources, typically those ported from a UNIX system.

6.4.9 C++ specific pragmas

The following `#pragma` directives are only relevant for C++ compilations.

VIRTUAL_FUNCTION_TAB pragma

```
#pragma VIRTUAL_FUNCTION_TAB { GLOBALLY_DEFINED }
                             { EXTERNALLY_DECLARED }
```

This pragma has the same effect as the corresponding compiler options:

The GLOBALLY-DEFINED entry is equivalent to the options
VIRTUAL-FUNCTION-TAB=*GLOBALLY-DEFINED and `-K force_vtbl`.

The EXTERNALLY-DECLARED entry is equivalent to the options
VIRTUAL-FUNCTION-TAB=*EXTERNALLY-DECLARED and `-K suppress_vtbl`.

This pragma can only be used as an alternative to option control.

As soon as an equivalent SDF or POSIX option is explicitly specified at compilation (even if this is only the default `VIRTUAL-FUNCTION-TAB= *INTERNALLY-DEFINED` or `-K normal_vtbl`), the pragma is ignored.

Pragmas to control template instantiation

```
#pragma { instantiate
         do_not_instantiate
         can_instantiate } argument
```

The instantiation of individual templates or even a group of templates can be controlled with the following pragmas:

- The `instantiate` pragma causes the template entity that is specified as an argument to be created. This pragma can be used in all instantiation modes.
- The `do_not_instantiate` pragma suppresses the instantiation of the template entity specified as an argument. The typical candidates for this pragma are template entities for which specific definitions (specializations) have been provided. This pragma can be used in all instantiation modes.
- The `can_instantiate` pragma is a hint to the compiler that the template entity specified as an argument can, but need not, be created in the compilation unit. This pragma is required in connection with libraries and is only evaluated in automatic instantiation mode.

The following arguments can be specified with the pragmas:

<i>Argument</i>	<i>Examples</i>
a template class name	<code>A<int></code>
a member function name	<code>A<int>::f</code>
a static data member name	<code>A<int>::i</code>
a member function declaration	<code>void A<int>::f(int, char)</code>
a template function declaration	<code>char * f(int, float)</code>

When a template class name is specified as an argument (e.g. `A<int>`), the net effect is the same as if the pragma were specified for each member function and for each static data member of that template class. When instantiating an entire class, individual member functions or static data members can be excluded from the instantiation process by using the `do_not_instantiate` pragma.

Example

```
#pragma instantiate A<int>
#pragma do_not_instantiate A<int>::f
```

In order to instantiate templates, the appropriate template definitions must be available in the current compilation unit. If an instantiation is requested explicitly with the `instantiate` pragma, and no template definition or only a specific definition (specialization) is available, an error message (ERROR) is issued.

Example

```
template <class T> void f1(T); // no body provided
template <class T> void g1(T); // no body provided
void f1(int) { } // specific definition
void main()
{
    int i;
    double d;
    f1(i);
    f1(d);
    g1(i);
    g1(d);
}
#pragma instantiate void f1(int) // error - specific definition
#pragma instantiate void g1(int) // error - no body provided
```

`f1(double)` and `g1(double)` will not be instantiated either (due to the missing template definitions), but no error message will be issued during the compilation in this case. The missing template definitions will, however, result in a linker error at link time.

If a member function name (e.g. `A<int>::f`) is specified as a pragma argument, it must not be an overloaded function. Overloaded member functions can be instantiated by providing the complete member function declaration, as in

```
#pragma instantiate char * A<int>::f(int, char *)
```

The argument to an instantiation pragma may not be a compiler-generated function, an inline function, or a pure virtual function.

7 C++ language support of the compiler

The compiler optionally supports the C++ language scope compatible with Cfront V3.0.3 as well as the scope defined in the ANSI/ISO C++ draft standard proposed at the end of 1996.

The C++ programming language is described in detail in the “The C++ Programming Language”, Second Edition (Cfront V3.0.3) and Third Edition (ANSI/ISO C++) by Bjarne Stroustrup.

The following sections describe the vendor-specific and implementation-defined language features of C++. They are intended as a supplement to the C++ language definitions given in the manuals by B. Stroustrup.

7.1 Overview of the C++ language modes

In accordance with the different language definitions for C++, the compiler supports three C++ compilation modes:

*Cfront C++ mode (POSIX option -X d, SDF option MODE=*CPP)*

The compiler accepts C++ code compatible with Cfront V3.0.3.

*extended ANSI C++ mode (POSIX option -X w, SDF option MODE=*ANSI) or*

*strict ANSI C++ mode (SDF option -X e, MODE=*STRICT-ANSI)*

The compiler accepts C++ code based on the ANSI/ISO definition.

The following table contains an overview of the main differences between the various C++ language modes.

Features / Language attributes	Cfront 3.0.3	extended ANSI	strict ANSI
Reserved keywords ¹⁾			
Exception handling catch, throw, try	no	yes	yes
Templates template	no	yes	yes
New ANSI C++ language features ²⁾		yes	yes
<ul style="list-style-type: none"> - Runtime type information (RTTI) typeid, dynamic_cast - Arrays: new/delete new [], delete[] - Name space namespace, using - Template parameter typename - Constructor type explicit - Data type wchar_t - Boolean data type bool 	no		
<ul style="list-style-type: none"> - Storage class mutable - Casting keywords const_cast, reinterpret_cast, static_cast 	yes		
long long	yes	yes	yes ³⁾
__STDC__	==0	==0	==1
__STDC_VERSION__	==199409L	==199409L	==199409L
__cplusplus	==1	==2	==199612L ⁴⁾
Object layout and name mangling	compatible	new ⁵⁾	new ⁵⁾

Overview of differences between the various C++ language modes

1) Reserved keywords

All keywords listed in the [chapter “C language support of the compiler” on page 205](#) are also reserved in the C++ language modes in addition to the following C++-specific keywords.

asm	explicit	new	template ²	using
bool ¹	export	operator	this	virtual
catch ²	false ¹	private	throw ²	wchar_t
class	friend	protected	true ¹	
const_cast	inline	public	try ²	
delete	mutable	reinterpret_cast	typeid	
dynamic_cast	namespace	static_cast	typename	

Reserved names in C++

The keywords shown in **boldprint** are not reserved in the Cfront C++ mode.

- 1 The keywords `bool`, `true` and `false` are reserved in the ANSI C++ modes, depending on the setting of the `-K bool`, `-K no_bool` or `KEYWORD-BOOL=*YES/*NO` options. `-K bool` or `KEYWORD-BOOL=*YES` is the default.
- 2 No exception handling and templates are supported in the Cfront C++ mode. The keywords `catch`, `throw`, `try` and `template` are nonetheless not freely available and will result in an error message if used.

The following keywords may be used as alternative tokens for C operators. They may or may not be reserved, depending on the settings of the `-K alternative_token`, `-K no_alternative_token` or `ALTERNATIVE-TOKENS=*YES/*NO` options. By default, they are “not reserved” in the Cfront C++ mode and “reserved” in the ANSI C++ modes.

and	&&	not_eq	!=
and_eq	&=	or	
bitand	&	or_eq	=
bitor		xor	^
compl	~	xor_eq	^=
not	!		

Keyword operators in C++

2) New ANSI C++ language features

In accordance with the future ANSI/ISO C++ standard, several new C++ language features have been added since the language definition presented by Bjarne Stroustrup in “The C++ Programming Language”, Second Edition. These new features are now documented in Stroustrup’s Third Edition.

3) long long

The data type `long long` is an extension to ANSI C++ (see also [page 219](#)).

4) __cplusplus

This value will increase in subsequent versions of the compiler and may also change before the final release of the ANSI/ISO C++ standard.

5) Object layout and name mangling

Due to the support for templates, exception handling, RTTI, `new[]` and `delete[]`, the object layout and name mangling strategies differ with respect to Cfront V3.0.3. Consequently, modules generated with the earlier C++ compiler as of V2.1 or with the C/C++ compiler V3.0 in the Cfront C++ mode cannot be linked with modules generated in one of the ANSI C++ modes!

7.2 Implementation-defined behavior based on the ANSI/ISO C++ standard

All of the implementation-defined features based on the ANSI/ISO C standard that have already been described in the [chapter “C language support of the compiler”](#) (see [page 208ff](#)) are also applicable in the C++ language modes and are therefore not listed individually here.

Only the implementation-defined C++ language features that extend beyond the scope of the C language are described below. A list of the individual C++ language features which are only supported in the ANSI C++ modes and not in the Cfront C++ mode of the compiler can be found in the tabular overview on [page 236](#).

Identifiers with external linkage

- Compilation in the Cfront C++ mode
External C++ names are truncated by the compiler to 32 characters. When generating shared code (with the options `-K share` or `SHAREABLE-CODE=*YES`), the maximum permissible length is restricted to 30 characters.
Lowercase letters are converted to uppercase by default, but can be retained in external names with the option `-K llm_case_lower` or `LOWER-CASE-NAMES=*YES`.
- Compilation in the ANSI C++ modes
All characters are significant for the construction of external C++ names. No name truncation and no conversion from lowercase to uppercase occurs. The compiler generates entry names in the EEN format, which can have a length of up to 32000 characters. Longer names result in an error.

To enable processing by the link editor, names with external C++ linkage are transformed internally by the compiler so that they only contain permitted characters and are unique (name mangling). This internal name mangling process differs in the Cfront C++ mode and in the ANSI C++ modes. A list of the originally used external C++ names and those which were generated internally by the compiler for the linkage editor can be obtained with the option `-N project` or `PROJECT-INFORMATION=*YES`.

Linkage of the main function

The `main` function has external C linkage.

Data type `bool`

The size of type `bool` is defined as `sizeof(bool) == 1`.

reinterpret_cast

The destination object contains the same bit pattern as the expression evaluated by `reinterpret_cast`, but might not be a valid object of the desired type (e.g. `reinterpret_cast<float>(int)`).

The following conversions are possible:

1. A pointer can be explicitly converted to an integral type that is large enough to hold it. Depending on whether the destination type is `signed` or `unsigned`, the value of the result obtained from the conversion may or may not be signed.
2. The value of an integral type can be explicitly converted to a pointer.

Allocation overhead for new arrays

For each allocated array, a structure is reserved at the start of the memory block allocated for that array. This structure contains two `size_t` members, one for the size of the array (in bytes) and one for the number of elements in the array (this field is encoded in order to detect whether the structure has been overwritten).

The asm keyword

The `asm` keyword is an extension to the ANSI C standard, but is defined as a normal reserved keyword in the ANSI C++ standard. However, since the inline substitution of Assembler code is not supported, any attempt to use this keyword will result in an error message.

Linkage specification

The supported linkage specifications are "C++" and "C".

The default linkage specification is "C++". Names with external C++ linkage are transformed internally by the compiler to enable processing by the linkage editor (name mangling). Since this internal name mangling process differs in the Cfront C++ mode and in the ANSI C++ modes, Cfront C++ modules cannot be linked with ANSI C++ modules.

In the case of names with external C linkage, no internal name mangling is performed. C linkage can be used to link and call functions written in C or in a language that behaves like C on its name mangling interface.

A pointer to a C function is compatible with a pointer to a C++ function of the same type if the argument types are C-compatible PODs.

Linkage of templates

Only templates with C++ linkage are supported.

Template instantiation

See [page 244ff](#)

Reference types

The reference for an value is bound to a temporary object created by the compiler. Internally, the C++ data type `reference` is treated as a pointer (see [page 210](#)).

Allocation of non-static data elements of a class

The allocation of memory for these data elements occurs in the strict order of their declarations, i.e. without taking the access specifiers `public`, `private` or `protected` into account.

Bitfields within classes

Bitfields within classes are handled in the same way as bitfields within structures. This applies to the allocation and alignment as well as the handling of “plain” bitfields without the `signed` or `unsigned` attribute.

Constructors and destructors for global and local static objects

C++ supports the initialization and finalization of objects with constructors and destructors. Constructor and destructor calls can be applied on both dynamic as well as global and local static objects.

Constructors and destructors for dynamic objects

Constructors for dynamic objects are called when objects are created with the `new` operator, on entering functions or local blocks, on processing current parameters, and when temporary objects are created.

Destructors for dynamic objects are called in the reverse order of their construction, e.g. on exiting a function (`return`, `exit`), block, etc.

Constructors and destructors for global and local static objects

The type and order of constructor and destructor calls for global and local static objects is “implementation-defined”. Seen from a technical viewpoint, this means the following:

Constructor calls are collected in one function per compilation unit, and the address of each such function is stored in a specially marked variable in the data section of the program. During the initialization of the runtime system, a list of these variables is generated, and the constructors are called before the `main` function. Destructors are called on exiting a program normally, i.e. on calling `exit`, `_exit`, `bs2exit` or `return` from the `main` function, but not on calling `abort`.

Seen from the user’s perspective, the following points must be observed when using global and local static objects with constructors and destructors:

- The redirection of standard I/O files (see [page 180](#)) has no effect within global and local static objects.
- The order in which constructors are called is totally undefined and may actually vary even when running the same program. It is therefore crucial to ensure that no dependencies between the individual constructor calls exist.
- The names of the special variables for constructor function addresses (see above) begin with ICP. This prefix and the initial letter `I` in general must not be used when constructing user-defined external names, since this produces errors in constructor handling.
- ESD information must not be suppressed when linking.
- The data of the program must be in class-6 memory.
- In the case of shared code, all code segments to call constructors are loaded dynamically.
- No inter-language linkage is permitted within a destructor (only linkage to other languages).
- No code may be “unloaded”; otherwise, on exiting the program, this would result in destructors being called for code that no longer exists.
- In the case or true subsystems for which data and code are loaded dynamically with an explicit call, the runtime system must be reinitialized. During this process, the table of constructors is extended, and the new constructors are called.

Exception handling

Thrown exceptions

The memory for exception objects is taken from a preallocated storage, which can be extended by calls to `malloc`.

Handling of exceptions

The stack is not unwound, i.e. no destructors are called for automatic objects if the program exits with `terminate()` due to a missing exception handler (see also the function `unwind_exit()` on [page 282](#)).

The `unexpected()` function

The thrown exception object which causes the `unexpected()` function to fail is destroyed by calling a destructor and is replaced by a new `bad_exception` object.

More details on exception handling can be found on [page 282ff](#).

Return type of `operator->`

The return type of `operator->()` is not checked any more at the point where the function is declared, but is now only checked at the point where it is used as an “->” operator. (This check was performed later on just for elements of class templates in versions up to V3.0C.)

7.3 Template instantiation

7.3.1 Fundamentals

The C++ language includes the concept of templates. A template is a description of a class or function that serves as a model for a family of derived classes or functions. For example, one can write a template for a `Stack` class, and then use a stack of integers, a stack of floats, or a stack of any user-defined type. These stacks could then be typically written in the source as `Stack<int>`, `Stack<float>` and `Stack<X>`. From a single source description of the template for a stack, the compiler can create instantiations of the template for each of the types required.

The instantiation of a class template is always created as soon as it is required during compilation. The instantiations of template functions and member functions or static data members of a class template (referred to as **template entities** below), by contrast, need not be created immediately. This is due to the following reasons:

- In the case of template entities with external linkage (functions and static data members), it is important to have only one copy of the instantiated template entity throughout the program.
- The ANSI C++ language allows one to write a specialization for a template entity, which means that the programmer can supply a specific version to be used instead of the instantiation generated from the template for a specific data type. Since the compiler cannot know, when compiling a reference to a template entity, if a specialization for that entity is available in another compilation unit, it cannot create the instantiation immediately.
- The ANSI C++ language dictates that template functions which are not referenced should not be compiled and should be checked for errors. Consequently, a reference to a template class should not automatically instantiate all the member functions of that class.

Note that some template entities such as inline functions are always instantiated when used.

From the requirements listed above, it is evident that if the compiler is responsible for the entire instantiation (i.e. if the instantiation is done “automatically”), these instantiations can only be performed meaningfully on a program-wide basis. In other words, the compiler cannot make decisions about the instantiation of template entities until it has seen the source code of all compilation units in the program.

The C/C++ compiler provides an instantiation mechanism by which automatic instantiation is carried out at link time (with the aid of a “prelinker”). This mechanism is discussed in detail in the [section “Automatic instantiation” on page 246](#).

More explicit control over the instantiation process is available to the programmer via different instantiation modes that can be selected using options and by means of `#pragma` directives.

- The options to select instantiation modes are:
MODIFY-SOURCE-PROP INSTANTIATION=`*NONE` / `*AUTO` / `*LOCAL` / `*ALL`
(see [page 144ff](#)).
- The instantiation of individual templates or even a template group can be controlled with the pragmas:
`instantiate`, `do_not_instantiate` and `can_instantiate` (see [page 233ff](#)).

Important notes

The default template instantiation method of this compiler (i.e. automatic instantiation by the prelinker and implicit inclusion) is also the recommended method and should generally be used. Deviations from this default method via control options should be restricted to exceptional cases and should only be attempted if the entire application is known in detail, including every template that is defined and used.

Implicit inclusion must not be disabled (with `IMPLICIT-INCLUDE=*NO`) when using templates from the standard C++ library (`SYSLNK.CRTE.STDCPP`). Otherwise, the required definitions will not be found.

Instantiation modes \neq `INSTANTIATION=*AUTO`: In this case, there is a high risk that unresolved external references (`*NONE`), duplicates (`*ALL`) or runtime errors (`*LOCAL`) may occur.

7.3.2 Automatic instantiation

The C/C++ compiler supports automatic instantiation (`INTEGRATION=*AUTO`) by default in the ANSI C++ language modes. This enables you to compile source code and link the generated objects without being concerned about any required instantiations.

Note that the discussion which follows refers to the automatic instantiation of template entities for which there is no explicit instantiation request (`template declaration`) and no `instantiate pragma`.

Requirements

For each instantiation, the compiler expects a source file that contains both a reference to the required instantiation and the definition of the template entity as well as all types required for the instantiation of that template entity. The latter two requirements can be satisfied by the following methods:

- Each `.h` file that declares a template entity also contains either the definition of the entity or includes another file containing the definition.
- Implicit inclusion
When the compiler sees a template declaration in a `.h` file and discovers a need to instantiate that entity, it looks for a source file with the same base name as the `.h` file and a suffix that satisfies the conventions for C++ source file names (`.C`, `.CPP`, `.CXX` or `.CC`). This file is then implicitly included by the compiler on instantiation at the end of each compilation unit without a message being issued. Further details see the [section “Implicit inclusion” on page 259](#).
- The programmer makes sure that the files that define template entities also contain the definitions of all required types and adds C++ code or instantiation pragmas in those files to request the instantiation of the template entities therein.

First instantiation without a definition list

The definition list method can also be used as an alternative to the following procedure (see [page 248](#)).

The following steps are performed internally during automatic instantiation:

1. Create instantiation information files
The first time that one or more source files are compiled with the `COMPILE` statement, no template entities are instantiated. For each source file that makes use of a template, an associated instantiation information file is created if no such file exists. This instantiation information file is placed in the PLAM library of the module to be generated and is assigned the base name of the module with the suffix `.II` by default (see the [section “Rules for constructing module names” on page 52](#)). For example, if a source program

named `ABC.CC` were to be compiled (without explicitly specifying a module name), the file `ABC.II` would be generated as the instantiation information file. This file must not be modified by the user.

2. Create modules

The created modules contain information on which instantiations could have been created and on those possibly required when compiling a source file.

3. Assign template instantiations

When the modules are linked with the `BIND` statement, the prelinker is called before the actual linking takes place. The prelinker examines the modules, looking for references and definitions of template entities and for additional information about entities that could be instantiated. If the prelinker cannot find a definition for a required template entity, it looks for a module that indicates that it could instantiate that template entity. When it finds such a module, it assigns the instantiation to it.

4. Update the instantiation information file

All instantiations that were assigned to a given module are recorded by name in the associated instantiation information file.

5. Recompile

The compiler is internally called again to recompile each file for which the instantiation information file was changed.

6. Create new modules

When the compiler compiles a file, it reads the instantiation information file for that compilation unit and generates a new module with the required instantiations (i.e. template definitions).

7. Repetition

Steps 3 to 6 are repeated until all instantiations which are required and which can be generated have been created.

8. Linkage

The modules are linked together.

First instantiation with the help of the definition list (temporary repository)

Since the method above (see [page 246](#)) needs to recompile some files more than once, an option was added to accelerate the entire process.

When this option is used, the files are generally only recompiled once. Most of the instantiations are assigned to the first few files to be recompiled in this process. This results in disadvantages in some cases since the object size increases due to this (although other objects decrease in size to balance this out).

Steps 3-5 above are modified. The resulting algorithm appears as follows:

1. Create instantiation information files
The first time that one or more source files are compiled with the COMPILE statement, no template entities are instantiated. For each source file that makes use of a template, an associated instantiation information file is created if no such file exists. This instantiation information file is placed in the PLAM library of the module to be generated and is assigned the base name of the module with the suffix `. I I` by default (see the [section "Rules for constructing module names" on page 52](#)). For example, if a source program named `ABC . CC` were to be compiled (without explicitly specifying a module name), the file `ABC . I I` would be generated as the instantiation information file. This file must not be modified by the user.
2. Create modules
The created modules contain information on which instantiations could have been created and on those possibly required when compiling a source file.
3. Assign template instantiations to a source file
If there are references for template entities for which there are no definitions in the set of modules, then a file that could instantiate one of the template entities is selected. All template entities that can be instantiated in this file are assigned to the file.
4. Update the instantiation information file
The set of instantiations that were assigned to the file are recorded by name in the associated instantiation file.
5. Store the definition list
A definition list is stored internally in memory. It contains a list of all definitions found in all modules that relate to templates. This list can be read in and changed during compilation.
Note
This list is not stored in a file.
6. Recompile
The compiler is internally called again to recompile the corresponding source files.

7. Create new modules

When the compiler recompiles a file, it reads the instantiation information file for that compilation unit and generates a new module with the required instantiations.

When the compiler obtains an opportunity during compilation to instantiate additional template entities that are not mentioned in the definition list and were not found in the libraries resolved, then it will also instantiate these (e.g. for templates that are contained in templates). It sends the list of instantiations that it obtained during recompilation to the prelinker so that the prelinker can assign it to the file.

This process results in faster instantiations and reduces the necessity of recompiling an existing file more than once during the prelinking process.

8. Repetition

Steps 3 to 7 are repeated until all instantiations that are required and can be generated have been created.

9. Linkage

The modules are linked together.

Further development

Once a program has been correctly linked, the associated instantiation information files contain all the names of the defined and required instantiations. From then on, whenever source files are compiled, the compiler will consult the instantiation information file and do the instantiations therein as in a normal compilation run. In other words, except in cases where the set of required instantiations changes, the prelinker will find all required instantiations stored in the modules, so no further instantiation adjustments are needed. This applies even if the entire program is recompiled.

If a specialization of a template entity has been provided somewhere in the program, the prelinker will treat it as a definition. Since this definition will satisfy any references to the template entity, the prelinker will see no need to request an instantiation for that template entity. If a specialization is added to a program that has already been compiled, the prelinker will remove the assignment of the instantiation from the corresponding instantiation information file.

The instantiation information file must not be modified (e.g. renamed or deleted) by the user, except in the following case: if a source file in which a definition was changed and another source file in which a specialization was added are being compiled in sequence in the same compiler run, and the compilation of the first file (with the changed definition) has aborted with an error, the associated instantiation information file must be deleted manually to allow the prelinker to regenerate it.

Interaction between the compilation and prelinker runs

Since the automatic template instantiation by the prelinker can, among other things, also involve subsequent compilations, the conditions under which the compilation occurs (COMPILE) play an important role in the prelinker run (BIND ACTION=*PRELINK,...) as well. Technically speaking, all options relevant for the compilation, code generation and compiler outputs are entered in the corresponding instantiation information files and interpreted during the automatic template instantiation. In order to ensure that the follow-up compilations and other updates function correctly during instantiation, the following points, in particular, must be observed:

- The input of source programs via SYSDTA and the output of listings to SYSLST or SYSOUT are not supported and are rejected with a corresponding error message.
- All I/O files and libraries of the compilation run must also be locatable and accessible in the prelinker run and should hence be neither renamed nor deleted. If the prelinker run occurs in some other environment (e.g. user ID), fully-qualified file names must be used in the compilation, i.e., names including the <cat-id> and <user-id> for BS2000 files and PLAM libraries, and absolute path names (beginning with /) for POSIX files.
- The *INCREMENT option is not permitted for the following output files:
 - module element name
 - cif element name
 - listing element name

Warning: Compilation is aborted in this case.

- When you specify an explicit version string for 'module name', an ii-element file with the same version string is searched for, read in and written. (If an existing ii-element with a different version string is to be used, then this element must be copied and stored under a new name before it can be linked and before the template instantiation starts.) This behavior is not compatible with the C++ Compiler V3.0.

Prelinking and dynamic linkage

The prelinker does the automatic instantiation only in the individual modules generated by the compiler, since it requires the unique assignment between the module and the instantiation file (.II file) for this purpose. The prelinker is activated only with the BIND statement of the compiler (ACTION=*PRELINK). If only ACTION=*MODULE-GENERATION is specified in the BIND statement, no template instantiation is performed.

When generating the finished program (via static or dynamic linkage), all “prelinked modules” of the the BIND statement and all dynamically linked modules that require instances of template entities must either

- already contain these instances (which may be achieved by explicit instantiation and/or the preinstantiation of modules with the BIND statement (ACTION=*PRELINK)
- or provide appropriate headers with `can_instantiate` pragmas.

During the preinstantiation with the BIND statement, it is absolutely essential to ensure that the C++ libraries and C++ runtime systems of the CRTE (standard C++ library, ANSI C++ runtime system, Tools.h++ library) are also considered by the prelinker, since duplicate definitions may otherwise be created. The libraries are automatically taken into account if the STDLIB option of the MODIFY-BIND-PROPERTIES statement is set to *DYNAMIC or *STATIC and the TOOLSLIB option is set to *YES. If the CRTE libraries are to be linked in dynamically themselves, duplicate definitions can be prevented as follows:

1. The first step is to preinstantiate the modules, taking the used CRTE libraries into account:

```
//MODIFY-BIND-PROPERTIES STDLIB=*DYNAMIC / *STATIC [,TOOLSLIB=*YES], ...
//BIND ACTION=*PRELINK, ...
```

2. All modules with unresolved external references to the used CRTE libraries can then be linked in a second step:

```
//MODIFY-BIND-PROPERTIES STDLIB=*NONE [,TOOLSLIB=*NO], ...
//BIND ACTION=*MODULE-GENERATION, ...
```

If the prelinker and linkage runs are to be executed concurrently, the ADD-PRELINK-FILES option must be used. The main disadvantage of this method is that the names of the CRTE libraries would then need to be specified explicitly.

In the case of a standard installation of the CRTE, for example, this is achieved as follows:

```
//MODIFY-BIND-PROPERTIES ADD-PRELINK-FILES=(*LIB(LIB=$.SYSLNK.CRTE.STDCPP),-
 *LIB(LIB=$.SYSLNK.CRTE.RTSCPP) [, *LIB(LIB=$.SYSLNK.CRTE.TOOLS)] ),-
 STDLIB=*NONE, ...
//BIND ACTION=( *PRELINK, *MODULE-GENERATION), ...
```

7.3.3 Generating explicit template instantiation statements (ETR files)

In some cases, for example, when automatic instantiation cannot be used effectively, the programmer has the option of using explicit (manual) instantiation in order to extend the sources as required.

To make this process easier, it is possible to create an ETR file (ETR - Explicit Template Request) which contains the instantiation statements for the templates used and which can be incorporated into a source.

The options for creating this ETR file can be specified with the GENERATE-ETR-FILE operand of the SDF statement MODIFY-DIAGNOSTIC-PROPERTIES (see [page 96](#)).

The important options are ***ALL...**, which outputs all relevant information and ***ASSIGNED...**, which only outputs part of this information.

The templates taken into account during the ETR analysis can be divided into the following classes:

- Templates that are instantiated explicitly in the compilation unit. These are output with “ALL”.
- Templates that are assigned by the prelinker to the compilation unit and then instantiated within the compilation unit. These can be output using both “ALL” and “ASSIGNED”.
- Templates that are used in the compilation unit and that can be instantiated here. These are output with “ALL”.
- Templates that are used in the compilation unit, but cannot be instantiated here. These are output with “ALL”.

The contents of an ETR file have the following format:

- Comments in the header will indicate that the file is a generated file.
- Four logical lines are created for each template (see the example below):
 - a comment line containing the text 'The following template was'
 - a comment line containing the type of the instance (for example, 'explicitly instantiated')
 - a comment line containing the external name of the instance. This name is the same as the entry in the ii file (see [section “The II-UPDATE tool” on page 305](#)) and can also be obtained from the binder listing or the binder error listing
 - a line which describes the explicit instantiation for this instance

Notes

- If the lines described above are too long, they will be wrapped in the usual C++ fashion using “*Backslash newline*”.
- The sequence of the output templates is not defined. If recompilation takes place or a source is modified, the sequence may change.
- The fourth logical line is particularly interesting when copying to a source.
- The comments are always in English.

The following two scenarios describe sensible uses of an ETR file:

1. The compiler is called during development using the `INSTANTIATION=*AUTO` option of the SDF statement `MODIFY-SOURCE-PROPERTIES` and the `GENERATE-ETR-FILE=*ASSIGNED` option of the SDF statement `MODIFY-DIAGNOSTIC-PROPERTIES`.
The instantiation statements output to the ETR files are incorporated into the appropriate sources. Productive operation is then activated using the `INSTANTIATION=*NONE` or `*AUTO` option in the SDF statement `MODIFY-SOURCE-PROPERTIES` the next time the compiler is called.
The advantage of this method is the considerable reduction in the time it takes to complete prelinking during productive operation.
2. The compiler is called during development using the `INSTANTIATION=*NONE` and `GENERATE-ETR-FILE=*ALL-INSTANTIATIONS` option (of the SDF statements as described above).
After linking the developer checks each unresolved external reference to see whether it is a template, and if it is a template, when it can be instantiated. Particularly helpful in this case are the output external names. Then, the developer selects a source for the instantiation and inserts the instantiation statements there. In addition, the correct header files must also be included.
This method requires a considerable amount of manual work. But you do not subsequently need to call the prelinker.
This procedure offers you precise control over the placing of instances (which is important when using components with high performance requirements).

Example 1

For a single ETR file compiled using two files, **x.c** and **y.c**:

The following statements are used for compilation:

```
//MOD-DIAG GEN-ETR=ALL
//COMPILE (X.C, Y.C), *LIB(OLIB), MODULE-OUTPUT=*SOURCE-LOCATION
```

Source x.c:

```
template <class T> void f(T) {}
template <class T> void g(T);
```

```
template void f(long);
```

```
void foo()
{
    f(5);
    f('a');
    g(5);
}
```

Source y.c:

```
template <class T> void f(T) {}
void bar()
{
    f(5);
}
```

ETR-File x.etr (is in the OLIB library as a type S member):

```
// This file is generated and will be changed when the module is compiled
// The following template was
// explicitly instantiated
// __Of__Fl&_
template void f(long);
```

```
// The following template was
// used in this module and can be instantiated here
// __Of__Fi&_
template void f(int);

// The following template was
// used in this module and can be instantiated here
// __Of__Fc&_
template void f(char);

// The following template was
// used in this module
// __Og__Fi&_
template void g(int);
```

ETR-File y.etr (is in the OLIB library as a type S member):

```
// This file is generated and will be changed when the module is compiled
// The following template was
// used in this module and can be instantiated here
// __Of__Fi&_
template void f(int);
```

The user can now decide in which source they wish to make explicit instantiations (this decision must always be made for entries with “used in this module and can be instantiated here”), for example, insertion of `template void f(int)` and `template void f(char)` in `x.c` (see the Source in Example 2). Then you will subsequently not need to use automatic template instantiation.

Example 2

This example uses two file **x.c** and **y.c** in a library called **test**.

Source x.c

```
template <class T> void f(T){}
template <class T> void g(T);

template void f(long);

void foo()
{
    f(5);
    f('a');
    g(5);
}
```

Source y.c:

```
template <class T> void f(T) {}

void bar()
{
    f(5);
}
```

These programs are compiled using the following statements and the prelinking carried out:

```
//mod-source-prop language=*cplusplus(*ansi), instantiation=*auto
//mod-diagnostic-prop generate-etr-file=*assigned-instantiations
//compile *lib-elem(test, X.C), module-output=*source-location
//compile *lib-elem(test, Y.C), module-output=*source-location
//mod-bind-prop start-llm-creation=*yes
//mod-bind-prop include=*lib-elem(test,X)
//mod-bind-prop include=*lib-elem(test,Y)
//bind action=*prelink
```


This creates a file called **x.etr** in the library **test**:

```
// This file is generated and will be changed when the module is compiled
// The following template was
// instantiated automatically by the compiler
// __Of__Fi&_
template void f(int);

// The following template was
// instantiated automatically by the compiler
// __Of__Fc&_
template void f(char);
```

The important lines are inserted in the file **x.c**, which creates the file **x1.c**:

```
template <class T> void f(T){}
template <class T> void g(T);

template void f(long);

void foo()
{
    f(5);
    f('a');
    g(5);
}
template void f(int);
template void f(char);
```

Then you can carry out production using the following commands:

```
//mod-source-prop language=*cplusplus(*ansi), instantiation=*none
//mod-diagnostic-prop generate-etr-file=*no
//compile *lib-elem(test,X1.C), module-output=*source-location
//compile *lib-elem(test,Y.C), module-output=*source-location
```

Example 3

The following example shows the four classes of template that can be output.

The same assumptions are made as for example 1.

The following commands are input:

```
//mod-source-prop language=*cplusplus(*ansi), instantiation=*auto
//mod-diagnostic-prop generate-etr-file=*no
//compile *lib-elem(test,Y.C), module-output=*source-location
//mod-bind-prop start-llm-creation=*yes
//mod-bind-prop include=*lib-elem(test,y)
//bind action=*prelink (this will assign y to f(int))
//mod-diagnostic-prop generate-etr-file=*all-instantiations
//compile *lib-elem(test,X.C), module-output=*source-location
//mod-bind-prop start-llm-creation=*yes
//mod-bind-prop include=*lib-elem(test,X)
//mod-bind-prop include=*lib-elem(test,Y)
//bind action=*prelink
```

This creates an ETR file **x.etr** in the library called **test**:

```
// This file is generated and will be changed when the module is compiled

// The following template was
// explicitly instantiated
// __Of__F1&
template void f(long(;

// The following template was
// used in this module and can be instantiated here
// __Of__Fi&
template void f(int);

// The following template was
// instantiated automatically by the compiler
// __Of__Fc&
template void f(char);

// The following template was
// used in this module
// __Og__Fi&
template void g(int);
```

7.3.4 Implicit inclusion

The implicit inclusion of source files is a method of finding definitions of template entities. This method is enabled for the compiler by default (see also the option `IMPLICIT-INCLUDE` on [page 139](#)) and can be disabled with `IMPLICIT-INCLUDE=*NO`. Implicit inclusion must not be disabled when using templates from the standard C++ library (see the notes on [page 245](#) for details).

When implicit inclusion is enabled, the compiler looks for the definition of a template entity in accordance with the following principle:

If a template entity is declared in a header file named *basename.H* and no definition for it is available in the compiled source code, the compiler will assume that the definition for that template entity is contained in a source file

- which is located as a source program element in the same PLAM library as the header file and
- which has the same base name as the header file and a standard suffix that is valid for C++ source files, i.e.: `.C`, `.CPP`, `.CXX` or `.CC` (e.g. *basename.CC*).

Let us assume, for example, that a template entity `ABC::f` is declared in the header file `XYZ.H` in the library `PLAM.T`. If the instantiation of `ABC::f` is requested on compilation, but no definition of `ABC::f` exists in the compiled source code, the compiler will search the library `PLAM.T` for a source file with the base name `XYZ` and a standard suffix that applies to C++ source files, i.e.: `.C`, `.CPP`, `.CXX` or `.CC` (e.g. `XYZ.CC`). If such a file exists, it will be treated as if it were included at the end of the source file containing the `#include` directive for `XYZ.H`.

7.4 Deviations from ANSI/ISO C++

7.4.1 Extensions to ANSI-/ISO-C++

The extensions described below are accepted in all C++ language modes, except when the option `-R strict_errors` or `ANSI-VIOLATIONS=*ERROR` has been set in the strict ANSI C++ mode. Note that all extensions to ANSI/ISO C (see the [section “Extensions to ANSI/ISO C” on page 219ff](#)) are also supported in the C++ modes.

Declarations and definitions in classes

The following extensions will result in warnings if used in any of the C++ modes. In the strict ANSI C++ mode, the output of errors can be forced with the option `-R strict_errors` or `ANSI-VIOLATIONS=*ERROR`.

- A friend declaration for a class may omit the `class` keyword:

```
class A {  
    friend B; // ANSI requires friend class B;  
};
```

- Constants of scalar type may be defined within a class:

```
class A {  
    const int size = 10;  
    int a[size];  
};
```

- A qualified name may be used in the declaration of a class member:

```
struct A {  
    int A::f(); // ANSI requires int f();  
};
```

Preprocessor

The preprocessor macro `cplusplus` is defined in addition to the standard-compliant macro `__cplusplus`. No warning is issued.

7.4.2 extern inline vs. static inline

An inline function can be interpreted as `static inline` or `extern inline`. This question is important when local static variables are used in the function or when the address of such a function is used in a comparison.

Examples 1 and 2 at the end of this section illustrate the problem.

From the programmer's viewpoint, a static inline function behaves in exactly the same way as a static function. The function itself and all the elements declared in the function are local to the current compilation unit. The function itself does not interact in any way with elements of other compilation units; these must be produced by other (external) functions.

An extern inline function behaves like a normal external function. The body of the function must occur in every compilation unit which uses it. It must, however, always be identical. The behavior can be described by means of a comparison: it is as if the body were the only definition in an additional compilation unit and this function were called in all the other compilation units.

In addition to the definition above, the optimizer is also informed that an inline expansion is desirable. However, the meaning must be retained when this optimization is implemented.

The view of the C++ standard

If a function is declared with the keyword `inline`, it is implicitly regarded as `extern inline`. A `static inline` must be declared with the keywords `static` and `inline`.

A member function which is defined within a class is implicitly regarded as `extern inline`.

Implementation in the compiler

The compiler does not know the concept of an extern inline function. It interprets all inline functions as `static inline`.

This affects both the functions declared with the keyword `inline` and also the member functions defined within a class.

The problematical constructs

In many cases it is irrelevant whether a function is interpreted as `static inline` or `extern inline`. To permit this, one of three constructs must be used. The situation is problematical only when the function is used in multiple compilation units.

The functions affected are those which are conceived as `extern inline` (see *Example 3*). These functions are almost always contained in a header file, together with a body. Templates or member functions can be concerned here.

In the case of templates it must be borne in mind that the CPP compiler includes other sources if this function is not disabled. In this context, see the options `//MODIFY-SOURCE-PROPERTIES IMPLICIT-INCLUDE =` and `-K implicit_include` which are active in the default case (see *Example 4*).

- Local static variables

In such a situation a local `static` variable exists several times over in the compiler implementation, while the standard requires a single occurrence. Whether this leads to a problem depends on the reason why the variable was defined as local `static` (see *Example 5*).

If a single occurrence is indeed expected, a problem exists. Whether this will result in runtime errors depends very much on the actual usage.

The situation is different if the reason is the lifetime of the memory. A great deal of code sets up a string internally in order to return it as the return value. No `auto` variable can be used here because the memory is released when the value is returned. However, a local `static` remains valid until the function is called again. This is frequently used when texts are edited for output. In this case the variable is duplicated, but this has practically no effect on the program run.

- Comparison of addresses

An extern inline function has an unambiguous address, a static inline function one address per compilation unit. This is not relevant as long as the addresses are only used to call the function.

Things become critical when such addresses are compared with each other. However, this tends to happen only rarely. It would be conceivable when registering callback functions in which each function should only be entered once.

- String literal

This construct only returns a problem in exceptional cases. The standard does not define whether string literals with the same content also have the same address. It is also very unusual to use the address of a string literal in comparisons or similar functions.

Example 1: A problematical situation

```
// file bsp.h
#include <stdio.h>
class BSP {
public:
    inline int mf1(void);
};
inline int BSP::mf1(void)
{
    static int i = 1;
    return i++;
}
extern void f();

// file bsp1.c
#include "bsp.h"
void f()
{
    BSP bsp;
    printf ("Value 1: %d\n", bsp.mf1() );
}

// file bsp2.c
#include "bsp.h"
void g()
{
    BSP bsp;
    printf ("Value 2: %d\n", bsp.mf1() );
}
int main()
{
    f();
    g();
    return 0;
}
```

Explanation

The function `mf1` is critical, as is the variable `i` which it contains. According to the standard this should only exist once. The two calls in `bsp1.c` and `bsp2.c` should therefore apply to the same variable. The required output is

```
Value 1: 1
Value 2: 2
```

However, in the implementation of the CPP compiler the function is processed separately in `bsp1.c` and `bsp2.c`. A different variable is addressed with the call in `bsp2.c` from the call in `bsp1.c`. Both variants are initialized with 1.

The output is then

Value 1: 1

Value 2: 1

Example 2: A problematical situation with templates

```
// file tpl.h
#include <stdio.h>
template <class T>
inline int tpl(T t)
{
    static int i = 1;
    return i++;
}
extern void f();

// file tpl1.c
#include "tpl.h"
void f()
{
    printf ("Value 1: %d\n", tpl(5) );
}

// file tpl2.c
#include "tpl.h"
void g()
{
    printf ("Value 2: %d\n", tpl(7) );
}
int main()
{
    f();
    g();
    return 0;
}
```

Explanation

The same notes apply as for *Example 1*, but a template is involved here rather than a member function. The output expected by the standard is

Value 1: 1

Value 2: 2

The output supplied by the CPP compiler is

Value 1: 1

Value 2: 1

Example 3: Functions affected

```

// file bf.h
    inline int f1(T) { }           // affected
    int f2(T);                   // not affected
    static inline int f3(T) { }  //not affected
                                as explicitly static

    template <class T> inline int tf1(T) { } //affected
    template <class T> int tf2(T);         //not affected
    template <class T> static inline int tf3(T) { } //not affected
                                                as explicitly static

class BSP {
public:
    inline int mf1(void);           //affected
    int mf2(void) { }             //affected
    int mf3(void);                 //not affected
};
inline int BSP::mf1(void) { }

```

Example 4: Implicit include

```

// file ii.h
    template <class T> inline int ii(T);
// file ii.c
    template <class T> inline int ii(T) //this function is affected,
                                        although it is contained in a
                                        .c file

    {
        static int i;
        return i++;
    }

```

Example 5: Problematical static variables

Usage is only presented as problematical/not problematical here. To become a real problem, the function must be a problematical inline function.

```
inline void no_recursion (void)
{
    static int active = 0;           //problematical
    active ++;
    if (active > 1)
    {
        illegal_recursion ();
    } else {
        do_something ();
    }
    active --;
}
```

This is problematical because the recursion is not reliably detected.

```
inline char * debug_text ()
{
    static char buffer [200];       // not problematical
    sprintf (buffer, "...", ...);
    return buffer;
}
```

This is not problematical because the content of the buffer is always used before the next call. There is normally nobody who saves and accepts the address, which means that its content changes reliably.

```
inline void f (void)
{
    static int actions = 0;         //(see below)
    actions++;
    if (actions > 200)
    {
        actions = 0;
        optimize_datastructures (); //Only cleaning up, no relevant
                                    change
    }
    do_something ();
}
```

Whether this is problematical depends very much on the `optimize_datastructures` function. If it is really the case that only optimization is being performed there, the program runs correctly despite actions being duplicated.

However, the performance is different because `optimize_datastructures` is no longer called on a regular basis.

7.5 Variations in the Cfront C++ mode

The behavior of the compiler in the Cfront C++ mode is compatible with Cfront C++ V3.0 and later versions, i.e. the compiler supports many of the corresponding function attributes and specific features. The Cfront C++ mode is supported so that existing code containing extensions to Cfront versions as of V3.0 can be compiled without manual intervention. It does not guarantee full compatibility with the earlier C++ compilers (V2.1, V2.2).

Consequently, note that if a program produces an error when compiled with the C++ V2.1/V2.2 compiler, it is possible that the C/C++ V3.0 compiler may produce a different error or no error at all in the Cfront C++ mode. Some of the special aspects to be noted for the Cfront C++ mode are described below.

- `const` qualifiers on the `this` parameter may be dropped in some contexts, as in this example:

```
struct A {
    void f() const;
};

void (A::*fp)() = &A::f;
```

This is actually a safe operation. A pointer to a function of type `const` may be put into a pointer to a non-`const` type, since a call using the pointer is permitted to modify the object, and the function pointed to will actually not modify the object. An assignment in the reverse direction would not be safe.

- Conversion operators specifying conversion to `void` are allowed.
- A `friend` declaration may introduce a new type. A `friend` declaration that omits the elaborated type specifier is allowed in the ANSI C++ mode, but is also allowed to introduce a new type name in the Cfront mode.

```
struct A {
    friend B;
};
```

- The third operator of the `?` operator is a conditional expression instead of an assignment expression.

- A reference to a pointer type may be initialized from a pointer value without using a temporary variable even if the reference pointer type has supplementary type qualifiers in addition to those present in the pointer value. For example:

```
int *p;
const int *&r = p;      // No temporary used
```

- A reference variable may be initialized with 0.
- Since the accessibility of types is not checked in the Cfront mode, access errors for types are issued as warnings instead of errors.
- When calling overloaded functions, a null pointer must be written as a string in the form "0". Other notations such as `const` variables with the value 0 or constants in the form `'\0'` are not interpreted as a null pointer by the compiler in the case of overloaded functions.
- No warning is issued when an `operator()()` function has default argument expressions.
- An alternate form of declaring pointer-to-member-function variables is supported. This is illustrated in the example below:

```
struct A {
    void f(int);
    static void sf(int);
    typedef void A::T3(int);    // nonstd typedef decl
    typedef void T2(int);      // std typedef
};

typedef void A::T(int);        // nonstd typedef decl
T* pmf = &A::f;                // nonstd ptr-to-member decl
A::T2* pf = A::sf;             // std ptr to static mem decl
A::T3* pmf2 = &A::f;           // nonstd ptr-to-member decl
```

In this example, `T` names a routine type for a nonstatic member function of class `A` that takes an `int` argument and returns `void`. The use of such types is restricted to non-standard pointer-to-member declarations. The declarations of `T` and `pmf` in combination are equivalent to a single standard pointer-to-member declaration in the form:

```
void (A::* pmf)(int) = &A::f;
```

A non-standard pointer-to-member declaration that appears outside of a class declaration, such as the declaration of `T`, is normally invalid and would cause an error to be issued. For declarations that appear within a class declaration, such as `A::T3`, this feature changes the meaning of a valid declaration.

- protected class members are not checked when the address of a protected member is specified.

```
class B { protected: inti; };
class D : public B {void mf(); };

void D::mf() {
    int B::* pm1 = &B::i;    // error in ANSI mode, OK in Cfront mode
    int D::* pm2 = &D::i;    // OK
}
```

Note that the checking of protected class members for other operations (i.e. everything except the declaration of pointer-to-member addresses) is handled as defined by the standard in Cfront mode.

- The destructor of a derived class may implicitly call the private constructor of a base class. This is an error in the ANSI C++ mode, but is reduced to a warning in the Cfront C++ mode. For example:

```
class A {
    ~A();
};

class B : public A {
    ~B();
};

B::~B(){}           // Error except in Cfront mode
```

- When disambiguation requires deciding whether something is a parameter declaration or an argument expression, the pattern *type-name-or-keyword (identifier...)* is treated as an argument. For example:

```
class A { A(); };
double d;
A x(int(d));
A(x2);
```

According to the standard, `int(d)` is interpreted as a parameter declaration (with redundant parentheses), which means that `x` is a function. In the Cfront C++ mode, `int(d)` is interpreted as an argument, so `x` is a variable.

Note that the declaration `A(x2);` is also interpreted differently in the Cfront C++ mode as compared to the standard. The standard dictates that it should be interpreted as the declaration of an object named `x2`, but in the Cfront C++ mode it is interpreted as a cast of `x2` to type `A`.

A similar deviation from the standard can be seen in the interpretation of the following declaration:

```
int xyz(int());
```

According to the standard, this declares the function `xyz`, which takes a parameter that is a function without arguments and returns an `int`. In the Cfront mode, this is interpreted as the declaration of an object that is initialized with the value 0.

- Bitfields

A named bitfield may have a size of 0. The declaration is treated as if no name were declared.

Plain bitfields, i.e. bitfields declared with the type `int`, are always unsigned.

- The name for a type specifier may be a typedef name that is a synonym for a class name:

```
typedef class A T;
class T *pa;          // No error in Cfront mode
```

- No warning is issued on duplicate size and sign (signed or unsigned) type specifiers:

```
short short int i;   // No warning in Cfront mode
```

- Virtual function table pointer update code is not generated in destructors for base classes of classes without virtual functions, even if the base class virtual functions might be overridden in a further derived class. For example:

```

struct A {
    virtual void f() {}
    A() {}
    ~A() {}
};

struct B : public A {
    B() {}
    ~B() {f();}           // Should call A::f according to ARM 12.7
};

struct C : public B {
    void f() {}
} c;

```

In the Cfront mode, `B::~~B` calls the function `C::f`.

- An extra comma is allowed after the last argument in an argument list:

```
f(1,2,);
```

- A constant pointer-to-member function may be cast to a pointer-to-function. Only a warning is issued:

```

struct A {int f();};
main() {
    int (*p)();
    p = (int (*)( ))A::f;    // OK, with warning
}

```

- Arguments of class types that allow bitwise copy construction but also have destructors are passed by value (like C structures), and the destructor is not called on the “copy”. In ANSI mode, the class object is copied to a temporary object; the address of the temporary object is passed as the argument, and the destructor is called on the temporary object after the call returns. In practice, this is not much of a problem, since classes that allow bitwise copying usually do not have destructors.
- A union member may be declared to have the type of a class for which the user has defined an assignment operator (as long as the class has no constructors or destructors). However, a warning is issued.

- If an unnamed class appears in a typedef declaration, the typedef name may be used as the class name.

```
typedef struct {int i, j; } S;
struct S x;           // No error in Cfront mode
```

- A typedef name may be used in an explicit destructor call:

```
struct A { ~A(); };
typedef A B;
int main() {
    A *a;
    a->~B();           // Permitted in Cfront mode
}
```

- Two member functions may be declared with the same parameter types when one is static and the other is nonstatic with a function qualifier. For example:

```
class A {
    void f(int) const;
    static void f(int);    // no error in Cfront mode
}
```

- When two functions have the same name and very similar parameter types, they cannot be used together. This is the case when the parameter types differ only in the following aspects:

- char vs. signed char;

```
f(char);
f(signed char);
```

- Array limits

```
f(char(*x)[15]);
f(char(*x)[18]);
```

- A const qualification of a typedef

```
typedef char c;
f(const c*);
f(c*);
```


- Indirect or combined use of these aspects

```
f(char*,int);  
f(signed char*,int);  
typedef char c;  
g(char,c*);  
g(signed char,const c*);
```

If both functions are defined nevertheless, this is interpreted as a duplicate. No message is issued during compilation.

8 The C++ libraries and C++ runtime system

The following C++ libraries are provided with CRTE:

- a standard C++ library based on the ANSI C++ draft
- a C++ library compatible with Cfront V3.0.3
- Tools.h++ V7.0

8.1 The standard C++ library

The standard C++ library can only be used in the ANSI C++ modes of the compiler. This library includes the following interfaces:

- A string class
 - `<string>`
- Container classes
 - `<bitset>`
 - `<deque>`
 - `<list>`
 - `<map>`
 - `<queue>`
 - `<set>`
 - `<stack>`
 - `<vector>`
- Iterators
 - `<iterator>`
- Generic algorithms
 - `<algorithm>`
- Numeric classes and operations
 - `<complex>`
 - `<numeric>`

- I/O classes

```
<iostream.h>
<fstream.h>
<strstream.h>
<stdiostream.h>
<iomanip.h>
```

The I/O classes are currently not ANSI-compliant and correspond to the Cfront V3.0.3-compatible I/O library `iostream`.

The header files for the above interfaces are contained in the library `SYSLIB.CRTE`, and the modules are contained in the library `SYSLNK.CRTE.STDCPP`.

All names of the standard C++ library, except for the I/O classes, are available in the `std` namespace.

The following header files (also contained in `SYSLIB.CRTE`), in which all ANSI C library functions are defined in the `std` namespace, constitute a further component of the standard C++ library. The names of these header files are derived from the names of the ANSI C headers as follows: each name is preceded by the letter `c`, and the `.h` suffix is dropped.

<cassert>	<ciso646>	<csetjmp>	<cstdio>	<ctime>
<cctype>	<climits>	<csignal>	<cstdlib>	<cwchar>
<cerrno>	<locale>	<cstdlibarg>	<cstring>	<cwctype>
<cfloat>	<cmath>	<stddef>		

Using the library

Program development in the BS2000 environment (SDF)

The library `SYSLIB.CRTE` must be searched for standard headers at compilation. This can be ensured by specifying the search for standard headers with the option `STD-INCLUDE-LIBRARY= *STANDARD-LIBRARY` (default) in the `MODIFY-INCLUDE-LIBRARY` statement.

The program must be linked with the `BIND` statement of the compiler, and the (default) entry `RUNTIME-LANGUAGE=*CPLUSPLUS(MODE=*ANSI)` must be specified in the `MODIFY-BIND-PROPERTIES` statement.

For further details see [page 79](#).

Program development in the POSIX environment

In order to include header files and link in the required modules, one of the ANSI C++ modes (`-X w`, `-X e`) must be specified in the `CC` command. The default setting for the `CC` command is `-X w` (extended ANSI C++). For more details, see also the manual "POSIX Commands of the C/C++ Compiler".

Documentation

The standard C++ library is described in detail in the following manual: "Standard C++ Library V1.2, User's Guide and Reference" [6] (one volume).

A description of the I/O classes compatible with Cfront V3.0.3 can be found in the "Cfront C++ Library", Reference Manual [5].

8.2 The Cfront C++ library

The C++ library compatible with Cfront V3.0.3 can only be used in the Cfront C++ mode of the compiler. This library includes the following interfaces:

- A class for complex math

`<complex.h>`

- Classes for stream-oriented I/O

`<iostream.h>`

`<fstream.h>`

`<strstream.h>`

`<stdiostream.h>`

`<iomanip.h>`

`<generic.h>`

`<new.h>`

The header files for the above interfaces are contained in the library `SYSLIB.CRTE.CPP`, and the modules are located in the library `SYSLNK.CRTE.CPP`.

Using the library*Program development in the BS2000 environment (SDF)*

The headers of the Cfront C++ library contain declarations which use the "at" sign (@). You should note that it will not be possible to use the Cfront C++ library where *NO is specified in the option `AT-ALLOWED=...` in the `MODIFY-SOURCE-PROPERTIES` statement.

In the search for standard headers at compilation, the library SYSLIB.CRTE.CPP must be searched before the library SYSLIB.CRTE. This can be ensured by specifying the search for standard headers with the option `STD-INCLUDE-LIBRARY= *STANDARD-LIBRARY` (default) in the `MODIFY-INCLUDE-LIBRARY` statement.

When linking with the `BIND` statement of the compiler, only the following entry must be specified in the `MODIFY-BIND-PROPERTIES` statement:
`RUNTIME-LANGUAGE=*CPLUSPLUS(MODE=*CPP)`.

Program development in the POSIX environment



If you use the Cfront C++ library, you cannot use the option `-K no_at`.

In order to include header files and link in the required modules, only the language mode option `-X d` needs to be specified in the `CC` command. Note, however, that the `-X d` option must be specified for both compilation and linkage:

```
CC -X d -c x.C y.C      # Compilation
CC -X d x.o y.o        # Linkage
```

Documentation

The Cfront C++ library is described in detail in the following manual:
“C++ V2.1 (BS2000) C++ Library Functions” [5].

8.3 The Tools.h++ library

The Tools.h++ V7.0 library can only be used in the ANSI C++ modes of the compiler.

This library offers a broad spectrum of “foundation classes”, i.e.:

- string classes with pattern-matching mechanisms
- classes to handle the date and time
- virtual streams
- file and file manager classes
- container classes (collectable) with the option of implementing persistence and associated iterator classes:
 - Smalltalk-like container classes (without template usage)
 - template container classes to store values (RWTVal1...)
 - template container classes to store pointers (RWTPtr...)
- classes for internationalization

The header files for the above interfaces are contained in the library SYSLIB.CRTE, and the modules are contained in the libraries SYSLNK.CRTE.TOOLS.

Using the library

Program development in the BS2000 environment (SDF)

The library SYSLIB.CRTE must be searched for standard headers at compilation. This can be ensured by specifying the search for standard headers with the option STD-INCLUDE-LIBRARY= *STANDARD-LIBRARY (default) in the MODIFY-INCLUDE-LIBRARY statement.

The program must be linked with the BIND statement of the compiler, and the following options must be specified in the MODIFY-BIND-PROPERTIES statement:
RUNTIME-LANGUAGE=*CPLUSPLUS(MODE=*ANSI) (default) and TOOLSLIB=*YES.

For further details see [page 79](#).

Program development in the POSIX environment

In order to include header files and link in the required modules, one of the ANSI C++ modes (`-X w`, `-X e`) must be specified in the `CC` command. The default setting for the `CC` command is `-X w` (extended ANSI C++).

Furthermore, when linking the program, the option `-l Rwttools` must also be specified. For more details, see also the manual “POSIX Commands of the C/C++ Compiler” [1].

Documentation

The Tools.h++ library is described in detail in the following manuals:

“Tools.h++ V7.0, User’s Guide” and

“Tools.h++ V7.0, Class Reference”

Information on the documentation

In order to make the *tools.h++* library independent of incompatible future versions of the standard library, it is not based on the standard library (i.e. the `RW_NO_STL` directive has been placed in the central configuration header `w/compiler.h`).

Certain classes are therefore not available or only available with restrictions. These are identified accordingly in the “Tools.h++ Class Reference” manual [8], e.g.

“`RWTValMap` requires the Standard C++ Library.”

When the associated headers are used, a `#error` error message is issued, e.g.

```
“Cannot include header if RW_NO_STL macro is defined for your compiler”
```

```
or
```

```
“You must have both Standard Library and Exceptions to use this class.”
```

Some classes implement functions which are linked to particular systems:

header `rw/xdrstrea.h`; class `RWXDRistream` and `RWXDRostream`

These classes can be used in BS2000/POSIX, but not in BS2000 native.

header `rw/winstrea.h`; class `RWCLIPstreambuf`

The “Tools.h++ Class Reference” manual [8] contains the following information on this:

“Class `RWCLIPstreambuf` is a specialized `streambuf` that gets and puts sequences of characters to Microsoft Windows global memory. It can be used to exchange data through Windows clipboard facility.”

As neither the concept of a “Microsoft Windows global memory” nor that of a “Windows clipboard” exists in Windows, this class is not available here.

8.4 The C++ runtime system

8.4.1 Initialization

Exceptions thrown during the initialization of global objects result in a call to `terminate` and thus cause the program to abort without diagnostics. If desired, you can use `set_terminate` to specify some other exception-handling routine to be used in the event of an unforeseen program abort. This function must, however, be called before initializing the global objects.

The C++ runtime system offers the following solution to specify functions to be used as the “initial current handler”:

- You can link your own `__initial_terminate_handler` function of type `terminate_handler` into your program. This function is declared weak in the C++ runtime system. If `__initial_terminate_handler` is defined, the function will then be called as the “initial handler” to terminate exception processing.
- You can also use the functions `__initial_unexpected_handler` and `__initial_new_handler` with the same mechanism. These routines are of type `unexpected_handler` and `new_handler`, respectively.

You must include the header file `<exception>` for the `__initial_terminate_handler` and `__initial_unexpected_handler` interfaces and the header file `<new>` for the `__initial_new_handler`.

8.4.2 Exception handling

Additional runtime functions

Besides the runtime functions defined by the language standard, the C++ runtime system offers a number of useful functions with which programs can be made more reliable. Note, however, that programs which use these functions are not ANSI C++-compliant and are hence not portable.

The following additional functions are available: `unwind_exit`, `get_caught_object_typeid`, `can_throw` and `can_rethrow`.

These functions are all part of the `_SNI_extensions` namespace.

`unwind_exit` - Unwind stack before exiting program

The `unwind_exit` function, like `exit`, is used to terminate a program. In contrast to `exit`, however, a call to `unwind_exit` causes the following additional actions to be performed before the program is exited:

- All automatic objects on the runtime stack that have not yet been deleted are destroyed.
- All exception objects that have not been exited are destroyed.

This is followed by the destruction of global objects as in the case of `exit`.

Note that neither `exit` nor `unwind_exit` will destroy objects on the heap that have not been released.

If a destructor called by `unwind_exit` ends with an exception, `terminate` is called implicitly. It is therefore advisable to call `unwind_exit` in the `terminate_handler` as well. This will guarantee that the destructors for all automatic and exception objects are eventually called in any case. This cannot result in an endless loop.

The `unwind_exit` function can be called from any part of the program (like `exit`), especially from a `terminate_handler`. Like `exit`, it is supplied with an exit status as an argument (and with the same effect).

The prototype of the `unwind_exit` function is declared in the `<exception>` header:

```
#include <exception>

namespace _SNI_extensions {
    void unwind_exit(int status);
}
```

get_caught_object_typeid - Determine type of caught exception object

The function `get_caught_object_typeid` can be used to determine the type of a caught exception object. It returns the type of the exception object that was most recently caught and was not finished. If the exception object is a pointer type, the type of the object pointed to is returned. If no caught and unfinished exception object exists, an exception of type `bad_typeid` is thrown.

The type of the caught exception object is returned as a reference to a `type_info` object. The class `type_info` and the prototype of the function `get_caught_object_typeid` are declared in the `<typeinfo>` header:

```
#include <typeinfo>

namespace _SNI_extensions {
    const type_info &get_caught_object_typeid(EO_flag_set *pflags);
}
```

If the argument to `get_caught_object_typeid` is non-zero, it is interpreted as an address containing information on whether the caught object is a pointer, and if it is, also the type attributes applicable to the object pointed to.

```
EO_NO_FLAGS           : not a pointer
EO_IS_POINTER         : pointer
EO_POINTER_TO_CONST   : pointer to constant
EO_POINTER_TO_VOLATILE : pointer to volatile
```

The function `get_caught_object_typeid` can be called in any handler (`terminate_handler`, `unexpected_handler`, `catch(...)` {...}) to obtain information about the type of the caught exception object. This can be useful in diagnosing program runtime errors.

can_throw - Check for terminate or unexpected on throwing an object

The function `can_throw` (predicate) checks whether an exception of the specified type can be thrown without resulting in a call to `terminate` or `unexpected`.

```
#include <typeinfo>

namespace _SNI_extensions {
    bool can_throw(const std::type_info &typeid_to_check,
                  EO_flag_set flags = EO_NO_FLAGS);
}
```

`typeid_to_check` must be a non-pointer type of an exception object.

`flags` can be used to specify whether the type of exception to be actually checked is a pointer or whether it is a `const` or `volatile`:

```
EO_NO_FLAGS           : not a pointer type
EO_IS_POINTER         : pointer type
EO_POINTER_TO_CONST   : pointer to const
EO_POINTER_TO_VOLATILE : pointer to volatile
```

If the `flags` argument is not specified, `EO_NO_FLAGS` (not a pointer type) is set by default.

can_rethrow - Check for terminate or unexpected on rethrowing an object

The function `can_rethrow` (predicate) checks whether an exception can be rethrown (`throw;`) without resulting in a call to `terminate` or `unexpected`.

```
#include <typeinfo>

namespace _SNI_extensions {
    bool can_rethrow(void);
}
```

C signal handling and C++ exception handling

According to the ANSI-C++ language definition, the use of C++ exception handling is not allowed in C signal routines. If a C signal routine is nonetheless reached by the runtime system during stack processing triggered by an exception, the system behaves as if the end of the stack were reached. Consequently, it is not possible to throw an exception from a signal routine or to rethrow an exception that was caught with `catch` before the signal routine was called. In such cases, `terminate` is invoked instead.

The function `unwind_exit` (see [page 282](#)) does not destroy all objects when called from signal routines. All `automatic` objects which were not deleted before the call to the signal routine and all exception objects that had not yet terminated are not destroyed in this case. However, the program still terminates with `exit`.

longjmp support

After a `longjmp` call, the following actions are performed by the C++ runtime system for each function that is skipped:

- All `automatic` objects that were constructed in the function but not deleted are destroyed.
- All exception objects which were caught in the function but which have not terminated are destroyed.

Note, however, that these destructors are not invoked for the target function of the `longjmp` call itself. Consequently, in a function containing a `setjmp` call, no object should be constructed, destroyed or caught between the `setjmp` call and the entry of the routine aborted by the `longjmp`. This can be achieved by relocating all code for constructors, destructors and catching exceptions to a separate function. Furthermore, in order to prevent this function from being inlined, it must be called via an external function pointer. Consider the following example:

Example

Instead of using the code of the function `f()` in Version 1, the code of Version 2 is to be used:

```
/* Version 1 */
#include <setjmp>

jmp_buf target;
void g();
class X { ~X(); };

void g()
{
    longjmp(target, 1);
}

void f()
{
    if (setjmp(target) == 0)
    {
        X x;           // No destruction for longjmp call by g()
        (g);
    }
}

/* Version 2 */

void f1()
{
    X x;           // Destruction for longjmp call by g()
    g();
}

extern void (*f1p)() = f1;

void f()
{
    if (setjmp(target) == 0)
    {
        (*f1p)();     // f1() is not inlined here
    }
}
```

longjmp and signal routines

When `longjmp` is called from a signal routine, no destructors for automatic and exception objects are executed in the function in which the signal occurred.

Linking old C modules with ANSI C++ modules

C modules generated with the C or C++ Compiler V2.2 can be linked with ANSI C++ modules. Note, however, that the following restrictions apply with respect to stack handling by the C++ runtime system:

- No exception may be thrown if there is still an active C V2.2 function between the call to `throw` and the exception handler (`catch`) which catches the exception.
- No exception may be rethrown if there is still an active C V2.2 function between the call to `throw`; and the exception handler (`catch`) that has caught the rethrown exception.
- No `longjmp` across a C V2.2 function may be executed from a C++ function.

9 Appendix

9.1 Description of listings

The listing generators of the compiler produce the following listings, depending on what is requested in the MODIFY-LISTING-PROPERTIES statement:

Listing	Options of the MODIFY-LISTING-PROPERTIES statement
Option listing	OPTIONS
Source/error listing	SOURCE
Preprocessor listing	PREPROCESSING-RESULT
Map listing	DATA-ALLOCATION-MAP
Cross-reference listing	CROSS-REFERENCE
Project listing	PROJECT-INFORMATION
Object code listing	ASSEMBLER-CODE
Summary listing	SUMMARY

The following listings are described below by means of examples: source/error, map, cross-reference, and object listings.

Source/error listing

The source/error listing is requested with the SOURCE option of the MODIFY-LISTING-PROPERTIES statement.

*** SOURCE - ERROR - LISTING ** C/C++(BS2000/OSD) COMPILER 03.2D00 DATE:2011-11-02 PAGE: 1
 SOURCENAME:*BS2000(MAINPROG) TIME=17:37:07

(1)	(2)	(3)	(4)	(5)	
EXP	INC	FILE	SRC	BLOCK	
LIN	LEV	NO	LIN	LEV	
1	0	0	1	0	#include <stdio.h>
1746	0	0	2	0	#include "incl1.h"
1747	1	10	1	0	class A
1748	1	10	2	0	{
1749	1	10	3	0	int i;
1750	1	10	4	0	public:
1751	1	10	5	0	A(int x = 1) : i(x) {};
1752	1	10	6	0	void foo() { printf("A::foo called\n"); };
1753	1	10	7	0	} a;
1754	0	0	3	0	#include "incl2.h"
1755	1	11	1	0	class B : public A
1756	1	11	2	0	{
1757	1	11	3	0	int i;
1758	1	11	4	0	public:
1759	1	11	5	0	B(int x = 2) : i(x) {};
1760	1	11	6	0	void foo() { printf("B::foo called\n"); };
1761	1	11	7	0	} b;
1762	0	0	4	0	extern "C" int jj;
1763	0	0	5	0	extern int ii;
1764	0	0	6	0	
1765	0	0	7	0	int main(void)
1766	0	0	8	0	{
1767	0	0	9	1	char *string = "AbCdEfG";
1768	0	0	10	1	float xx = 1.0;
1769	0	0	11	1	int ii = 1;
1770	0	0	12	1	int jj = 2;
1771	0	0	13	1	A* aptr = &a;
1772	0	0	14	1	A* bptr = &b;
1773	0	0	15	1	
1774	0	0	16	1	printf("%d\n", ii);
1775	0	0	17	1	printf("%d\n", jj);
1776	0	0	18	1	printf("%s\n", string);
1777	0	0	19	1	printf("%f\n", xx);
1778	0	0	20	1	a.foo();
1779	0	0	21	1	aptr->foo();
1780	0	0	22	1	b.foo();
1781	0	0	23	1	bptr->foo();
1782	0	0	24	1	
1783	0	0	25	1	return 0;
1784	0	0	26	1	}

- (1) Serial line number in the source listing, including all lines of the header elements used in the source program. The lines from the header elements are always included in the count, regardless of whether or not they are displayed in the source listing (see the INCLUDE-INFORMATION option).
- (2) Nesting level of the header elements.
- (3) Number of the file (source file or header element) for which the respective contents are mapped in the source listing. This number (starting with 0 for the source file) is incremented by 1 for each `#include` or `#line` directive. At the end of each header element, the number is reset to the value of the file containing the associated `#include` directive. This number is relevant for the source reference for debugging with AID if header elements contain executable statements or if `#line` directives are interspersed with executable statements in source programs. Only the user-defined header elements, i.e. `incl1.h` (10) and `incl2.h` (11), are expanded in the example. The standard header element `stdio.h` and the other `#include` directives contained in it are assigned the numbers 1 to 9 (see also the FILETABLE section in the cross-reference listing).
- (4) Original line number in the source file or header element, taking `#line` directives into account.
- (5) Nesting level of the statement blocks.
- (6) The contents of header elements, depending on what is specified in the INCLUDE-INFORMATION option of the MODIFY-LISTING-PROPERTIES statement (only the user-defined header elements in this case).

Map listing

The map listing is requested with the DATA-ALLOCATION-MAP option of the MODIFY-LISTING-PROPERTIES statement. It provides information on all the symbolic addresses used in the program (names of variables and functions).

***** MAP - LISTING ***** C/C++ (BS2000/OSD) COMPILER 03.2D00 DATE:2011-11-02 PAGE: 3
 SOURCENAME:*BS2000(MAINPROG) TIME=17:37:33

(1) name	(2) stcl/type	(3) size	(4) slice	(5) offs	(6) xoffs	(7) enuval	(8) stroffs	(9) xstroffs	(10) xstroffs
a	nospec class	4	1	104	0x0068	-	-	-	-
{									
}									
aptr	auto pointer to class	4	1	24	0x0018	-	-	-	-
b	nospec class	8	1	108	0x006C	-	-	-	-
{									
}									
bptr	auto pointer to class	4	1	28	0x001C	-	-	-	-
ii	extern signed int	4	-	-	-	-	-	-	-
ii	auto signed int	4	1	16	0x0010	-	-	-	-
jj	extern signed int	4	-	-	-	-	-	-	-
jj	auto signed int	4	1	20	0x0014	-	-	-	-
string	auto pointer to char	4	1	8	0x0008	-	-	-	-
x	param signed int	4	-	-	-	-	-	-	-
x	param signed int	4	-	-	-	-	-	-	-
xx	auto float	4	1	12	0x000C	-	-	-	-
A	nospec class	4	-	-	-	-	-	-	-
{									
}									
B	nospec class	8	-	-	-	-	-	-	-
{									
}									

- (1) Name of the symbolic address.
- (2) stcl (storage class): Storage class of the symbolic address. The following specifiers are used:
 - extern External variables and functions that are defined in a different module.
 - entry_var Functions that are defined or declared in the source file.
 - static Static variables at block level, i.e. that are valid at block level.
 - param Function parameters.
 - auto Variables at block level, excluding static variables (see istat).
 - enum Member of an enumeration type.
 - member Elements of classes, structures or unions.
 - statmem Static element of a class.
 - typedef typedef name.
 - lab_const Label.
- (3) type: Data type of the symbolic address (in a separate line under the storage class). The following abbreviations are used:
 - funct ret function returning
 - long long int
 - ptr pointer
 - short short int
 - struct structure
 - uchar unsigned char
 - schar signed char
- (4) size: Size of the variable in memory (in bytes).
- (5) slice: A slice is an area (code or data fragment) with a length of 4096 bytes that can be addressed via a base register. The digit specifies in which slice of the data module the variable is located.

- (6) offs: Relative address within a slice (decimal).
- (7) xoffs: Relative address within a slice (hexadecimal).
- (8) enuval: For members of an enumeration type (enum), enuval specifies the value of the member.
- (9) stroffs: Byte position of the symbolic address within a structure (decimal).
- (10) xstroffs: Byte position of the symbolic address within a structure (hexadecimal).

Cross-reference listing

The cross-reference listing is requested with the CROSS-REFERENCE option of the MODIFY-LISTING-PROPERTIES statement.

It consists of the following parts:

- The FILETABLE section contains the names of all files or libraries/elements that were used by the compiler as a source (source program or header element). A number is assigned to each of these names. These numbers are referenced in the other sections of the cross-reference listing.
- The PREPROCESSING-INFO section contains a list of the names processed by the preprocessor in `#include` and `#define` directives (macros, header element names, etc.).
- The TYPES section contains a list of the user-defined types (typedefs, classes, structure, union, and enumeration types).
- The VARIABLES section contains a list of variables.
- The FUNCTIONS section contains a list of functions.
- The LABELS section contains a list of labels.
- The TEMPLATES section contains a list of templates (in ANSI C++ mode only).

The names in the individual lists are sorted in alphabetical order.

The PREPROCESSING-INFO, TYPES and TEMPLATES sections are not included in the cross-reference listing by default and must be explicitly specified with PREPROCESSING-INFO=*YES, TYPES=*YES or TEMPLATES=*YES, respectively.

FILETABLE section of the cross-reference listing

```
***** XREF - LISTING *****      C/C++ (BS2000/OSD) COMPILER 03.2D00      DATE:2011-11-02      PAGE: 1
FILETABLE                          SOURCENAME:*BS2000(MAINPROG)      TIME=17:37:52
```

```
SOURCE FILE      0      = *BS2000(:20SC:$TST30B.MAINPROG)
INCLUDE FILE     1      = *LIBRARY-ELEMENT(:20SL:$TSOS.SYSLIB.CRTE,STDIO.H(V02.3B08),S)
INCLUDE FILE     2      = *LIBRARY-ELEMENT(:20SL:$TSOS.SYSLIB.CRTE,IOPUF.H(V02.3B08),S)
INCLUDE FILE     3      = *LIBRARY-ELEMENT(:20SL:$TSOS.SYSLIB.CRTE,POSFILE.H(V02.3B08),S)
INCLUDE FILE     4      = *LIBRARY-ELEMENT(:20SL:$TSOS.SYSLIB.CRTE,STDIO.BS21.H(V02.3B08),S)
INCLUDE FILE     5      = *LIBRARY-ELEMENT(:20SL:$TSOS.SYSLIB.CRTE,STDIO.COMMON.H(V02.3B08),S)
INCLUDE FILE     6      = *LIBRARY-ELEMENT(:20SL:$TSOS.SYSLIB.CRTE,STDIO.BS22.H(V02.3B08),S)
INCLUDE FILE     7      = *LIBRARY-ELEMENT(:20SL:$TSOS.SYSLIB.CRTE,CGLOBALS.H(V02.3B08),S)
INCLUDE FILE     8      = *LIBRARY-ELEMENT(:20SL:$TSOS.SYSLIB.CRTE,IOPUF.H(V02.3B08),S)
INCLUDE FILE     9      = *LIBRARY-ELEMENT(:20SL:$TSOS.SYSLIB.CRTE,ERRNO.H(V02.3B08),S)
INCLUDE FILE    10      = *LIBRARY-ELEMENT(:20SC:$TST30B.PLAM.INCL,INCL1.H(*UPPER-LIMIT),S)
INCLUDE FILE    11      = *LIBRARY-ELEMENT(:20SC:$TST30B.PLAM.INCL,INCL2.H(*UPPER-LIMIT),S)
```

File number in global cross-reference listings

In a cross-reference listing that was created with the global listing generator from multiple CIF files (see the [section “Controlling the global listing generator” on page 156](#)), the file number is indicated in the form $n(m)$, where n is the sequential number of the source and header files (analogous to the local cross-reference listing; see above) used for each compilation unit (= CIF file), and m is the number of the respective compilation unit. The numbering of compilation units begins with 0.

PREPROCESSING-INFO section in the cross-reference listing

```
***** XREF - LISTING *****      C/C++ (BS2000/OSD) COMPILER 03.2D00      DATE:2011-11-02      PAGE: 2
PREPRO                          SOURCENAME:*BS2000(MAINPROG)      TIME=17:37:52
```

```
56/12:5
*LIBRARY-ELEMENT(:20SC:$TST30B.PLAM.INCL,INCL1.H(*UPPER-LIMIT),S) / include file
2%0
*LIBRARY-ELEMENT(:20SC:$TST30B.PLAM.INCL,INCL2.H(*UPPER-LIMIT),S) / include file
3%0
```

```
.'.applied ':def ':^'undef '%included
```


TYPES section in the cross-reference listing

```
***** XREF - LISTING *****          C/C++ (BS2000/OSD) COMPILER 03.2D00          DATE:2011-11-02  PAGE:   3
      TYPES                             SOURCENAME:*BS2000(MAINPROG)          TIME=17:37:52
```

```
-----
a0000270          / struct of size 16 (0x10)
std              / namespace
A               / class of size 4 (0x4) ----- (1)
  1/7:10        1/18.11      13/3.0      14/3.0 ----- (2)
                public baseclass of class 'B'
  i             / member, signed int, private ----- (3)
  A             / inline constructor( signed int ), public
  foo          / member, inline function( void ) ret void, public
B               / class of size 8 (0x8)
  1/7:11
  A             / baseclass, public
  i             / member, signed int, private
  B             / inline constructor( signed int ), public
  foo          / member, inline function( void ) ret void, public
-----
```

```
'.'used ':'def '&'decl
```

- (1) Name and description of the user-defined type, possibly with a size specification (decimal and hexadecimal).
- (2) From left to right:
 source program line and column in which the type appears,
 abbreviation symbol for usage of the type, and
 number of the source file or (header) element from the FILETABLE.
 For example, 14/3.0 means that a variable of type class A is defined in line 14,
 column 3 of the source program mainprog (0), and that the type is used (.).
- (3) In the case of structured types, the respective components of these types are also described (indented).
 The data members of structures, classes, and unions are listed only in the TYPES section (they are not variables). Members of functions are repeated in the FUNCTIONS section.

VARIABLES section in the cross-reference listing

***** XREF - LISTING ***** C/C++ (BS2000/OSD) COMPILER 03.2D00 DATE:2011-11-02 PAGE: 4
 VARIABLES SOURCENAME:*BS2000(MAINPROG) TIME=17:37:52

```

a          / class 'A' ----- (1)
7/3=10    7/3:10      13/14&0      20/3&0 ----- (2)
aptr      / automatic, pointer to class 'A',
          local in main( void ) ret signed int, init value = &a-
13/6=0    13/6:0      21/3.0
b          / class 'B'
7/3=11    7/3:11      14/14&0      22/3&0
bptr      / automatic, pointer to class 'A',
          local in main( void ) ret signed int
14/6=0    14/6:0      23/3.0
i          / member, signed int,
          member of class 'A'
3/10:10   5/21=10
i          / member, signed int,
          member of class 'B'
3/10:11   5/21=11
ii         / automatic, signed int,
          local in main( void ) ret signed int, init value = 1
11/7=0    11/7:0      16/18.0
ii        / extern, signed int
5/12:%0
jj         / extern, signed int
4/16:%0
jj         / automatic, signed int,
          local in main( void ) ret signed int, init value = 2
12/7=0    12/7:0      17/18.0
string    / automatic, pointer to char,
          local in main( void ) ret signed int, init value = "AbCdEfG"
9/9=0     9/9:0      18/18.0
x          / param of constructor A::A, signed int
5/12:10   5/23.10
x          / param of constructor B::B, signed int
5/12:11   5/23.11
xx        / automatic, float,
          local in main( void ) ret signed int, init value = 1
10/9=0    10/9:0      19/18.0

```

'='write '.'read '*='indir-write '*.'indir-read '&'read-addr ':'def '%'decl ':'%extdecl '%%'use

- (1) Name, storage class and data type of the variables.
- (2) From left to right:
 - source program line and column in which the variable appears,
 - abbreviation symbol for usage of the variable, and
 - number of the source file or (header) element from the FILETABLE

For example, 7/3:10 means that the variable a is defined (:) in line 7, column 3 of the header element incl1.h (10).

FUNCTIONS section in the cross-reference listing

```
***** XREF - LISTING *****      C/C++ (BS2000/OSD) COMPILER 03.2D00      DATE:2011-11-02      PAGE:   5
FUNCTIONS                          SOURCENAME:*BS2000(MAINPROG)      TIME=17:37:52
```

```
-----
foo      / public member of class 'B', inline function( void ) ret void ----- (1)
6/11:11  22/5.0
foo      / public member of class 'A', inline function( void ) ret void ----- (2)
6/11:10  20/5.0      21/9.0      23/9.0
main     / function( void ) ret signed int
7/5:0
A        / public member of class 'A', inline constructor( signed int )
5/6:10   7/3.10      5/26.11
B        / public member of class 'B', inline constructor( signed int )
5/6:11   7/3.11
```

```
-----
'.call ':def '&'decl '%extdecl ':^'forward '&.'read-addr
```

- (1) Name of the function, its scope, signature (type of parameters and return type), and storage class. For member functions, additional information on the access rights (e.g. public) and the class or union in which it is contained.
- (2) From left to right:
 - source program line and column in which the function appears,
 - abbreviation for usage at that position, and
 - number of the source file or (header) element from the FILETABLE.

For example, 6/11:11 means that the function `foo` is defined (:) in line 6, column 11 of the header element `incl2.h` (11).

Object listing

The object listing is requested with the ASSEMBLER-CODE option of the MODIFY-LISTING-PROPERTIES statement.

It contains

- the hexadecimal representation of object code generated by the code generator,
- the object code in Assembler notation,
- comments concerning the object code in Assembler notation,
- the source program lines in C/C++ notation as comments.

The overall listing is arranged into separate sections for code and data modules. Each module listing is divided into areas. The beginning of a given module or area listing is indicated by comment lines in the Assembler source program.

***** ASSEMBLER - LISTING *****

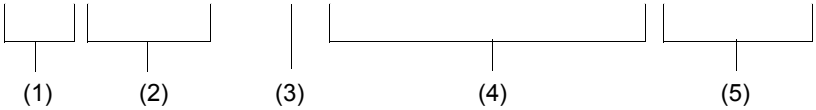
C/C++ (BS2000/OSD) COMPILER 03.2D00
 SOURCE: *BS2000(MAINPROG)

DATE:2011-11-02
 TIME=17:37:22

PAGE: 1

```

1 *****
2 *          CODE MODULE
3 *
4 *          OPTIMIZER:    ON
5 *          SHARED CODE:  OFF
6 *          OBJECT FORMAT: LLM
7 *****
8 MAINPROG\&@ CSECT      READ
000000 9 MAINPROG\&@ AMODE      ANY
000000 10 MAINPROG\&@ RMODE      ANY
11 *****
12 *          CODE AREA (main)
13 *****
14          ENTRY MAIN
000000 15 MAIN      DS          0A
000000 16          USING      *,15
000000 17          STM        14,15,12(13)
000004 18          STM        2,12,28(13)
000008 19          L          9,76(0,13)
00000C 20          LM        10,11,#DC#CONT#10      104(15)
000010 21          L          14,24(0,9)
000014 22          LA        0,216(0,14)
000018 23          CL        0,16(0,9)
00001C 24          BC        2,#OFLOW#10      68(0,15)
000020 25          ST        0,24(0,9)
000000 26          #OFLOWOK#10 EQU *
000024 27          ST        14,12(0,9)
000028 28          L          0,#DC#SAVID#10      136(0,15)
00002C 29          ST        0,0(0,14)
000030 30          L          0,#DC#EHL#10      140(0,15)
000034 31          ST        0,80(0,14)
000038 32          ST        13,4(0,14)
00003C 33          ST        9,76(0,14)
000040 34          LR        13,14
000042 35          BCR      15,10
.
.
.
000094 58 80 F084 69          L          8,132(0,15)
70          ***** LINE 7: (*BS2000(:20SC:$TST30B.MAINPROG))
71          ***** int main(void)
000098 72 @0000001 DS      0H
73          ***** LINE 8
74          ***** {
000098 75 @0000002 DS      0H
000098 76          L          15,36(0,11)      <17>      <17>
00009C 77          LA        0,0(0,0)
0000A0 78          BASR     14,15
79          ***** LINE 16
80          ***** printf("%d\n", ii);
    
```



- (1) Location counter, in hexadecimal notation
- (2) Object code, in hexadecimal notation
- (3) Line number of the Assembler code
- (4) Assembler code (symbolic address, Assembler mnemonics, operands) and source program line as a comment
- (5) Explanation of the Assembler code

9.2 Predefined preprocessor names

When the C/C++ compiler is used for compilation in the SDF environment (COMPILE, CHECK-SYNTAX, and PREPROCESS statements), some preprocessor macros and assertions are predefined, depending on which language mode is selected and which additional options are specified.

Predefined preprocessor macros (defines)

The options specified below are all part of the MODIFY-SOURCE-PROPERTIES statement:

<code>__BOOL</code>	In ANSI C++ mode with the option <code>KEYWORD-BOOL=*YES</code>
<code>__CGLOBALS_PRAGMA</code>	Always set
<code>__cplusplus</code>	In all C++ language modes: == 1 in Cfront C++ mode == 2 in extended ANSI C ++ mode == 199612L in strict ANSI C ++ mode
<code>cplusplus</code>	In all C++ language modes
<code>__CFRONT_V3</code>	In Cfront C++ mode
<code>__EDG_NO_IMPLICIT_INCLUSION</code>	In ANSI C++ modes, when implicit inclusion has been disabled with the option <code>IMPLICIT-INCLUDE=*NO</code> within the framework of template instantiation
<code>__EXISTCGLOB</code>	Always set
<code>LANGUAGE_C</code>	Always set
<code>__LANGUAGE_C</code>	Always set
<code>__LONGLONG</code>	Option <code>LONGLONG=*YES</code>
<code>__OLD_SPECIALIZATION_SYNTAX</code>	In ANSI C++ modes: == 1 with the option <code>SPECIALIZATION=*OLD</code> == 0 with the option <code>SPECIALIZATION=*NEW</code> (default)
<code>__SHORT_NAMES</code>	Is defined, if <code>C-NAMES=*SHORT</code> is specified
<code>__SIGNED_CHARS__</code>	Option <code>SIGNED-CHARACTER=*YES</code>
<code>__SNI</code>	In all C modes and in Cfront C++ mode

<code>__SNI_HOST_BS2000</code>	Always set
<code>__SNI__STDCplusplus</code>	In all C++ language modes: == 0 in Cfront and extended ANSI C++ mode == 1 in strict ANSI C++ mode
<code>__SNI_TARG_BS2000</code>	Always set
<code>__STDC__</code>	Always set: == 0 in the K&R C, extended ANSI C, Cfront C++ and extended ANSI C++ modes == 1 in strict ANSI C and strict ANSI C++ modes
<code>__STDC_VERSION__</code>	Undefined in K&R C mode == 199409L in ANSI C modes and in all C++ modes
<code>_STRICT_STDC</code>	In strict ANSI C and C++ modes
<code>_WCHAR_T</code>	In ANSI C++ modes with the option <code>KEYWORD-WCHAR=*YES</code> (default) If this option is not set (e.g. in C modes or in Cfront C++ mode), <code>_WCHAR_T</code> is defined in various standard headers to issue a typedef for <code>wchar_t</code>
<code>_WCHAR_T_KEYWORD</code>	In ANSI C++ modes with the option <code>KEYWORD-WCHAR=*YES</code> (default)

Predefined preprocessor assertions (`#assert`)

The options specified below are all part of the `MODIFY-SOURCE-PROPERTIES` statement:

<code>data_model(bit32)</code>	Always set
<code>cpu(7500)</code>	On generating /390 code
<code>machine(7500)</code>	On generating /390 code
<code>system(bs2000)</code>	Always set

9.3 Concept of a name adapter module in the C runtime system

One of the problems with regard to C library functions is the fact that these functions are addressed on the source program level with predefined names (e.g. `printf`, `fopen`), while BS2000 naming conventions require entry names beginning with the prefix “IC”.

Furthermore, C library functions also need to be replaceable by user-defined functions, where entry names can be constructed from the function names without the prefix “IC”.

Up until CRTE Version 1.0B, this problem was solved with the aid of a table that enabled the compiler to recognize the function names and convert them accordingly. Consequently, changes in the C runtime system were always associated with changes in the compiler. For reasons of compatibility, this technique has been retained for the existing C library functions in the C runtime system (all ANSI functions and approx. 50 BS2000-specific extensions). For all POSIX functions that were added by name as of CRTE V2.0A and all functions to be added in future, the problem is now solved on a compiler-independent basis by means of name adapter modules. These adapter modules contain the entry names derived from the function name minus the prefix “IC” and call the actual “standard” function with the entry name IC....

The following adapter modules are available for each C library function:

- An object module (OM) with the function name (possibly abbreviated to 8 characters) as an entry name in which lowercase letters have been converted to uppercase, and the underscore to a dollar character, e.g. `FPA$KON` for the function `fpathkonv`.

This is only relevant when linking object modules generated with the earlier C/C++ compilers (until V2.2). As of C/C++ V3.0 and higher, only modules in LLM format are generated.

- Up to four LLMs in which the unabbreviated function name is contained as the entry name, but once in lowercase or uppercase, and once with the underscore retained or converted (cf. the `LOWER-CASE-NAMES` and `SPECIAL-CHARACTERS` options in the `MODIFY-MODULE-PROPERTIES` statement).

For example, two LLMs exist for the `fpathkonv` function containing the entry names `FPA$KONV` and `fpathkonv`, respectively.

Adapter modules belong to the non-preloadable components of the C runtime system and must consequently be linked into the application program. They are contained both in the `SYSLNK.CRTE` library and in the `SYSLNK.CRTE.PARTIAL-BIND` library.

If a user-defined function is to be called instead of the standard library function, the corresponding user module must be linked with precedence before the `SYSLNK.CRTE` or `SYSLNK.CRTE.PARTIAL-BIND` library. The user module can be either an object module or an LLM (with or without conversion of the lowercase letters and underscores).

9.4 The II-UPDATE tool

When binding or recompiling ANSI C++ programs the system accesses existing ii information (ii=instantiation information). This information contains, among other things, the names of source files/libraries and source elements, of include libraries and lists or CIF files/libraries/elements. If these files, libraries and/or elements are renamed or located under a different ID (or another cat-id), then these changes **must** also be made in the ii files.

ii files, also known as ii elements, are created by the compiler from programs that contain templates.

II-UPDATE is a compiler-independent tool with an SDF interface. If a source or include library has been renamed, the tool is able to make these changes automatically in the ii file without the need to recompile. It is only possible to adjust the ii elements of a **single** library per call of the tool.

Entries in ii elements which contain the names of BS2000 or POSIX files or POSIX directories cannot be changed, but II-UPDATE can display these without error. This applies both for source, include, CIF and for listing entries.

START-II-UPDATE

```

CONTAINER = *LIBRARY-ELEMENT(...)

*LIBRARY-ELEMENT(...)
  |
  | LIBRARY = <filename 1..54>
  | ,ELEMENT = *ALL / <composed-name 1..64 with-under>(…)
  | <composed-name 1..64 with-under>(…)
  | | VERSION = *HIGHEST-EXISTING / <composed-name 1..24 with-under>
,OLD-NAME = *LIBRARY-ELEMENT(...) / *LIBRARY(...)

* LIBRARY-ELEMENT(...)
  |
  | LIBRARY = <filename 1..54>
  | ,ELEMENT = <composed-name 1..64 with-under>(…)
  | <composed-name 1..64 with-under>(…)
  | | VERSION = *DEFAULT / <composed-name 1..24 with-under>

*LIBRARY(...)
  | LIBRARY = <filename 1..54>
,NEW-NAME = *LIBRARY-ELEMENT(...) / *LIBRARY(...)

* LIBRARY-ELEMENT(...)
  |
  | LIBRARY = *SAME / <filename 1..54>
  | ,ELEMENT = *SAME / <composed-name 1..64 with-under>(…)
  | <composed-name 1..64 with-under>(…)
  | | VERSION = *UNCHANGED / <composed-name 1..24 with-under>

*LIBRARY(...)
  | LIBRARY = <filename 1..54>
,CONTEXT = *INCLUDE / *CIF / *SOURCE / *LISTING / *ALL
,ACTION = list-poss(2): *REPLACE / *SHOW

```

CONTAINER = *LIBRARY-ELEMENT(...)

This option is used to specify which ii elements in which object library are to be adapted.

LIBRARY = <filename 1..54>

<filename> is the name of a PLAM library.

ELEMENT = *ALL

All the ii elements in the PLAM library specified using LIBRARY= are to be adapted.

ELEMENT = <composed-name 1..64 with-under>(…)

<composed-name> identifies the fully qualified name of an ii element from the PLAM library specified using LIBRARY=.

VERSION = *HIGHEST-EXISTING

If the ii element information does not contain a version number, II-UPDATE uses the ii element with the highest version.

VERSION = <composed-name 1..24 with-under>

II-UPDATE uses the ii element with the specified version.

OLD-NAME = *LIBRARY-ELEMENT(…) / *LIBRARY

This option is used to specify the original library name or original library element names (for example, from the source library elements) that is to be modified in an ii file. The name must be specified in exactly the same way as in the ii file, i.e. if required with `cat=id` and `user=id` (see also example 6).

LIBRARY-ELEMENT (…)**LIBRARY = <filename 1..54>**

<filename> is the name of the PLAM library that has changed or in which a change has been made.

ELEMENT = <composed-name 1..64 with-under>(…)

<composed-name> identifies the fully qualified name of an ii element that has changed.

VERSION = *DEFAULT

Regardless of context, the version value of the ii element is accepted.

VERSION = <composed-name 1..24 with-under>

Identifies the version of the element that has changed.

LIBRARY = <filename 1..54>

<filename> is the name of a PLAM library that has changed.

NEW-NAME = *LIBRARY-ELEMENT(…) / *LIBRARY

This option is used to specify the new library or element name that is to be substituted for that specified in OLD-NAME.

LIBRARY-ELEMENT(…)**LIBRARY = *SAME / <filename 1..54>**

If *SAME is specified, the name of the PLAM library as entered in the OLD-NAME parameter is used.

<filename> is the new name of the PLAM library that is specified under OLD-NAME.

ELEMENT = *SAME / <composed-name 1..64 with-under>(…)

If *SAME is specified, the name of the element as entered in the OLD-NAME parameter is used.

<composed-name> is the changed, fully qualified name of an element that was specified under OLD-NAME.

VERSION = *UNCHANGED

The existing version of the library elements (in the ii element) should not be changed.

VERSION = <composed-name 1..24 with-under>

The new version name of the element.

LIBRARY = <filename 1..54>

<filename> is the new name of the PLAM library as specified under OLD-NAME.

Note

Only the same type of OLD-NAME / NEW-NAME combinations are supported, for example, OLD-NAME = *LIBRARY / NEW-NAME = *LIBRARY or OLD-NAME = *LIBRARY-ELEMENT / NEW-NAME = *LIBRARY-ELEMENT

CONTEXT = *INCLUDE / *CIF / *SOURCE / *LISTING / *ALL

If *INCLUDE is specified, the adaption of the library is only carried out if OLD-NAME and NEW-NAME are also library names.

If OLD-NAME and NEW-NAME are library element names, then adaptation would not take place for *INCLUDE.

CONTEXT = *ALL specifies the renaming of include libraries, of source libraries, of elements, lists or CIF outputs.

Note

No message is output indicating the total number or replacements carried out for the various contexts.

ACTION =

This option enables II-UPDATE to be instructed to replace or output all library or library element names of an ii element which can potentially be replaced.

ACTION = *REPLACE

Replacements are performed. Only the replacements which are performed are output.

ACTION = *SHOW

No replacements are performed. All possible replacements are output, however. The names of libraries and elements which would not be replaced are also output.

ACTION = (*SHOW, *REPLACE)

Replacements are performed. In addition to the replacements which were performed, the replacements which were not performed are also output.

Note

The specified context is taken into account in all cases. II-UPDATE remains compatible to older versions with regard to call, replacement and output behavior.

The following examples will provide further information about the points mentioned above.

Example 1

The include library *INC-LIB-V1* has been renamed in *INC-LIB-V2*. The following statement carries out the appropriate changes to all the ii elements in the *MY-OBJ-LIB* library.

```
/START-II-UPDATE CONTAINER=*LIB-ELEM(MY-OBJ-LIB, *ALL),-
/OLD-NAME=*LIBRARY( INC-LIB-V1 ),-
/NEW-NAME=*LIBRARY( INC-LIB-V2 ),-
/CONTEXT=*INCLUDE
```

Example 2

The source library *MY-SOURCE-LIB.V1* has been renamed in *MY-SOURCE-LIB.V2*. The following statement makes the appropriate changes to the library element *EL.O.II* in the *MY-OBJ-LIB* library:

```
/START-II-UPDATE CONTAINER=*LIB-ELEM(MY-OBJ-LIB, EL.O.II),-
/OLD-NAME=*LIBRARY(MY-SOURCE-LIB.V1),-
/NEW-NAME=*LIBRARY(MY-SOURCE-LIB.V2),-
/CONTEXT=*SOURCE
```

Example 3

The source libraries *MY-SOURCE-LIB1* and *MY-SOURCE-LIB2* contain a program system. The associated objects are located in the libraries *MY-OBJ-LIB1* and *MY-OBJ-LIB2*. The source libraries are to be renamed in *SOURCE-LIB1-V1* and *SOURCE-LIB2-V2*. The assumption has been made that each source library makes use of the other (include):

```
/MODIFY-FILE-ATTRIBUTES MY-SOURCE-LIB1, SOURCE-LIB1-V1
/MODIFY-FILE-ATTRIBUTES MY-SOURCE-LIB2, SOURCE-LIB2-V1
```

The ii files/elements are adapted using the following statements:

```
/START-II-UPDATE CONTAINER=*LIB-ELEM(MY-OBJ-LIB1, *ALL),-
/OLD-NAME=*LIBRARY(MY-SOURCE-LIB1).-
/NEW-NAME=*LIBRARY(SOURCE-LIB1-V1),-
/CONTEXT=*SOURCE
```

```

/START-II-UPDATE CONTAINER=*LIB-ELEM(MY-OBJ-LIB1, *ALL),-
/OLD-NAME=*LIBRARY(MY-SOURCE-LIB2),-
/NEW-NAME=*LIBRARY(SOURCE-LIB2-V1),-
/CONTEXT=*INCLUDE

/START-II-UPDATE CONTAINER=*LIB-ELEM(MY-OBJ-LIB2, *ALL),-
/OLD-NAME=*LIBRARY(MY-SOURCE-LIB1),-
/NEW-NAME=*LIBRARY(SOURCE-LIB1-V1),-
/CONTEXT=*INCLUDE

/START-II-UPDATE CONTAINER=*LIB-ELEM(MY-OBJ-LIB2, *ALL),-
/OLD-NAME=*LIBRARY(MY-SOURCE-LIB2),-
/NEW-NAME=*LIBRARY(SOURCE-LIB2-V1),-
/CONTEXT=*SOURCE

```

Example 4

Using the program system described in Example 3 but this time, instead of renaming the source libraries, we are going to move the source library element *MY-ELEM.C* from the *MY-SOURCE-LIB1* library to the *MY-SOURCE-LIB2* library and both the associated object *MY-ELEM.O* and the associated ii element *MY-ELEM.O.II* to the *MY-OBJ-LIB2* library:

```

/LMS
//O-L MY-SOURCE-LIB1,*U
//COP-EL (,MY-ELEM.C,S),(MY-SOURCE-LIB2,MY-ELEM.C)
//DEL-EL (,MY-ELEM.C,S)
//O-L MY-OBJ-LIB1,*U
//COP-EL (,MY-ELEM.O,L),(MY-OBJ-LIB2,MY-ELEM.O)
//COP-EL (,MY-ELEM.O.II,S),(MY-OBJ-LIB2,MY-ELEM.O.II)
//DEL-EL (,MY-ELEM.O,L)
//DEL-EL (,MY-ELEM.O.II,S)
//END

```

The ii files/elements are adapted using the following statements:

```

/START-II-UPDATE CONTAINER=*LIB-ELEM(MY-OBJ-LIB2, MY-ELEM.O.II),-
/OLD-NAME=*LIB-ELEM(MY-SOURCE-LIB1,MY-ELEM.C)
/NEW-NAME=*LIB-ELEM(MY-SOURCE-LIB2, MY-ELEM.C),-
/CONTEXT=*SOURCE

```

Example 5

The source libraries *MY-SOURCE-LIB1* and *MY-SOURCE-LIB2* contain a program system (see Example 3). The associated listing libraries *MY-LISTING-LIB1* and *MY-LISTING-LIB2* are to be renamed to *LISTING-LIB1-V1* or *LISTING-LIB2-V1*:

```
/MODIFY-FILE-ATTRIBUTES MY-LISTING-LIB1, LISTING-LIB1-V1
/MODIFY-FILE-ATTRIBUTES MY-LISTING-LIB2, LISTING-LIB2-V2
```

The ii files/elements are adapted using the following statements:

```
/START-II-UPDATE CONTAINER=*LIB-ELEM(MY-OBJ-LIB1,*ALL),-
/OLD-NAME=*LIBRARY(MY-LISTING-LIB1),-
/NEW-NAME=*LIBRARY(LISTING-LIB1-V1),-
/CONTEXT=*LISTING

/START-II-UPDATE CONTAINER=*LIB-ELEM(MY-OBJ-LIB2,*ALL),-
/OLD-NAME=*LIBRARY(MY-LISTING-LIB2),-
/NEW-NAME=*LIBRARY(LISTING-LIB2-V1),-
/CONTEXT=*LISTING
```

Example 6

The include library *HELLO-MAP.INCLIB* was renamed *HELLO-MAP.INCLIB.NEW*. The following string of statements is initially used to determine the exact notation of the library which is to be renamed (*ACTION=*SHOW*; in this case the values of the *old-name* and *new-name* parameters are of no significance); subsequently the replacements are performed using *ACTION>(*SHOW, *REPLACE)*:

```
/START-II-UPDATE CONTAINER=*LIB-ELEM(HELLO-MAP.OLIB1,*ALL),-
/OLD-NAME=*LIBRARY(OLIB),-
/NEW-NAME=*LIBRARY(NLIB),-
/CONTEXT=*ALL,-
/ACTION=*SHOW
```

Output:

```
% BLS0523 ELEMENT 'II-UPDATE', VERSION '032', TYPE 'L' FROM LIBRARY ': 20SG:
$TSOS.SYSLNK.CPP.032' IN PROCESS
% BLS0524 LLM 'II-UPDATE', VERSION '03.2D00' OF '2011-11-02 14:33:19' LOADED
% BLS0551 COPYRIGHT (C) Fujitsu Technology Solutions 2011. ALL RIGHTS RESERVED
% CDR9992 BEGIN II-UPDATE VERSION 03.2D00
% CDR9810 Processing ii-element *LIBRARY-ELEMENT(:20SC:$MTZ.HELLO-MAP.OLIB1,
HELLO-MAP-HP.II(*UPPER-LIMIT),S)
```

```

% CDR9819 include lib no replacement: *LIBRARY(:20SC:$MTZ.HELLO-MAP.INCLIB)
% CDR9819 include lib no replacement: *LIBRARY(:20SC:$MTZ.HELLO-MAP.INCLIB)
% CDR9819 include lib no replacement: *LIBRARY(:20SC:$TSOS.SYSLIB.CRTE)
% CDR9819 source lib no replacement: *LIBRARY(:20SC:$MTZ.HELLO-MAP.SLIB1)
% CDR9810 Processing ii-element *LIBRARY-ELEMENT(:20SC:$MTZ.HELLO-MAP.OLIB1,
HELLO-MAP-UP.II(*UPPER-LIMIT),S)
% CDR9819 include lib no replacement: *LIBRARY(:20SC:$MTZ.HELLO-MAP.INCLIB)
% CDR9819 include lib no replacement: *LIBRARY(:20SC:$MTZ.HELLO-MAP.INCLIB)
% CDR9819 include lib no replacement: *LIBRARY(:20SC:$TSOS.SYSLIB.CRTE)
% CDR9819 source lib no replacement: *LIBRARY(:20SC:$MTZ.HELLO-MAP.SLIB1)
% CDR9814 Summary: 0 include / 0 source / 0 cif / 0 listing replaced
% CCM0998 CPU TIME USED: 0.1485 SECONDS

```

```

/START-II-UPDATE CONTAINER=*LIB-ELEM(HELLO-MAP.OLIB1,*ALL),-
/OLD-NAME=*LIBRARY(:20SC:$MTZ.HELLO-MAP.INCLIB),-
/NEW-NAME=*LIBRARY(:20SC:$MTZ.HELLO-MAP.INCLIB.NEW),-
/CONTEXT=*ALL,-
/ACTION=(*REPLACE,*SHOW)

```

Output:

```

% BLS0523 ELEMENT 'II-UPDATE', VERSION '032', TYPE 'L' FROM LIBRARY
':20SG:$TSOS.SYSLNK.CPP.032' IN PROCESS
% BLS0524 LLM 'II-UPDATE', VERSION '03.2D00' OF '2011-11-02 14:33:19' LOADED
% BLS0551 COPYRIGHT (C) Fujitsu Technology Solutions 2011. ALL RIGHTS RESERVED
% CDR9992 BEGIN II-UPDATE VERSION 03.2D00
% CDR9810 Processing ii-element *LIBRARY-ELEMENT(:20SC:$MTZ.HELLO-MAP.OLIB1,
HELLO-MAP-HP.II(*UPPER-LIMIT),S)
% CDR9819 include lib no replacement: *LIBRARY(:20SC:$TSOS.SYSLIB.CRTE)
% CDR9811 include lib replaced: *LIBRARY(:20SC:$MTZ.HELLO-MAP.INCLIB) -->
*LIBRARY(:20SC:$MTZ.HELLO-MAP.INCLIB.NEW)
% CDR9819 source lib no replacement: *LIBRARY(:20SC:$MTZ.HELLO-MAP.SLIB1)
% CDR9810 Processing ii-element *LIBRARY-ELEMENT(:20SC:$MTZ.HELLO-MAP.OLIB1,
HELLO-MAP-UP.II(*UPPER-LIMIT),S)
% CDR9819 include lib no replacement: *LIBRARY(:20SC:$TSOS.SYSLIB.CRTE)
% CDR9811 include lib replaced: *LIBRARY(:20SC:$MTZ.HELLO-MAP.INCLIB) -->
*LIBRARY(:20SC:$MTZ.HELLO-MAP.INCLIB.NEW)
% CDR9819 source lib no replacement: *LIBRARY(:20SC:$MTZ.HELLO-MAP.SLIB1)
% CDR9814 Summary: 2 include / 0 source / 0 cif / 0 listing replaced
% CCM0998 CPU TIME USED: 0.2243 SECONDS

```


9.5 EBCDIC table (EDF041)

Decimal	Hexadecimal	EBCDIC	Meaning
0	00	0000 0000	\0 (null byte)
.	.	.	
.	.	.	
5	05	0000 0101	\t (tabulator character)
.	.	.	
.	.	.	
11	0B	0000 1011	\v (vertical tabulator)
12	0C	0000 1100	\f (page feed)
13	0D	0000 1101	\r (carriage return)
.	.	.	
.	.	.	
21	15	0001 0101	\n (newline character)
22	16	0001 0110	
.	.	.	
.	.	.	
64	40	0100 0000	␣ (space)
.	.	.	
.	.	.	
.	.	.	
74	4A	0100 1010	` (accent grave)
75	4B	0100 1011	.
76	4C	0100 1100	< (less than)
77	4D	0100 1101	((open parenthesis)
78	4E	0100 1110	+ (plus)
79	4F	0100 1111	(vertical)
80	50	0101 0000	& (ampersand)
.	.	.	
.	.	.	
.	.	.	
90	5A	0101 1010	! (exclamation mark)
91	5B	0101 1011	\$ (dollar sign)

Decimal	Hexadecimal	EBCDIC	Meaning
92	5C	0101 1100	* (asterisk)
93	5D	0101 1101) (close parenthesis)
94	5E	0101 1110	; (semicolon)
.	.	.	
96	60	0110 0000	- (minus)
97	61	0110 0001	/ (slash)
.	.	.	
.	.	.	
.	.	.	
106	6A	0110 1010	^ (exclusive OR)
107	6B	0110 1011	,
108	6C	0110 1100	% (percent)
109	6D	0110 1101	_ (underscore)
110	6E	0110 1110	> (greater than)
111	6F	0110 1111	? (question mark)
.	.	.	
.	.	.	
.	.	.	
122	7A	0111 1010	: (colon)
123	7B	0111 1011	# (number sign)
124	7C	0111 1100	@ (commercial at)
125	7D	0111 1101	' (apostrophe)
126	7E	0111 1110	= (equals sign)
127	7F	0111 1111	" (quote)
.	.	.	
.	.	.	
.	.	.	
129	81	1000 0001	a
130	82	1000 0010	b
131	83	1000 0011	c
132	84	1000 0100	d
133	85	1000 0101	e
134	86	1000 0110	f

Decimal	Hexadecimal	EBCDIC	Meaning
135	87	1000 0111	g
136	88	1000 1000	h
137	89	1000 1001	i
138	8A	1000 1010	
.	.	.	
.	.	.	
.	.	.	
145	91	1001 0001	j
146	92	1001 0010	k
147	93	1001 0011	l
148	94	1001 0100	m
149	95	1001 0101	n
150	96	1001 0110	o
151	97	1001 0111	p
152	98	1001 1000	q
153	99	1001 1001	r
.	.	.	
.	.	.	
.	.	.	
162	A2	1010 0010	s
163	A3	1010 0011	t
164	A4	1010 0100	u
165	A5	1010 0101	v
166	A6	1010 0110	w
167	A7	1010 0111	x
168	A8	1010 1000	y
169	A9	1010 1001	z
.	.	.	
.	.	.	
.	.	.	
187	BB	1011 1011	[(open square bracket)
188	BC	1011 1100	\ (backslash)
189	BD	1011 1101] (close square bracket)

Decimal	Hexadecimal	EBCDIC	Meaning
.	.	.	
.	.	.	
.	.	.	
193	C1	1100 0001	A
194	C2	1100 0010	B
195	C3	1100 0011	C
196	C4	1100 0100	D
197	C5	1100 0101	E
198	C6	1100 0110	F
199	C7	1100 0111	G
200	C8	1100 1000	H
201	C9	1100 1001	I
.	.	.	
.	.	.	
.	.	.	
209	D1	1101 0001	J
210	D2	1101 0010	K
211	D3	1101 0011	L
212	D4	1101 0100	M
213	D5	1101 0101	N
214	D6	1101 0110	O
215	D7	1101 0111	P
216	D8	1101 1000	Q
217	D9	1101 1001	R
.	.	.	
.	.	.	
.	.	.	
226	E2	1110 0010	S
227	E3	1110 0011	T
228	E4	1110 0100	U
229	E5	1110 0101	V
230	E6	1110 0110	W
231	E7	1110 0111	X

Decimal	Hexadecimal	EBCDIC	Meaning
232	E8	1110 1000	Y
233	E9	1110 1001	Z
.	.	.	
.	.	.	
.	.	.	
240	F0	1111 0000	0
241	F1	1111 0001	1
242	F2	1111 0010	2
243	F3	1111 0011	3
244	F4	1111 0100	4
245	F5	1111 0101	5
246	F6	1111 0110	6
247	F7	1111 0111	7
248	F8	1111 1000	8
249	F9	1111 1001	9
.	.	.	
251	FB	1111 1011	{ (open brace)
.	.	.	
253	FD	1111 1101	} (close brace)
.	.	.	
255	FF	1111 1111	~ (bit complement)

9.6 ASCII table (ISO 8859-1)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			SP	0	@	P	`	p			NBSP	°	À	Ð	à	ð
1			!	1	A	Q	a	q			ı	±	Á	Ñ	á	ñ
2			"	2	B	R	b	r			¢	²	Â	Ò	â	ò
3			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó
4			\$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô
5			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ
6			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö
7			'	7	G	W	g	w			§	•	Ç	x	ç	÷
8			(8	H	X	h	x			"	,	È	Ø	è	ø
9)	9	I	Y	i	y			©	¹	É	Ù	é	ù
A			*	:	J	Z	j	z			ª	º	Ë	Ú	ë	ú
B			+	;	K	[k	{			«	»	Ê	Û	ë	û
C			,	<	L	\	l				¬	¹ / ₄	Ì	Ü	ì	ü
D			-	=	M]	m	}			SHY	¹ / ₂	Í	Ý	í	ý
E			.	>	N	^	n	~			®	³ / ₄	Î	Þ	î	þ
F			/	?	O	-	o				¯	¿	Ï	ß	ï	ÿ

Related publications

The manuals are available as online manuals, see <http://manuals.ts.fujitsu.com>, or in printed form which must be paid and ordered separately at <http://manualshop.ts.fujitsu.com>.

- [1] **C/C++ V3.2D** (BS2000/OSD)
POSIX Commands of the C/C++ Compiler
User Guide
- [2] **C Library Functions** (BS2000/OSD)
Reference Manual
- [3] **C Library Functions for POSIX Applications** (BS2000/OSD)
Reference Manual
- [4] **CRTE** (BS2000/OSD)
Common RunTime Environment
User Guide
- [5] **C++** (BS2000)
C++ Library Functions
- [6] **Standard C++ Library V1.2**
User's Guide and Reference
- [7] **Tools.h++ V7.0**
User's Guide
- [8] **Tools.h++ V7.0**
Class Reference
- [9] **AID** (BS2000/OSD)
Debugging of C/C++ Programs
User Guide
- [10] **AID** (BS2000)
Advanced Interactive Debugger
Core Manual
User Guide

- [11] **BS2000/OSD-BC
Commands**
User Guide
- [12] **SDF (BS2000/OSD)
SDF Dialog Interface**
User Guide
- [13] **BLSSERV
Dynamic Binder Loader / Starter in BS2000/OSD**
User Guide
- [14] **BINDER
Binder in BS2000/OSD**
User Guide
- [15] **EDT V16.6B (BS2000/OSD)
Statements**
User Guide
- [16] **LMS (BS2000)
SDF Format**
User Guide
- [17] **BS2000/OSD-BC
Executive Macros**
User Guide
- [18] **BS2000/OSD-BC
Introductory Guide to DMS**
User Guide
- [19] **JV (BS2000/OSD)
Job Variables**
Reference Manual
- [20] **POSIX (BS2000/OSD)
POSIX Basics for Users and System Administrators**
User Guide
- [21] **POSIX (BS2000/OSD)
Commands**
User Guide

Other reference literature and standards

- [22] The C Programming Language
2nd Edition - ANSI-C
by Brian W. Kernighan und Dennis M. Ritchie
- [23] The C++ Programming Language
(Third Edition)
by Bjarne Stroustrup
- [24] „American National Standard for Information Systems - Programming Language C“,
Doc.No. X3J11/90-013, February 14, 1990 bzw.
„International Standard ISO/IEC 9899 : 1990, Programming languages - C“
- [25] „International Standard ISO/IEC 9899 : 1990, Programming languages - C /
Amendment 1 : 1994“
- [26] „Working Paper for Draft Proposed International Standard for Information Systems -
Programming Language C++“,
Doc.No. X3J16/96-0219R1, WG21/N0137, Dec 2 1996
- This document can be ordered from:
American National Standards Institute (ANSI), Standards Secretariat: ITIC,
1250 Eye Street NW, Suite 200, Washington DC 20005 (USA)
or from:
Normenausschuß Informationstechnik im DIN
Deutsches Institut für Normung e.V.
10772 Berlin
- [27] „International Standard ISO/IEC 14882 : 1998, Programming languages - C++“

Index

- [#assert directive](#) [221](#)
- [#define directive](#) [137](#)
- [#ident directive](#) [221](#)
- [#include directive](#) [100](#), [214](#)
- [#line directive](#) [221](#)
- [#pragma directive](#) [214](#), [223](#)
 - [ETPND area](#) [117](#)
 - [for template instantiation](#) [233](#)
 - [inline substitution of functions](#) [123](#)
 - [layout of listings](#) [111](#), [165](#)
- [#unassert directive](#) [222](#)
- [*UNCHANGED, explanation of operand value](#) [64](#)
- [__cplusplus](#) [136](#), [236](#), [302](#)
- [__DATE__](#) [214](#)
- [__STDC__](#) [135](#), [303](#)
- [__STDC_VERSION__](#) [135](#), [207](#), [303](#)
- [__TIME__](#) [214](#)
- [_OSD_POSIX](#) [35](#), [103](#)
- [_SNI_extensions, namespace](#) [282](#)
- [_STRICT_STDC](#) [135](#), [303](#)

- A**
- [ACTION option, BIND statement](#) [66](#)
- [adapter modules, C runtime system](#) [26](#), [304](#)
- [ADD-OPTION option](#) [69](#)
- [ADD-PRELINK-FILES option](#) [84](#)
- [address passing](#)
 - [of parameters](#) [190](#)
- [ADD-STATEMENT option](#) [86](#)
- [ADVANCED mode, DBL](#) [171](#)
- [AID](#) [184](#)
- [aligned pragma](#) [223](#)
- [alignment of data types](#) [215](#)
 - [aligned pragma](#) [223](#)
- [alphanumeric-name, SDF data type](#) [15](#)
- [ALTERNATIVE-TOKENS option](#) [141](#)
- [ANSI C language mode](#)
 - [extended](#) [135](#), [200](#)
 - [strict](#) [135](#), [200](#)
- [ANSI C++ language mode](#)
 - [extended](#) [136](#), [235](#)
 - [strict](#) [137](#), [235](#)
- [ANSI-VIOLATIONS option](#) [96](#)
- [argc](#)
 - [parameter for the main function](#) [183](#)
- [argv](#)
 - [parameter for the main function](#) [183](#)
- [arithmetic conversions](#) [206](#)
- [arrays](#)
 - [internal representation](#) [210](#)
- [ASCII notation](#) [142](#)
- [ASCII table](#) [318](#)
- [asm, inline substitution of Assembler code](#) [220](#), [240](#)
- [ASSERT option](#) [139](#)
- [assertions \(see preprocessor assertions\)](#) [303](#)
- [AT-ALLOWED option](#) [140](#)
- [auto, storage class](#) [216](#)
- [automatic instantiation](#) [246](#)

- B**
- [BEM, message class](#) [56](#)
- [BIND statement](#) [66](#)
- [BINDER](#) [169](#)
 - [control statements](#) [174](#)
 - [linking with](#) [174](#)
- [bitfields](#) [211](#), [220](#), [241](#)
- [bitwise operations](#) [213](#)

- BLSLIBnn
 - link name 171
- bool, C++ data type 239
- BUILTIN-FUNCTIONS option 124
- by reference
 - parameter passing 190
- by value
 - parameter passing 189
- by value, parameter passing 217

- C**
- C ++ functions
 - calling in C 194
- C functions
 - calling in C++ 193
- C language modes 200
- C language support of the compiler 199
 - extensions to ANSI/ISO C 219
 - implementation-defined behavior 208
 - overview of the C language modes 200
 - pragmas 223
- C runtime system
 - overview 25
 - POSIX library functions 35
- C++ language modes 235
- C++ language support of the compiler 235
 - constructors and destructors calls 241
 - extensions to ANSI/ISO C++ 260
 - implementation-defined behavior 239
 - overview of the C++ language modes 235
 - template instantiation 244
 - variations in the Cfront C++ mode 267
- C++ libraries 275
- C++ runtime system, overview 27
- C/C++ compiler
 - call 60
 - construction of default names 48
 - exit status 60
 - I/O in the POSIX file system 32
 - input sources and output destinations 45
 - messages 55
 - statements 61
- C/C++ development system
 - components 23
 - introductory examples 37
 - overview 21
- calling the DBL 172
- can_instantiate pragma 233
- can_rethrow() 284
- can_throw() 284
- CDR, message class 56
- cerr
 - redirecting 180
- CFE, message class 56
- Cfront
 - explained 9
- Cfront C++ language mode 137, 235, 267
- Cfront C++ library 277
- CHANGE-MSG-WEIGHT option 95
- char, C data type 209
- characters 209
- CHECK-SYNTAX statement 70
- CIF information
 - global listing generator 156
 - MODIFY-CIF-PROPERTIES 90
- cin
 - redirecting 180
- classes 241
- clog
 - redirecting 180
- C-NAMES option 118
- COMMENTS option 139
- compilation 45
 - introductory examples 37
- COMPILE statement 72
- compiler messages 55
- composed-name, SDF data type 15
- CONSTANTS option 118
- constructors 241
- CONSUMER option 90
- conversions 212
 - arithmetic 206
 - data types 212
- cout
 - redirecting 180

- CPU-LIMIT option 60
- cross-reference listing 108, 163
 - layout 295
- CRTE
 - components 25
 - POSIX support 35
- c-string, SDF data type 15
- CURRENT-LIBRARY option 102

- D**
- data types in C
 - internal representation 215
 - size and value ranges 215
- data types in SDF 15
- DBL 169
 - linking and loading with 171
- debugging with AID 184
 - requirements for symbolic debugging 186
- declarators, number 214
- default names
 - generated by compiler 48
- DEFINE option 137
- definition list 246, 248
- destructors 241
- digraph sequences 205
- division remainder, sign 213
- do_not_instantiate pragma 233
- DOLLAR-ALLOWED option 140
- dynamic binder loader (DBL) 169
- dynamic linking and loading 171

- E**
- EBCDIC table 313
- editing source programs 29
- EDT, file editor 29
- EENs 68, 119
- END statement
 - compiler 78
 - global listing generator 157
- END-OF-LINE-COMMENTS option 142
- enter options line 179
- enum 212
- ENUM-TYPE option 140

- envp, parameter for the main function 219
- error listing (see source/error listing) 107, 161
- error messages of the compiler 55
- ERROR, message weight 57
- ETPND area, creation 226
- ETPND-GENERATION option 117
- ETR files 252
- exception handling 243, 282
- executable program 169
- expansion of loops 122, 130
- external (global)
 - storage class 217
- external C 240
- external C++ 240
- external names
 - external C declarations 191, 196
 - in C 208
 - in C++ 239
- external symbols
 - masking 176
- EXTERNAL-DEFINITION option 141

- F**
- FATAL, message weight 57
- filename, SDF data type 15
- first instantiation
 - with definition list 248
 - without definition list 246
- FP-ARITHMETICS 119
- functions
 - linkage specifications 189

- G**
- GENERATE-ETR-FILE option 96
- GENERATE-LISTING statement, global listing generator 158

- H**
- header files
 - MODIFY-INCLUDE-LIBRARIES 100
 - storage 29

I

- ident pragma [232](#)
- identifier [208](#), [219](#)
- IEEE floating-point arithmetics [119](#)
- ii files [54](#), [305](#)
- II-UPDATE [305](#)
- ILCS (Inter-Language Communication Services) [189](#)
- ILCS interface [189](#)
 - special conventions for C/C++ [189](#)
- implementation-defined behavior
 - C language mode [208](#)
 - C++ language mode [239](#)
- implicit inclusion [259](#)
- IMPLICIT-INCLUDE option [139](#)
- INCLUDE option [81](#)
- INCLUDE-INFORMATION option [111](#), [165](#)
 - MODIFY-CIF-PROPERTIES [91](#)
- INITIAL-TITLE-TEXT option [112](#), [166](#)
- inline generation of functions [231](#)
- inline pragma [231](#)
- inline substitution of functions [122](#), [129](#)
- INLINING option [122](#)
- instantiate pragma [233](#)
- instantiation of templates [244](#)
- integer, SDF data type [17](#)
- interactive debugger AID [184](#)
- interfacing C and C++
 - calling C functions [193](#)
- ISO C mode (see ANSI C mode) [135](#)

J

- job variables [60](#)
 - status indicator [60](#)

K

- K&R C language mode [136](#), [200](#)
- key assignments, overview [31](#)
- keyword operators in C++ [237](#)
- KEYWORD-BOOL option [146](#)
- KEYWORD-WCHAR option [146](#)

L

- language interfacing
 - special conventions for C/C++ [189](#)
- LANGUAGE option [134](#)
- language scope of the compiler
 - C [199](#)
 - C++ [235](#)
- languages
 - linkage specifications [189](#)
- LAYOUT option [110](#), [164](#)
- LEVEL option [121](#)
- library
 - Cfront C++ [277](#)
 - standard C++ [275](#)
 - Tools.h++ [279](#)
- link name
 - BLSLIBnn [171](#)
- linkage
 - between C and C++ [191](#)
 - of templates [240](#)
 - to ILCS programs in other languages [196](#)
- linkage between C and C++ [191](#)
 - calling C++ functions [193](#), [194](#)
 - common types [192](#)
- LINKAGE option [116](#)
- linkage specification in C++ [240](#)
- linkage to functions (see linkage to languages) [189](#)
- linkage to languages [189](#)
- linking
 - general [169](#)
 - POSIX linkage option [36](#)
 - temporary [171](#)
 - with BINDER [174](#)
 - with DBL [171](#)
 - with the BIND statement [66](#)
- listing generator
 - description [289](#)
- listing generator, global [156](#)
- LISTING pragma [228](#)
- LISTING-PRAGMAS option [111](#), [165](#)

- listings
 - #pragma directive 228
 - global listing generator 156
- list-poss, SDF notational convention 14
- LLM
 - element name 52
 - generation with BINDER 169, 174
 - module and CSECT name 53
 - output to POSIX file 33
 - shareable 177
- LLM formats 1 to 4 68
- LOAD-EXECUTABLE-PROGRAM 172
- LOAD-EXECUTABLE-PROGRAM command
 - calling DBL 172
- loading
 - general 169
- long long, C data type 219
- LONGLONG option 142
- loop expansion 122, 130
- LOOP-INIT option 146
- LOOP-UNROLLING option 122
- LOWER-CASE-NAMES option 117
- M**
- macro arguments, empty 220
- main function 209, 219
 - calling from other languages 196
 - definition with parameters 183
 - linkage in C++ 239
 - parameter input 181
- mandatory-quotes, suffix for SDF data types 19
- map listing 108, 162
 - layout 292
- masking symbols 176
- MAX-ERROR-NUMBER option 95
- MAX-INstantiate-ITER option 86
- message weight 56
- messages of the compiler 55
- MINIMAL-MSG-WEIGHT option
 - MODIFY-DIAGNOSTIC-PROPERTIES 95
- MODE option 135
- MODIFY-BIND-PROPERTIES statement 79
- MODIFY-CIF-PROPERTIES statement 90
- MODIFY-DIAGNOSTIC-PROPERTIES
 - statement 94
- MODIFY-INCLUDE-LIBRARIES statement 100
- MODIFY-LISTING-PROPERTIES statement
 - compiler 105
 - global listing generator 160
- MODIFY-MODULE-PROPERTIES
 - statement 115
- MODIFY-MSG-ATTRIBUTES command 58
- MODIFY-OPTIMIZATION-PROPERTIES
 - statement 121
- MODIFY-RUNTIME-PROPERTIES
 - statement 131
- MODIFY-SOURCE-PROPERTIES
 - statement 133
- MODIFY-SYMBOL-VISIBILITY
 - BINDER statement 176
- MODIFY-TEST-PROPERTIES statement 147
- module names 52
- module properties, MODIFY-MODULE-PROPERTIES 115
- MODULE-OUTPUT option 73
- MONJV option 60, 156
- multibyte character constant 209
- multibyte characters 210
- multiple definitions of external variables 220
- N**
- name mangling in C++ 238
- name, SDF data type 17
- new array 240
- notational conventions
 - general 12
 - SDF 13
- NOTE, message weight 56
- O**
- object code listing 109, 164
- object layout in C++ 238
- object listing
 - layout 299
- object listing (see object code listing) 109, 164
- object program
 - temporary 171

- optimization
 - MODIFY-OPTIMIZATION-PROPERTIES 121
 - process 126
 - output of listings
 - compiler 105
 - global listing generator 156
 - output of messages
 - MODIFY-DIAGNOSTIC-PROPERTIES 94
 - OUTPUT option
 - BIND 67
 - MODIFY-CIF-PROPERTIES 91
 - MODIFY-DIAGNOSTIC-PROPERTIES 97
 - MODIFY-LISTING-PROPERTIES 112, 166
 - PREPROCESS 149
 - OUTPUT-FORMAT option, BIND statement 68
- P**
- pack-Pragma 225
 - PAGE pragma 230
 - parameter input at program start
 - for the main function 181
 - redirecting standard I/O files 180
 - parameter list, internal structure 217
 - parameter passing
 - by reference 190
 - by value 189
 - specific to C/C++ 189
 - parameter, internal structure of the parameter list 217
 - PARAMETER-PROMPTING option 131, 179
 - patch area 226
 - PLAIN-FIELDS option 140
 - pointer 210
 - POSIX
 - C library functions 35
 - I/O in POSIX files 32
 - posix-filename
 - SDF data type 17
 - posix-pathname 34
 - SDF data type 17
 - pragmas 223
 - predefined preprocessor macros 302
 - PREINCLUDE option 139
 - prelinker, automatic template instantiation 144
 - PREPROCESS statement 148
 - preprocessor assertions, predefined 303
 - preprocessor directives 214, 220
 - preprocessor listing 107, 162
 - preprocessor macros, predefined 302
 - PRESERVING option 140
 - program execution 179
 - program interfacing (see linkage specifications) 189
 - PROGRAM-INTERRUPT option 132
 - project listing 109, 164
 - prototyping 206
 - ptrdiff_t 212
 - PUBLIC-SLICING option 115
- R**
- reference types in C++ 241
 - register, storage class 216
 - reinterpret_cast 240
 - repository
 - temporary 248
 - reserved keywords
 - in C 205
 - in C++ 237
 - RESET-TO-DEFAULT statement 153
 - RESOLVE option 82
 - return code display
 - contents 60
 - return values of functions
 - specific to C/C++ 190
 - right shift 213
 - RUNTIME-LANGUAGE option 87
- S**
- SDF
 - data types 15
 - notational conventions 13
 - suffixes for data types 18
 - shareability 115
 - C/C++ programs 177
 - COMPILER-ACTION option 115
 - SHAREABLE-CODE option 115
 - SHAREABLE-CODE option 115

- SHOW-COLUMN option 96
- SHOW-DEFAULTS statement 154
- SHOW-INCLUDES option 96
- SHOW-PROPERTIES statement 155
- sign of division remainder 213
- sign propagation 212
- SIGNED-CHARACTER option 140
- SIGNED-FIELDS option 140
- SIS, message class 56
- size_t 212
- source listing
 - layout 290
- SOURCE option
 - CHECK-SYNTAX 70
 - COMPILE 72
 - PREPROCESS 148
- source program
 - editing 29
 - input via SYSDTA 76
 - preparing 29
 - saving to POSIX file 32
 - storage 29
- source/error listing 107, 161
- SPACE pragma 231
- special characters 219
- SPECIAL-CHARACTERS option 118
- SPECIALIZATION option 146
- spin-off mechanism, SDF 64
- STACK-SIZE option 131
- standard C++ library 275
- standard I/O files 180
 - default assignment 180
 - redirection 180
- standard optimizations 126
- START-CPLUS-COMPILER 60
- START-CPLUS-LISTING-GENERATOR 156
- START-EXECUTABLE-PROGRAM 172
- START-EXECUTABLE-PROGRAM command
 - calling DBL 172
 - starting a C program 179
- starting
 - general 169
- START-LLM-CREATION option 80
- static (global)
 - storage class 217
- static (local)
 - storage class 217
- static variable, storage class 217
- STATISTIC-MESSAGES option 131
- statistics 164
 - in summary listing 109
- status indicator
 - in job variables 60
- stderr
 - redirecting 180
- stdin
 - redirecting 180
- STD-INCLUDE-LIBRARY option 101
- STDLIB option 87
- stdout
 - redirecting 180
- storage classes 216
- string literals 219
- STRING-LITERALS option 118
- Stroustrup, Bjarne 235
- structure, C data type 206
- structures 210
- STXIT event handling
 - for linkage between languages 197
- SUBROUTINE-CALL option 117
- subroutines
 - specifications (see linkage to languages) 189
- summary listing 109, 164
- SUPPRESS-MSG option
 - MODIFY-DIAGNOSTIC-PROPERTIES 95
- switch statement 214
- symbolic debugging
 - with AID 184
- symbolic debugging with AID 184
- SYSDTA, input of source program 76
- SYSLIB.CRTE 25
- SYSLIB.CRTE.CPP 25
- SYSLNK.CRTE 26
- SYSLNK.CRTE.CFCPP 27
- SYSLNK.CRTE.COMPL 27
- SYSLNK.CRTE.CPP 27
- SYSLNK.CRTE.CPP-COMPL 28

SYSLNK.CRTE.PARTIAL-BIND [27](#)
SYSLNK.CRTE.POSIX [28, 36](#)
SYSLNK.CRTE.RTSCPP [28](#)
SYSLNK.CRTE.STDCPP [28](#)
SYSLNK.CRTE.TOOLS [28](#)

T

TEMPLATE-DEF-LIST option [86](#)
templates
 C++ linkage [240](#)
 instantiation of [244](#)
TEST-SUPPORT option [88, 147](#)
TITLE pragma [230](#)
Tools.h++ library [279](#)

U

UMP, message class [56](#)
UNCHANGED, explanation of operand value [64](#)
UNDEFINE option [138](#)
union, C data type [206](#)
USER-INCLUDE-LIBRARY option [100](#)
USE-STD-NAMESPACE option [145](#)

V

VERBOSE option [96](#)
VIRTUAL_FUNCTION_TAB pragma [232](#)
visibility
 of external symbols [176](#)
void, C data type [205](#)
volatile, type qualifier [212](#)

W

WARNING, message weight [56](#)
wchar_t [210](#)
weak pragma [232](#)
with, suffix for SDF data types [18](#)
without, suffix for SDF data types [18](#)
WORKSPACE option [116](#)

X

XREF listing (see cross-reference listing) [108, 163](#)