English

Fujitsu Software BS2000

# POSIX commands of the C/C++-Compilers

User Guide

Valid for:
C/C++ V4.0B05

Edition November 2024

## 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: bs2000.info@fujitsu.com.

## Certified documentation according to DIN EN ISO 9001:2015

To ensure a consistently high quality standard and user-friendliness, this documentation was created to meet the regulations of a quality management system which complies with the requirements of the standard DIN EN ISO 9001:2015.

## Copyright and Trademarks

# Table of Contents

# C/C++ POSIX Commands

# 1 Preface

This chapter provides information concerning the following topics:

- Brief product description
- Summary of contents
- Changes since the previous manual
- Notational conventions

## 1.1 Brief product description

The BS2000 C/C++ Compiler can be called from and controlled with options from the BS2000 (SDF) or POSIX (POSIX shell) environment.

This manual describes controlling the compiler from the POSIX environment, where the following POSIX commands are available:

| | |
|---|---|
| `cc`, `c11`, `c89` | Calls the compiler as a C compiler |
| `CC` | Calls the compiler as a C++ compiler |
| `cclistgen` | Calls the global listing generator |

The options and operands of the above calling commands cover most of the services and functions available for controlling the compiler via the SDF interface (see the “C/C++ User Guide” [4]). The syntax of the POSIX commands is based on the definition in the XPG4 Standard or on the normal UNIX shell commands.

The `cc`, `c11`, `c89` and `CC` calling commands also include linking the compiled objects together to form an executable program.

The software products CRTE and POSIX-HEADER are required for creating and running C and C++ programs in the POSIX environment. CRTE also contains the standard header files and modules of the C and C++ library functions. The headers of CRTE and also the POSIX headers are required for using the POSIX library functions.

## 1.2 Summary of contents

This manual describes how C and C++ programs are compiled, linked and executed with the C/C++ compiler and additional development tools in the POSIX environment.

Chapter "Basics" summarizes the C/C++ program development in the POSIX environment.

The compiler is called with the `cc`, `c11`, `c89` and `CC` commands, which are described in detail with their options and the effects of these in chapter "The cc, c11, c89 and CC commands".

Chapter "Global listing generator (cclistgen)" describes the `cclistgen` command, which is used to call and control the global listing generator.

All compiler options are listed alphabetically in the appendix together with page references.

In order to work effectively with this manual, you will need to be familiar with the C and C++ programming languages and the POSIX shell.

This manual is primarily intended for use as a reference manual for the POSIX commands of the C/C++ compiler.

Detailed information on the services and functions of the C/C++ compiler beyond the POSIX control can be found in the following manual:

“C/C++ BS2000/OSD, C/C++ Compiler”, User Guide [4].

In addition to SDF control of the C/C++ compiler, the above manual contains further information on topics not dealt with in this manual. These topics include:

- process and effects of optimization
- structure of the compiler listings and messages
- compiler C language support (a summary of the C language modes, implementationdependent behavior, `#pragma` directives, extensions to the ANSI/ISO C standard)
- compiler C++ language support (a summary of the C++ language modes, implementation-dependent behavior, extensions to the ANSI/ISO C++ standard)
- links between functions and language
- brief description of the C++ libraries supplied with CRTE

## 1.3 Changes since the previous manual

### Compared to C/C++ version V4.0B03 some significant topics have changed.

The `switch` statement accepts an expression of type `long long`.

The content of the ETR file has changed. The new content can not be used with a compiler V4.0B03 or older.

### Compared to C/C++ version V4.0B00 some significant topics have changed.

The library for the language modes C++2017 and C++ 2020 is now available in two versions. Version 1 is the default and compatible with the previous library. Version 2 implements some features of the standard C++ 2020. See the C/C++ user guide [4] for more details.

### Compared to C/C++ version V4.0A30 some significant topics have changed.

The compiler now supports the language features of the C++ 2020 standard. See the C/C++ user guide [4] for some restrictions. Full library support is currently not available and instead the existing C++ 2017 library is used.

There is a new option for this language mode. The default language mode for the command `CC` is now C++ 2020. Up until version 4.0A30 it was C++ 2017.

The arguments of a `printf` call will be checked against the specifiers in the format string (see __printf_args pragma and __scanf_args pragma in the C/C++ user guide [4]).

### Compared to C/C++ version V3.2D some significant topics have changed.

The changes in this manual compared to the C/C++ V3.2D User Guide mainly affect the new language modes C11 and C++ 2017 and the changes to compiler options due to them.

The most important point is the default setting of the compiler. If the language mode is not specified explicitly, it always uses the most modern implemented language standard. This was and is the case with version 3. For Version 3 the (then) most modern standard was C89. Now the compiler supports C11 and this is also the default. This is similar for C++. The version 3 had as (at that time) most modern standard a preliminary version of C++ 98, the current compiler supports C++ 17.

The new compiler now supports 10 language modes, 5 for C and 5 for C++. For better handling of this variety, the syntax for specifying the language mode has been redesigned. The options used in Version 3 will continue to be recognized and mapped to new options. The mapping is:

| | |
|---|---|
| `-X a` | `-X cc -X 1990 -X nostrict` |
| `-X c` | `-X cc -X 1990 -X strict` |
| `-X t` | `-X cc -X kr` |
| `-X w` | `-X CC -X V3 -X nostrict` |
| `-X e` | `-X CC -X V3 -X strict` |
| `-X d` | `-X CC -X V2` |

With the support of the new language modes, the recognition of questionable source constructs has also been revised. In some situations, messages are now different than C/C++ V3.2D. The error weight, the error number and the text may have changed. There are a few situations where either C/C++ V3.2 gives a message or C/C++ V4.0 but not both.

## 1.4 Notational conventions

The following notational conventions are used to depict commands, options and program directives described in this User Guide:

| | |
|---|---|
| *STD | Uppercase letters, digits and special characters which do not belong to the metalanguage characters designate keyword or constants which must be entered exactly as shown. |
| `-R msg_id` | Uppercase and lowercase letters, digits and special characters in `typewritten text` are constants and must be entered exactly as shown, except for the `-K` option arguments, which are shown in the manual in lowercase letters, but may be entered in both uppercase and lowercase (see "Calling syntax and general rules (C/C++ POSIX Commands, #26)"). |
| *name* | Lowercase letters in *italics* denote variables, which must be replaced by current values at the time of input. |
| {`cc` \| `c89`} | Braces enclose alternatives from which one must be selected. The separator character \| must not be specified. |
| [ ] | Square brackets enclose options that may be omitted. |
| ( ) | Parentheses are constants and must be specified. |
| 'BLANK' | This symbol indicates that at least one white space character is necessary for the syntax. |
| ... | Ellipses signify repetition, i.e. the preceding unit can be repeated several times in succession. |

# 2 Basics

This chapter provides information concerning the following topics:

## 2.1 Delivery structure and software environment

The files required for controlling the BS2000 C/C++ compiler from the POSIX shell are stored as follows in the POSIX file system:

This chapter provides information concerning the following topics:

| | |
|---|---|
| `/opt/C/bin/c89`<br>`/opt/C/bin/cclistgen` | Links to the compiler and listing generator installed in BS2000 (PLAM library) |
| `/opt/C/bin/cc`<br>`/opt/C/bin/c11`<br>`/opt/C/bin/CC` | Links to `/opt/C/bin/c89` |
| `/usr/bin/cc` | Link to `/opt/C/bin/cc` |
| `/usr/bin/c11` | Link to `/opt/C/bin/c11` |
| `/usr/bin/c89` | Link to `/opt/C/bin/c89` |
| `/usr/bin/CC` | Link to `/opt/C/bin/CC` |
| `/usr/bin/cclistgen` | Link to `/opt/C/bin/cclistgen` |

Installation of the above POSIX files is described in the Release Notice for C/C++ (BS2000/OSD) V4.0.

C/C++ uses the C and C++ library function header (or include) files and modules supplied with CRTE and the header files supplied with POSIX-HEADER for all POSIX library functions. The libraries for programs `lex` and `yacc` are part of the software product POSIX-SH.

The C and C++ library function modules are installed in BS2000 as PLAM libraries and not in the POSIX file system. When linking with the `cc`/`c11`/`c89`/`CC` commands, the link options are issued to the relevant PLAM libraries as RESOLVE directives (of the LINK EDITOR).
See also the link option `-l` *x* ("Link editor options").

The header files for the C and C++ library functions are stored as POSIX files in the standard `/usr/include`, `/usr/include/sys`, `/usr/include/CXX01`, `/usr/include/CXX02` and `/usr/include/CC` directories.
Installation of these header files is described in the CRTE Release Notice or in the manual “POSIX Basics” [1].

## 2.2 From source code to program execution

This section provides you with an overview of the following program creation stages in the POSIX subsystem:

- Providing the source code and header files
- Compiling
- Linking
  - Linking user modules
  - Linking the CRTE runtime libraries
- Debugging
- Using the POSIX library functions

### 2.2.1 Providing the source code and header files

The source code and header files may be provided in EBCDIC or ASCII code. The default is EBCDIC in the POSIX file system and ASCII in the file systems of remote UNIX hosts. All files in a file system (POSIX file system or merged in, remote file system) must be available in the same codeset. The compiler does not query the codeset of individual files, it only queries the codeset of a file system. The files of ASCII file systems are converted automatically to EBCDIC, as long as the POSIX variable IO_CONVERSION is set to YES.

The the source code file names must contain one of the following standard suffixes:

`c`, `C` C source code (`cc`, `c11`, `c89`) or C++ source code (`CC`) before the preprocessor run

`cpp`, `CPP`, `cxx`, `CXX`, `cc`, `CC`, `c++`, `C++`

C++ source code before the preprocessor run (`CC`)

`i` C source code (`cc`, `c11`, `c89`) after the preprocessor run

`I` C++ source code after the preprocessor run (`CC`)

In addition to the above suffixes, the `-Y F` option (see "General options") can be used to define additional input file suffixes which are then also accepted by the compiler.

Source code and include members stored in BS2000 files or PLAM libraries cannot be processed with the compiler in the POSIX file system.

The POSIX `bs2cp` command is provided for transferring BS2000 files and PLAM library members into the POSIX file system and vice versa. The POSIX `edt` command is provided for editing POSIX files in the POSIX shell. If the POSIX shell is accessed via `rlogin`, the `vi` editor may also be used. See the manual “POSIX Commands” [3].

The standard header files for the C and C++ library functions available with CRTE are in the standard directories `/usr/include`, `/usr/include/sys`, `/usr/include/CC`, `/usr/include/CXX01 and /usr/include/CXX02`. These directories are searched automatically by the compiler (or preprocessor).

### 2.2.2 Compiling

C sources are compiled with the `cc`, `c11` and `c89` commands and C++ sources with the `CC` command.

These commands are described in detail in chapter "The cc, c11, c89 and CC commands".

#### C and C++ language modes

The C and C++ sources can be compiled in various language modes via the following options:

C language modes (`cc`/`c11`/`c89` commands):

- extended c11 mode (`-X 2011 -X nostrict`), default for `cc` and `c11`
- strict c11 mode (`-X 2011 -X strict`)
- extended c89 mode (`-X 1990 -X nostrict`), default for `c89`
- strict c89 mode (`-X 1990 -X strict`)
- Kernighan&Ritchie C (`-X kr`)

C++ language modes (`CC` command):

- extended C++ 2020 mode (`-X 2020 -X nostrict`), default
- strict C++ 2020 mode (`-X 2020 -X strict`)
- extended C++ 2017 mode (`-X 2017 -X nostrict`)
- strict C++ 2017 mode (`-X 2017 -X strict`)
- extended C++ V3 mode (`-X v3-compatible -X nostrict`)
- strict C++ V3 mode (`-X v3-compatible -X strict`)
- Cfront C++ mode (Cfront-V3.0.3-compatible C++) (`-X v2-compatible`)

See "Options for selecting the language mode" for the language mode options.

#### Creating an object file (“.o” file)

If the compilation run is not terminated after the preprocessor phase (see the `-E` and `-P` options on "Options for selecting compilation phases"), the compiler creates an LLM for each compiled source file and stores it by default in a POSIX object file named *basename*`.o` in the current directory. *basename* is the name of the source file without the directory part or the standard suffixes (`.c`, `.C` etc.).

A different directory and/or file name may be defined for the object file with the `-o` option (see "General options").

By default, a link run is started after compilation. If only one source file is compiled and linked in one step, the object file is stored temporarily and then deleted. If at least two source files or one source and one object file (`.o` file) are specified, the object files are not deleted.

Linking can be prevented by specifying the `-c` option (see "Options for selecting compilation phases (C/C++ POSIX Commands, #29)").

### Creating an expanded, recompilable source program (“.i” file)

If the `-P` option is specified, only the preprocessor run is executed and one expanded, recompilable source program is generated for each compiled source file. The result is written by default into a POSIX source file named *basename*`.i` (`cc`, `c11`, `c89`) or *basename*`.I` (`CC`) in the current directory.

The `-o` option can be used to specify a different destination directory and/or file name for the expanded source program (see "General options ").

### Creating compilation listings

The `-N` *listing* option can be used to request various compiler listings (e.g. source/error listing, cross-reference listing, etc.). The compiler either writes the requested listings separately for each compiled source file into a list file named *basename*`.lst` or collectively for all compiled source files into a list file *file* specified with the `-N output` option (see "Options for outputting listings and CIF information)").

You can also create CIFs (Compilation Information Files) for the output of compilation listings, which are subsequently processed with the global listing generator `cclistgen`. See the `-N cif` option ("Options for outputting listings and CIF information") and chapter “Global listing generator (cclistgen)”.

List files can be printed out with the POSIX `bs2lp` command (see the manual “POSIX Commands” [3]).

### Output destinations and codeset

The compiler saves the output files by default in the current directory, i.e. in the directory from which the compiler run was started.

The `-o` option (see "General options") can be used to specify a different directory and/or file name as the output destination. This can be a directory in the local POSIX file system or in a merged in file system on a remote host. However, it must be noted that it is only meaningful to further process text files on UNIX hosts or PCs, i.e. only expanded source programs (“.i” files) and list files (“.lst” files).

The codeset of the destination file system determines the output codeset used for the files (ASCII or EBCDIC) itf the environment variable IO-CONVERSION has the value YES (see sections "Environment variables" und "Support for file systems in ASCII" in manual "C Library Functions for POSIX Applications“ [2] ).

How characters and strings are stored is controlled by the `-K literal_encoding_`... option (see "Common frontend options in C and C++").

### 2.2.3 Linking

A C or C++ program can only be linked in the POSIX shell to form an executable program with the `cc`, `c11`, `c89` and `CC` calling commands. A “standalone” link editor, normally found in UNIX systems, does not exist. From the technical viewpoint, linking in the POSIX shell calls the BS2000 link editor and supplies it with the appropriate directives (INCLUDE-
MODULES, RESOLVE-BY-AUTOLINK etc.).

A link run is started if none of the `-c`, `-E`, `-M`, `-P` or `-y` options are specified (see "Options for selecting compilation phases"), as long as no errors occurred during a prior compilation. By default, the linked program is written as an LLM into an executable POSIX file with the standard name `a.out`, in the current directory. The `-o` option can be used to specify a different directory and/or file name (see "General options").

No link listings can be generated when linking in the POSIX shell. If errors occur, appropriate error messages are output to `stderr`.

The `-N binder` option can be used to generate the standard listings of BINDER (see "Options for outputting listings and CIF information").

#### 2.2.3.1 Linking user modules

User modules can only be linked in statically and not dynamically (i.e. at runtime). Programs containing “unresolved externals” to user modules cannot be loaded in the POSIX shell.

The following can be input sources to the link editor:

- Object files generated by the compiler (“.o” files)
- Libraries created with the `ar` utility (“.a” files)
- LLMs that were copied from PLAM libraries into POSIX object files with the POSIX `bs2cp` command (see “Introductory examples”). These may be LLMs that were generated directly by a compiler in the BS2000 environment (SDF) or object modules which were written into an LLM with the link editor.
- LLMs and object modules in BS2000 PLAM libraries. The PLAM libraries must be assigned with the BLSLIB*nn* environment variable (see the `-l BLSLIB` option on "Link editor options").

The modules from the PLAM libraries may be modules generated by any BS2000 compiler with ILCS capabilities (e. g. C/C++, COBOL85, COBOL2000, ASSEMBH).
You must observe language-specific requirements with the above (parameter passing, required runtime systems, etc.).

Internal INCLUDE-MODULES directives are issued during linking with POSIX object files and RESOLVE-BY-AUTOLINK directives are issued for `ar` libraries and PLAM libraries.

#### 2.2.3.2 Linking the CRTE runtime libraries

The link editor resolves the unresolved external references to the C and C++ runtime systems via autolink (RESOLVE-BY-AUTOLINK) from the CRTE PLAM libraries.

### C runtime system

When code is generated, the C runtime system modules can be linked or loaded with the `cc`, `c11`, `c89` and `CC` commands as follows:

1. Loading the C runtime system dynamically (partial bind). There are two variants of the partial bind linkage method:
   - Standard partial bind (`-d y`)

     Linking is carried out by default from the SYSLNK.CRTE.PARTIAL-BIND library if no special linker options are specified. This library contains link modules that resolve all unresolved external references to the C and COBOL runtime systems. Only the connection modules required are linked. If a module loaded by the application to be linked requires entries of the runtime system, this can result in unresolved external references because the link modules to the runtime system's entries need not necessarily already be linked. In this case the complete partial bind method should be used for linking (see also CRTE-BHB).

     The C and COBOL runtime systems themselves are loaded dynamically at runtime, either from class 4 memory, if it has been preloaded, or from the SYSLNK.CRTE library.

     The linked program requires considerably less disk storage space than if the C runtime system is linked statically from the SYSLNK.CRTE library (see 2.). The load time is also faster. The appropriate CRTE must be available when the program is called.
   - Complete partial bind (`-d compl`)

     In this case, linking is done from the SYSLNK.CRTE.COMPL library. Basically, the procedure for the complete partial bind is the same as that for the standard partial bind method. With complete partial bind, the link modules provided in
     SYSLNK.CRTE.COMPL contain all the entries and the external data of the complete C and COBOL runtime systems. This means that the unresolved external references which may occur when modules of an application which was linked in a standard partial bind are loaded cannot occur in complete partial bind. When you use shared libraries in POSIX, successful linking is only guaranteed with a complete partial bind.

   For more information on the partial bind linking method, see the manual "CRTE" [5)].
2. Linking the complete C runtime system statically (

   `-d n`)

   If the `-d n` link option is specified (see "Link editor options"), all required C runtime system modules are linked in from the SYSLNK.CRTE library.
3. Leaving the external references to the C runtime system unresolved (`-z nodefs`)

   If the `-z nodefs` link option is specified (see "Link editor options)"), the program is linked without a RESOLVE to the C runtime library. The unresolved external references are then resolved at runtime from the C runtime system preloaded in class 4 memory. `-z nodefs` is not supported when linking C++ programs (`CC` command).

### C++ library for the Cfront C++ language mode C++ V2

The modules of the Cfront C++ library (SYSLNK.CRTE.CPP) and of the Cfront C++ runtime system (SYSLNK. CRTE.CFCPP) can only be linked in statically. If the Cfront C++ mode(`-X v2-compatible` option) is specified in the `CC` command, these libraries are linked in automatically in addition to the C runtime system.

See also the `-l` link option in section "Link editor options".

### C++ library for the Cfront C++ language mode C++ V3

The modules of the standard C++ V3 library (SYSLNK.CRTE.STDCPP) and of the C++ runtime system (SYSLNK.CRTE.RTSCPP) can only be linked in statically. These libraries are linked in automatically in addition to the C runtime system if the C++ V3 mode is specified in the `CC` command ( `-X v3-compatible`).

See also the `-l` link option in section "Link editor options".

### C++ V3 library Tools.h+

The modules of the Tools.h++ library (SYSLNK.CRTE.TOOLS) can only be linked in statically. The library is only available in the C++ V3 mode (`-X v3-compatible` option) and is only linked in if the `-l RWtools` link option is also specified.

See also the `-l` link option in section "Link editor options".

### modern C++ library for the C++ language modes C++ 2017 and C++ 2020

The modules of the modern C++ library can only be linked in statically. This library is linked in automatically in addition to the C runtime system if the C++ 2017 mode (option `-X 2017`) or C++ 2020 mode (default or option `-X 2020`) is specified in the `CC` command. The file being used depends on the library version: for library version 1 it is SYSLNK.CRTE.CXX01, for library version 2 it is SYSLNK.CRTE.CXX02.

See also the `-l` link option in section "Link editor options".

### POSIX link switch

The link switches `posix.o` and `postime.o` available with CRTE (correspond to the CRTE SYSLNK.CRTE.POSIX library in the BS2000 environment) are linked in automatically. The time, signal handling and `clock` functions, which are duplicated in the C runtime system, are therefore generally executed with POSIX semantics. Mixed processing of POSIX and BS2000 is generally possible. Please refer to the manual “C Library Functions for POSIX Applications” [2 ] for further details.

### 2.2.4 Debugging

Linked programs can be debugged with the dialog debugger AID, provided the required debugging information (LSD) has been generated by the compiler by specifying the `-g` option (see "Debug option").

*Note*

When the `-g` option is used, the objects created may be much larger under some circumstances due to the LSD information!

The AID debugger is activated with the POSIX `debug` *programname* command. When this command is input, the BS2000 environment becomes the current environment. This is indicated by the %xxxxyyyy/ prompt, where xxxxyyyy stands for the PID of the process started using `debug`. The debugging commands as described in the manual “AID Debugging of C/C++ Programs” [11] can then be input. Once the program is terminated, the POSIX shell then becomes the current environment again.

The `debug` command is described with all operands in the manual “POSIX Commands” [3].

### 2.2.5 Using the POSIX library functions

In contrast to developing programs in the BS2000 environment (SDF), no special provisions are required for using the POSIX library functions in the POSIX environment. The following actions are executed automatically:

- Setzen des Präprozessor-Defines `_OSD_POSIX`
- merging in the standard header files supplied with CRTE and POSIX-HEADER from the standard directories `/usr/include` or `/usr/include/sys`. These depend on the language mode:

| language mode | searched directories |
|---|---|
| all C modes | `/usr/include`<br>`/usr/include/sys` |
| Cfront C++ | `/usr/include`<br>`/usr/include/sys` |
| C++ V3 | `/usr/include/CC`<br>`/usr/include`<br>`/usr/include/sys` |
| C++ 2017 / C++ 2020 library version 1 | `/usr/include/CXX01`<br>`/usr/include`<br>`/usr/include/sys` |
| C++ 2017 / C++ 2020 library version 2 | `/usr/include/CXX02`<br>`/usr/include`<br>`/usr/include/sys` |

  The default setting can be overwritten by specifying `-Y -I`.
- linking in the POSIX link switches `posix.o` and `postime.o` (corresponds to the PLAM library SYSLNK.CRTE. POSIX in the BS2000 environment)

The PROGRAM-ENVIRONMENT variable is set to “Shell” when the program is started.

Please refer to the manual “C Library Functions for POSIX Applications” [2] for further details.

## 2.3 C++ template instantiation under POSIX

This section provides information concerning the following topics:

- Basic aspects
- Automatic instantiation
- Generating explicit template instantiation statements (ETR files)
- Implicit inclusion
- Libraries and templates

### 2.3.1 Basic aspects

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>`. The compiler can create instantiations of the template for each of the types required from a single source description of the template for a stack.

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 mainly 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 C++ language allows one to write a specialization for a template entity, which means that the user can supply a specific version to be used instead of the instantiation generated from the template for a specific data type. In the language mode C++V3 such a specialization need not be declared when it is used. 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 C++ standard 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 (“automatic” instantiation), 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”). Refer to section “Automatic instantiation” for further details.

Explicit control over the instantiation process is available to the programmer via selectable instantiation modes and `#pragma` directives:

- The instantiation mode selection options are `-T auto`, `-T none`, `-T local` and `-T all`.They are described in detail in section “"Template options".

- The following `#pragma` directives can be used to control instantiation of single templates or a group of templates:
  - The `instantiate` pragma creates the template instance specified as the argument. This pragma can be used in place of the C++ language feature `template` *declaration* for explicit instantiation requests. See also the example on "Libraries and templates".
  - The `do_not_instantiate` pragma suppresses instantiation of the template instance specified as the argument. Typical candidates for this pragma are template entities for which specific definitions have been provided (specializations).
  - The `can_instantiate` pragma informs the compiler that the template instance specified as the argument can be, but does not have to be created in the compilation unit. This pragma is required in conjunction with libraries and is only evaluated in automatic instantiation mode. See also the example on "Libraries and templates".

  The exact syntax and general rules regarding these pragmas can be found in theC/C++ User Guide [4], in the section “Pragmas for controlling template Instantiation”.
- It is possible to implement explicit control using the “explicit instantiation statements” specified (in the source). These “explicit instantiation statements” can be generated using `-T etr_file_all` or `-T etr_file_assigned` (see section “Generatingexplicit template instantiation statements (ETR files)”) and can then be incorporated into the sources by the user.

## Important information

The method of template instantiation preset for this compiler (automatic instantiation by the prelinker and implicit inclusion) is also the method we recommend. There are options allowing you to change the settings for this method, but you should do this only in exceptional cases and only when you are very familiar with the entire application, including all the templates which are defined and used.

Implicit inclusion: implicit inclusion must not be disabled (with `-K no_implicit_include`) when templates from the C++ V3 library (SYSLNK.CRTE.STDCPP) are used, since otherwise no definitions are found.

Instantiation modes other than `-T auto` : there is a danger here that unsatisfied external references (`-T none`), duplicates (`-T all`) or runtime errors (`-T local`) may occur.

### 2.3.2 Automatic instantiation

Automatic instantiation (`-T auto` option) is supported by the compiler by default in the language modes C++ V3, C++ 2017 and C++ 2020. This allows you to compile your source code and link the generated objects without having to worry about how the necessary instantiations are done.

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 last 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 (see the rules for input file names on "Calling syntax and general rules"). This file is then implicitly included by the compiler on instantiation at the end of each compilation unit without a message being issued. See also section “Implicit inclusion” for details.
- 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 First instantiation using the definition list (temporary repository)).

The following steps are performed internally during automatic instantiation:

1. Create instantiation information files
   No template entities are instantiated the first time that one or more source files are compiled. For each source file that makes use of a template, an associated instantiation information file is created if no such file already exists. An instantiation information file has the suffix `.o.ii`. For example, the compilation of `abc.C` would result in the creation of the file `abc.o.ii`. The instantiation information file must not be modified by the user.
2. Create object files
   The created objects contain information on which instantiations could have been and were created when compiling a source file.
3. Assign template instantiations
   When the object files are linked, the prelinker is called before the actual linking takes place. The prelinker examines the object files, looking for references and definitions of template entities and for added information about entities that could be instantiated. If it does not find a definition for a required template entity, it searches for an object file which can instantiate the template entity.When it finds such a file, it assigns the instantiation to it.

4. Update the instantiation information file
   All instantiations that were assigned to a given file 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 object file
   When the compiler compiles a file, it reads the instantiation information file for that compilation unit and creates a new object file with the required instantiations.
7. Repetition
   Steps 3 to 6 are repeated until all instantiations that are required and can be generated have been created.
8. Linkage
   The object files are linked together.

### First instantiation using the definition list (temporary repository)

Since the procedure above (see "First instantiation without a definition list") recompiles some files more than once, an option was added that is intended to accelerate the overall process.

Generally the files are only recompiled once. The majority of instantiations are associated with the first files to be recompiled in the method. This has disadvantages in some cases since their object sizes increase due to this (although the sizes of other objects decrease to compensate for this).

Increasing the size of individual modules can be a disadvantage in user applications when, for example, precisely this module needs to be loaded often. The user must therefore decide if the method that more evenly distributes the instances (default) is desired or if this method is desired (due to improved response during prelinking).

This schema can be enabled by specifying the `-T definition_list` option.

Steps 3-5 above are modified. The resulting algorithm appears as follows then:

1. Create instantiation information files
   When one or more source files are compiled for the first time, no template entities are instantiated. One instantiation information file is created (if not already present) for every source file that uses a template. An instantiation information file has the file suffix `.o.ii` . When compiling `abc.C`, for example, the file `abc.o.ii` would be created. The instantiation information file may not be modified by the user.
2. Create object files
   The objects created contain information on which instances could be created and may be needed when compiling a source file.
3. Assign template instances to a source file
   If there are references for template entities for which there are no definitions in the set of object files, then a file is selected that could instantiate one of the template entities. All template entities that can be instantiated in this file are assigned to it.
4. Update instantiation information
   The set of instances that this file is assigned to is recorded in the associated instantiation file.
5. Save the definition list
   A definition list is maintained internally in memory. It contains a list of all definitions relating to templates that were found in all object files. This list can be read in and changed during recompilation.

   *Note*

This list is not stored in a file.

6. Recompilation
   The compiler is called again internally to recompile the corresponding source file.
7. Create new object file
   When the compiler recompiles a file, it reads the instantiation information file for this compilation unit and creates a new object file with the required instances.If the compiler gets the chance during compilation to instantiate additional referenced template entities that are not mentioned in the definition list or were not found in the libraries resolved, then it also instantiates these template entities (e.g. for templates that are contained in templates). It passes a list of instantiations received to the prelinker so that the prelinker can assign them to the file.

   This process permits faster instantiation. It also reduces the necessity of recompiling and existing file more than once during the prelink process.
8. Repeat
   Steps 3 - 7 are repeated until all necessary and generatable instances have been created.
9. Link
   The object files are linked.

### Further development

Once a program has been linked correctly, the associated instantiation information files contain all the names of the 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 object files, 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.

### Automatic instantiation, libraries and prelinked object files

When an executable file is generated with the `CC` command in automatic instantiation mode, the prelinker will do the automatic instantiation only in individual object files (`.o` files), but not in objects that are part of a library (`.a` library) or an object file that was prelinked with the `-r` option.

When generating the executable file, libraries or prelinked object files that require instances of template entities must either

- already contain these instances (which may be achieved by explicit instantiation and/or the preinstantiation of objects using the `-y` option; see "Options for selecting compilation phases")
- or provide appropriate header files with `can_instantiate` pragmas.

More details can be found in section “Libraries and templates”.

The option `-T add_prelink_files` provides a further method of controlling automatic instantiation in connection with libraries (see "Template options").

### Controlling instantiation assignments

The assignment of instantiations to local object files can be enabled and disabled with the `-K assign_local_only` and `-K no_assign_local_only` options (see "Template options").

### 2.3.3 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 are described in section “Template options”.

The option has three possibilities: `-T etr_file_none (default) /_all /_assigned.`If `_none` is specified, the file will not be generated, if `_all` is specified, all relevant information is output, if `_assigned` is specified, then only the specified information is output.

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 using `_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 using `_all`.
- Templates that are used in the compilation unit, but cannot be instantiated here. These are output using `_all`.

The contents of an ETR file have the following format:

- Comments in the header will indicate that the file is a generated file.
- Three 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 line which describes the explicit instantiation for this instance. This line contains the external name of the instance. This name is the same as the entry in the ii file and can also be obtained from the binder listing or the binder error listing

#### 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 third logical line is particularly interesting when copying to a source.
- The comments are always in English.

The following scenarios describe two possible uses of an ETR file:

1. The compiler is called during development using the `-T auto` and `-T etr_file_assigned` options.
   The instantiation statements output to the ETR files are incorporated into the appropriate sources. Productive operation is then activated using the `-T none` or `-T auto` option 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 `-T none` and `-T etr_file_all` options.
   After binding 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 (`-T none`).
   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** (when using `etr_file_all`).

The following command is used for compilation:

```
CC -c -T etr_file_all x.c y.c
```

### 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.o.etr

```
// This file is generated and will be changed when the module is compiled

// The following template was
// explicitly instantiated
#pragma instantiate_mangled_id __1f__tm__2_l__FZ1Z_v&_

// The following template was
// used in this module and can be instantiated here
#pragma instantiate_mangled_id __1f__tm__2_i__FZ1Z_v&_

// The following template was
// used in this module and can be instantiated here
#pragma instantiate_mangled_id __1f__tm__2_c__FZ1Z_v&_

// The following template was
// used in this module
#pragma instantiate_mangled_id __1g__tm__2_i__FZ1Z_v&_
```

### ETR-file y.o.etr

```
// 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
#pragma instantiate_mangled_id __1f__tm__2_i__FZ1Z_v&_
```

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”), see **Example 2**.
Then you will subsequently not need to use automatic template instantiation.

### Example 2

Example of the use of `etr_file_assigned`.
Two files are specified **x.c** and **y.c:**

### 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 first compiled using the following commands and then prelinked:

```
CC -c -T auto -T etr_file_assigned x.c
CC -c -T auto -T etr_file_assigned y.c
CC -y -T auto -T etr_file_assigned x.o y.o
```

Then a file is created **x.o.etr** (since only x template instantiations are assigned) which looks like this:

```
// This file is generated and will be changed when the module is compiled

// The following template was
// instantiated automatically by the compiler
#pragma instantiate_mangled_id __1f__tm__2_i__FZ1Z_v&_

// The following template was
// instantiated automatically by the compiler
#pragma instantiate_mangled_id __1f__tm__2_c__FZ1Z_v&_
```

The important lines are inserted in the file **x.c**, thus creating 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);
}
#pragma instantiate_mangled_id __1f__tm__2_i__FZ1Z_v&_
#pragma instantiate_mangled_id __1f__tm__2_c__FZ1Z_v&_
```

Then production can be carried out using the following commands:

```
CC -c -T none x1.c
CC -c -T none y.c
```

### Example 3

The following example shows the four classes of template that can be output. The assumptions are as in Example 1

The following commands are entered:

```
CC -c -T auto y.c
CC -y -T auto y.o (this assigns y.o to f(int))
CC -c -T auto -T etr_file_all x.c
CC -y -T auto -T etr_file_all x.o y.o
```

This creates the following ETR file, **x.o.etr**:

```
// This file is generated and will be changed when the module is compiled

// The following template was
// explicitly instantiated
#pragma instantiate_mangled_id __1f__tm__2_l__FZ1Z_v&_

// The following template was
// used in this module and can be instantiated here
#pragma instantiate_mangled_id __1f__tm__2_i__FZ1Z_v&_

// The following template was
// instantiated automatically by the compiler
#pragma instantiate_mangled_id __1f__tm__2_c__FZ1Z_v&_

// The following template was
// used in this module
#pragma instantiate_mangled_id __1g__tm__2_i__FZ1Z_v&_);
```

### 2.3.4 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 `-K implicit_include` option on "Template options") and can be disabled with `-K no_implicit_include`. Please refer to the notes on "Basic aspects" with regard to disabling implicit inclusion.

If 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 in a source file with the same base name as the header file and with a suitable suffix (e.g. *basename* `.C` ).
Let us assume, for example, that a template entity `ABC::f` is declared in the header file `xyz.h`. 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 directory containing the header file for a source file with the base name `xyz` and a suitable suffix (e.g. `xyz.C`). If such a file exists, it will be treated as if it were included at the end of the source file.

The following suffixes are checked in this order: c, C, cpp, CPP, cxx, CXX, cc, CC, c++ und C++. These are the suffixes that are interpreted as C ++ source by default. The `-Y F` option has no effect on the suffixes checked for implicit inclusion.

To ensure that the file containing the definition of a particular template entity can be found during instantiation, the complete path name of the file with the declaration of the template must be known. This information is not available in files containing `#line` directives. Consequently, implicit inclusion is not possible in such cases.

*Implicit inclusion and the* `make` *utility*

When working with the `make` utility, implicit inclusions must be taken into account when generating file dependency lines. In other words, the object file depends on explicitly included headers as well as implicitly included files with template definitions.

When using the `-M` option, implicit inclusions will be taken into account in automatic instantiation mode only if the instantiation information files have been correctly built.

The following steps are required for this purpose:

1. Compile all source files.
2. Link the program together so that all instantiations are assigned.
3. Generate file dependency lines with the `make` program using the `-M` option (see also "Options for selecting compilation phase").
4. Repeat steps 2 and 3 if the generated template instances have changed.

### 2.3.5 Libraries and templates

Instantiations for template entities (template functions, member functions and static data members of template classes) can be generated in automatic instantiation mode only if the object meets the following conditions:

- It is not part of a `.a` library,
- it contains a reference to the template entity or the `can_instantiate` pragma for that template entity,
- and it contains all definitions needed for the instantiation.

A library that requires instances for its implementation must either contain these instances or provide special headers with `can_instantiate` pragmas. These two options are explained individually below.

1. The library contains all required instances

   The main point to be observed here is to ensure that no duplicates are created when using multiple libraries.

   The instantiation of template entities in libraries can be achieved by the following methods:

   a. automatic instantiation of the template unit using the prelinker with the option `-y` (see "Options for selecting compilation phases").

      *Caution*

      If multiple libraries that require the same entity are used, there is a potential risk of duplicates being created, since a separate object is not created per entity. This can be avoided by using the `-T add_prelink_files` option (see "Template options").

   b. by explicitly instantiating all template entities with the instantiation directive `template` *declaration* or the `instantiate` pragma.

      The main point to be observed here is to ensure that a separate object is created per entity.

      *Example*

      Given:

      - a library `l.a` with references to the instances `t_list(Foo1)` and `t_list(Foo2),`
      - a header file `listFoo.h` with the declarations of `t_list`, `Foo1` and `Foo2`
      - and a source file `listFoo.C` with the definitions of `t_list`, `Foo1` and `Foo2`

```
// l.h
#ifndef L_H
#define L_H
#include "listFoo.h"
void g();
#endif

// l.C (l.o is an element of l.a)
#include "l.h"
void g()   {
   Foo1 f1;
   Foo2 f2;
   //...
   t_list(f1);
   t_list(f2);
   //...
   }

//listFoo.h
#ifndef LIST_FOO_H
#define LIST_FOO_H
template <class T> void t_list (T t);
class Foo1 {...};
class Foo2 {...};
#endif

//listFoo.C
template <class T> class t_list (T t)
{
   ...
};
```

Each of the referenced instances are contained in separate objects in the library l.a.

```
// lf1.C (lf1.o is an element of l.a)
// and contains an explicit instantiation for t_list(Foo1)
#include "listFoo.h"
template void t_list(Foo1);

// lf2.C (lf2.o is element of l.a)
// and contains an explicit instantiation for t_list(Foo2)
#include "listFoo.h"
#pragma instantiate void t_list(Foo2)
```

2. The header files contain `can_instantiate` pragmas for all required instances.

   *Example*

   Given:

   - a library `l.a` with a reference to the instance `t_list(Foo)`,
   - a header file `listFoo.h` with the declarations of `t_list` and `Foo`
   - and a source file `listFoo.C` with the definitions of `t_list` and `Foo`.

```
// l.h
#ifndef L_H
#define L_H
#include "listFoo.h"
void g();
#endif

// l.C (l.o is an element of l.a)
#include "l.h"
void g()
{
   Foo f;
   //...
   t_list(f);
   //...
}

//listFoo.h
#ifndef LIST_FOO_H
#define LIST_FOO_H
template <class T> void t_list (T t);
class Foo {...};
#pragma can_instantiate t_list(Foo)
#endif

//listFoo.C
template <class T> void t_list (T t) {...};
```

The object `user.o` and the library `l.a` are linked together (`CC user.o l.a`).

```
// user.C
#include "l.h"
int f ()
{
   g();
}
```

`user.C` includes `l.h`, which in turn includes `listFoo.h`. Consequently, `user.C` contains notification that `list(Foo)` can be instantiated.

Automatic instantiation by the prelinker produces only one instance `t_list(Foo)`.

*Note*

In order to generate the needed instances, the `can_instantiate` pragma must be contained in a header file of the library that will be included by the user programs.

## 2.4 Porting software

When porting C source programs from UNIX systems into POSIX BS2000, note must made of the different, implementation-dependent handling of externally visible names by the compiler.

The BS2000 C/C++ compiler uses the external name of the source program (e.g. function name) to create a corresponding external name for the link editor (entry name). As default, lowercase letters are converted to uppercase and the underscore character (_) is converted to a dollar sign ($) (see also “Generating the entry names with LLMs” (Options for controlling object generation )). These conversions ensure that the objects created by the compiler can be linked to other objects (e.g. objects created by BS2000 compilers in other languages or objects in object module format).

When selecting the externally visible name for C source programs, it is therefore imperative that two names not only differ in uppercase/lowercase. For example, the function names `getc` and `getC` will be mapped to the same external name `GETC`. Provided no names of BS2000 compilers in other languages are affected, this behavior can be prevented using the `-K llm_case_lower` option (see "Options for controlling object generation").

## 2.5 Introductory examples

### Compiling and linking with the c89 command

```
c89 hello.c
```

Compiles `hello.c` and creates the executable file `a.out`

```
c89 -o hello hello.c
```

Compiles `hello.c` and creates the executable file `hello`

```
c89 -c hello.c upro.c
```

Compiles `hello.c` and `upro.c` and creates the object files `hello.o` and `upro.o`

```
c89 -o hello hello.o upro.o
```

Links `hello.o` and `upro.o` to the executable file `hello`

### Copying with the bs2cp command

```
bs2cp bs2:hello hello.c
```

Copies the cataloged BS2000 file `HELLO` to the POSIX file `hello.c`

```
bs2cp 'bs2:plam.bsp(hello.l,l)' hello.o
```

Copies the LLM `HELLO.L` from the PLAM library `PLAM.BSP` to the POSIX object file `hello.o`

# 3 The cc, c11, c89 and CC commands

The C/C++ compiler can be called and supplied with options from the POSIX shell. The options cover most of the services and functions available for controlling the compiler via the SDF interface.
The syntax of the options, names of the processed or created objects and other conventions are based on the definition in the XPG4 Standard. POSIX shell interface extensions not covered by the XPG4 standard are based on the normal compiler or utility interface in UNIX systems.

The compiler includes an integrated link phase which converts the normal shell link options into corresponding link editor directives. A “standalone” link editor which is independent of the calling command is not available in the POSIX shell.

Only POSIX files can be read and written when compiling with the C/C++ compiler in the POSIX shell. BS 2000 files are not supported.
The source and header files may exist in either EBCDIC or ASCII code. It is assumed that all files (from a remote, merged in or a POSIX file system) are in the same codeset.

## 3.1 Calling syntax and general rules

{ `cc` | `c11` | `CC` } { *option* | *operand* } ...

or

`c89` [*option*]... *operand*...

The differences between the `cc`, `c11`, `c89` and `CC` commands are summarized below.

### The cc, c11, c89 and CC commands

`cc`

If the compiler is called with `cc`, it works as a C compiler, and the default language mode is set to the latest supported C language mode. In this version of the compiler this mode is C11 (see the `-X 2011` option in section "Options for selecting the language mode)").
Options and operands may be specified in any order on the command line.
In contrast to the `c89` command, `-L` *dir* is interpreted as an operand (see option -L and the `--` option in section "General options").

`c11`

If the compiler is called with `c11`, it works as a C compiler, and the default language mode is set to C11 (see the `-X 2011` option in section "Options for selecting the language mode").

Options and operands may be specified in any order on the command line.
In contrast to the `c89` command, `-L` *dir* is interpreted as an operand (see option -L and the `--` option in section "General options").

`c89`

If the compiler is called with `c89`, it works as a C compiler, and the default language mode is set to C89 (see the `-X 89` option in section "Options for selecting the language mode").

In this case, options and operands cannot be mixed on the command line, i.e. the "options before operands" sequence must be maintained.
In contrast to the `cc/CC` commands, `-L` *dir* is interpreted here as an option (see option -L and the `--` option in section "General options").).

`CC`

If the compiler is called with `CC`, it works as a C++ compiler, and the default language mode is set to the latest supported C++ language mode. In this version of the compiler this mode is C++ 2020 (see the `-X 2020` option in section "Options for selecting the language mode)").

Options and operands may be specified in any order on the command line.
In contrast to the `c89` command, `-L` *dir* is interpreted as an operand (see option -L and the `--` option in section "General options").

### Options

No *option* specified

If the source code contains no syntax errors, and all unresolved references are resolved, the compiler generates an executable file `a.out`, which contains the executable program.
The compiler only stores the object code of the separate source files in `.o` files with the same names if at least two source files or one source file and one (`.o`) object file are specified.
If only a source file *file.c* is specified, no object file *file.o* is available after compilation as it is a temporary file and is subsequently deleted. If an object file *file.o* exists before compilation, this is also deleted.

*option*

You can specify options in the compiler call to control the compilation process and to determine which arguments are passed to the programs for the individual compilation phases.

Options can also be used to instruct the compiler to perform only some of the compilation phases (see "Options for selecting compilation phases"). If the compilation process is not completed fully, all options that refer to the skipped compilation phases are ignored by the compiler. If multiple options are used to select the compilation phases to be performed, the compiler will stop after the earliest specified phase.

An option always consists of a single letter that is identified by a leading hyphen ("-").

Multiple options may be grouped, i.e. specified in succession after a single hyphen without any delimiting whitespace, only if none of the listed options take any arguments (e.g. `-V -c` could also be entered as `-Vc`).

Options that take arguments must be specified in accordance with the XPG4 Standard by separating the option and its argument with a space. This XPG4-compliant notation is strongly recommended, but is not enforced by the compiler for compatibility reasons (e.g. the compiler will accept `-ohello` instead of `-o hello` ).

Arguments that contain delimiters (`:` or `,`) or the equals sign (=) must not be specified with any whitespace before or after these characters.

*Examples*

```
-D MAKRO = 1     illegal
-D MAKRO=1       legal
-R limit,  20    illegal
-R limit,20      legal
```

If the same option is specified more than once with conflicting arguments (e.g. `-K at` and `-K no_at`), the last option specified on the command line applies.

Options that are not known to the compiler, i.e. options that begin with an unrecognized letter after the leading hyphen ("-"), are ignored. A corresponding warning is issued. If the unknown option and the argument are separated by whitespace, the option is interpreted as an option without an argument.

Options with unrecognized arguments are ignored, and a corresponding warning is issued.

*Special input rules for the* `-K` *option*

`-K` *arg1*[,*arg2*...]

The `-K` option can be used to specify one or more arguments in succession, with a delimiting comma between each such argument. The delimiter between the arguments (i.e. the comma) must not be preceded or followed by any whitespace.Multiple `-K` options with one argument each have the same effect as a single `-K` option with multiple arguments delimited by commas. The arguments specified with the `-K` option may be entered in uppercase and/or lowercase letters (e.g. the arguments `PIC`, `pic`, `Pic`, etc. are equivalent). In the case of conflicting specifications (e.g. `-K uchar` and `-K schar`), the last entry is taken without issuing a warning.

## Operands

The "operands" category includes the following entries:

- the names of input files, i.e.: *file*.*suffix*
- the link editor options `-l` *x* and `-lBLSLIB`
- only for the `cc/c11`/`CC` commands: also the link editor option `-L` *dir*

The compiler processes all options first, and then the operands, in the order in which they are specified on the command line.
All arguments that follow the `--` option (which ends the input of options) on the command line are interpreted as operands, even if they begin with a "-" character (see the `--` option in section "General options").

*file.suffix*

The name of an input file.

The compiler determines the contents of a file, and thus the compilation steps to be performed in each case, from the file name extension. The file name must therefore have a suffix (or extension) that matches its contents. The possible suffixes that can be used to identify source files will depend on the mode in which the compiler is invoked and whether the compiler was called with the `cc`/`c11`/`c89` command (C mode) or with `CC` (C++ mode).

The following suffixes are interpreted in individual cases as listed below:

`c`, `C` C source code (`cc`, `c11`, `c89`) or C++ source code (`CC`) before the preprocessor run

`cpp`, `CPP`, `cxx`, `CXX`, `cc`, `CC`, `c++`, `C++`

C++ source code before the preprocessor run (`CC`)

`i` C source code (`cc`, `c11`, `c89`) after the preprocessor run

`I` C++ source code after the preprocessor run (`CC`)

`o` Object file

`a` Static library with object modules created with the `ar` utility.

In addition to the suffixes above, the `-Y F` option may be used to specify other user defined suffixes, which are then recognized by the individual compiler components (see "General options").

File names with no suffix or an unrecognized suffix are passed through to the link editor without issuing a warning.

At least one input file (*file*`.`*suffix*) or one library in the form `-l x` is required for each compiler call.

If more than one input file is specified, these files need not be of the same type, i.e. source files and object files may all be specified in the same compiler call. In the case of object files and libraries, the order and position in which they are entered on the command line are significant for linking.

`-L` *dir*

`-L` *dir* is only interpreted as an operand when the compiler is called with the `cc`, `c11` and `CC` commands. *dir* can be used to specify an additional directory that is to be searched by the link editor for the libraries specified with the `-l` option (see "Link editor options" for more details).

`-l` *x*

This operand instructs the link editor to search for libraries named `lib`*x*`.a` (see “Link editor options” for more details).

`-l BLSLIB`

This operand instructs the link editor to search through PLAM libraries which were assigned with the BLSLIB*nn* shell environment variable (00 >= *nn* <= 99) (see “Link editor options” for more details).

## Exit status

0 Normal termination of the compiler run; no errors, but possibly with notes and warnings

1 Normal termination of the compiler run; with error

2 Abnormal termination of the compiler run; with the occurrence of a fatal error

## 3.2 Description of options

The following sections describe the possible options for the `cc`, `c11`, `c89` and `CC` commands in groups, depending on the context in which they are used. The options are classified as follows:

- Options for selecting the language mode
- General options
- Options for selecting compilation phases
- Preprocessor options
- Common frontend options in C and C++
- C++-specific frontend options
  - General C++ options
  - Template options
- Optimization options
- Options for controlling object generation
- Debug option
- Runtime options
- Link editor options
- Options for controlling message output
- Options for outputting listings and CIF information

The general aspects to be observed when entering options are discussed in section “Calling syntax and general rules”).

A list of all options in alphabetic order with references to the pages on which they appear can be found in the appendix.

### 3.2.1 Options for selecting the language mode

`-X cc`

This option is used to select the C language mode. It only needs to be specified if the compiler is to run in C mode, but is not called by `cc`, `c11` or `c89`.

`-X CC`

This option is used to select the C++ language mode. It only needs to be specified if the compiler is to run in C++ mode, but is not called by `CC`.

`-X 89`
`-X 90`
`-X 1990`

C89 mode (the default setting when calling the compiler with `c89`)
This option is only allowed in C mode. The compiler supports C code, as defined in the ANSI/ISO C standard from 1990. This standard is also known as the ANSI C89 standard. This specification corresponds to the specification `-X a` or `-X c` of the C/C++ V3 compiler.
`__STDC_VERSION__` has a value of 199409L.

`-X 11`
`-X 2011`

C11 mode (the default setting when calling the compiler with `cc` or `c11` or specifying `-X cc`)
This option is only allowed in C mode. The compiler supports C code according to the 2011 C standard.
`__STDC_VERSION__` has a value of 201112L.

`-X kr`
`-X KR`

K&R C mode

This option is only allowed in 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 system (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 `-K kr_cpp` can be specified to convert the preprocessor behavior to K&R C (as required when porting old C sources, for example).
`__STDC_VERSION__` is undefined. The option -X strict has no effect, i.e. -X nostrict always applies.

`-X 17`
`-X 2017`

C++ 2017 mode
This option is only allowed in C++ mode. The compiler supports C++ code according to the 2017 C++ standard.

`__cplusplus` has a value of 201703L and `__STDC_VERSION__` a value of 199409L.

`-X 20`
`-X 2020`

C++ 2020 mode (the default setting when calling the compiler with CC or specifying -X CC)
This option is only allowed in C++ mode. The compiler supports C++ code according to the 2020 C++ standard. Full C++ 2020 library support is currently not available and instead the existing C++ 2017 library is used.

__cplusplus has a value of 202002L and __STDC_VERSION__ a value of 199409L.

-X V2-COMPATIBLE

-X v2-compatible

Cfront-C++ mode
This option is only allowed in 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, which was first released with the C++ V2.1 compiler. The Cfront-compatible C++ library for complex mathematics and stream-oriented I/O is available.
More information on the Cfront C++ library can be found in the C/C++ User Guide [4 ].
C++ sources must be compiled and linked with -X v2-compatible if their objects are to be linkable with C++ V2.1/V2.2 objects.
This specification corresponds to the specification -X d of the C/C++ V3 compiler.
__cplusplus has a value of 1 and __STDC_VERSION the value 199409L. The option -X strict has no effect, i. e. -X nostrict always applies.

-X V3-COMPATIBLE

-X v3-compatible

C++ V3 mode
This option is only allowed in ++C mode. The compiler supports C++ code corresponding to the C/C++ V3 compiler. The specification corresponds to the specification -X w or -X e of the C/C++ V3 compiler.
The following C++ libraries are available:

- the standard C++ library (strings, containers, iterators, algorithms, and numerics), including the Cfront-compatible I/O classes
- the C++ V3 library Tools.h++

For more information on C++ libraries, see also the C++ User Guide [4].

__cplusplus has a value of 2 (for -X nostrict) or 199612L (for -X strict) and __STDC_VERSION has the value 199409L.

-X strict

Strict C- resp. C++ mode
The namespace is restricted to the names defined in the standard, and only the C resp. C++ library functions defined in the standard are available. Certain extensions (such as the asm keyword) and some commonly expected header file declarations (stdio.h, stdlib.h etc.) are not available.
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 -R strict_errors (see "Options for controlling message output").
__STDC__ has a value of 1, _STRICT_STDC is defined

`-X nostrict`

Extended C- resp. C++ mode
Some required diagnostics are omitted, the name space is not restricted to names that are specified by the standard and some extensions are included.

`__STDC__` has a value of 0, `_STRICT_STDC` is not defined

`-K library_version=`*n*

`-K library_version=high`

This option specifies the library version for C++ 2017 and C++ 2020.

The library version has an effect on the libraries used for include file search and resolving. With library version 1 they are `/usr/include/CXX01` and `SYSLNK.CRTE.CXX01`, with library version 2 they are `/usr/include/CXX02` and `SYSLNK.CRTE.CXX02`.

See the C/C++ user guide [4] for more details.

### 3.2.2 General options

`-K` *arg1*[,*arg2*...]

General input rules for the `-K` option can be found on "Calling syntax and general rules".
The following entries are possible as *arg* arguments for general control of the compilation run:

`verbose`
`no_verbose`

Note that the `-K verbose` specification, which causes additional information on template instantiation to be written to the standard error output `stderr`, is presently only meaningful with the `CC` command.
`-K no_verbose` is the default setting.

`-o` *output_destination*

If this option is omitted, the compiler generates output files with default names and places them in the current directory.
The `-o` option can be used to rename the various output files of a compiler run and/or have them written to a different directory.

*output_destination* can be any of the following: only the name of a directory, only a file name, or a file name including directory components. The specified directories must already exist.

*output_destination* = directory name *dir*

The output files are created with default names and placed in the specified directory *dir* as follows:

- When an executable file is generated, the file is assigned the name *dir*`/a.out`.
- If the `-c` option is specified, the object file is named *dir/sourcefile*`.o`.
- If the `-E` option is specified, the preprocessor output is written to the file*dir/sourcefile*`.i` (`cc`/`c11`/`c89` command) or *dir/sourcefile*`.I` (`CC` command) ) instead of the standard output `stdout`.
- If the `-M` option is specified, the preprocessor output (dependency lines for further processing with `make`) is written to the file *dir/sourcefile*`.mk` instead of the standard output `stdout`.
- If the `-P` option is specified, the preprocessor output is written to the file`dir/`*sourcefile*`.i` (`cc`/`c11`/`c89` command) or *dir/sourcefile*`.I` (`CC` command).

With the exception of the executable file generated by the link editor, independent output files are created for each compiled source file in cases where multiple source files are specified for compilation.

*output_destination* = a specified *file_name* or
*output_destination* = a specified directory and file name: *dir/file_name*

If an executable file is being generated or if the `-o` option is specified in combination with option `-c`, `-E`, `-M` or `-P`, the compiler writes the result to a file named *file_name* and places it in the current directory or in the directory specified with *dir*. Apart from the executable file generated by the link editor, a different file name may be specified for all other output files, but only if a single source file is listed for compilation in each compiler call.
If more than one input file is specified but only **one** output file is specified, then a warning is output and *output_destination* is reset to the default value.
If an executable file is created, the file name specified with `-o` must differ from an object file generated by the compiler or specified explicitly in the command line. For example, the following commands are rejected with errors:

```
cc -o hello.o hello.o
cc -o hello.o hello.c
```

`-V`

For each compiler component that is implicitly called during the execution of `cc/c89/CC`, the version and possibly a copyright note are written in a separate line. In the linking procedure the version of the CRTE being used and a list of the libraries used are also output.

`-Y F,`*file-type*`,`*user_suffix*

This option can be used to define user-specific suffixes in the form *user_suffix* for input files of type *file-type* in addition to the standard suffixes (see "Calling syntax and general rules").

The following entries are possible for *file-type*:

| | |
|---|---|
| `c++` | C++ source file |
| `c` | C source file |
| `prec++` | C++ preprocessor output file |
| `prec` | C preprocessor output file |
| `obj` | Object file |
| `lib` | Static library |

*Example*

```
-Y F,obj,llm
```

An input file named *file*`.llm` is recognized by the compiler as an object file.

`--`

This option ends the input of options, i.e. causes all following arguments (except for the link editor options that fall under the "operands" category) to be interpreted as file names, even if they begin with a hyphen. This makes it possible to specify file names that start with a hyphen (e.g. `-hello.c`).
The following link editor options are permitted after the `--` option:
`-l` *x*

`-l BLSLIB`

`-L` *dir* (only with the `cc`, `c11` and `CC` commands; in the `c89` command, this entry would be interpreted as a file name!)

### 3.2.3 Options for selecting compilation phases

All of the options listed below always suppress the linkage run and cause any link editor options and operands that may have been specified to be ignored.

`-c`

Terminates the compiler run after an LLM has been created and placed into an object file *file*`.o` for each compiled source file. The object is written by default into the current directory. The `-o` option (see "General options") can be used to define a different file name and/or directory.

`-E`

The compiler run is terminated after the preprocessor phase and the result is written to the standard output `stdout`. Any blank lines present in the file are combined in the process, and the corresponding `#line` directives are generated. By default, C and C++ comments are removed from the preprocessor output (see the `-C` option in section "Preprocessor options"). If the `-o` option is specified (see "General options"), the result of the preprocessor run is written to a file instead of the standard output `stdout`.

`-M`

The compiler run is terminated after the preprocessor phase; however, instead of the normal preprocessor output (cf. `-E`, `-P`), a list of dependency lines that is suitable for further processing with the POSIX `make` program is generated and written to the standard output `stdout`. If the `-o` option is specified (see "General options"), the file dependency list is written to a file instead of the standard output `stdout`.

#### Note

Templates in C++ 2017, C++ 2020 and C++ V3 sources are not included implicitly.

`-P`

The compiler run is terminated after the preprocessor phase, and the result is written to a file named *file*`.i` (`cc`/`c11`/`c89` command) or *file*`.I` (`CC` command) and placed in the current directory instead of the standard output `stdout` as in the `-E` option. The output does not contain any additional `#line` directives. By default, C or C++ comments are removed from the preprocessor output (see the `-C` option on "Preprocessor options"). *file*`.i` can be subsequently compiled further with the `cc`/`c11`/`c89`/`CC` commands, whereas *file*`.I` can only be compiled with the `CC` command. If desired, the `-o` option (see "General options") may be used to specify another file name and/or directory.

`-y`

This option can only be specified with the `CC` command in the modes C++ 2017, C++ 2020 and C++ V3.
The compiler run is terminated after the prelinker phase (automatic template instantiation), and an object file named *sourcefile*`.o` containing the instantiated templates is generated for each compiled source file. This is meaningful for objects that are to be subsequently incorporated in a library (`.a` library) or in a prelinked object file (`-r`); no automatic instantiation is performed for templates within libraries or prelinked object files. Note that the `-y` option can only be meaningfully used in the default automatic instantiation mode (`-T auto`).

*Example*

Contents of the source files (extracts):

```
// a.h:
```

```
class A {int i;};

// f.h:

template <class T> void f(T)
{
   /* any code */
}

// b.c:

#include "a.h"
#include "f.h"
void foo() {
   A a;
   f(a);
}

// main.c:

extern void foo();

int main(void)
{
   foo();
}
```

Commands:

`CC -c b.c`

The first compilation produces an object file `b.o` and a template information file `b.o.ii`, where each contains an entry that the function `f(A)` is not instantiated.

`CC -y b.o`

The `b.o` and `b.o.ii` files generated in the first compilation run are updated, and the function `f(A)` is instantiated.

`ar -r x.a b.o`

The module in `b.o` is added to the library `x.a`.

`CC main.c x.a`

An executable file named `a.out` is generated.

The following command sequence, by contrast, would not produce the desired result:

```
rm *.o *.ii *.a a.out   /* Cleanup the current directory */
CC -c b.c
ar -r x.a b.o
CC main.c x.a
```

This command sequence results in an error message. This is because the function `f(A)` cannot be found, since no automatic instantiation is performed for the templates in the library `x.a`.

### 3.2.4 Preprocessor options

`-A "`*name*`(`*tokens*`)"`

This option can be used to define an assertion, as if by a preprocessor `#assert`directive (see the section “Extensions over ANSI/ISO C” in the C/C++ User Guide [4]). The quotes are required because of the special significance of parentheses in the POSIX shell. The parentheses can alternatively be nullified with the backslash:
`-A` *name*`\(`*tokens*`\)`

`-C`

This option is evaluated only if the `-E` or `-P` option is also specified (see "Options for selecting compilation phases"). It causes C or C++ comments to be retained in the preprocessor output. Such comments are removed by default.

`-D` *name*[=*value*]

This option can be used to define names, symbolic constants and macros (as if by a preprocessor `#define` directive).
`-D` *name* has the same effect as the statement `#define` *name* `1`;
`-D` *name*=*value* corresponds to the `#define` directive for text substitutions,
i.e. `#define` *name value.*

`-H`

Causes a list of all header files used during the compilation run to be written to the standard error output `stderr`.

`-i` *header*

This option specifies an include file *header* which is inserted before the source program text (pre-include).

You can specify the *header* as follows:

- by giving the fully qualified path name of the include file
- by giving the relative path name of the include file on the basis of the option `-I`

The include file specified as the *header* will be handled in exactly the same way as any include file where an `#include` statement is specified at the beginning of the source program file. If several header files are to be pre-included, the corresponding #include instructions must be collected together in a single include file which is then specified using the option `-i`.

`-I` *dir*

*dir* is added to the list of directories that are searched by the preprocessor for header files. If this option is entered more than once, the order of entry determines the search order for header files.

If the relative pathname of the header file (which does not begin with a slash /) is specified in the `#include` directive enclosed within quotes "...", the preprocessor searches the directories in the following order:

1. the directory of the source or header file containing the `#include` directive
2. the directories that were specified with the preprocessor `-I` option

3. either the directories specified with the `-Y I` option or the standard directories (see "Using the POSIX library functions")

If the relative pathname of the header file is specified in the `#include` directive enclosed within angular brackets <...>, the preprocessor will only search the directories listed under points 2 and 3 above.

If you want the preprocessor to search some other directories last instead of the standard directories listed above, you can specify such directories by using the `-Y I` option (see below).

`-K` *arg1*[,*arg2*...]

General input rules for the `-K` option can be found in section "Calling syntax and general rules".
The following entries are possible as *arg* arguments to control preprocessor behavior:

`ansi_cpp`

`kr_cpp`

`-K ansi_cpp` is the default setting in all C and C++ language modes of the compiler. This means that preprocessor behavior in accordance with the ANSI/ISO C standard is also supported in the K&R C mode. The obsolete preprocessor behavior based on Reiser cpp and Johnson pcc can be turned on with `-K kr_cpp`.

`-U` *name*

Undefines a macro or a symbolic constant *name* (as when using the preprocessor directive `#undef`), where *name* is a predefined preprocessor name (see "Predefined preprocessor names") or a name that was defined with the `-D` option in the command line before or after option `-U` .This option has no effect on `#define` directives in the source program.

`-Y I`,*dir*[:*dir*...]

Instructs the preprocessor which directory or directories are to be searched for header files last. *dir* specifies the directory.
Without this option, the last directories to be searched are the standard directories (see "Using the POSIX library functions").

### 3.2.5 Common frontend options in C and C++

`-K` *arg1*[,*arg2*...]

General input rules for the `-K` option can be found on "Calling syntax and general rules".
The following entries are possible as *arg* arguments to control the compiler frontend in the C and C++ modes:

`uchar`
`schar`

The default data type `char` is `unsigned char`. If `-K schar` is specified, `char` is treated as a `signed char` in expressions and conversions.
Note that the use of this option may result in portability problems!

`at`
`no_at`

If `-K no_at` is specified, the “at” sign ‘@’ is not allowed in identifiers.
The default is `-K at`. The ‘@’ sign in identifiers is an extension.

> If you are using a Cfront C++ library, you cannot use the option `-K no_at`.

`dollar`
`no_dollar`

If `-K no_dollar` is specified, the dollar sign ‘$’ is not allowed in identifiers.
The default is `-K dollar`. The ‘$’ sign in identifiers is an extension.

`literal_encoding_native`
`literal_encoding_ascii`
`literal_encoding_ascii_full`
`literal_encoding_ebcdic`
`literal_encoding_ebcdic_full`

This option determines whether the C/C++ compiler creates the code for characters or for 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: `\x[0-9A-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_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. `literal_encoding_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`.

### Prerequisites for using the ASCII Format:

- You must not explicitly declare the C library functions in your source program, but only indirectly by including the corresponding CRTE header. Otherwise a tranlation error ‘CFE1079[ERROR]..: expected a type specifier‘ may occur.
- 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 `#include <stdio.h>`.
- If you are using CRTE functions you must also specify the following options:
  `-K llm_keep`
  `-K llm_case_lower`

`signed_fields_signed`
`signed_fields_unsigned`

If `-K signed_fields_unsigned` is specified, `signed` bit fields are always interpreted as `unsigned`. This option is only offered for compatibility with older C versions and is only meaningful in K&R C mode. The default is `-K signed_fields_signed`.

`plain_fields_signed`
`plain_fields_unsigned`

These arguments control whether integer bit fields (`short`, `int`, `long`, `long long`) are treated as `signed` or `unsigned` types by default.
The default is `-K plain_fields_signed`.

`long_preserving`
`unsigned_preserving`

These arguments control whether arithmetic operations with operands of type `long` and `unsigned int` return a result of type `long` (`long_preserving`) in accordance with K&R mode (first edition; appendix 6.6) or of type `unsigned long` (`unsigned_preserving`) in accordance with ANSI/ISO C.
The default is `-K unsigned_preserving`.

`alternative_tokens`
`no_alternative_tokens`

These arguments control whether alternative tokens are to be recognized by the compiler:

- Digraph sequences in the C and C++ language modes (e.g. `<:` for `[`),
- Additional keyword operators which are only valid in the C++ language modes (e.g. `and` for `&&`, `bitand` for `&`).

`-K alternative_tokens` is the default in the modes C11, C++ V3, C++ 2017 and C++ 2020,
`-K no_alternative_tokens` is the default for all other modes.

`longlong`
`no_longlong`

These arguments control whether the data type `long long` is recognized by the compiler.
`-K longlong` is the default. In this case, the preprocessor define `_LONGLONG` is set.
If `-K no_longlong` is set, the use of the data type `long long` will result in an error. This specification is only allowed in the language modes C89 and C++ V3.

`end_of_line_comments`
`no_end_of_line_comments`

These arguments control whether the compiler accepts C++-style comments (//...) in the extended C89 mode as well. `-K no_end_of_line_comments` is the default in the extended C89 mode.
Only one entry is permitted in the other language modes, the other is ignored with a warning. In the strict C89 mode and in the K&R mode, C++ comments are not allowed. In the C++ modes and in the C11 mode they are always allowed.

### 3.2.6 C++-specific frontend options

The options described in the following sections on "General C++ options" and "Template options" are only applicable to the `CC` command.

#### 3.2.6.1 General C++ options

General C++ options can be used to control the following C++ features:

- whether the keywords `wchar_t` and `bool` are recognized
- the scope of initialization directives in `for` and `while` loops
- whether the old specialization syntax is accepted

None of the language features listed above is supported in Cfront C++ mode.

`-K` *arg1*`[,`*arg2*`...]`

General input rules for the `-K` option can be found on "Calling syntax and general rules". The following entries are possible as *arg* arguments to control the C++ frontend:

`using_std`

`no_using_std`

These arguments determine the use of the C++ library functions for which names have been defined in the standard name space `std`.

If `-K using_std` is specified, the compiler behaves as if the following lines were entered at the start of a compilation unit:

```
namespace std{}

using namespace std;
```

`-K using_std` is the default in extended C++ V3 mode (`-X v3-compatible -X nostrict`).

`-K no_using_std` is the default in strict C++ V3 mode (`-X v3-compatible -X strict`), in C++ 2017 mode (`-X 2017`) and in C++ 2020 mode (`-X 2020`) and the only possible behavior in Cfront C++ mode (`-X v2-compatible`).
If `-K no_using_std` is set in the mode C++ V3, C++ 2017 or C++ 2020, the source program must contain the directive `using namespace std;` before the first call to a C++ library function or the names must be qualified appropriately.

`wchar_t_keyword`
`no_wchar_t_keyword`

These arguments can be used to define whether `wchar_t` is recognized as a keyword.

`-K wchar_t_keyword` is the default in mode C++ V3 and the only possible behavior in mode C++ 2017 and C++ 2020. In this case, the preprocessor macro `_WCHAR_T` is defined.

`-K no_wchar_t_keyword` is the default and the only possible behavior in the Cfront C++ mode.

`bool`
`no_bool`

These arguments can be used to define whether `bool` is recognized as a keyword.

`-K bool` is the default in mode C++ V3 and the only possible behavior in mode C++ 2017 and C++ 2020. In this case, the preprocessor macro `_BOOL` is defined.

`-K no_bool` is the default and the only possible behavior in the Cfront C++ mode.

`old_for_init`
`new_for_init`

These arguments define how an initialization directive in `for` and `while` loops is to be handled.

`-K old_for_init`

Specifies that an initialization directive has the same scope as the entire loop. This is the default setting in the Cfront C++ mode.

`-K new_for_init`

Specifies the new ANSI C++-compliant scope rule, which surrounds the entire loop in its own implicitly generated scope.This is the default setting in mode C++ V3 and the only possible behavior in mode C++ 2017 and C++ 2020.

`no_old_specialization`
`old_specialization`

These arguments are only relevant in the modes C++ V3, C++ 2017 and C++ 2020. They are used to specify whether template specializations need the new syntax `template<>`.

`-K no_old_specialization` is the default setting in mode C++ V3 and the only possible behavior in mode C++ 2017 and C++ 2020. In this case, the compiler implicitly defines the macro `__OLD_SPECIALIZATION_SYNTAX` with the value 0.

If `-K old_specialization` is specified, the the compiler implicitly defines the macro `__OLD_SPECIALIZATION_SYNTAX` with the value 1.

#### 3.2.6.2 Template options

The following options are only relevant in the modes C++ 2017, C++ 2020 and C++ V3 as templates are not supported in the Cfront C++ mode.

`-T none`
`-T auto`
`-T local`
`-T all`

These options control 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 class templates. These template types are combined under the generic term "template entity" below.

In all instantiation modes, the compiler generates all instances per compilation unit that are requested with the explicit instantiation directive `template` *declaration* or with the instantiation pragma `#pragma instantiate` *template entity.*

The remaining template entities are instantiated as follows:

`-T none`

No instances are generated unless explicitly requested.

`-T auto` (default)
Instantiation is carried out globally for all compilation units by a prelinker. The prelinker is activated when an executable file is generated with the `CC` command or if the `-y` option (see "Options for selecting compilation phases") is specified. No instantiations are performed by the prelinker when generating a prelinked object file ( `-r` option). The principle of automatic instantiation is described in detail in section “Automatic instantiation”.

`-T local`

Instantiation is carried out per compilation unit. All template entities used in a compilation unit are instantiated, with internal linkage for the functions generated in the process. This provides a very simple mechanism for starting template programming. The compiler instantiates the functions required in each compilation unit as local functions. The program links them and then terminates correctly. However, this method produces a large number of copies of the instantiated functions and is therefore not recommended for production. And for the same reasons this mode is not suitable if one of the templates contains `static` variables.

### Warning:

The `basic_string` template contains a `static` variable which is used to represent an empty string. If you use the `-T local` option and select the `string` type from the library the empty string is no longer recognized. Try to avoid using this combination as it can lead to serious problems.

`-T all`

Instantiation is carried out per compilation unit. All template entities that are used or declared in a compilation unit are instantiated. All member functions and static variables of a class template are instantiated regardless of whether they are used or not. Function templates are also instantiated if they are just declared.

`-T add_prelink_files,` *pl_file1*[`,` *pl_file2...*]

This option can be used to specify objects and libraries that are taken into account as described below when the prelinker determines the instances to be generated:

*pl_filei* is the name of an object file (`.o` file) or a static library (`.a` file).

- If an object file or library *pl_filei* contains the definition of a function or static data member, no instance of a template entity that is a duplicate this is generated.
- If an object file or library *pl_filei* needs instances for template entities, these instances are not generated.

*Problem*

The `libX.a` and `libY.a` libraries contain references to the same template instances. Duplicates occur if the objects of the two libraries were each preinstantiated with the `-y` option.

In such cases, the prelinker must be informed that symbols are defined elsewhere and it should therefore not generate any instances. The `-T add_prelink_files` option is provided for this purpose.

*Solution*

The objects of the `libX.a` library are initially preinstantiated with the `-y` option. Then the objects of the `libY.a` library are preinstantiated, using the `-T add_prelink_files,libX.a` option to inform the prelinker to consider `libX.a` and ensure that no duplicate of this is generated.

`-T max_iterations,`*n*

In automatic instantiation mode ( `-T auto` ), this option specifies the maximum number *n* of prelinker runs. The default is *n* = 30. The number of prelinker runs is unlimited if *n* is set to the value 0.

`-T etr_file_none`
`-T etr_file_all`
`-T etr_file_assigned`

These three options are used to control the creation of an ETR file `file.etr` (ETR=Explicit Template Request) for the application of explicit template instantiation (see section “Generating explicit template instantiation statements (ETR files)”).

**Warning:**

The `etr_file_all` and `etr_file_assigned` options are ignored if they are used in conjunction with the preprocessor options `-P / -E / -M`.

`-T etr_file_none`

This is the default setting and suppresses the output of instantiation information.

`-T etr_file_all`

This option outputs all the possible template information.

`-T etr_file_assigned`

This option ensures that only those instantiation templates assigned by the prelinker are output.

`-T [no_]definition_list or -T [no_]dl`

This options allows for internal communication between the front end and the prelinker during the recompilation phase of the automatic template instantiation. You will find more information in the section "Automatic instantiation".

`-K` *arg1* `[,` *arg2* `...]`

General input rules for the `-K` option can be found in section "Calling syntax and general rules". The following entries are possible as *arg* arguments to control template instantiation:

`assign_local_only`
`no_assign_local_only`

These arguments determine whether or not instantiation assignments are only supported locally. If `-K assign_local_only` is set, the following applies:

- Instantiations can only be assigned to object files that are located in the current directory (local files).
- Instantiations can only be assigned to an object file if the current directory at the time of the instantiation matches the current directory at compile time.

*Example*

```
cd dir1          # The current directory when
CC -c test1.c    # compiling test1.c is dir1
cd ../dir2       # The current directory when
CC -c test2.c    # compiling test2.c is dir2
cd ../dir1       # The current directory for the
                 # prelinker is dir1
CC -K assign_local_only -o test test1.c ../dir2/test2.o
```

In this example, the assignment of instantiations is restricted to the local object file `test1.o` .

`-K no_assign_local_only` is the default setting.

`implicit_include`
`no_implicit_include`

These arguments determine whether the definition of a template is implicitly included (see section "Implicit inclusion").

`-K implicit_include` is the default setting.

`instantiation_flags`
`no_instantiation_flags`

`-K instantiation_flags` is the default setting and causes special symbols to be generated for use by the prelinker during automatic instantiation.
If `-K no_instantiation_flags` is set, no such symbols are generated, so the object size is reduced. Consequently, no automatic instantiation with `-T auto` is possible in this case.

### 3.2.7 Optimization options

If none of the following optimization options are specified, the compiler does not carry out any optimization. This corresponds to the SDF `LEVEL=*LOW` option.

The separate optimization options and their effects are described in detail in the C/C++ User Guide [4] in the section “Optimization”.

`-O`
`-F O2`

These options enable the standard optimization of the compiler. The only difference between the two options is that every optimization strategy is internally executed only once for `-O`, but several times for `-F O2`. Consequently, the overall compile time at
the `-O` optimization level is significantly less than the compile time required for the “highly-optimized” `-F O2` level.

The following standard optimizations are performed by the compiler:

- calculates constant expressions at compilation time
- optimizes the indexing in loops
- eliminates unnecessary assignments
- propagates constant expressions
- eliminates redundant expressions
- optimizes jumps and unconditional jump commands

In addition, registers are also optimized.

In contrast to the SDF option (where the optimization level can be set as *HIGH or *VERY-HIGH without parameters), loop unrolling is disabled here.

If the standard optimization has not been explicitly enabled with `-O` or `-F O2`, it is automatically activated at level `-O` if the `-F loopunroll` (loop expansion) or `-F i`, `-F inline_by_source` (inline substitution of user-defined functions) options are specified.

`-F I`[*name*]

This option allows you to specify the C library functions for which the implementation in CRTE can be assumed. This permits better optimization of the program.
When -F I is specified without *name*, all calls for known C library functions are handled separately.
When the -F I option is not specified, no call is handled separately.
When -F I*name* is specified (without a separating blank), only the *name* function is handled separately.
If several functions are to be handled separately, the -F I*name* option must be specifed several times.
The -F I option can be specified independently of normal optimization.

The compiler achieves the greatest effect by means of inline generation of a function. In this case the function code is inserted directly in place of the function call. This eliminates time-consuming management activities required of the runtime system (e.g. saving and restoring registers or returning from the function), thus shortening program runtime.

The following C library functions can be generated inline:

| | |
|---|---|
| abs | strcat |
| fabs | strlen |
| labs | strcmp |
| memcmp | strncmp |
| memcpy | strcpy |
| memset | |

Functions which are generated inline cannot be replaced by other functions at linkage time, nor can they be used as test points when debugging with AID.

The default compiler optimization does not have to be activated for generating C library functions inline.

The compiler knows the semantics of the CRTE library functions. With the `-F I`*name* option you command the compiler to generate optimized functions that observe the CRTE library function semantics. If no name is specified, then the compiler should use all its knowledge of the CRTE functions (the compiler knows of about 150 functions).

Functions which are not generated inline are retained as calls. However, optimizations are possible which are not feasible with the user functions. For example, the compiler can use the information that the `isdigit()` function has no side effects.

Some functions are highly specialized since they are generated to be completely inline. For these functions the compiler creates the code directly without passing it to CRTE. These functions are listed in the table above.

In some cases this optimization may not be desired. If the program is to be debugged, you may need to set a breakpoint in such a function. This is not possible for functions generated to be completely inline, or more precisely, you can set a breakpoint, but it will not be reached. The code generated by the compiler is used and not the function where the breakpoint was set.

Another case is when a function is defined with a name that is already known to the compiler. In most cases this function will use semantics different from the CRTE semantics. If a conflict between such a function and this option arises, then all calls assume the CRTE semantics. Warning CFE2067 is output in this case.

Note that the CRTE semantics are used in every compilation unit. The warning is only output in the compilation unit that contains the private definition.

`-F i`[*name*]
`-F inline_by_source`

These alternative options control the inline substitution of user-defined functions. As in the case of some C library functions from the standard library (see `-F I`), each call to an inline function is replaced by the corresponding function code. This saves the code sequence for the call and return and thus results in faster execution times. Specifying the `-F i`, `-F i`*name* or `-F inline_by_source` options automatically activates the standard optimization (`-O`) as well, unless `-F O2` was explicitly set.

`-F i` and `-F i`*name*

When `-F i` is specified with or without *name*, the compiler selects functions for inline substitution in accordance with its own criteria. Any existing inline pragmas and C++-specific inline functions in the source program are automatically considered by the compiler in the search for suitable candidates (see also `-F inline_by_source`).

If *name* is specified (without a leading blank!), the function *name* will also be inlined. If multiple user-selected functions are to be considered by the compiler for inline substitution, the `-F i`*name* option must be specified more than once.
The `-F i`*name* option is ignored for C++ compilations, i.e. by the `CC` command.

`-F inline_by_source`

If this option is specified, only the following user-defined functions are inlined:

- For C compilations (`cc`, `c11`, `c89`): C functions declared with the `#pragma inline` *name* directive (see also the section “inline pragma” in the C/C++ User Guide [4]). The inline pragma in not supported in C++.
- For C++ compilations (`CC`): the C++-specific inline functions. These are the functions defined within classes and functions with the `inline` attribute.

**Note on inline functions in C++**

The inline substitution of C++-specific inline functions is also performed when optimization is not enabled or if the `-F i` or `-F inline_by_source` options are not set. This can be suppressed with the `-F no_inlining` option.

`-F loopunroll[,`*n*`]`

This option controls loop unrolling. Multiple unrolling of the loop body speeds up loop execution. This optimization option is not used by default. If it is specified, it automatically activates the standard optimization (`-O`), unless `-F O2` was explicitly set.

If `-F loopunroll` is specified without *n*, the compiler unrolls loop bodies four times.You can use *n* to specify your own unroll factor, where *n* can be set to a value between 1 and 100.

Specifying `-F loopunroll[,`*n*`]` does not guarantee that the optimizer will always carry out the loop expansion. The optmizer decides whether or not to run the loop expansion on the basis of the loop structure and specified factor *n*.

`-F no_inlining`

This option suppresses the inline substitution of C++-specific inline functions, which is performed by default even if the `-F i` or `-F inline_by_source` options have not been specified.
If the `-F no_inlining` option in combination with the `-F i` or `-F inline_by_source` option, the last specification on the command line applies.
If `-F no_inlining` is the last specified option, even the originally requested inlining of user-defined C functions is suppressed (however, the implicitly set `-O` optimization remains enabled).
The inlining of C library functions set with the `-F I` option is not affected by `-F no_inlining`.

### 3.2.8 Options for controlling object generation

`-K` *arg1*[,*arg2*...]

General input rules for the `-K` option can be found on "Calling syntax and general rules".
The following entries are possible as *arg* arguments to control object generation:

*Assembler commands for subroutine entries*

```
subcall_basr
subcall_lab
```

`-K subcall_basr` (default)
The BASR command is generated by default.

`-K subcall_lab`

The processor-independent LA and B assembler commands are generated. Programs using these commands will run on all 7500 systems.

**Warning:** This option is not allowed in the modes C++ 2017, C++ 2020 and C++ V3.

*Generating the ETPND area*

The following options are used to ignore the `#pragma` directive for generating an ETPND area (see the section “ETPND pragma” in the C/C++ User Guide [4]) or to define the date format of the ETPND area.

```
no_etpnd
calendar_etpnd
julian_etpnd
```

`-K no_etpnd` (default)
By default, no ETPND area is generated.

`-K calendar_etpnd`

The date format in the ETPND area is defined as follows:8 bytes calendar date - 4 bytes load address.

`-K julian_etpnd`

The date format in the ETPND area is defined as follows:6 bytes calendar date - 3 bytes Julian date - 4 bytes load address.

*Generating the entry code for function calls*

```
ilcs_opt
ilcs_out
```

`-K ilcs_opt` (default)
The ILCS entry code is generated inline. This speeds up the runtime of the created object.

`-K ilcs_out`

A branch to the ILCS entry code for function calls in the runtime system is generated. This reduces the module code volume.

*Handling enum data*

`enum_value`
`enum_long`

`-K enum_value` (default)
By default, the `enum` data is handled as `char`, `short` or `long`, depending on the value range.

`-K enum_long`

`enum` data is always handled as type `long` objects.

*Generating the entry names with LLMs*

`llm_convert`

`llm_keep`

`-K llm_convert` (default)
By default, underscore characters are converted to dollar signs when entry names are generated.

`-K llm_keep`

The underscore characters are retained when generating entry names.

The underscore character conversion applies to all external symbols in the C language modes and only the symbols declared with `external "C"` in the C++ language modes (not the entry names of the C library functions). The underscore character is always retained when coding external C++ symbols.

`no_llm_case_lower`

`llm_case_lower`

`-K no_llm_case_lower` (default)
By default, lowercase letters are converted to uppercase when entry names are generated.

`-K llm_case_lower`

The lowercase letters are retained when entry names are generated.

The lowercase to uppercase conversion applies to all external symbols in the C language modes and the Cfront C++ mode, and only the symbols declared with `extern "C"` in the modes C++ 2017, C++ 2020 and C++ V3. Lowercase letters are always retained when coding external C++ symbols in the modes C++ 2017, C++ 2020 and C++ V3.

### Warning:

The C library functions are only available in full if one of the following combinations of options has been specified:

- `-K llm_convert` and `-K no_llm_case_lower`
- `-K llm_keep` and `-K llm_case_lower`

`csect_suffix=` *suffix*
`csect_hashpath`

These options specify how CSECT names are formed. By default the CSECT name is derived from the module name, and the module name is derived from the source name as long as it is not explicitly specified.
The options can be sued to generate different CSECT names when the object names are the same.

With the help of these two options, a 30 character long string is created as the basis for the real CSECT names. This basis can be output using `'-K verbose / -v'`.

The basis is changed in the usual manner by:

- converting all lower case letters to upper case letters ,
- converting all special characters such as '_' or '.' to '$' and
- adding '&@' or '&#' to generate real CSECT names.

With the help of these options you select different suffixes that are appended to the object names. If an object name is longer than 30 characters (not including the length of the suffix), then it is truncated.

`-K csect_suffix=`

With this option you specify a user-defined suffix. A maximum of 10 characters are used.

`-K csect_hashpath`

With this option you generate a 7 character long string from the full object path (including '..'; links are not expanded). This character string is used as the suffix.

*Storing const objects*

```
roconst
no_roconst
```

`-K no_roconst` (default)
By default, type `const` objects are stored in the data module (WRITEABLE). This allows the values to be overwritten if the `const` attribute is removed with a `cast` operator.

`-K roconst`

Type `const` objects are stored in the code module (READ-ONLY). The constants cannot be overwritten even if the `const` attribute is removed with a `cast` operator.

**Caution:** only global or local `static` constants are affected. Local `auto` variables with the `const` type attribute cannot be stored in the READ-ONLY area.

*Storing string constants*

```
no_rostr
rostr
```

`-K no_rostr` (default)
By default, string constants are stored in the data module (WRITEABLE). This allows the values to be overwritten if the `const` attribute is removed with a `cast` operator.

`-K rostr`

String constants and aggregate initialization constants are stored in the code module (READ-ONLY).

*Floating-arithmetics in /390 and IEEE formats*

```
no_ieee_floats
ieee_floats
```

`-K no_ieee_floats` (default setting)
By default, floating point data types and operations in /390 format are used.

`-K ieee_floats`

The IEEE format is used for floating-point data types and operations. This applies to all variables and constants of the `float`, `double` and `long double` data types inside the C/C++ programms.

### Important:

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 floatingpoint 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.
- C++ library functions do no support the IEEE format and must therefore be replaced with C functions where necessary (see the example below).

### Prerequisites for using the IEEE-Format:

- 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>`.
- CRTE contains some C library functions which use the IEEE format for floatingpoint arithmetics. To ensure that the IEEE function names are correctly used, you should specify the following two options for the option `ieee_floats`:

  ```
  -K llm_keep
  -K llm_case_lower
  ```

*Generating shareable code*

```
no_share
share
```

`-K no_share` (default)
By default, the compiler does not generate any shareable code.

`-K share`

The compiler generates shareable code comprising a shared code CSECT and a non-shared data CSECT.Modules containing shareable code can only be meaningfully further processed in a BS2000 environment (SDF).

*Storing workspace variables*

```
workspace_static
workspace_stack
```

`-K workspace_static` (default)
By default, workspace variables are stored in the static data area.

`-K workspace_stack`

The data required for workspace variables is stored on the stack.

*Multiple definition of externally visible variables*

`external_multiple`
`external_unique`

`-K external_multiple`

An externally visible variable that is defined in several modules is only assigned one memory area.
In order to achieve this, the variable may not be statically initialized in any of the definitions. The compiler places the memory for this variable in the COMMON area. If the variable is statically initialized during definition, the memory is placed in the data area. It is then not possible to assign it just one memory area. This behavior is the default in K&R C mode.

`-K external_unique`

Externally visible variables may only be defined in just one module and must be declared as `external` in all other modules. The memory space for such variables is placed in the data module of the object in which the variable was defined.This behavior is the default in the language modes C89, C11 and all C++ language modes. The default may not be changed in the C++ language modes.

*Length of external C names*

The following options define the length of external C names and affect all external symbols in the C language modes, but only the symbols declared with `extern "C"` in the C++ modes (not the entry names of C library functions).

`c_names_std`
`c_names_unlimited`
`c_names_short`

`-K c_names_std` (default)
By default, external C names may be a maximum of 32 characters long. Longer names are truncated by the compiler to 32 characters. Only 30 characters are allowed when generating shareable code (`-K share`).

`-K c_names_unlimited`

Names are not truncated. In this case, the compiler generates entry names in EEN format. EENs can have a length of up to 32000 characters. Modules containing EENs are saved by the compiler in LLM Format 4. More detailed information on how LLMs in Format 4 are processed further can be found on "Link editor options" (`-B extended_external_names`). EENs are not supported in the Cfront C++ mode.

`-K c_names_short`

External C names are truncated to 8 characters.

**Note**

Options which affect the length of external names also affect the names of static functions as the compiler handles the names of static functions like the names of external functions.

### 3.2.9 Debug option

`-g`

The compiler generates additional information (LSD) for the AID debugger. By default, no debugging information is generated.

A program, i.e. an executable file generated by the link editor, can be debugged in the POSIX shell with the `debug` command. Once this command is input, the user is in BS2000 mode (indicated by %DEBUG/). The AID commands are then input as described in the manual “AID Debugging of C/C++ Programs” [11]. The POSIX shell is the current environment after the program is terminated.

A description of the `debug` command can be found in the manual “POSIX Commands” [3].

### 3.2.10 Runtime options

The following options can be used to influence runtime behavior when compiling the module containing the `main` function. When compiling other modules These options have no effect.

`-K` *arg1*[,*arg2*...]

General input rules for the `-K` option can be found on "Calling syntax and general rules".
The following entries are possible as *arg* arguments to control the runtime behavior:

`integer_overflow`
`no_integer_overflow`

`-K integer_overflow` (default)
By default, the program mask is set to X‘0C‘ in compliance with the ILCS convention.

`-K no_integer_overflow`

The program mask is set to X’00’.

The two program masks have the following effect:

| | X‘0C‘ | X‘00‘ |
|---|---|---|
| Fixed point overflow | allowed | suppressed |
| Decimal overflow | allowed | suppressed |
| Exponent underflow | suppressed | suppressed |
| Mantissa null | suppressed | suppressed |

*Notes*

The ILCS program mask may not be changed with mixed code!

The `-K integer_overflow` option does not affect he selection of generated commands. The result is that permitting INTEGER-OVERFLOW does not necessarily mean that an overflow is triggered in all cases.

`prompting`
`no_prompting`

`-K prompting` (default)
If the program is called from the BS2000 environment (SDF), a prompt line is output in which parameters can be specified for the `main` function or for redirecting the standard I/O stream.

`-K no_prompting`

No prompt line is output.

This option has no effect if the program is started from the POSIX shell as the parameters are always specified in the command line in this case.

`statistics`
`no_statistics`

`-K statistics` (default)
The used CPU time is output when a program generated with this option is terminated. However, this only occurs when the program is transferred to BS2000 and is started there.

`-K no_statistics`

The used CPU time is not output.

`stacksize=`*n*

The `-K stacksize` option can be used to input a number *n* (8 to 99999) which defines the number of kilobytes to be reserved for the first segment of the C runtime stack. The default is 64 kilobytes.

`environment_encoding_std`
`environment_encoding_ebcdic`

These options enable the encoding of external strings, such as arguments of *main* and environment variables, to be controlled.

`–K environment_encoding_std` (default).
The external strings are encoded in the way specified in the options `-K literal_encoding_ascii`, `-K literal_encoding_ascii_full`, `-K literal_encoding_ebcdic` or `-K literal_encoding_ebcdic_full` (see "Common frontend options in C and C+").

`-K literal_encoding_ebcdic_full`
This option is offered for reasons of compatibility. Despite `-K literal_encoding_ascii` or `–K literal_encoding_ascii_full` being specified, external strings are encoded in EBCDIC.

The table below explains the option combinations and the encoding of the external strings:

| | **environment_encoding_std** | **environment_encoding_ebcdic** |
|---|---|---|
| **literal_encoding_ebcdic*** | EBCDIC | EBCDIC |
| **literal_encoding_ascii*** | ASCII | EBCDIC |

### 3.2.11 Link editor options

The following link editor options are not evaluated if one of the `-c`, `-E`, `-M` or `-P` options are specified (terminate the compiler run after compilation or after the preprocessor run, see "Options for selecting compilation phases").

`-B extended_external_names`
`-B short_external_names`

This option is required if the program to be linked is to run on very old systems. LLMs with long names (EEN) cannot be processed on systems with BS2000/OSD versions 1 to 3 or the BLSSERV product with version less than 2.0. For such systems, the generated element must have an LLM format 1.

EENs, i.e. untruncated external C++ symbols are generally contained in modules that were generated with the compiler in mode C++ 2017, C++ 2020 or C++ V3.
Untruncated external C symbols are generated only if the `-K c_names_unlimited` option is specified at compilation (see "Options for controlling object generation").
If this is the case, even external C symbols are not truncated to 32 bytes by the compiler.
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 mode C++ 2017, C++ 2020 or C++ V3 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 link editor always generates LLM Format 1 in accordance with the input format in this case.

By default, the link editor 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 the link editor.

`-B extended_external_names`

This entry is supported for compatibility reasons only.

`-B short_external_names`

This entry is needed if the link editor is to generate LLM Format 1.
In this case, all symbols with long names (EEN) must be satisfied. There is no EEN in the generated LLM. If an open reference remains here, it remains open in the long run and cannot be satisfied subsequently.

*Summary of generated LLM formats*

| Input format | Option -B | Output format |
|---|---|---|
| LLM 1 | no entry / extended_external_names / short_external_names | LLM 1 |
| LLM 4 (EEN) | no entry / extended_external_names | LLM 4 |
| | short_external_names | LLM 1 |

`-d y`
`-d n`
`-d compl`

This option affects linking the C runtime system.

By default, i.e. if the option is either not specified or `-d y` is specified, a RESOLVE for the standard C library `libc.a` is issued to the SYSLNK.CRTE.PARTIAL-BIND library. Instead of the complete C runtime system being linked in, only a connection module is linked that satisfies all unresolved external references to the C runtime system. The C runtime system itself is loaded dynamically at runtime, either from class 4 memory if the C runtime system has been preloaded, or from the SYSLNK.CRTE.

If `-d n` is specified, the C runtime system is linked in completely from the SYSLNK.CRTE.

The complete partial bind method of the CRTE is supported with the `-d compl` option. To accomplish this the SYSLNK.CRTE.COMPL library is linked in.
You will find a detailed description of the complete partial bind method in the "CRTE" [5] manual.

In the mode C++ V3 the special library SYSLNK.CRTE.CPP-COMPL is linked instead of the standard libraries SYSLNK.CRTE.RTSCPP and SYSLNK.CRTE.STDCPP. This library is also used instead of SYSLNK.CRTE.TOOLS.

*Note*

The complete partial bind method is not supported in the CFRONT-C++ mode. The option `-d compl` is reset to `-dy` in this case.
In the modes C++2017 and C++ 2020 the "complete partial bind" technique is not supported. In this case, the `-d compl` option is rejected with an error.

`-K` *arg1*[,*arg2*...]

General input rules for the `-K`-option can be found in section "Calling syntax and general rules".
The following entries are possible as *arg* arguments to control the link editor:

`link_stdlibs`
`no_link_stdlibs`

`-K link_stdlibs` is the default setting and causes certain standard libraries to be automatically linked in (see also the `-l` option). This means that the corresponding `-l` options are automatically set for these libraries:

1. Only with the `CC` command

   `-l Cxx` in C++ 2017 and C++ 2020 mode
   `-l Cstd` in C++ V3 mode
   `-l C` in Cfront C++ mode

2. Always

   `-l c`

If `-K no_link_stdlibs` is specified, the above libraries are not linked in automatically. `-K no_link_stdlibs` is set automatically, if a prelinked object file is generated with the `-r` option (see "Link editor options").

`-l` *x*

This option instructs the link editor to search the library named `lib` *x*.a for resolving external references via autolink. By default the link editor searches for the library in the following directories in the order given below:

1. The directories specified with `-L`
2. Either the directories specified with the `-Y P` option (see "Link editor options") or the standard `/usr/lib` directory.

`-l` *x* falls into the category of operands and can also be specified with `--` after options have been input (see also "Operands" (Calling syntax and general rules)).

The standard libraries of the C and C++ runtime system are not installed in the POSIX file system `/usr/lib` directory, they are stored as PLAM libraries in BS2000.

Assignment of standard libraries `-l` *x* to the BS2000 PLAM libraries:

| ***x*** | **Library name** | **Contents** |
|---|---|---|
| `c` | `SYSLNK.CRTE.PARTIAL-BIND` | Connection module for loading the C runtime system dynamically (default) |
| | `SYSLNK.CRTE` | Separate modules for completely linking the C runtime system (with `-d n`) |
| `m` | see `c` | |
| `C` | see `c` | |
| | `SYSLNK.CRTE.CFCPP` | Cfront C++ runtime system |
| | `SYSLNK.CRTE.CPP` | Cfront C++ library for input/output and complex mathematics |
| `Cstd` | see `c` | |
| | `SYSLNK.CRTE.RTSCPP` | C++ V3 runtime system |
| | `SYSLNK.CRTE.STDCPP` | Standard C++ V3 library |
| `Cxx` | see `c` | |
| | see Cxx1 or Cxx2 | C++ 2017 or C++ 2020 library, according to option `-K library_version` |
| `Cxx1` | see `c` | |
| | `SYSLNK.CRTE.CXX01` | C++ 2017 library in library version 1 |
| `Cxx2` | see `c` | |
| | `SYSLNK.CRTE.CXX02` | C++ 2017 or C++ 2020 library in library version 2 |
| `RWtools` | `SYSLNK.CRTE.TOOLS` | C++ V3 library Tools.h++ |

The link editor only resolves the unresolved external references from these PLAM libraries if the `-l x` option is used and not if the path name is specified explicitly (e.g. `/usr/lib/libRWtools.a`) using the *file*.*suffix* operand (see "Calling syntax and general rules")!
When calling the compiler in the C language modes, `-lc` is implicitly added as the last -l option, when calling in Cfront-C++ mode the implicitly added option is `-l C`, when calling in C++ V3 mode it is `-l Cstd` and when calling in C++ 2017 or C++ 2020 mode it is `-l Cxx` (does not apply to `-K no_link_stdlibs`).

The order and position in which the `-l` options and any object files (or source files from which the compiler generates object files) are specified in the command line is significant to the link process.
For example, the program would be linked correctly with the `CC -v3-compatible test.c -l RWtools` command, but the `CC -v3-compatible -l RWtools test.c` command would lead to an error.

`-l BLSLIB`

This option instructs the link editor to search PLAM libraries which were assigned with the BLSLIB*nn* (00 <= *nn* <= 99) shell environment variable.
The environment variables must be supplied with the library names and exported with the POSIX `export` command before the compiler is called. The libraries are searched in ascending order *nn*.

All libraries specified with BLSLIBxx are searched cyclically. They are processed by BINDER as if they had been specified as a list in **one** RESOLVE directive to the link editor.

`-l BLSLIB` falls into the category of operands and can also be specified with `--` after options have been input (see also “Operands (Calling syntax and general rules) ”).

*Example*

The library assigned with BLSLIB00 contains unresolved external references to the library assigned with BLSLIB01 and this in turn contains unresolved external references to the BLSLIB00 library (reverse references).

```
BLSLIB00='$RZ99.SYSLNK.CCC.999'

BLSLIB01='$MYTEST.LIB'
export BLSLIB00 BLSLIB01
c89 mytest.o -l BLSLIB
```

`-L dir`

The *dir* option can be used to specify an additional directory in which the link editor is to search for libraries specified with `-l` options. By default, only the `/usr/lib` directory is searched for the libraries. A directory specified with `-L` is searched before the standard directory `/usr/lib` or before the directories specified with the `-Y P` option. The order in which the `-L` options are specified in the command line determines the link editor search order.
This option only falls into the category of operands for the `cc`, `c11` and `CC` commands and can therefore only be specified with `--` after the options have been input (see also “Operands” (Calling syntax and general rules)).

`-r`

Several object files can be prelinked to form a single object file with this option. A prelinked object file is not executable, but contains the relocation information required to repeat a linkage run.
The following options are set implicitly when prelinking with `-r`:
`-K no_link_stdlibs` and `-B extended_external_names` . This means that the C/C++ standard libraries are not linked and that LLM Format 4 is generated in the case of long C and C++ names (EENs).
The `-K link_stdlibs` and `-B short_external_names` options, if specified, are ignored. No instantiations by the prelinker are performed when generating a prelinked object file.Unresolved references do not cause error messages to be output.
The prelinked object file is given the name `a.out` or the name specified with the `-o` option. The object file can only be meaningfully further processed (linked) if the name of the prelinked object file is suffixed with `.o` or with a suffix which can be defined with the `-Y F` option (see "General options").

`-s`

Symbol table information is stripped from the output file. The sections with additional information for troubleshooting and with line numbers and associated offset information are also removed.
The option is ignored if debugging information for AID is simultaneously requested (`-g` options). It is also ignored in all C++ modes as the symbol tables are required at runtime for global initialization. The option corresponds to the link editor SAVE-LLM SYMBOL-DICTIONARY=*NO directive.

`-Y P,` *dir1*[`:` *dir2*...]

Instructs the link editor to search for libraries in the directories specified with *dir* last. Without this option, the last directory to be searched is the standard directory `/usr/lib`.

`-z nodefs`

This option is only supported when linking C programs (`cc`, `c11`, `c89`). Specifying this option allows a C program to be linked in which all external references to the standard C library `libc.a` remain unresolved, i.e. no RESOLVE is issued to the library SYSLNK.CRTE.PARTIAL-BIND or SYSLNK.CRTE. The unresolved external references are resolved dynamically at runtime from the C runtime system which is preloaded into class 4 memory.

If this option is used, unresolved externals to user modules are ignored and not reported. Information on unresolved external references is only output when the program is loaded.

`–z dup_ignore`
`–z dup_warning`
`–z dup_error`

These options control the behavior of duplicates during linking.

`–z dup_ignore`

Duplicates are ignored during linking.This is the default.

`–z dup_warning`

Duplicates during linking result in a warning being issued.

`–z dup_error`

Duplicates during linking result in an error.

A program which contains duplicates cannot be executed in POSIX. It immediately returns with error code 127.
The compiler cannot specify any duplicates which are found by name. They are output in the message BLS0339 when the program is loaded in BS2000 interface (SDF) with `/LOAD-EXECUTABLE-PROGRAM`.
The modules which contain the duplicates can be found using the BINDER listing (see -N binder, "Options for outputting listings and CIF information") and possibly also with the support program `nm`.

### 3.2.12 Options for controlling message output

More detailed information on the compiler message output can be found in the C/C++ User Guide [4] in the section “Structure of the compiler messages”.

`-R diagnose_to_listing`

This option allows you to sort diagnostic information (normally sent to `stderr`) as a special “result listing” and to copy this to the end of the listing file. Note: the messages are sorted according to their message weighting.

`-R limit,`*n*
This option defines the maximum number of errors tolerated by the compiler before it aborts the compilation run. Notes and warnings are counted separately. The default value is *n* = 50. If *n* = 0, the compiler will attempt to continue compiling as long as possible, regardless of the number of errors that have occurred.

`-R min_weight,`*min_weight*

This option defines the minimum error weight (i.e. severity code) as of which diagnostic messages from the compiler are to be output to the standard error output `stderr`.
`-R min_weight,warnings` is the default setting. The following entries are possible for

*min_weight*:

| | |
|---|---|
| `notes` | All messages are output, i.e. even the notes. |
| `warnings` | The output of notes is suppressed (default). |
| `errors` | The output of notes and warnings is suppressed. |
| `fatals` | The output of notes, warnings and errors is suppressed. |

`-R note,`*msgid*,[*msgid*...]
`-R warning,`*msgid*,[*msgid*...]
`-R error,`*msgid*,[*msgid*...]

These options can be used to change the default severity code of diagnostic messages. *msgid* is the corresponding message number. The severity code for fatal errors cannot be changed. This also applies to errors, unless they have been explicitly marked in the original message with an asterisk: `[*error]`. Depending on the language mode or the position in the code, the same message ID *msgid* can have a different severity code (warning or error).

`-R show_column`
`-R no_show_column`

This option determines whether the diagnostic messages of the compiler are generated in short or long form. `-R show_column` 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` is specified, the marked source program line is not output.

`-R strict_errors`
`-R strict_warnings`

This option will be ignored in the non-strict language modes (`-X nostrict`). `-R strict_warnings` is the default, which means that warnings are issued for language constructs that deviate from the ANSI/ISO standard, but do not represent a serious violation of the semantic rules defined therein (e.g. implementation-dependent language extensions; see the C/C++ User Guide [4]).
If `-R strict_errors` is specified, such cases are treated as errors with corresponding messages. Serious violations automatically lead to errors.

`-R suppress,`*msgid*,[*msgid*...]

Suppresses the output of the message with the message ID *msgid*. Some messages (e.g. fatal errors) cannot be suppressed. The message is still counted for the summary message. When an error is detected, no module will be generated. This is also the case when the output of the error is suppressed.

`-R use_before_set`
`-R no_use_before_set`

`-R use_before_set` is the default setting and causes warnings to be issued if local `auto` variables are used in the program before being assigned a value.
If `-R no_use_before_set` is specified, the output of such warnings is suppressed.

`-v`

The output of messages with this option is the same as for the option combination `-R min_weight,notes` and `-K verbose`.

`-w`

This option is a synonym for `-R min_weight,errors`.

### 3.2.13 Options for outputting listings and CIF information

`-N binder[,`*file*`]`

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 created only when an executable file or a prelinked object file (`-r`) is generated. If *file* is not specified, the BINDER listings are written to an output file *file*`.lst`, where *file* is the name of the executable or the prelinked object file (`a.out` or the name defined with the `-o` option). *file* can be used to specify some other output file name. The `-N binder` option is ignored if specified in combination with any of the options `-c`, `-E`, `-M`, `-P` or `-y`.

`-N cif,`[*output-spec*]`,`*consumer1*[`,`*consumer2* ...]

(*output-spec* is a file or a directory).
The compiler generates a CIF (Compilation Information File) containing information for the specified *consumers*. If *output-spec* is not specified, the CIF is written into a separate file named *sourcefile*`.cif` for each compiled source file. A different output file name can be defined with *output-spec*. In this case, only one source file can be compiled. The global listing generator `cclistgen` is provided to further process the generated CIF information (see "Global listing generator (cclistgen)").
The following entries can be made for *consumer*:

`option` or `lo` (options)

`prepro` or `lp` (result of the preprocessor)

`source_error` or `ls` (errors in the source program)

`data_allocation_map` or `lm` (addresses)

`cross_reference` or `lx` or `xref` (references)

`object` or `la` (object code)

`project` or `lP` (project information, only with the `CC` command)

`summary` or `lS` (statistics)

`ALL`

If `ALL` is specified, all possible CIF information is generated, e.g. when the compiler run is terminated after the preprocessor phase (`-E`, `-P` options), CIF information for an options, preprocessor and statistics listing. The CIF may be very large if `ALL` is specified!

`-N` *listing1*[`,`*listing2*...]

The compiler writes the listings requested with this option either into a separate*sourcefile*`.lst` file for each compiled source file or for all compiled source files into the listing file `file` specified with the `-N output` option.
When the maximum number of errors is reached (controlled by `-R limit`), no source program information will be output to the source/error list. At this point the source/error list can no longer taken as a reliable guide to real error status.
The following entries are possible for *listing*:

`option` or `lo` (options listing)

`prepro` or `lp` (preprocessor listing)

`source_error` or `ls` (source program/error listing)

`data_allocation_map` or `lm` (map listing)

`cross_reference` or `lx` (cross-reference listing, see also the `-N xref` option)

`object` or `la` (object listing)

`project` or `lP` (project listing, only with the `CC` command)

`summary` or `lS` (statistics listing)

`ALL`

If `ALL` is specified, all possible CIF information is generated, e.g. when the compiler run is terminated after the preprocessor phase (`-E`, `-P` options), an options, preprocessor and statistics listing.

`-N map_structlevel,`*n*

This option controls the nesting level up to which structure elements are included in the list requested with the option `-N data_allocation_map`. Values from 0 to 256 inclusively can be specified for *n*.
Structure elements up to the nesting level specified by *n* are represented in the map listing. If a nesting level of 0 is specified, no structure elements are output.

Examples for the structure of compiler listings can be found in the section “Description of listings” of the C/C++ User Guide [4].

`-N output[,`[*output-spec*][`,`*layout*][`,`[*lpp*][,*cpl*]]]

This option can be used to specify the name of the output file (*output-spec*) or output directory in which the compiler listings for all source files are to be written.
If *output-spec* is not specified, a separate listing file *sourcefile*`.lst` is generated for each compiled source file.
If *output-spec* specifies an existing output directory, the name *output-spec/sourcefile.lst* is assigned by default. If this is not the case, the name *output-spec* is interpreted as the file name.

The following entries can be made for *layout*:

`normal` or `for_normal_print` (default)
The default page length is 64 lines and the line width 132 characters.

`rotation` or `for_rotation_print`
The page length for the compiler listing is defined as 84 lines and the line width as 120 characters.

*lpp* can be used to define a page length of from 11 to 255 lines per page.

*cpl* can be used to define a line width of from 120 to 255 characters per line.

*Note*

Since the output file is prepared for printing under POSIX, there are up to 3 control characters at the beginning of some lines in the file. In addition, every line is terminated with the printer control character for a carriage return. If the output file is printed out, then the line length is `cpl–1`.

`-N title,`*text*

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 `-N title` option applies to all compiler listings. In order to ensure that the desired text is transferred correctly, it is advisable to enter it within quotes ("*text*"). In the case of source and preprocessor listings, TITLE and PAGE pragmas (if any) override the `-N title` specification. See also the section on “Pragmas to control the layout of listings” in the C/C++ User Guide [4].

`-N xref,`*xrefopt1*[`,`*xrefopt2*...]

The sections contained in the cross-reference listing requested with `-N cross_reference` can be controlled with this option.
If the `-N xref` option is not specified, the cross-reference listing contains a list of the variables, functions and labels (equivalent to `-N xref,v,f,l`).
The cross-reference listing always contains a FILETABLE section containing the names of all files, libraries and members that the compiler used as sources.

If the `-N xref` option is specified, the cross-reference listing only contains the FILETABLE section and the sections requested with the *xrefopt* argument:

| | |
|---|---|
| `p` | List of the names processed by the preprocessor in `#include` and `#define` statements |
| `y` | List of the user-defined types (typedefs, structure, union, classes and counter types) |
| `v` | List of variables |
| `f` | List of functions |
| `l` | List of labels |
| `t` | List of templates (only with C++ compilations) |
| `o=`*str* | The order in which the separate sections are listed in the cross-reference listing. *str* is a string of up to 6 characters (letters for the lists shown above). The default is the order as shown above (i.e., `o=pyvflt`). If the order specified with `o=`*str* does not include all letters for the listings requested with `-N xref`, the omitted letters are implicitly appended to the end of *str* in the default order shown above. |

`-K` *arg1*[,*arg2*...]

General input rules for the `-K` option can be found on "Calling syntax and general rules (C/C++ POSIX Commands, #26)".
The following entries are possible as *arg* arguments to control the listing output:

```
include_user
include_all
include_none
```

These arguments control whether and which header files are mapped to the source program, preprocessor and cross-reference listings.

`-K include_user` is the default and only maps the user header files.

If `-K include_all` is specified, all header files are mapped, i.e. the standard header files and those of the user.

If `-K include_none` is specified, no header files are mapped.

`cif_include_user`
`cif_include_all`
`cif_include_none`

These arguments control whether and from which header files (also called include files) the CIF information for source/error, preprocessor and cross-reference listings is to be generated.

`-K cif_include_user` is the default and causes only the user-defined header files to be considered in the CIF.

If `-K cif_include_all` is specified, all header files, i.e., the standard headers and the user-defined headers, are considered in the CIF.

If `-K cif_include_none` is specified, none of the header files are considered in the CIF.

`pragmas_interpreted`
`pragmas_ignored`

These arguments control whether `#pragma` directives for controlling the layout of listings are evaluated (see also the section “Pragmas to control the layout of listings” in the C/C++ User Guide [4]).
`-K pragmas_interpreted` is the default.

## 3.3 Files

| | |
|---|---|
| *file*`.c/.C` | (`cc`, `c11`, `c89`) or C++ source file (`CC`) before the preprocessor run |
| *file*`.cpp/.CPP/.cxx/.CXX/.cc/.CC/.c++/.C++` | C++ source file before the preprocessor run |
| *file*`.i` | C source file (`cc`, `c11`, `c8`) after the preprocessor run |
| *file*`.I` | C++ source file) after the preprocessor run |
| *file*`.o` | LLM object file |
| *file*`.a` | Static library containing object files created with the `ar` utility |
| *file*`.lst` | File containing compilation listings |
| *file*`.cif` | File containing CIF information for further processing with the global listing generator `cclistgen` |
| *file*`.etr` | File containing explicit instantiation statements |
| *file*`.o.ii` | Information file for automatic template instantiation (used internally) |
| `a.out` | Executable file |
| *file*`.mk` | Preprocessor output file for further processing with `make` |
| `/var/tmp/`... | Temporary files used during compilation |

## 3.4 Environment variables

The cc/c11/c89/CC commands can be influenced with the following environment variables:

| | |
|---|---|
| LANG, LC_MESSAGES | Message output language |
| TMPDIR | Name of the directory in which temporary files are stored |
| BLSLIB*nn* | Assignment of PLAM libraries which the link editor is to search with autolink |
| IO_CONVERSION | Automatic conversion (IO_CONVERSION=YES) from ASCII to EBCDIC. |

## 3.5 Predefined preprocessor names

When the compiler is called with `cc`, `c11`, `c89` or `CC`, preprocessor macros and predicates are predefined, depending on the command and some options.

For a few macros the value is fixed. It cannot be changed, neither on the command line nor in the source (via `#define` or `#undef`). These macros are: `__cplusplus`, `__STDC__`, `__STDC_VERSION__` and `__SNI__STDCplusplus`.

### Predefined preprocessor macros (defines)

| | |
|---|---|
| `_BOOL` | In the language modes C++ V3, C++ 2017 and C++ 2020 with the option `-K bool`, which is the default |
| `__CGLOBALS_PRAGMA` | Always set |
| `__cplusplus` | In all C++ language modes:<br>== 1 in Cfront C++ mode<br>== 2 in extended C++ V3 mode<br>== 199612L in strict C++ V3 mode<br>== 201703L in C++ 2017 mode<br>== 202002L in C++ 2020 mode |
| `c_plusplus` | In all C++ language modes |
| `__CFRONT_V3` | In Cfront C++ mode |
| `__EDG_NO_IMPLICIT_INCLUSION` | In the modes C++ V3, C++ 2017 and C++ 2020, if implicit inclusion has been disabled with the option `-K no_implicit_include` within the framework of template instantiation |
| `__EXISTCGLOB` | Always set |
| `__HALF_TAG_LOOKUP` | Always set |
| `_IEEE` | Option `-K ieee_floats` |
| `LANGUAGE_C` | Always set |
| `_LANGUAGE_C` | Always set |
| `_LIBCPP_STD_VER` | In mode C++ 2020: the compiler will set it to the value `17`<br>In mode C++ 2017: the library headers will set it to the value `17` |
| `__LITERAL_ENCODING_ASCII` | Option `-K literal_encoding_ascii[_full]` |
| `__LITERAL_ENCODING_EBCDIC` | Option `-K literal_encoding_{ebcdic[_full]\|native}` |

| | |
|---|---|
| _LONGLONG | Option -K longlong |
| __OLD_SPECIALIZATION_SYNTAX | == 1 with the option -K old_specialization |
| _OSD_POSIX | Always set |
| __OSD_POSIX | Always set |
| __SHORT_NAMES | Option –K c_names_short |
| __SIGNED_CHARS__ | Option -K schar |
| __SMALL_VA_DCL | Always set |
| __SNI | In all C modes and in C++ V2 mode |
| __SNI_HOST_BS2000 | Never set (reserved for compilation in BS2000 (SDF)) |
| __SNI_HOST_BS2000_POSIX | Always set |
| __SNI_PRINTF_CHECK | Always set |
| __SNI__STDCplusplus | in all C++ language modes:<br>== 0 in extended language modes (-X nostrict)<br>== 1 strict language modes (-X strict) |
| __SNI_TARG_BS2000 | Never set (reserved for compilation in BS2000 (SDF)) |
| __SNI_TARG_BS2000_POSIX | Always set |
| __STDC__ | Always set:<br>== 0 in extended language modes (-X nostrict)<br>== 1 in strict language modes (-X strict) |
| __STDC_HOSTED__ | Always set |
| __STDC_NO_ATOMICS__ | Always set |
| __STDC_NO_COMPLEX__ | Always set |
| __STDC_NO_THREADS__ | Always set |
| __STDCPP_DEFAULT_NEW_ALIGNMENT__ | Always set<br>== 8U |
| __STDC_UTF_16__ | Always set |
| __STDC_UTF_32__ | Always set |

| | |
|---|---|
| `__STDC_VERS_CRTE__` | Undefined in K&R C mode<br>== 199409L in language modes C89, C++ V2, C++ V3<br>== 201112L in language modes C11, C++ 2017 and C++ 2020 |
| `__STDC_VERSION__` | Undefined in K&R C mode<br>== 199409L in language mode C89 and in all C++ language modes<br>== 201112L in language mode C11 |
| `_STRICT_STDC` | in strict language modes (`-X strict`) |
| `_WCHAR_T` | Option `-K wchar_t_keyword` (default in language modes C++ V3, C++ 2017 and C++ 2020)<br>If this option is not set (e.g. in C modes or in C++ V2 mode), `_WCHAR_T` is defined in various standard headers to issue a `typedef` for `wchar_t`. |
| `_WCHAR_T_KEYWORD` | Option `-K wchar_t_keyword` (default in language modes C++ V3, C++ 2017 and C++ 2020) |
| `_XPG_IV` | If called with `c11` or `c89` |

### Predefined preprocessor predicates (#assert)

| | |
|---|---|
| `data_model(bit32)` | Always set |
| `cpu(7500)` | With /390 code generation |
| `machine(7500)` | With /390 code generation |
| `system(bs2000)` | Always set |

# 4 Global listing generator (cclistgen)

The global listing generator is called with the `cclistgen` command. The input sources for the listing generator are the CIFs (Compilation Information Files) generated by the compiler and written into a file *sourcefile*`.cif` or into an explicitly specified file *file* (see the `-N cif` option on "Options for outputting listings and CIF information"). The generated listings are written by default to `stdout` or into an output file specified in the `-o` option. The listing generator creates global module cross-reference and project listings from the local module cross-reference and project listing CIF information. The remaining listings are generated per source file.

## 4.1 Calling syntax

`cclistgen [`*option*`] ... `*operand* ...

Mixing options and operands is not allowed. The “options first, operands last” order must be adhered to.

### Options

No *option* specified

A source/error listing is generated and output to `stdout`.

*option*

Options can be used to control the type and scope of the listings to be generated.The options are described in the section "Options").

If `cclistgen` is called with illegal options, the program outputs an error message and terminates with an exit status other than `0`.

### Operands

*cif file*

Name of the CIF file from which a listing is to be generated. An unlimited number of CIF files can be specified, but at least one must be. The syntax of the `.cif` extension is not checked, i.e. other file names are accepted (see also the `-N cif` compiler option in section "Options for outputting listings and CIF information").

### Exit status

The exit value 0 is returned if the listing was generated successfully; an exit value other than 0 is returned if an error occurred.

## 4.2 Options

`-o` *outputfile*

The global listing is written to the file *outputfile*. If *outputfile* contains a directory path section, the file is written into it, otherwise into the current directory. The listing is output to `stdout` by default. If the `-o` option is used, the output codeset (ASCII or EBCDIC) is determined by the destination system codeset. However, BS2000 print control characters are always generated.

`-V`

The version and a copyright message are output to `stderr`.

`-N` *listing1*[`,` *listing2*...]

The listing generator writes the listings requested with this option either to `stdout` or into the file specified with the `-o` *outputfile* option.
The following can be specified for *listing*:

`option` or `lo` (options listing)

`prepro` or `lp` (preprocessor listing)

`source_error` or `ls` (source/error listing)

`data_allocation_map` or `lm` (map listing)

`cross_reference` or `lx` (cross-reference listing)

`object` or `la` (object listing)

`project` or `lP` (project listing, only useful for C++ programs)

`summary` or `lS` (statistics listing)

`ALL`

If `ALL` is specified, all listings are generated.

In order to generate a listing, the corresponding information must be requested during the compile (see `-Ncif` in section "Options for outputting listings and CIF information"). If a listing is requested but the information is not available, an error is generated.

`-N output` [`,` *layout*][`,` [*lpp*][`,`*cpl*]]

The layout of the global listing can be influenced with this option.

Four specifications are possible for *layout*:

`normal` or `for_normal_print` (default)
The default for the page length is 64 lines and for the line width 132 characters.

`rotation` or `for_rotation_print`
Defines the page length as 84 lines and the line width as 120 characters.

A page length of from 11 to 255 lines can be defined with *lpp*.

A line width of from 120 to 255 characters can be defined with *cpl*.

*Note*

Since the output file is prepared for printing under POSIX, there are up to 3 control characters at the beginning of some lines in the file. In addition, every line is terminated with the printer control character for a carriage return. If the output file is printed out, then the line length is `cpl-1`.

`-N title,`*text*

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 `-N title` option applies to all compiler listings. In order to ensure that the desired text is transferred correctly, it is advisable to enter it within quotes ("*text*"). In the case of source and preprocessor listings, TITLE and PAGE pragmas (if any) override the `-N title` specification. See also the section on “Pragmas to control the layout of listings” in the C/C++ User Guide [4].

`-N map_structlevel,`*n*

See compiler option `-N map_structlevel,`*n* in section "Options for outputting listings and CIF information".

`-N xref,`*xrefopt1*[`,`*xrefopt2*...]

This option can be used to control which sections are included in the cross-reference listing requested with the `-N cross_reference` option. If the `-N xref` option is not specified, the cross-reference listing contains a list of the variables, functions and labels (equivalent to `-N xref,v,f,l`).
The cross-reference listing always contains a FILETABLE section with the names of all files, libraries and members that the compiler used as sources.

If the `-N xref` option is specified, the cross-reference listing contains the FILETABLE section and only the sections requested with the *xrefopt* argument. The following can be specified for *xrefopt*:

`p` List of the names in `#include` and `#define` directives processed by the preprocessor.

`y` List of the user-defined types (typedefs, structure, union, classes and counter types)

`v` List of variables

`f` List of functions

`l` List of labels

`t` List of templates (only with C++ compilations)

`o=`*str* The order in which the separate sections are listed in the cross-reference listing.
*str* is a string of 6 characters maximum ( letters for the listings shown above).
The default is the order shown above (i.e. `o=pyvflt`).
If the order specified with `o=`*str* does not include all letters for the listings requested with `-N xref`, the omitted letters are implicitly appended to the end of *str* in the default order shown above.

`-K` *arg1*[,*arg2*...]

General input rules for the `-K` option can be found on "Calling syntax and general rules.
The following entries are possible as *arg* arguments to control the listing output:

`include_user`
`include_all`
`include_none`

These options control whether and which header files are mapped to the source program, preprocessor and cross-reference listing.

`-K include_user` is the default and causes only the user header files to be mapped.

If `-K include_all` is specified, all header files, i.e. the standard and the user header files, are mapped.

No header files are mapped if `-K include_none` is specified.

A mapping of header files is only possible when the information is contained in the CIF file (see the compiler option `-K cif_include_user`).

`pragmas_interpreted`
`pragmas_ignored`

These arguments control whether `#pragma` directives are evaluated to control the t layout of the listing (see also the section “Pragmas to control the layout of listings” in the C/C++ User Guide [4]).
`-K pragmas_interpreted` is the default.

# 5 Appendix: overview of options (alphabetic)

| Option | Category | Section |
|---|---|---|
| — | General | General options |
| –A | Preprocessor | Preprocessor options |
| –B extended_external_names | Link | Link editor options |
| –B short_external_names | Link | Link editor options |
| –C | Preprocessor | Preprocessor options |
| –c | Compilation phases (object code) | Options for selecting compilation phases |
| –D *name*[=*value*] | Preprocessor | Preprocessor options |
| –d compl | Link | Link editor options |
| –d n | Link | Link editor options |
| –d y | Link | Link editor options |
| –E *name* | Compilation phases (preprocessor) | Options for selecting compilation phases |
| –F I | Optimization | Optimization options |
| –F i[*name*] | Optimization | Optimization options |
| –F inline_by_source | Optimization | Optimization options |
| –F loopunroll | Optimization | Optimization options |
| –F no_inlining | Optimization | Optimization options |
| –F O2 | Optimization | Optimization options |
| –g | Debug | Debug option |
| –H | Preprocessor | Preprocessor options |
| –i *header* | Preprocessor | Preprocessor options |
| –I *dir* | Preprocessor | Preprocessor options |
| –K [no_]alternative_tokens | C and C++ frontend | Common frontend options in C and C++ |
| –K ansi_cpp | Preprocessor | Preprocessor options |
| –K [no_]assign_local_only | C++ frontend (templates) | Template options |

| | | |
|---|---|---|
| –K [no_]at | C and C++ frontend | Common frontend options in C and C++ |
| –K [no_]bool | C and C++ frontend (general) | General C++ options |
| –K c_names_short | Object generation | Options for controlling object generation |
| –K c_names_std | Object generation | Options for controlling object generation |
| –K c_names_unlimited | Object generation | Options for controlling object generation |
| –K calendar_etpnd | Object generation | Options for controlling object generation |
| –K cif_include_all | CIF | Options for outputting listings and CIF information |
| –K cif_include_none | CIF | Options for outputting listings and CIF information |
| –K cif_include_user | CIF | Options for outputting listings and CIF information |
| –K csect_hashpath | Object generation | Options for controlling object generation |
| –K csect_suffix= | Object generation | Options for controlling object generation |
| –K [no_]dollar | C and C++ frontend | Common frontend options in C and C++ |
| –K [no_] end_of_line_comments | C frontend | Common frontend options in C and C++ |
| –K enum_long | Object generation | Options for controlling object generation |
| –K enum_value | Object generation | Options for controlling object generation |
| –K environment_encoding_ebcdic | Runtime | Runtime options |
| –K environment_encoding_std | Runtime | Runtime options |
| –K external_multiple | Object generation | Options for controlling object generation |
| –K external_unique | Object generation | Options for controlling object generation |
| –K [no_]ieee_floats | Object generation | Options for controlling object generation |
| –K ilcs_opt | Object generation | Options for controlling object generation |
| –K ilcs_out | Object generation | Options for controlling object generation |
| –K [no_]implicit_include | C and C++ frontend (templates) | Template options |
| –K include_all | Listings | Options for outputting listings and CIF information |

| | | |
|---|---|---|
| –K include_none | Listings | Options for outputting listings and CIF information |
| –K include_user | Listings | Options for outputting listings and CIF information |
| –K [no_]instantiation_flags | C and C++ frontend (templates) | Template options |
| –K [no_]integer_overflow | Runtime | Runtime options |
| –K julian_etpnd | Object generation | Options for controlling object generation |
| –K kr_cpp | Preprocessor | Preprocessor options |
| –K library_version= | Language mode | Options for selecting the language mode |
| –K [no_]link_stdlibs | Link | Link editor options |
| –K literal_encoding_ascii | C and /C++ frontend | Common frontend options in C and C++ |
| –K literal_encoding_ascii_full | C and /C++ frontend | Common frontend options in C and C++ |
| –K literal_encoding_ebcdic | C and /C++ frontend | Common frontend options in C and C++ |
| –K literal_encoding_ebcdic_full | C and /C++ frontend | Common frontend options in C and C++ |
| –K literal_encoding_native | C and /C++ frontend | Common frontend options in C and C++ |
| –K [no_]llm_case_lower | Object generation | Options for controlling object generation |
| –K llm_convert | Object generation | Options for controlling object generation |
| –K llm_keep | Object generation | Options for controlling object generation |
| –K [no_]longlong | C and C++- frontend | Common frontend options in C and C++ |
| –K long_preserving | C and C++- frontend | Common frontend options in C and C++ |
| –K new_for_init | C++ frontend (general) | General C++ options |
| –K no_etpnd | Object generation | Options for controlling object generation |
| –K old_for_init | C++ frontend (general) | General C++ options |
| –K [no_]old_specialization | C++ frontend (general) | General C++ options |
| –K plain_fields_signed | C and C++ frontend | Common frontend options in C and C++ |
| –K plain_fields_unsigned | C and C++ frontend | Common frontend options in C and C++ |
| –K pragmas_ignored | Listings | Options for outputting listings and CIF information |

| | | |
|---|---|---|
| –K pragmas_interpreted | Listings | Options for outputting listings and CIF information |
| –K [no_]prompting | Runtime | Runtime options |
| –K [no_]roconst | Object generation | Options for controlling object generation |
| –K [no_]rostr | Object generation | Options for controlling object generation |
| –K schar | C and C++ frontend | Common frontend options in C and C++ |
| –K [no_]share | Object generation | Options for controlling object generation |
| –K signed_fields_signed | C and C++ frontend | Common frontend options in C and C++ |
| –K signed_fields_unsigned | C and C++ frontend | Common frontend options in C and C++ |
| –K stacksize=*n* | Runtime | Runtime options |
| –K [no_]statistics | Runtime | Runtime options |
| –K subcall_basr | Object generation | Options for controlling object generation |
| –K subcall_lab | Object generation | Options for controlling object generation |
| –K uchar | C and C++ frontend | Common frontend options in C and C++ |
| –K unsigned_preserving | C and C++ frontend | Common frontend options in C and C++ |
| –K [no_]using_std | C++ frontend (general) | General C++ options |
| –K [no_]verbose | General | General options |
| –K [no_]wchar_t_keyword | C++ frontend (general) | General C++ options |
| –K workspace_stack | Object generation | Options for controlling object generation |
| –K workspace_static | Object generation | Options for controlling object generation |
| –l BLSLIB | Link | Link editor options |
| –L *dir* | Link | Calling syntax and general rules<br>Link editor options |
| –l *x* | Link | Calling syntax and general rules<br>Link editor options |
| –M | Compilation phases (preprocessor) | Options for selecting compilation phases |
| –N binder,... | Link (listings) | Options for outputting listings and CIF information |

| | | |
|---|---|---|
| –N cif,... | CIF | Options for outputting listings and CIF information |
| –N *listing*,... | Listings | Options for outputting listings and CIF information |
| –N map_structlevel | Listings | Options for outputting listings and CIF information |
| –N output | Listings | Options for outputting listings and CIF information |
| –N title | Listings | Options for outputting listings and CIF information |
| –N xref | Listings | Options for outputting listings and CIF information |
| –O | Optimization | Optimization options |
| –o *ausgabeziel* | General | General options |
| –P | Compilation phases (preprocessor) | Options for selecting compilation phases |
| –r | Link | Link editor options |
| –R diagnose_to_listing | Compiler messages | Options for controlling message output |
| –R error | Compiler messages | Options for controlling message output |
| –R limit | Compiler messages | Options for controlling message output |
| –R min_weight,... | Compiler messages | Options for controlling message output |
| –R note | Compiler messages | Options for controlling message output |
| –R [no_]show_column | Compiler messages | Options for controlling message output |
| –R strict_errors | Compiler messages | Options for controlling message output |
| –R strict_warnings | Compiler messages | Options for controlling message output |
| –R suppress | Compiler messages | Options for controlling message output |
| –R [no_]use_before_set | Compiler messages | Options for controlling message output |
| –R warning | Compiler messages | Options for controlling message output |
| –s | Link | Link editor options |
| –T add_prelink_files | C++ frontend (templates) | Template options |
| –T all | C++ frontend (templates) | Template options |

| | | |
|---|---|---|
| –T auto | C++ frontend (templates) | Template options |
| –T [no_]definition_list | C++ frontend (templates) | Template options |
| –T [no_]dl | C++ frontend (templates) | Template options |
| –T etr_file_all | C++ frontend (templates) | Template options |
| –T etr_file_assigned | C++ frontend (templates) | Template options |
| –T etr_file_none | C++ frontend (templates) | Template options |
| –T local | C++ frontend (templates) | Template options |
| –T max_iterations | C++ frontend (templates) | Template options |
| –T none | C++ frontend (templates) | Template options |
| –U *name* | Preprocessor | Preprocessor options |
| –V | General | General options |
| –v | Compiler messages | Options for controlling message output |
| –w | Compiler messages | Options for controlling message output |
| –X cc | Language mode (C) | Options for selecting the language mode |
| –X CC | Language mode (C++) | Options for selecting the language mode |
| –X kr | Language mode (C) | Options for selecting the language mode |
| –X KR | Language mode (C) | Options for selecting the language mode |
| –X [no]strict | Language mode | Options for selecting the language mode |
| –X v2-compatible | Language mode (C++) | Options for selecting the language mode |
| –X V2-COMPATIBLE | Language mode (C++) | Options for selecting the language mode |
| –X v3-compatible | Language mode (C++) | Options for selecting the language mode |
| -X V3-COMPATIBLE | Language mode (C++) | Options for selecting the language mode |
| –X 11 | Language mode (C) | Options for selecting the language mode |
| –X 17 | Language mode (C++) | Options for selecting the language mode |
| –X 1990 | Language mode (C) | Options for selecting the language mode |
| –X 20 | Language mode (C++) | Options for selecting the language mode |
| –X 2011 | Language mode (C) | Options for selecting the language mode |
| –X 2017 | Language mode (C++) | Options for selecting the language mode |

| | | |
|---|---|---|
| –X 2020 | Language mode (C++) | Options for selecting the language mode |
| –X 89 | Language mode (C) | Options for selecting the language mode |
| –X 90 | Language mode (C) | Options for selecting the language mode |
| –y | Compilation phases (Prelinker) | Options for selecting compilation phases |
| –Y F,... | General | General options |
| –Y I,... | Preprocessor | Preprocessor options |
| –Y P,... | Link | Link editor options |
| –z dup_error | Link | Link editor options |
| –z dup_ignore | Link | Link editor options |
| –z dup_warning | Link | Link editor options |
| –z nodefs | Link | Link editor options |

# 6 Related publications

The manuals are available as online manuals at https://bs2manuals.ts.fujitsu.com.


[1] **POSIX** (BS2000/OSD)
POSIX Basics for Users and System Administrators
User Guide

[2] **C Library Functions for POSIX Applications** (BS2000/OSD)
Reference Manual

[3] **POSIX** (BS2000/OSD)
Commands
User Guide

[4] **C/C++ V4.0B03** (BS2000/OSD)
C/C++ Compiler
User Guide

[5] **CRTE** (BS2000/OSD)
Common RunTime Environment
User Guide

[6] **C++** (BS2000)
C++ Library Functions

[7] **Standard C++ Library V1.2**
User‘s Guide and Reference

[8] **Tools.h++ V7.0**
User Guide

[9] **Tools.h++ V7.0**
Class Reference

[10] **C++ Library Functions** (BS2000)
Reference Manual

[11] **AID** (BS2000/OSD)
Advanced Interactive Debugger
**Debugging of C/C++ Programs**
User Guide

[12] **AID** (BS2000)
**Advanced Interactive Debugger Core Manual**
User Guide


**Other reference literature and standards**

[13] The C Programming Language
by Brian W. Kernighan und Dennis M. Ritchie

[14] The C++ Programming Language
(Third Edition)
by Bjarne Stroustrup

[15] „International Standard ISO/IEC 9899 : 1990, Programming languages - C“

[16] „International Standard ISO/IEC 9899 : 1990, Programming languages - C / Amendment 1 : 1994“

[17] „International Standard ISO/IEC 9899 : 2011, Programming languages - C“

[18] „International Standard ISO/IEC 14882 : 1998, Programming languages - C++“

[19] „International Standard ISO/IEC 14882 : 2017, Programming languages - C++“

[20] „International Standard ISO/IEC 14882 : 2020, Programming languages - C++“