Deutsch

Fujitsu Software BS2000

POSIX-Kommandos des C/C++-Compilers

Benutzerhandbuch

Stand der Beschreibung:
C/C++ V4.0B05

Ausgabe November 2024

Kritik… Anregungen… Korrekturen…

Die Redaktion ist interessiert an Ihren Kommentaren zu diesem Handbuch. Ihre Rückmeldungen helfen uns, die Dokumentation zu optimieren und auf Ihre Wünsche und Bedürfnisse abzustimmen.

Sie können uns Ihre Kommentare per E-Mail an bs2000.info@fujitsu.com senden.

Zertifizierte Dokumentation nach DIN EN ISO 9001:2015

Um eine gleichbleibend hohe Qualität und Anwenderfreundlichkeit zu gewährleisten, wurde diese Dokumentation nach den Vorgaben eines Qualitätsmanagementsystems erstellt, welches die Forderungen der DIN EN ISO 9001: 2015 erfüllt.

Copyright und Handelsmarken

Inhaltsverzeichnis

POSIX-Kommandos des C/C++-Compilers

1 Einleitung

In diesem Kapitel werden folgende Themen behandelt:

- Kurzbeschreibung des Produkts
- Konzept des Handbuchs
- Änderungen gegenüber dem Vorgängerhandbuch
- Darstellungsmittel

1.1 Kurzbeschreibung des Produkts

Der BS2000-Compiler C/C++ kann sowohl aus der BS2000-Umgebung (SDF) als auch aus der POSIX-Umgebung (POSIX-Shell) aufgerufen und mit Optionen gesteuert werden.

In diesem Handbuch wird die Steuerung des Compilers aus der POSIX-Umgebung beschrieben. Hierfür stehen folgende POSIX-Kommandos zur Verfügung:

`cc`, `c11`, `c89`	Aufruf des Compilers als C-Compiler
`CC`	Aufruf des Compilers als C++-Compiler
`cclistgen`	Aufruf des globalen Listengenerators

Mit den Optionen und Operanden der o.g. Aufrufkommandos sind weitgehend die Leistungen und Funktionen abgedeckt, die mit der Compiler-Steuerung über die SDF-Schnittstelle zur Verfügung stehen (siehe „C/C++-Benutzerhandbuch“ [4]). Die Syntax der POSIX-Kommandos ist an der Definition im XPG4-Standard bzw. an den im UNIX-System üblichen Shell-Kommandos orientiert.

In die Aufrufkommandos `cc`, `c11` , `c89` und `CC` ist auch eine Bindephase integriert, in der die übersetzten Objekte zu einer ausführbaren Einheit gebunden werden können.

Für das Erstellen und den Ablauf von C- und C++-Programmen in POSIX-Umgebung werden die Softwareprodukte CRTE und POSIX-HEADER benötigt. In CRTE sind u.a. die Standard-Include-Dateien und die Module der C- und C++-Bibliotheksfunktionen enthalten. Für die Anwendung der POSIX-Bibliotheksfunktionen werden die Header von CRTE und zusätzlich die POSIX-Header benötigt.

1.2 Konzept des Handbuchs

Das vorliegende Handbuch beschreibt, wie C- und C++-Programme mit dem C/C++-Compiler und weiteren Entwicklungswerkzeugen in der POSIX-Umgebung übersetzt, gebunden und zum Ablauf gebracht werden.

Im Kapitel „Grundlagen" wird die C/C++-Programmentwicklung in POSIX-Umgebung im Überblick dargestellt.

Die Kommandos zum Aufruf des Compilers heißen `cc`, `c11`, `c89` und `CC`. Diese Kommandos werden im Kapitel „Die Kommandos cc, c11, c89 und CC“ mit den möglichen Optionen und ihren Auswirkungen ausführlich dargestellt.

Das Kapitel „Globaler Listengenerator (cclistgen)“ beschreibt das Kommando `cclistgen`, mit dem der globale Listengenerator aufgerufen und gesteuert wird.

Im Anhang sind alle Compileroptionen alphabetisch mit Verweisen auf den die Beschreibung enthaltenden Abschnitt aufgelistet.

Voraussetzung für die Arbeit mit diesem Handbuch sind Kenntnisse der Programmiersprachen C bzw. C++ sowie Grundkenntnisse im Umgang mit der POSIX-Shell.

Das Handbuch ist in erster Linie ein Nachschlagewerk zu den POSIX-Kommandos des C/C++-Compilers.

Ausführliche, über die POSIX-Steuerung hinausgehende Informationen zum Leistungs- und Funktionsumfang des C /C++-Compilers finden Sie im Handbuch:

„C/C++ BS2000/OSD, C/C++-Compiler“, Benutzerhandbuch [4]

Dieses Handbuch enthält neben der Beschreibung der SDF-Steuerung des C/C++-Compilers weitergehende Informationen zu Themen, die im vorliegenden Handbuch nicht behandelt werden. Dies sind u.a.

- Verlauf und Auswirkungen der Optimierungsmaßnahmen
- Aufbau der Compilerlisten und Meldungen
- C-Sprachunterstützung des Compilers (die C-Sprachmodi im Überblick, Implementierungsabhängiges Verhalten, `#pragma`-Anweisungen, Erweiterungen gegenüber dem ANSI-/ISO-C-Standard)
- C++-Sprachunterstützung des Compilers (die C++-Sprachmodi im Überblick, Implementierungsabhängiges Verhalten, Erweiterungen gegenüber dem ANSI-/ISO-C++-Standard)
- Funktions- und Sprachverknüpfung
- Kurzbeschreibung der mit CRTE ausgelieferten C++-Bibliotheken

1.3 Änderungen gegenüber dem Vorgängerhandbuch

Gegenüber C/C++ Version V4.0B03 haben sich einige wesentliche Punkte geändert.

Das `switch`-Statement akzeptiert jetzt eine Expression vom Typ `long long`.

Der Inhalt der ETR-Datei hat sich geändert. Der neue Inhalt kann nicht mit einem Compiler V4.0B03 oder älter genutzt werden.

Gegenüber C/C++ Version V4.0B00 haben sich einige wesentliche Punkte geändert.

Die Bibliothek für die Sprachmodi C++2017 und C++2020 wird jetzt in zwei Versionen angeboten. Die Version 1 ist der Default und kompatibel zur früheren Bibliothek. Die Version 2 implementiert einige Features des C++ 2020 Standards. Zu Details siehe das C/C++ Benutzerhandbuch [4].

Gegenüber C/C++ Version V4.0A30 haben sich einige wesentliche Punkte geändert.

Der Compiler unterstützt jetzt die Sprachfeatures des C++ 2020 Standards. Siehe das C/C++ Benutzerhandbuch [4] zu ein paar Einschränkungen. Eine volle Unterstützung der Bibliothek ist aktuell nicht verfügbar, statt dessen wird die existierende C++ 2017-Bibliothek verwendet.

Damit gibt es eine neue Option für diesen Sprachmodus. Die Voreinstellung beim Aufruf mit `CC` ist jetzt C++ 2020. Bis Version 4.0A30 war die Voreinstellung C++ 2017.

Es gibt jetzt eine Überprüfung, ob die Argumente bei einem `printf`-Aufruf mit den Angaben im Formatstring übereinstimmen (siehe __printf_args-Pragma und __scanf_args-Pragma im C/C++ Benutzerhandbuch [4]).

Gegenüber C/C++ Version V3.2D haben sich einige wesentliche Punkte geändert.

Die Änderungen des vorliegenden Handbuchs gegenüber dem Benutzerhandbuch zu C/C++ V3.2D betreffen im Wesentlichen die neuen Sprachmodi C11 und C++ 2017 sowie die darauf zurückzuführenden Änderungen von Compiler-Optionen.

Der wichtigste Punkt ist die Voreinstellung des Compilers. Wenn der Sprachmodus nicht explizit angegeben wird, nimmt er immer den modernsten realisierten Sprachstandard. Dies war bei der Version 3 der Fall und ist es jetzt auch. Bei der Version 3 war der (damals) modernste Standard C89. Jetzt unterstützt der Compiler C11 und dies ist auch die Voreinstellung. Bei C++ ist dies ähnlich. Die Version 3 hatte als (damals) modernsten Standard eine Vorabversion von C++98, der aktuelle Compiler unterstützt C++17.

Der neue Compiler unterstützt jetzt 10 Sprachmodi, je 5 für C und C++. Um diese Vielzahl besser zu handhaben, wurde die Syntax zur Angabe des Sprachmodus neu gestaltet. Die in der Version 3 gebräuchlichen Optionen werden weiterhin erkannt und auf neue Optionen abgebildet. Die Abbildung ist:

`-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`

Mit der Unterstützung der neuen Sprachmodi wurde auch die Erkennung von fragwürdigen Source-Konstrukten überarbeitet. In manchen Situationen kommen jetzt andere Meldungen als bei C/C++ V3.2D. Dabei kann sich sowohl das Fehlergewicht, die Fehlernummer als auch der Text geändert haben. Es gibt ein paar Situationen, wo entweder C/C++ V3.2 eine Meldung bringt oder C/C++ V4.0, aber nicht beide.

1.4 Darstellungsmittel

Für die Darstellung von Kommandos, Optionen und Programmanweisungen wird in diesem Benutzerhandbuch folgende allgemeine Metasprache verwendet:

*STD	Großbuchstaben, Ziffern und Sonderzeichen, die nicht zu den metasprachlichen Zeichen gehören, bezeichnen Schlüsselwörter bzw. Konstanten, die in dieser Form angegeben werden müssen.
`-R msg_id`	Groß- und Kleinbuchstaben, Ziffern und Sonderzeichen in `Schreibmaschinenschrift` sind Konstanten, die in dieser Form angegeben werden müssen. Eine Ausnahme bilden die Argumente der Option `-K` , die im Handbuch in Kleinbuchstaben geschrieben sind, jedoch beliebig in Groß- und/oder Kleinbuchstaben geschrieben werden können (siehe "Aufruf-Syntax und allgemeine Regeln").
name	Kleinbuchstaben in *Kursivschrift* bezeichnen Variablen, die bei der Eingabe durch aktuelle Werte ersetzt werden müssen.
{`cc` | `c89`}	Geschweifte Klammern schließen Alternativen ein, von denen eine ausgewählt werden muss. Das Trennzeichen | darf nicht angegeben werden.
[ ]	Eckige Klammern schließen optionale Angaben ein, die weggelassen werden dürfen.
( )	Runde Klammern sind Konstanten und müssen angegeben werden.
'BLANK'	Dieses Zeichen deutet an, dass mindestens ein Leerzeichen syntaktisch notwendig ist.
...	Drei Punkte bedeuten, dass die davorstehende Einheit mehrmals wiederholt werden kann.

2 Grundlagen

In diesem Kapitel werden folgende Themen behandelt:

2.1 Lieferstruktur und Software-Umgebung

Die Dateien, die für die Steuerung des BS2000-Compilers C/C++ aus der POSIX-Shell benötigt werden, sind wie folgt im POSIX-Dateisystem abgelegt:

`/opt/C/bin/c89`
`/opt/C/bin`
`/cclistgen`
	Links auf den im BS2000 (PLAM-Bibliothek) installierten Compiler und Listengenerator

`/opt/C/bin/cc`
`/opt/C/bin/c11`
`/opt/C/bin/CC`
	Links auf `/opt/C/bin/c89`

`/usr/bin/cc`	Link auf `/opt/C/bin/cc`
`/usr/bin/c11`	Link auf `/opt/C/bin/c11`
`/usr/bin/c89`	Link auf `/opt/C/bin/c89`
`/usr/bin/CC`	Link auf `/opt/C/bin/CC`
`/usr/bin/cclistgen`	Link auf `/opt/C/bin/cclistgen`

Die Installation der oben aufgeführten POSIX-Dateien ist in der Freigabemitteilung zu C/C++ (BS2000/OSD) V4.0 beschrieben.

C/C++ nutzt die mit CRTE ausgelieferten Include-Dateien und Module der C- und C++-Bibliotheksfunktionen sowie die mit POSIX-HEADER ausgelieferten Include-Dateien für alle POSIX-Bibliotheksfunktionen. Die Bibliotheken für die Programme `lex` und `yacc` sind Bestandteil des Software-Produkts POSIX-SH.

Die Module für die C- und C++-Bibliotheksfunktionen sind nicht im POSIX-Dateisystem, sondern als PLAM-Bibliotheken im BS2000 installiert. Beim Binden mit den `cc`/`c11`/`c89`/`CC`-Kommandos werden die Binder-Optionen als RESOLVE-Anweisungen (des BINDER) auf die entsprechenden PLAM-Bibliotheken abgesetzt. Siehe auch Binder-Option -l *x* ("Binder-Optionen").

Die Include-Dateien für die C- und C++-Bibliotheksfunktionen sind als POSIX-Dateien in den Standard-Dateiverzeichnissen `/usr/include`, `/usr/include/sys`, `/usr/include/CXX01`, `/usr/include/CXX02` und `/usr/include/CC` abgelegt. Die Installation dieser Include-Dateien ist in der Freigabemitteilung zu CRTE bzw. im Handbuch „POSIX Grundlagen“ [1] beschrieben.

2.2 Vom Quellprogramm zum Programmablauf

Dieser Abschnitt gibt einen Überblick über folgende Programmerstellungsstufen im POSIX-Subsystem:

- Bereitstellen des Quellprogramms und der Include-Dateien
- Übersetzen
- Binden
  - Binden von Benutzermodulen
  - Binden der CRTE-Laufzeitbibliotheken
- Testen
- Benutzen der POSIX-Bibliotheksfunktionen

2.2.1 Bereitstellen des Quellprogramms und der Include-Dateien

Die Quellprogramm-Dateien und Include-Dateien können in EBCDIC- und ASCII-Code vorliegen. Im POSIX-Dateisystem ist der EBCDIC-Code voreingestellt, in Dateisystemen auf fernen UNIX-Rechnern der ASCII-Code. Alle Dateien eines Dateisystems (POSIX-Dateisystem oder eingehängtes fernes Dateisystem) müssen jeweils im selben Codeset vorliegen. Der Compiler fragt das Codeset eines Dateisystems zentral und nicht pro einzelne Datei ab. Dateien eines ASCII-Dateisystems werden intern automatisch nach EBCDIC konvertiert, wenn die POSIX-Variable IO_CONVERSION=YES gesetzt ist.

Die Dateinamen der Quellprogramme müssen eines der folgenden Standard-Suffixe enthalten:

`c`, `C` C-Quellcode (`cc`, `c11`, `c89`) oder C++-Quellcode (`CC`) vor dem Präprozessorlauf

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

C++-Quellcode vor dem Präprozessorlauf (`CC`)

`i` C-Quellcode (`cc`, `c11`, `c89`) nach dem Präprozessorlauf

`I` C++-Quellcode nach dem Präprozessorlauf (`CC`)

Zusätzlich zu den o.g. Suffixen können mit der Option `-Y F` (siehe "Allgemeine Optionen") weitere Suffixe für Eingabedateien vereinbart werden, die dann vom Compiler ebenfalls erkannt werden.

Quellprogramme und Include-Elemente, die in BS2000-Dateien oder PLAM-Bibliotheken abgelegt sind, können mit dem Compiler im POSIX-Subsystem nicht verarbeitet werden.

Für das Transferieren von BS2000-Dateien und PLAM-Bibliothekselementen in das POSIX-Dateisystem und umgekehrt steht das POSIX-Kommando `bs2cp` zur Verfügung. Für das Editieren von POSIX-Dateien in der POSIX-Shell steht das POSIX-Kommando `edt` zur Verfügung. Wenn der Zugang in die POSIX-Shell über `rlogin` erfolgt, kann auch mit dem Editor `vi` gearbeitet werden. Siehe Handbuch „POSIX-Kommandos“ [3].

Die Standard-Include-Dateien für die mit CRTE verfügbaren C- und C++-Bibliotheksfunktionen befinden sich in den Standard-Dateiverzeichnissen `/usr/include`, `/usr/include/sys`, `/usr/include/CC`, `/usr/include/CXX01 und /usr/include/CXX02`. Diese Dateiverzeichnisse werden vom Compiler (bzw. Präprozessor) automatisch durchsucht.

2.2.2 Übersetzen

Für die Übersetzung von C-Quellen stehen die Kommandos `cc`, `c11` und `c89` zur Verfügung, für die Übersetzung von C++-Quellen das Kommando `CC`.

Diese Kommandos sind ausführlich im Kapitel „Die Kommandos cc, c11, c89 und CC“ beschrieben.

C- und C++-Sprachmodi

Die C- und C++-Quellen können, durch Optionen steuerbar, in verschiedenen Sprachmodi übersetzt werden.

C-Sprachmodi (`cc`/`c11`/`c89`-Kommandos):

- erweiterter c11-Modus (`-X 2011 -X nostrict`), Voreinstellung für `cc` und `c11`
- strikter c11-Modus (`-X 2011 -X strict`)
- erweiterter c89-Modus (`-X 1990 -X nostrict`), Voreinstellung für `c89`
- strikter c89-Modus (`-X 1990 -X strict`)
- Kernighan&Ritchie-C (`-X kr`)

C++-Sprachmodi (`CC`-Kommando):

- erweiterter C++ 2020-Modus (`-X 2020 -X nostrict`), Voreinstellung
- strikter C++ 2020-Modus (`-X 2020 -X strict`)
- erweiterter C++ 2017-Modus (`-X 2017 -X nostrict`)
- strikter C++ 2017-Modus (`-X 2017 -X strict`)
- erweiterter C++ V3-Modus (`-X v3-compatible -X nostrict`)
- strikter C++ V3-Modus (`-X v3-compatible -X strict`)
- Cfront-C++-Modus (Cfront-V3.0.3-kompatibles C++) (`-X v2-compatible`)

Zu den Sprachmodus-Optionen siehe "Optionen zur Auswahl des Sprachmodus".

Erzeugen einer Objektdatei („.o“-Datei)

Wenn der Compilerlauf nicht nach der Präprozessorphase beendet wird (siehe Optionen `-E` und `-P`, "Optionen zur Auswahl von Übersetzungsphasen"), erzeugt der Compiler pro übersetzter Quelldatei ein LLM und legt dieses standardmäßig in eine POSIX-Objektdatei mit dem Namen *basisname*`.o` im aktuellen Dateiverzeichnis ab. *basisname* ist der Name der Quelldatei ohne die Dateiverzeichnisbestandteile und ohne die Standard-Suffixe (`.c`, `.C` etc.).

Mit der Option `-o` lassen sich für die Ausgabe der Objektdatei ein anderes Dateiverzeichnis und/oder ein anderer Dateiname vereinbaren (siehe "Allgemeine Optionen").

Standardmäßig wird nach dem Übersetzungslauf ein Bindelauf gestartet. Wenn in einem Arbeitsgang nur eine Quelldatei übersetzt und gebunden wird, wird die Objektdatei nur temporär angelegt und anschließend gelöscht. Wenn mindestens zwei Quelldateien oder zusätzlich zu einer Quelldatei eine Objektdatei (`.o`-Datei) angegeben werden, bleiben die Objektdateien erhalten.

Mit der Option `-c` (siehe "Optionen zur Auswahl von Übersetzungsphasen") kann der Bindelauf verhindert werden.

Erzeugen eines expandierten, weiterübersetzbaren Quellprogramms („.i“-Datei)

Bei Angabe der Option `-P` wird nur ein Präprozessorlauf durchgeführt und pro übersetzte Quelldatei ein expandiertes, weiterübersetzbares Quellprogramm erzeugt. Das Ergebnis wird standardmäßig in eine POSIX-Quelldatei mit dem Namen *basisname*`.i` (`cc`, `c11`, `c89`) bzw. *basisname*`.I` (`CC`) in das aktuelle Dateiverzeichnis geschrieben.

Mit der Option `-o` lassen sich für die Ausgabe des expandierten Quellprogramms ein anderes Dateiverzeichnis und /oder ein anderer Dateiname vereinbaren (siehe "Allgemeine Optionen").

Erzeugen von Übersetzungslisten

Mit der Option `-N` *listing* können diverse Übersetzungslisten angefordert werden (z.B. Quellprogramm-/Fehlerliste, Querverweisliste etc.). Die angeforderten Listen schreibt der Compiler entweder pro übersetzte Quelldatei in eine Listendatei mit dem Namen *basisname*`.lst` oder für alle übersetzten Quelldateien in eine mit der Option `-N output` angegebene Listendatei *file* (siehe "Optionen zur Ausgabe von Listen und CIF-Informationen").

Für die Ausgabe von Übersetzungslisten können auch CIFs (Compilation Information Files) erzeugt werden, die anschließend mit dem globalen Listengenerator `cclistgen` weiterverarbeitet werden. (Siehe Option `-N cif` ("Optionen zur Ausgabe von Listen und CIF-Informationen") und Kapitel „Globaler Listengenerator(cclistgen)“).

Für das Ausdrucken von Listendateien steht das POSIX-Kommando `bs2lp` zur Verfügung (siehe Handbuch „POSIX-Kommandos“ [3]).

Ausgabeziele und Ausgabe-Codeset

Standardmäßig legt der Compiler die Ausgabedateien im aktuellen Dateiverzeichnis ab, d.h. in dem Dateiverzeichnis, aus dem der Compilerlauf gestartet wird.

Mit der Option `-o` (siehe "Allgemeine Optionen") lässt sich als Ausgabeziel neben einem anderen Dateinamen auch ein anderes Dateiverzeichnis auswählen. Dies kann ein Dateiverzeichnis im lokalen POSIX-Dateisystem sein oder ein Dateiverzeichnis in einem eingehängten Dateisystem auf einem fernen Rechner. Dabei ist zu beachten, dass auf UNIX-Rechnern oder PCs nur Dateien mit Textdaten sinnvoll weiterverarbeitet werden können, also nur expandierte Quellprogramme („.i“-Dateien) und Listendateien („.lst“-Dateien).

Das Ausgabe-Codeset der Dateien (ASCII oder EBCDIC) richtet sich nach dem Codeset des Ziel-Dateisystems, wenn die Umgebungsvariable IO-CONVERSION den Wert YES hat (siehe Abschnitte „Umgebungsvariablen" und „Unterstützung von Dateisystemen in ASCII" im Handbuch „C-Bibliotheksfunktionen“ [2]).

Wie Zeichen und Zeichenketten im Ablaufcode abgelegt werden, wird durch die Option `-K literal_encoding_...` (siehe "Gemeinsame Frontend-Optionen in C und C++") gesteuert.

2.2.3 Binden

Ein C- oder C++-Programm kann in der POSIX-Shell ausschließlich mit den Aufrufkommandos `cc`, `c11`, `c89` und `CC` zu einer ausführbaren Datei gebunden werden. Ein, wie in UNIX-Systemen üblich, „stand alone“-Binder steht nicht zur Verfügung. Technisch gesehen wird beim Binden in der POSIX-Shell der BS2000-BINDER aufgerufen und mit entsprechenden Anweisungen (INCLUDE-MODULES, RESOLVE-BY-AUTOLINK etc.) versorgt.

Ein Bindelauf wird gestartet, wenn keine der Optionen `-c`, `-E`, `-M`, `-P` oder `-y` angegeben wird (siehe "Optionen zur Auswahl von Übersetzungsphasen") und wenn bei einer ggf. vorangegangenen Übersetzung kein Fehler auftrat. Standardmäßig wird das fertig gebundene Programm als LLM in eine ausführbare POSIX-Datei mit dem Standardnamen `a.out` geschrieben und im aktuellen Dateiverzeichnis abgelegt. Mit der Option `-o` kann ein anderes Dateiverzeichnis und/oder ein anderer Dateiname vereinbart werden (siehe "Allgemeine Optionen").

Mit der Option `-N binder` können Standardlisten des BINDER erzeugt werden (siehe "Optionen zur Ausgabe von Listen und CIF-Informationen").

2.2.3.1 Binden von Benutzermodulen

Benutzereigene Module können nur statisch und nicht dynamisch (d.h. zum Ablaufzeitpunkt) eingebunden werden. Programme, die „unresolved externals“ auf Benutzermodule enthalten, können in der POSIX-Shell nicht geladen werden.

Eingabequellen für den Binder können sein:

- vom Compiler erzeugte Objektdateien („.o“-Dateien)
- mit dem Dienstprogramm `ar` erstellte Bibliotheken („.a“-Dateien)
- LLMs, die mit dem POSIX-Kommando `bs2cp` aus PLAM-Bibliotheken in POSIX-Objektdateien kopiert wurden (siehe "Einführungsbeispiele“). Dies können LLMs sein, die in BS2000-Umgebung (SDF) direkt von einem Compiler erzeugt wurden oder Objektmodule, die mit dem BINDER in ein LLM geschrieben wurden.
- LLMs und Objektmodule, die in BS2000-PLAM-Bibliotheken stehen. Die PLAM-Bibliotheken müssen dazu mit den Umgebungsvariablen BLSLIB*nn* zugewiesen werden (siehe Option `-l BLSLIB`, "Binder-Optionen").

Die aus PLAM-Bibliotheken stammenden Module können von jedem ILCS-fähigen BS2000-Compiler erzeugte Module sein (z.B. C/C++, COBOL85, COBOL2000, ASSEMBH). Sprachspezifika sind dabei zu beachten (Paramaterübergaben, benötigte Laufzeitsysteme etc.).

Für POSIX-Objektdateien werden beim Bindelauf intern INCLUDE-MODULES-Anweisungen abgesetzt, für `ar`-Bibliotheken und PLAM-Bibliotheken RESOLVE-BY-AUTOLINK-Anweisungen.

2.2.3.2 Binden der CRTE-Laufzeitbibliotheken

Die offenen Externbezüge auf die C- und C++-Laufzeitsysteme werden vom Binder per Autolink (RESOLVE-BY-AUTOLINK) aus den PLAM-Bibliotheken des CRTE aufgelöst.

C-Laufzeitsystem

Wenn Code erzeugt wird, können die Module des C-Laufzeitsystems auf folgende Arten mit den Kommandos `cc`, `c11`, `c89` und `CC` gebunden bzw. nachgeladen werden:

1. Dynamisches Nachladen des C-Laufzeitsystems (Bindetechnik Partial-Bind)

   Die Bindetechnik Partial-Bind ist gibt es in zwei Varianten:

   - Standard Partial-Bind (`-d y`)

     Standardmäßig, d.h. ohne Angabe von speziellen Binder-Optionen, wird aus der Bibliothek SYSLNK.CRTE. PARTIAL-BIND eingebunden. Diese Bibliothek enthält Verbindungsmodule, die alle offenen Externbezüge des zu bindenden Bindemoduls auf das C- und COBOL-Laufzeitsystem befriedigen. Es werden nur die benötigten Verbindungsmodule eingebunden. Benötigt ein von der zu bindenden Anwendung nachgeladener Modul Entries des Laufzeitsystems, so kann es zu unbefriedigten Externbezügen kommen, da die Verbindungsmodule zu dessen Entries nicht zwangsläufig schon eingebunden sein müssen. In diesem Fall sollte mit der Technik Complete Partial-Bind gebunden werden (siehe auch CRTE-BHB).

     C- und COBOL-Laufzeitsystem selbst werden zum Ablaufzeitpunkt dynamisch nachgeladen, und zwar entweder aus dem Klasse-4-Speicher, falls das C-Laufzeitsystem vorgeladen ist, oder aus der Bibliothek SYSLNK.CRTE.

     Das fertig gebundene Programm benötigt deutlich weniger Plattenspeicher als beim statischen Einbinden des C-Laufzeitsystems aus der Bibliothek SYSLNK.CRTE (siehe 2.). Außerdem wird die Ladezeit verkürzt. Beim Programmaufruf muss das passende CRTE verfügbar sein.

   - Complete Partial-Bind (`-d compl`)

     In diesem Fall wird aus der Bibliothek SYSLNK.CRTE.COMPL eingebunden. Grundsätzlich wird beim Complete Partial-Bind analog wie beim Standard Partial-Bind verfahren. Beim Complete Partial-Bind enthalten jedoch die in SYSLNK.CRTE.COMPL bereitgestellten Verbindungsmodule alle Entries und externen Daten des kompletten C- und COBOL-Laufzeitsystems. Damit sind unbefriedigte Externbezüge, wie sie beim Nachladen von Modulen einer mit der Technik Standard Partial-Bind gebundenen Anwendung auftreten können, beim Complete Partial-Bind ausgeschlossen.
     Wenn Sie im POSIX Shared Libraries einsetzen, ist erfolgreiches Binden nur mit Complete Partial-Bind gewährleistet.

   Nähere Informationen zu den Partial-Bind-Bindetechniken finden Sie im Handbuch „CRTE“ [5].

2. Statisches Binden des gesamten C-Laufzeitsystems (`-d n`)

   Bei Angabe der Binder-Option `-d n` (siehe "Binder-Optionen") werden alle notwendigen Einzel-Module des C-Laufzeitsystems aus der Bibliothek SYSLNK.CRTE eingebunden.

3. Offenlassen der Externbezüge auf das C-Laufzeitsystem (`-z nodefs`)

   Bei Angabe der Binder-Option `-z nodefs` (siehe "Binder-Optionen") wird das Programm ohne ein RESOLVE auf die C-Laufzeitbibliothek gebunden. Die offenen Externbezüge werden erst zum Ablaufzeitpunkt aus dem in den Klasse-4-Speicher vorgeladenen C-Laufzeitsystem befriedigt. `-z nodefs` wird beim Binden von C++-Programmen (`CC`-Kommando) nicht unterstützt.

Die C++-Bibliothek für den Cfront-C++ Sprachmodus V2

Die Module der Cfront-C++-Bibliothek (SYSLNK.CRTE.CPP) und des Cfront-C++-Laufzeitsystems (SYSLNK.CRTE.CFCPP) können nur statisch eingebunden werden. Diese Bibliotheken werden zusätzlich zum C-Laufzeitsystem automatisch eingebunden, wenn im CC-Kommando der Cfront-C++-Modus (Option `-X v2-compatible`) angegeben wird.

Siehe auch Binder-Option `-l`, "Binder-Optionen".

Die C++-Bibliothek für den Sprachmodus C++ V3

Die Module der Standard-C++ V3-Bibliothek (SYSLNK.CRTE.STDCPP) und des C++-Laufzeitsystems (SYSLNK.CRTE.RTSCPP) können nur statisch eingebunden werden. Diese Bibliotheken werden zusätzlich zum C-Laufzeitsystem automatisch eingebunden, wenn im CC-Kommando der C++ V3-Modus (Option `-X v3-compatible`) angegeben wird.

Siehe auch Binder-Option `-l`, "Binder-Optionen".

C++-V3-Bibliothek Tools.h++

Die Module der Bibliothek Tools.h++ (SYSLNK.CRTE.TOOLS) können nur statisch eingebunden werden. Die Bibliothek steht nur im C++ V3-Modus zur Verfügung (Option `-X v3-compatible`) und wird nur eingebunden, wenn zusätzlich die Binder-Option `-l RWtools` angegeben wird.

Siehe auch Binder-Option `-l`, "Binder-Optionen".

Die moderne C++-Bibliothek für die Sprachmodi C++ 2017 und C++ 2020

Die Module der modernen C++-Bibliothek können nur statisch eingebunden werden. Diese Bibliothek wird zusätzlich zum C-Laufzeitsystem automatisch eingebunden, wenn im CC-Kommando der C++ 2017-Modus (Option `-X 2017`) oder C++ 2020-Modus (Default oder Option `-X 2020`) angegeben wird. Die genutzt Datei hängt von der Bibliotheks-Version ab: bei Bibliotheks-Version 1 ist es SYSLNK.CRTE.CXX01, bei Bibliotheks-Version 2 ist es SYSLNK.CRTE.CXX02.

Siehe auch Binder-Option `-l`, "Binder-Optionen".

POSIX-Bindeschalter

Die mit CRTE angebotenen „Bindeschalter“ `posix.o` und `postime.o` (entspricht in BS2000-Umgebung der CRTE-Bibliothek SYSLNK.CRTE.POSIX) werden automatisch eingebunden. Die im C-Laufzeitsystem doppelt vorhandenen Zeitfunktionen, Signalbehandlungsfunktionen und die Funktion `clock` werden deshalb generell mit POSIX-Funktionalität ausgeführt. Es ist grundsätzlich die gemischte Verarbeitung von POSIX- und BS2000-Dateien möglich. Weitere Einzelheiten entnehmen Sie bitte dem Handbuch „C-Bibliotheksfunktionen für POSIX-Anwendungen“ [2].

2.2.4 Testen

Fertig gebundene Programme können mit der Dialogtesthilfe AID getestet werden. Voraussetzung hierfür sind Testhilfeinformationen (LSD), die der Compiler bei Angabe der Option `-g` (siehe "Testhilfe-Option") erzeugt.

Hinweis

> Bei Verwendung der Option `-g` werden die erzeugten Objekte wegen der LSD-Information u. U. deutlich größer!

Die Testhilfe AID wird mit dem POSIX-Kommando `debug` *programmname* aktiviert. Nach Eingabe dieses Kommandos ist die BS2000-Umgebung die aktuelle Umgebung. Es wird %xxxxyyyy/ als Prompting ausgegeben, wobei für xxxxyyyy die PID des mit `debug` gestarteten Prozesses steht. In diesem Modus können die Testhilfe-Kommandos wie im Handbuch „AID Testen von C/C++-Programmen“ [11] beschrieben eingegeben werden. Nach Beendigung des Programms ist wieder die POSIX-Shell die aktuelle Umgebung.

Das `debug`-Kommando mit allen Operanden ist im Handbuch „POSIX-Kommandos“ [3] beschrieben.

2.2.5 Benutzen der POSIX-Bibliotheksfunktionen

Im Gegensatz zur Programmentwicklung in BS2000-Umgebung (SDF) müssen bei der Programmentwicklung in POSIX-Umgebung keine besonderen Vorkehrungen getroffen werden, um die POSIX-Bibliotheksfunktionen nutzen zu können. Die folgenden Aktionen werden automatisch durchgeführt:

- Setzen des Präprozessor-Defines `_OSD_POSIX`
- Einfügen der mit CRTE und POSIX-HEADER ausgelieferten Standard-Include-Dateien aus den Standard-Dateiverzeichnissen. Diese sind abhängig vom Sprachmodus:

Sprachmodus	durchsuchte Dateiverzeichnisse
alle C-Modi	`/usr/include` `/usr/include/sys`
Cfront-C++	`/usr/include` `/usr/include/sys`

C++ V3	`/usr/include/CC`
	`/usr/include`
	`/usr/include/sys`

C++ 2017 / C++ 2020 Bibliotheks-Version 1	`/usr/include/CXX01`
	`/usr/include`
	`/usr/include/sys`

C++ 2017 / C++ 2020 Bibliotheks-Version 2	`/usr/include/CXX02`
	`/usr/include`
	`/usr/include/sys`

  Die Standard-Einstellung kann durch die Option `-Y -I` überschrieben werden.
- Einbinden der POSIX-Bindeschalter `posix.o` und `postime.o` (entspricht in BS2000-Umgebung der PLAM-Bibliothek SYSLNK.CRTE.POSIX)

Die Umgebungsvariable PROGRAM-ENVIRONMENT ist bei Programmstart auf den Wert ’Shell’ gesetzt.

Weitere Einzelheiten entnehmen Sie bitte dem Handbuch „C-Bibliotheksfunktionen für POSIX-Anwendungen“ [2].

2.3 C++-Template-Instanziierung unter POSIX

In diesem Kapitel werden folgende Themen behandelt:

- Grundlegende Aspekte
- Automatische Instanziierung
- Generieren von expliziten Template-Instanziierungsanweisungen (ETR-Dateien)
- Implizites Inkludieren
- Bibliotheken und Templates

2.3.1 Grundlegende Aspekte

Die Sprache C++ beinhaltet das Konzept der Templates. Ein Template ist die Beschreibung einer Klasse oder Funktion, die als Modell für eine Familie von abgeleiteten Klassen oder Funktionen dient. So kann man z.B. ein Template für eine `Stack`-Klasse schreiben und als Integer-Stack, Float-Stack oder einen Stack für einen beliebigen benutzerdefinierten Typ verwenden. Im Quellcode könnten diese dann beispielsweise `Stack<int>`, `Stack<float>` und `Stack<X>` genannt werden. Aus der einmaligen Beschreibung eines Templates für einen Stack im Quellcode kann der Compiler Instanzen des Templates für jeden benötigten Typ generieren.

Die jeweilige Instanz eines Klassen-Templates wird immer dann erzeugt, wenn sie während der Übersetzung benötigt wird.
Demgegenüber werden die Instanzen von Funktions-Templates sowie von Elementfunktionen oder statische Datenelemente eines Klassen-Templates (im Folgenden **Template-Einheiten** genannt) nicht notwendigerweise sofort erzeugt. Die wichtigsten Gründe hierfür sind:

- Bei Template-Einheiten mit externer Linkage (Funktionen und statische Datenelemente) ist es wichtig, programmweit nur eine einzige Kopie der instanziierten Template-Einheit zu haben.
- Es ist erlaubt, eine Spezialisierung für eine Template-Einheit zu schreiben. Das heißt, der Programmierer kann für einen bestimmten Datentyp eine spezielle Implementierung anbieten, die an Stelle der generierten Instanz benutzt wird. Im Sprachmodus C++ V3 muss diese nicht gesondert deklariert werden. Da der Compiler beim Übersetzen einer Referenz auf eine Template-Einheit nicht wissen kann, ob es Spezialisierungen dieser Template-Einheit in einer anderen Übersetzungseinheit gibt, darf er nicht sofort die Instanz generieren.
- Template-Funktionen, die nicht referenziert werden, sollen gemäß des C++ Standard nicht übersetzt und auf Fehler überprüft werden. Deshalb sollte die Referenz auf ein Klassen-Template nicht bewirken, dass automatisch alle Elementfunktionen dieser Klasse instanziiert werden.

Bestimmte Template-Einheiten wie z.B. Inline-Funktionen werden immer instanziiert, wenn sie benutzt werden.

Die oben aufgeführten Anforderungen machen deutlich, dass der Compiler, wenn er für die gesamte Instanziierung verantwortlich ist („automatische“ Instanziierung), diese nur programmweit sinnvoll durchführen kann. Das heißt, er kann die Instanziierung von Template-Einheiten erst dann durchführen, wenn ihm der Quellcode sämtlicher Übersetzungseinheiten des Programms bekannt ist.

Mit dem C/C++-Compiler steht ein Instanziierungs-Mechanismus zur Verfügung, bei dem die automatische Instanziierung zum Binde-Zeitpunkt (und zwar mithilfe eines „Prälinkers“) durchgeführt wird. Nähere Einzelheiten siehe Abschnitt "Automatische Instanziierung“.

Für die explizite Kontrolle der Instanziierung durch den Programmierer stehen durch Optionen wählbare Instanziierungsmodi sowie `#pragma`-Anweisungen zu Verfügung:

- Die Optionen zur Auswahl der Instanziierungsmodi lauten `-T auto`, `-T none`, `-T local` und `-T all`. Sie sind ausführlich im Abschnitt "Template-Optionen“ beschrieben.

- Die Instanziierung einzelner Templates oder auch einer Gruppe von Templates kann mit folgenden `#pragma`-Anweisungen gesteuert werden:
  - Das Pragma `instantiate` bewirkt, dass die als Argument angegebene Template-Instanz erzeugt wird. Dieses Pragma kann analog zu dem C++-Sprachmittel für explizite Instanziierungsanforderungen `template` *declaration* verwendet werden. Siehe auch Beispiel im Abschnitt "Bibliotheken und Templates".
  - Das Pragma `do_not_instantiate` unterdrückt die Instanziierung der als Argument angegebenen Template-Instanz. Typische Kandidaten für dieses Pragma sind Template-Einheiten, für die spezifische Definitionen (Spezialisierungen) bereitgestellt werden.
  - Das Pragma `can_instantiate` ist ein Hinweis für den Compiler, dass die als Argument angegebene Template-Instanz in der Übersetzungseinheit erzeugt werden kann, aber nicht muss. Dieses Pragma wird im Zusammenhang mit Bibliotheken benötigt und wird nur im automatischen Instanziierungsmodus ausgewertet. Siehe auch Beispiel im Abschnitt "Bibliotheken und Templates".

  Die genaue Syntax und allgemeine Regeln zu diesen Pragmas finden Sie im C/C++-Benutzerhandbuch [4], Abschnitt „Pragmas zur Steuerung der Template-Instanziierung“.
- Durch die eingegebenen (in die Source eingemischten) „expliziten Instanziierungsanweisungen“ ist eine explizite Kontrolle möglich. Diese „expliziten Instanziierungsanweisungen“ können über `-T etr_file_all` bzw. `-T etr_file_assigned` (siehe Abschnitt "Generieren von expliziten Template-Instanziierungsanweisungen (ETR-Dateien)“) generiert und dann vom Benutzer in die Sourcen eingebracht werden.

Wichtige Hinweise

Die bei diesem Compiler voreingestellte Methode zur Template-Instanziierung (automatische Instanziierung durch den Prälinker und implizites Inkludieren) ist auch die von uns empfohlene Methode. Von der Möglichkeit, über Optionen steuerbar, von diesem voreingestellten Verfahren abzuweichen, sollte nur in Ausnahmefällen Gebrauch gemacht werden und nur bei genauester Kenntnis der gesamten Applikation einschließlich aller definierten und benutzten Templates.

Implizites Inkludieren: Das implizite Inkludieren darf nicht ausgeschaltet werden (mit `-K no_implicit_include`), wenn Templates aus der C++-V3-Bibliothek (SYSLNK.CRTE.STDCPP) benutzt werden, da in diesem Fall Definitionen nicht gefunden werden.

Andere Instanziierungsmodi als `-T auto` : Hier besteht die Gefahr, dass unbefriedigte Externverweise (`-T none`), Duplikate (`-T all`) oder ggf. Ablauffehler (`-T local`) auftreten können.

2.3.2 Automatische Instanziierung

Der Compiler unterstützt als Voreinstellung in den Sprachmodi C++ V3, C++ 2017 und C++ 2020 die automatische Instanziierung (Option `-T auto`). Dadurch können Sie Quellcode übersetzen und die erzeugten Objekte binden, ohne sich um die notwendigen Instanziierungen kümmern zu müssen.

Im Folgenden beziehen sich die Erläuterungen zur automatischen Instanziierung auf Template-Einheiten, für die es keine explizite Instanziierungsanforderung (`template` *declaration*) und kein `instantiate`-Pragma gibt.

Voraussetzungen

Der Compiler erwartet dabei für jede Instanziierung eine Quelldatei, die sowohl eine Referenz auf die benötigte Instanz enthält als auch die Definition der Template-Einheit und aller für die Instanziierung der Template-Einheit benötigten Typen. Um die letzten beiden Anforderungen zu erfüllen, haben Sie folgende Möglichkeiten:

- Jede `.h`-Datei, in der eine Template-Einheit deklariert ist, enthält entweder auch die Definition der Template-Einheit oder inkludiert eine Datei, die diese Definition enthält.
- Implizites Inkludieren
  Wenn der Compiler eine Template-Deklaration in einer `.h`-Datei findet und auf eine Instanziierungsanforderung stößt, sucht er nach einer Quelldatei mit dem Basisnamen der `.h`-Datei und einem Suffix, das den C++-Quelldatei-Konventionen genügt (siehe Regeln für Eingabedateinamen, "Aufruf-Syntax und allgemeine Regeln"). Diese Datei wird beim Instanziieren ohne Meldung am Ende der jeweiligen Übersetzungseinheit inkludiert. Weitere Einzelheiten siehe Abschnitt "Implizites Inkludieren“.
- Der Programmierer stellt sicher, dass die Dateien, die Template-Einheiten definieren, auch die Definitionen der benötigten Typen enthalten, und fügt in diese Dateien C++-Code oder Instanziierungs-Pragmas ein, mit denen die Instanziierung der dortigen Template-Einheiten angefordert wird.

Erste Instanziierung ohne Definitionsliste

Alternativ zu dem folgenden Verfahren kann auch das Definitionslisten-Verfahren angewandt werden (siehe "Erste Instanziierung mithilfe der Definitionsliste (temporäres Repository)").

Bei der automatischen Instanziierung werden intern folgende Schritte durchgeführt:

1. Instanziierungs-Informationsdateien erzeugen
   Wenn eine oder mehrere Quelldateien zum ersten Mal übersetzt werden, werden noch keine Template-Einheiten instanziiert. Für jede Quelldatei, die ein Template benutzt, wird, falls noch nicht vorhanden, eine Instanziierungs-Informationsdatei erzeugt. Eine Instanziierungs-Informationsdatei hat das Suffix `.o.ii`. Bei der Übersetzung von `abc.C` würde z.B. die Datei `abc.o.ii` erzeugt werden. Die Instanziierungs-Informationsdatei darf vom Benutzer nicht verändert werden.
2. Objektdateien erzeugen
   Die erzeugten Objekte enthalten Informationen darüber, welche Instanzen bei der Übersetzung einer Quelldatei erzeugt werden könnten und ggf. benötigt werden.
3. Template-Instanziierungen zuweisen
   Wenn die Objektdateien gebunden werden, wird vor dem eigentlichen Binden der Prälinker aufgerufen. Dieser durchsucht die Objektdateien nach Referenzen und Definitionen von Template-Einheiten und nach zusätzlichen Informationen über erzeugbare Instanzen. Wenn er keine Definition einer benötigten Template-Einheit findet, sucht er nach einer Objektdatei, in der angegeben ist, dass sie die Template-Einheit instanziieren könnte. Wenn er eine solche Datei findet, weist er die Instanziierung dieser Datei zu.

4. Instanziierungs-Informationsdatei aktualisieren
   Für alle Instanziierungen, die einer Datei zugewiesen wurden, werden in der zugehörigen Instanziierungs-Informationsdatei die Namen der entsprechenden Instanzen geschrieben.
5. Nachübersetzen
   Der Compiler wird intern erneut aufgerufen, um alle Dateien nachzuübersetzen, deren Instanziierungs-Informationsdatei verändert wurde.
6. Neue Objektdatei erzeugen
   Wenn der Compiler eine Datei übersetzt, liest er die Instanziierungs-Informationsdatei für diese Übersetzungseinheit und erzeugt eine neue Objektdatei mit den benötigten Instanzen.
7. Wiederholung
   Die Schritte 3 bis 6 werden solange wiederholt, bis alle benötigten und generierbaren Instanzen erzeugt sind.
8. Binden
   Die Objektdateien werden gebunden.

Erste Instanziierung mithilfe der Definitionsliste (temporäres Repository)

Da das obige Verfahren (siehe "Erste Instanziierung ohne Definitionsliste") einige Dateien mehr als einmal nachübersetzt (rekompiliert), wurde eine Option hinzugefügt, die den gesamten Prozess beschleunigen soll.

Dabei werden die Dateien in der Regel nur einmal nachübersetzt. Durch das Verfahren wird der Hauptanteil der Instanziierungen den ersten nachzuübersetzenden Dateien zugeordnet. Dies hat in einigen Fällen Nachteile, da dadurch ihre Objektgröße ansteigt (als Ausgleich werden andere Objekte kleiner).

Das Vergrößern einzelner Module kann in Benutzeranwendungen von Nachteil sein, wenn z.B. genau diese Module häufig geladen werden müssen. Der Benutzer muss deshalb selbst entscheiden, ob die gleichmäßigere Verteilung der Instanzen (default-Verfahren) oder dieses Verfahren (besseres Zeitverhalten während des Prälinkens) gewollt ist.

Dieses Schema kann durch Angabe der Option `-T definition_list` aktiviert werden. Die obigen Schritte 3-5 werden modifiziert. Damit sieht der Algorithmus folgendermaßen aus:

1. Instanziierungs-Informationsdateien erzeugen
   Wenn eine oder mehrere Quelldateien zum ersten Mal übersetzt werden, werden noch keine Template-Einheiten instanziiert. Für jede Quelldatei, die ein Template benutzt, wird, falls noch nicht vorhanden, eine Instanziierungs-Informationsdatei erzeugt. Eine Instanziierungs-Informationsdatei hat das Suffix `.o.ii`. Bei der Übersetzung von `abc.C` würde z.B. die Datei `abc.o.ii` erzeugt werden. Die Instanziierungs-Informationsdatei darf vom Benutzer nicht verändert werden.
2. Objektdateien erzeugen
   Die erzeugten Objekte enthalten Informationen darüber, welche Instanzen bei der Übersetzung einer Quelldatei erzeugt werden könnten und ggf. benötigt werden.
3. Template-Instanzen einer Sourcedatei zuordnen
   Falls es Referenzen für Template-Einheiten gibt, für die es keine Definitionen im Satz der Objektdateien gibt, so wird eine Datei ausgewählt, die eine der Template-Einheiten instanziieren könnte. Alle Template-Einheiten, die in dieser Datei instanziiert werden können, werden dieser zugeordnet.
4. Instanziierungs-Informationsdatei aktualisieren
   Der Satz an Instanziierungen, der dieser Datei zugeordnet ist, wird in der assoziierten Instanziierungs-Datei aufgezeichnet.

5. Speichern der Definitionsliste
   Intern wird eine Definitionsliste im Speicher gehalten. Sie enthält eine Liste aller Template-bezogenen Definitionen, die in allen Objektdateien gefunden wurden. Diese Liste kann während des Nachübersetzens gelesen und verändert werden.

   Hinweis
   Diese Liste wird nicht in einer Datei abgelegt.
6. Nachübersetzen
   Der Compiler wird intern erneut aufgerufen, um die korrespondierende Quelldatei nachzuübersetzen.
7. Neue Objektdatei erzeugen
   Wenn der Compiler eine Datei nachübersetzt, liest er die Instanziierungs-Informationsdatei für diese Übersetzungseinheit und erzeugt eine neue Objektdatei mit den benötigten Instanzen.
   Wenn der Compiler während der Übersetzung die Gelegenheit erhält, weitere referenzierte Template-Einheiten zu instanziieren, die nicht in der Definitionsliste erwähnt sind oder in den aufgelösten Bibliotheken nicht gefunden wurden, führt er auch diese Instanziierungen durch (z. B. bei Templates, die in Templates enthalten sind). Er führt dem Prälinker eine Liste der Instanziierungen zu, die er auf seinem Weg erhalten hat, so dass der Prälinker sie der Datei zuordnen kann.
   Dieser Prozess erlaubt schnellere Instanziierung. Zudem reduziert sich die Notwendigkeit, eine bestehende Datei während des Prälink-Prozesses mehr als einmal nachzuübersetzen.
8. Wiederholung
   Die Schritte 3 - 7 werden solange wiederholt, bis alle benötigten und generierbaren Instanzen erzeugt sind.
9. Binden
   Die Objektdateien werden gebunden.

Weiterentwicklung

Wenn ein Programm korrekt gebunden wurde, enthalten die zugehörigen Instanziierungs-Informationsdateien alle Namen der definierten und benötigten Instanzen. Von da an zieht der Compiler, wenn eine Quelldatei erneut übersetzt wird, die Instanziierungs-Informationsdatei zu Rate und instanziiert wie bei einem normalen Übersetzungslauf. Das heißt, außer in Fällen, in denen Veränderungen an den Instanzen gemacht werden, sind alle für den Prälinker nötigen Instanziierungen in den Objektdateien gespeichert und keine Instanziierungsanpassungen mehr nötig. Das ist auch der Fall, wenn das Programm vollständig neu übersetzt wird.

Eine irgendwo im Programm bereitgestellte Spezialisierung einer Template-Einheit betrachtet der Prälinker als Definition. Da diese Definition beliebig auftretende Referenzen auf diese Template-Einheit befriedigt, sieht der Prälinker keine Notwendigkeit, eine Instanz für die Template-Einheit anzufordern. Wenn eine Spezialisierung zu einem Programm hinzugefügt wird, das vorher schon einmal übersetzt wurde, löscht der Prälinker die Instanziierungszuweisung aus der entprechenden Instanziierungs-Informationsdatei.

Bis auf folgende Ausnahme darf die Instanziierungs-Informationsdatei vom Benutzer nicht verändert, z.B. umbenannt oder gelöscht werden: Im selben Compilerlauf wird zuerst eine Quelldatei übersetzt, in der eine Definition verändert wurde und anschließend eine Quelldatei, in die eine Spezialisierung eingefügt wurde. Wenn nun die Übersetzung der ersten Datei (mit der geänderten Definition) mit Fehler abgebrochen wird, muss die zugehörige Instanziierungs-Informationsdatei von Hand gelöscht werden, damit sie vom Prälinker neu generiert werden kann.

Automatische Instanziierung, Bibliotheken und vorgebundene Objektdateien

Wenn mit dem CC-Kommando eine ausführbare Datei im automatischen Instanziierungsmodus erzeugt wird, führt der Prälinker die automatische Instanziierung nur in einzelnen Objektdateien (`.o`-Dateien) durch, nicht jedoch in Objekten, die Bestandteil einer Bibliothek (`.a`-Bibliothek) oder einer mit der Option `-r` vorgebundenen Objektdatei sind.

Bibliotheken oder vorgebundene Objektdateien, die Instanzen von Template-Einheiten benötigen, müssen beim Erzeugen der ausführbaren Datei

- entweder diese Instanzen bereits enthalten; dies kann durch explizite Instanziierung und/oder das Vorinstanziieren der Objekte über die Option `-y` erreicht werden (siehe auch Option `-y`, "Optionen zur Auswahl von Übersetzungsphasen")
- oder entsprechende Include-Dateien mit `can_instantiate`-Pragmas bereitstellen.

Weitere Einzelheiten siehe Abschnitt "Bibliotheken und Templates“.

Die Option `-T add_prelink_files` stellt eine weitere Möglichkeit dar, die automatische Instanziierung im Zusammenhang mit Bibliotheken zu steuern (siehe "Template-Optionen").

Steuerung der Instanziierungszuweisungen

Die Zuweisung von Instanziierungen an lokale Objektdateien kann durch die Option `-K assign_local_only` an- und durch die Option `-K no_assign_local_only` abgeschaltet werden (siehe "Template-Optionen").

2.3.3 Generieren von expliziten Template-Instanziierungsanweisungen (ETR-Dateien)

In manchen Fällen, z.B. wenn die automatische Instanziierung nicht sinnvoll einsetzbar ist, bietet sich für den Programmierer die Möglichkeit, die explizite (manuelle) Instanziierung zu verwenden, um die Sourcen entsprechend zu erweitern.
Um diesen Vorgang zu erleichtern, kann eine ETR-Datei (ETR - Explicit Template Request) erstellt werden, die die Instanziierungs-Anweisungen für die verwendeten Templates enthält, die in eine Source übernommen werden können.
Die Optionen zur Erstellung dieser ETR-Datei sind im Abschnitt „Template-Optionen“ dargestellt.
Die Option beinhaltet drei Fälle: `-T etr_file_none (default) /_all /_assigned.` Bei der Angabe `_none` wird die Datei nicht generiert, bei `_all` werden alle relevanten Informationen ausgegeben, bei `_assigned` nur die angegebenen Informationen.

Die Templates, die bei der ETR-Analyse berücksichtigt werden, können in folgende Klassen eingeteilt werden:

- Templates, die in der Übersetzungseinheit explizit instanziiert wurden. Diese werden bei `_all` ausgegeben.
- Templates, die vom Prälinker der Übersetzungseinheit zugeordnet und dann darin instanziiert wurden. Die Ausgabe ist sowohl mit `_all` als auch mit `_assigned` möglich.
- Templates, die in der Übersetzungseinheit verwendet wurden und die hier auch instanziiert werden können. Sie werden mit `_all` ausgegeben.
- Templates, die in der Übersetzungseinheit verwendet wurden, hier aber nicht instanziiert werden können. Auch diese werden bei `_all` ausgegeben.

Der Inhalt einer ETR-Datei hat folgende Form:

- In einer Kopfzeile wird durch Kommentare darauf hingewiesen, dass es sich um eine generierte Datei handelt.
- Für jedes Template werden drei logische Zeilen erzeugt (vgl. Beispiel):
  - eine Kommentarzeile mit dem Text ’The following template was’
  - eine Kommentarzeile, die die Art der Instanz angibt (z.B ’explicitly instantiated’)
  - eine Zeile, die die explizite Instanziierung für diese Instanz beschreibt. Diese enthält den externen Namen der Instanz. Dieser Name entspricht dem Eintrag in der ii-Datei und kann auch dem Binder-Listing bzw. der Binder-Fehlerliste entnommen werden

Hinweise

- Sind die oben angegebenen Zeilen zu lang, so werden sie mit dem in C++ üblichen *„Backslash newline“* umbrochen.
- Die Reihenfolge der ausgegebenen Templates ist nicht definiert. Nach einer Recompilierung oder einer Änderung der Source kann die Reihenfolge anders sein.
- Die logische Zeile drei ist besonders interessant für die Übernahme in eine Source.
- Kommentare sind grundsätzlich in Englisch.

Für die Nutzung der ETR-Datei erscheinen folgende zwei Szenarien sinnvoll:

1. Der Compiler wird während der Entwicklung mit der Option `-T auto` und `-T etr_file_assigned` aufgerufen.
   Die in den ETR-Dateien ausgegebenen Instanziierungs-Anweisungen werden in die zugehörigen Sourcen mit aufgenommen. Der Produktivbetrieb wird dann mit der Option `-T none` oder `-T auto` beim nächsten Compile-Aufruf vorgenommen.
   Der Vorteil dieser Variante liegt in der deutlich reduzierten Zeit für das Prälinken im Produktivbetrieb.
2. Der Compiler wird während der Entwicklung mit der Option `-T none` und der Option `- T etr_file_all` aufgerufen.
   Nach dem Binden prüft der Entwickler jeden ungelösten Externverweis, ob dies ein Template ist und wenn ja, wo es instanziiert werden kann. Hilfreich dabei sind die ausgegebenen externen Namen. Anschließend wählt der Entwickler eine Source für die Instanziierung aus und nimmt die Instanziierungs-Anweisungen dort auf. Außerdem müssen noch die richtigen Header-Files inkludiert werden.
   In dieser Variante ist viel manuelle Arbeit erforderlich. Der Aufruf des Prälinkers kann allerdings dabei entfallen (`-T none`).
   Dieses Vorgehen erlaubt eine genaue Kontrolle über die Platzierung der Instanzen (wichtig z.B. bei Komponenten mit hohen Performance-Ansprüchen).

Beispiel 1

Für eine ETR-Datei, die beim Übersetzen (für die Nutzung `etr_file_all`) zweier Dateien **x.c** und **y.c** entsteht:

Beim Übersetzen wurde dieses Kommando verwendet:

```
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&_
```

Der Benutzer kann nun entscheiden, in welche Source er explizite Instanziierungen vornimmt (diese Entscheidung muss immer getroffen werden für Einträge mit „used in this module and can be instantiated here“), siehe **Beispiel 2**. Danach muss nicht mehr mit der automatischen Template-Instanziierung weitergearbeitet werden.

Beispiel 2

Beispiel für die Nutzung von `etr_file_assigned`.
Gegeben seien zwei Dateien **x.c** und **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):
}
```

Diese Programme werden mit folgenden Kommandos erstmal übersetzt und das Prälinken durchgeführt:

```
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
```

Danach existiert eine Datei **x.o.etr** (da nur x Template-Instanziierungen zugeordnet werden), die wie folgt aussieht

```
// 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&_
```

Die wichtigen Zeilen werden dann in die Datei **x.c** aufgenommen, wodurch folgende Datei **x1.c** entsteht:

```
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&_
```

Im Anschluss kann die Produktion mit folgenden Kommandos durchgeführt werden:

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

Beispiel 3

Folgendes Beispiel zeigt die vier Klassen von Templates, die ausgegeben werden können. Die Annahmen sind wie im Beispiel 1.

Es werden folgende Kommandos eingegeben:

```
CC -c -T auto y.c
CC -y -T auto y.o (dadurch wird f(int) y.o zugewiesen)
CC -c -T auto -T etr_file_all x.c
CC -y -T auto -T etr_file_all x.o y.o
```

Danach entsteht folgende ETR-Datei **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 Implizites Inkludieren

Das implizite Inkludieren von Quelldateien ist eine Methode, um Definitionen von Template-Einheiten zu finden. Diese Methode ist beim Compiler voreingestellt (siehe auch Option `-K implicit_include`, "Template-Optionen") und kann mit `-K no_implicit_include` ausgeschaltet werden. Zum Ausschalten der impliziten Inkludierung beachten Sie bitte auch die Hinweise im Abschnitt "Grundlegende Aspekte".

Wenn implizites Inkludieren eingeschaltet ist, sucht der Compiler die Definition zu einer Template-Einheit nach folgendem Prinzip: Wenn eine Template-Einheit in einer Include-Datei *basisname* `.h` deklariert ist und im übersetzten Quellcode keine Definition bereitsteht, nimmt der Compiler an, dass die Definition zu dieser Template-Einheit in einer Quelldatei steht, die den Basisnamen der Include-Datei und einen passenden Suffix enthält (z.B. *basisname* `.C`).

Es sei beispielsweise eine Template-Einheit `ABC::f` in der Include-Datei `xyz.h` deklariert. Wenn die Instanziierung von `ABC::f` bei der Übersetzung angefordert wird, aber keine Definition von `ABC::f` im übersetzten Quellcode existiert, sucht der Compiler in dem Verzeichnis, in dem die Include-Datei steht, nach einer Quelldatei mit dem Basisnamen `xyz` und einem passenden Suffix (z.B. `xyz.C`). Wenn sie existiert, wird sie so behandelt, als ob sie am Ende der Quelldatei inkludiert wäre.

Es werden die folgenden Suffixe in dieser Reihenfolge geprüft: c, C, cpp, CPP, cxx, CXX, cc, CC, c++ und C++. Das sind die Suffixe, die per Default als C++-Source interpretiert werden. Die Option `-Y F` hat keine Auswirkung auf die beim impliziten Inkludieren geprüften Suffixe.

Damit beim Instanziieren die Datei gefunden wird, in der die Definition einer bestimmten Template-Einheit steht, muss der vollständige Pfadname der Datei bekannt sein, in der die Deklaration des Templates steht. Diese Information ist in Dateien, die `#line`-Anweisungen enthalten, nicht verfügbar. In diesem Fall ist implizites Inkludieren nicht möglich.

Implizites Inkludieren und `make` *-Mechanismus*

Wenn Sie mit dem `make`-Mechanismus arbeiten, müssen die implizit inkludierten Teile bei der Generierung der Dateiabhängigkeitszeilen berücksichtigt werden. Die Objektdatei hängt also sowohl von explizit inkludierten Include-Dateien als auch von implizit inkludierten Dateien mit Template-Definitionen ab.

Wenn Sie die `-M`-Option benutzen, werden die impliziten Include-Teile im automatischen Instanziierungsmodus nur dann berücksichtigt, wenn die Instanziierungs-Informationsdateien korrekt erzeugt worden sind.

Folgende Arbeitsschritte sind dazu notwendig:

1. Alle Quelldateien übersetzen.
2. Das Programm binden, so dass alle Instanziierungen zugewiesen sind.
3. Die Dateiabhängigkeitszeilen für das Programm `make` mit der Option `-M` erzeugen (siehe auch "Optionen zur Auswahl von Übersetzungsphasen").
4. Die Schritte 2 und 3 wiederholen, wenn sich die generierten Template-Instanzen geändert haben.

2.3.5 Bibliotheken und Templates

Im automatischen Instanziierungsmodus können Instanzen für Template-Einheiten (Funktions-Templates, Elementfunktionen und statische Datenelemente von Klassen-Templates) nur generiert werden, wenn das Objekt folgende Bedingungen erfüllt:

- es ist nicht Bestandteil einer `.a`-Bibliothek
- es enthält eine Referenz auf die Template-Einheit oder das Pragma `can_instantiate` für diese Template-Einheit
- und es enthält alle für die Instanziierung notwendigen Definitionen.

Eine Bibliothek, die zu ihrer Implementierung Instanzen benötigt, muss entweder diese Instanzen enthalten oder spezielle Include-Dateien mit `can_instantiate`-Pragmas bereitstellen. Diese beiden Möglichkeiten werden im Folgenden erläutert.

1. Die Bibliothek enthält alle benötigten Instanzen

   Hierbei ist darauf zu achten, dass bei Verwendung mehrerer Bibliotheken keine Duplikate entstehen.

   Um Template-Einheiten in Bibliotheken zu instanziieren, haben Sie folgende Möglichkeiten:

   a. Automatische Instanziierung der Template-Einheiten durch den Prälinker mithilfe der Option `-y` (siehe "Optionen zur Auswahl von Übersetzungsphasen").

      Achtung

      Es besteht die Gefahr, dass bei der Verwendung mehrerer Bibliotheken, die die gleiche Instanz benötigen, Duplikate auftreten, da nicht pro Instanz ein separates Objekt erzeugt wird. Hier kann die Verwendung der Option `-T add_prelink_files` Abhilfe schaffen (siehe "Template-Optionen").

   b. Explizite Instanziierung aller Template-Einheiten mit der Instanziierungsanweisung `template` *declaration* oder mit dem Pragma `instantiate`.

      Hierbei sollte darauf geachtet werden, dass pro Instanz ein separates Objekt erzeugt wird.

      Beispiel

      Es seien gegeben:

      - eine Bibliothek `l.a` mit Referenzen auf die Instanzen `t_list(Foo1)` und `t_list(Foo2),`
      - eine Include-Datei `listFoo.h` mit den Deklarationen von `t_list`, `Foo1` und `Foo2`
      - und eine Quelldatei `listFoo.C` mit den Definitionen von `t_list`, `Foo1` und `Foo2`.

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

// l.C (l.o ist ein Element von 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)
{
   ...
};
```

In der Bibliothek `l.a` sind die referenzierten Instanzen jeweils in separaten Objekten enthalten.

```
// lf1.C : lf1.o ist ein Element von l.a
// und enthält eine explizite Instanziierung von t_list(Foo1)
#include "listFoo.h"
template void t_list(Foo1);

// lf2.C : lf2.o ist ein Element von l.a
// und enthält eine explizite Instanziierung von t_list(Foo2)
#include "listFoo.h"
#pragma instantiate void t_list(Foo2)
```

2. Die Include-Dateien enthalten `can_instantiate`-Pragmas für alle benötigten Instanzen.

   Beispiel

   Es seien gegeben:

   - eine Bibliothek `l.a` mit einer Referenz auf die Instanz `t_list(Foo)`
   - eine Include-Datei `listFoo.h` mit den Deklarationen von `t_list` und `Foo`
   - und eine Quelldatei `listFoo.C` mit den Definitionen von `t_list` und `Foo`

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

// l.C (l.o ist ein Element von 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) {...};
```

Das Objekt `user.o` und die Bibliothek `l.a` werden zusammengebunden(`CC user.o l.a`).

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

`user.C` inkludiert `l.h` und `l.h` inkludiert wiederum `listFoo.h`. Somit enthält `user.C` den Hinweis, dass `t_list(Foo)` instanziiert werden kann.

Bei der automatischen Instanziierung durch den Prälinker wird nur eine Instanz `t_list(Foo)` generiert.

Hinweis
Damit die benötigten Instanzen generiert werden können, muss das Pragma `can_instantiate` in einer Include-Datei der Bibliothek enthalten sein, die dann jeweils von den Benutzerprogrammen inkludiert wird.

2.4 Hinweise zur Software-Portierung

Bei der Portierung von C-Quellprogrammen aus UNIX-Systemen in das POSIX-BS2000 muss die unterschiedliche, implementierungsabhängige Behandlung von extern sichtbaren Namen durch die Compiler beachtet werden.

Der BS2000-Compiler C/C++ erzeugt aus den externen Namen des Quellprogramms (z.B. Funktionsnamen) entsprechende externe Namen für den Binder (Entry-Namen). Dabei werden standardmäßig die Kleinbuchstaben in Großbuchstaben und die Unterstriche (_) in Dollarzeichen ($) umgewandelt (siehe auch „Generierung der Entry-Namen bei LLMs“ (Optionen zur Objektgenerierung)). Durch diese Umwandlungen wird sichergestellt, dass die vom Compiler erzeugten Objekte auch mit anderen Objekten (z.B. von anderssprachigen BS2000-Compilern erzeugte Objekte oder Objekte im Objektmodul-Format) verknüpfbar sind.

Bei der Wahl der extern sichtbaren Namen in C-Quellprogrammen muss deshalb unbedingt darauf geachtet werden, dass sich zwei Namen nicht nur durch Groß-/Kleinschreibung unterscheiden. Z.B. werden die Funktionsnamen `getc` und `getC` standardmäßig auf den gleichen externen Namen `GETC` abgebildet. Soweit keine Namen von anderssprachigen BS2000-Compilern betroffen sind, kann man dieses Verhalten mit der Option `-K llm_case_lower` (siehe "Optionen zur Objektgenerierung") verhindern.

2.5 Einführungsbeispiele

Übersetzen und Binden mit dem c89-Kommando

`c89 hallo.c`
Übersetzt `hallo.c` und erzeugt ausführbare Datei `a.out`

`c89 -o hallo hallo.c`
Übersetzt `hallo.c` und erzeugt ausführbare Datei `hallo`

`c89 -c hallo.c upro.c`
Übersetzt `hallo.c` und `upro.c` und erzeugt die Objektdateien `hallo.o` und `upro.o`

`c89 -o hallo hallo.o upro.o`
Bindet `hallo.o` und `upro.o` zu der ausführbaren Datei `hallo`

Kopieren mit dem bs2cp-Kommando

`bs2cp bs2:hallo hallo.c`
Kopiert die katalogisierte BS2000-Datei `HALLO` in die POSIX-Datei `hallo.c`

`bs2cp 'bs2:plam.bsp(hallo.l,l)' hallo.o`
Kopiert das LLM `HALLO.L` aus der PLAM-Bibliothek `PLAM.BSP` in die POSIX-Objektdatei `hallo.o`

3 Die Kommandos cc, c11, c89 und CC

Der C/C++-Compiler kann über die POSIX-Shell aufgerufen und mit Optionen versorgt werden. Mit den Optionen sind weitgehend die Funktionen und Leistungen abgedeckt, die mit der SDF-Schnittstelle des Compilers angeboten werden.
Die Syntax der Optionen, die Namen der verarbeiteten bzw. erzeugten Objekte und andere Konventionen richten sich grundsätzlich nach der Definition im XPG4-Standard. So weit es sich um nicht im XPG4-Standard definierte Erweiterungen handelt, ist die POSIX-Shell-Schnittstelle des Compilers an die in UNIX-Systemen üblichen Compiler- bzw. Dienstprogramm-Schnittstellen angelehnt.

In den Compiler ist eine Bindephase integriert, die die Shell-üblichen Binder-Optionen und -Operanden in entsprechende BINDER-Anweisungen umsetzt. Ein von den Aufruf-Kommandos unabhängiger „stand alone“-Binder steht in der POSIX-Shell nicht zur Verfügung.

Beim Übersetzen mit dem C/C++-Compiler in der POSIX-Shell können grundsätzlich nur POSIX-Dateien eingelesen und ausgegeben werden. BS2000-Dateien werden nicht unterstützt.
Die Quell-Dateien und Include-Dateien können sowohl im EBCDIC- als auch im ASCII-Code vorliegen. Dabei wird davon ausgegangen, dass alle Dateien eines Dateisystems (fernes eingehängtes Dateisystem oder POSIX-Dateisystem) jeweils im gleichen Codeset vorliegen.

3.1 Aufruf-Syntax und allgemeine Regeln

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

oder

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

Die Unterschiede zwischen den cc/c11/c89/CC-Kommandos sind unten zusammengefasst.

Kommandos cc, c11, c89, CC

cc

Wenn der Compiler mit cc aufgerufen wird, arbeitet er als C-Compiler. Als Sprachmodus wird der letzte unterstützte C-Sprachmodus voreingestellt. In dieser Version des Compilers ist C11 voreingestellt (siehe Option -X 2011, "Optionen zur Auswahl des Sprachmodus"). Dies kann sich in zukünftigen Compiler-Versionen ändern.
Optionen und Operanden können in der Kommandozeile gemischt angegeben werden. -L *dvz* wird im Unterschied zu dem c89-Kommandos als Operand interpretiert (siehe -L und Option --, "Allgemeine Optionen").

c11

Wenn der Compiler mit c11 aufgerufen wird, arbeitet er als C-Compiler. Als Sprachmodus ist C11 voreingestellt (siehe Option -X 2011, "Optionen zur Auswahl des Sprachmodus").

Optionen und Operanden können in der Kommandozeile gemischt angegeben werden. -L *dvz* wird im Unterschied zu dem c89-Kommandos als Operand interpretiert (siehe -L und Option --, "Allgemeine Optionen").

c89

Wenn der Compiler mit c89 aufgerufen wird, arbeitet er als C-Compiler. Als Sprachmodus ist C89 voreingestellt (siehe Option -X 89, "Optionen zur Auswahl des Sprachmodus").
Die gemischte Angabe von Optionen und Operanden in der Kommandozeile ist nicht erlaubt. Die Reihenfolge „erst Optionen, dann Operanden“ muss eingehalten werden. -L *dvz* wird im Unterschied zu den cc/CC-Kommandos als Option interpretiert (siehe -L , und Option --, "Allgemeine Optionen").

CC

Wenn der Compiler mit CC aufgerufen wird, arbeitet er als C++-Compiler. Als Sprachmodus wird der letzte unterstützte C++-Sprachmodus voreingestellt. In dieser Version des Compilers ist C++ 2020 voreingestellt (siehe Option -X 2020, "Optionen zur Auswahl des Sprachmodus").
Optionen und Operanden können in der Kommandozeile gemischt angegeben werden. -L *dvz* wird im Unterschied zu dem c89-Kommando als Operand interpretiert (siehe -L und Option --, "Allgemeine Optionen").

Optionen

Keine *option* angegeben

Wenn der Quellcode syntaktisch korrekt ist und alle offenen Referenzen aufgelöst werden, erstellt der Compiler eine ausführbare Datei `a.out` mit dem ablauffähigen Programm. Nur wenn mindestens zwei Quelldateien oder zusätzlich zu einer Quelldatei eine Objektdatei (`.o`-Datei) angegeben werden, speichert der Compiler den Objektcode zu den einzelnen Quelldateien zusätzlich in gleichnamigen `.o`-Dateien ab.
Bei Angabe nur einer Quelldatei *datei* `.c` steht nach dem Compilerlauf keine Objektdatei *datei* `.o` zur Verfügung, da diese dann nur temporär angelegt und anschließend gelöscht wird; eine ggf. vor dem Compilerlauf vorhandene Objektdatei *datei* `.o` wird ebenfalls gelöscht.

option

Durch Angabe von Optionen im Compiler-Aufruf können Sie den Ablauf steuern und beeinflussen, mit welchen Argumenten die Programme für die einzelnen Übersetzungsphasen versorgt werden.

Mit Optionen können Sie den Compiler auch veranlassen, nur einen Teil der Übersetzungsphasen durchzuführen (siehe "Optionen zur Auswahl von Übersetzungsphasen"). Wenn der Übersetzungsprozess nicht vollständig durchgeführt wird, ignoriert der Compiler alle Optionen, die sich auf nicht durchlaufene Übersetzungsphasen beziehen. Wenn mehrere Optionen zur Auswahl von Übersetzungsphasen angegeben werden, stoppt der Compiler nach der frühesten Phase.

Eine Option ist immer genau ein Buchstabe und wird durch einen führenden Bindestrich („`-`“) gekennzeichnet.

Mehrere Optionen können auch gruppiert, d.h. hinter einem einzigen Bindestrich ohne trennende Leerzeichen angegeben werden, wenn sie keine Argumente besitzen (z.B. kann statt `-V -c` auch `-Vc` geschrieben werden).

Bei Optionen mit Argumenten wird gemäß dem XPG4-Standard zwischen der Option und ihrem Argument ein Leerzeichen geschrieben. Bei diesem Compiler ist die standardkonforme Schreibweise aus Kompatibilitätsgründen zwar nicht zwingend (z.B. wird statt `-o hello` auch `-ohello` akzeptiert), jedoch unbedingt empfehlenswert.

Bei Argumenten, die Trennzeichen (`:` oder `,`) oder das Gleichheitszeichen (=) enthalten, ist kein Leerzeichen vor und nach diesen Zeichen erlaubt.

Beispiele

```
-D MAKRO = 1     verboten
-D MAKRO=1       erlaubt
-R limit,  20    verboten
-R limit,20      erlaubt
```

Bei mehrmaliger Angabe der gleichen Option mit widersprüchlichen Argumenten (z.B. `-K at` und `-K no_at`) gilt die in der Kommandozeile zuletzt angegebene Option.

Dem Compiler unbekannte Optionen, d.h. Optionen, die nach dem Bindestrich („`-`“) mit einem unbekannten Buchstaben beginnen, werden ignoriert. Es wird eine entsprechende Warnungsmeldung ausgegeben. Wenn zwischen der unbekannten Option und einem eventuellen Argument ein Leerzeichen steht, wird sie als Option ohne Argument interpretiert.

Optionen mit unbekannten Argumenten werden ignoriert, und es wird eine entsprechende Warnungsmeldung ausgegeben.

Besondere Eingaberegeln für die Option `-K`

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

Mit der `-K`-Option lassen sich ein oder mehrere, jeweils durch ein Komma voneinander getrennte Argumente angeben. Vor oder nach dem Komma darf kein Leerzeichen geschrieben werden.
Mehrere `-K`-Optionen mit jeweils einem Argument haben die gleiche Wirkung wie eine `-K`-Option mit mehreren, durch Kommas voneinander getrennten Argumenten. Die mit der `-K`-Option angegebenen Argumente können in Groß- und/oder Kleinbuchstaben geschrieben werden (z.B. haben die Argumente `UCHAR`, `uchar`, `Uchar` etc. die gleiche Bedeutung). Bei widersprüchlichen Angaben (z.B. `-K uchar` und `-K schar`) wird ohne Warnungsmeldung die letzte Angabe genommen.

Operanden

Zur Kategorie der „Operanden“ zählen:

- die Namen von Eingabedateien *datei.suffix*
- die Binder-Optionen `-l` *x* und `-l BLSLIB`
- nur bei den `cc/c11`/`CC`-Kommandos zusätzlich die Binder-Option `-L` *dvz*

Der Compiler verarbeitet zuerst alle Optionen und dann die Operanden, und zwar in der Reihenfolge, in der sie in der Kommandozeile stehen.
Nach Angabe der Option `--` (beendet die Eingabe der Optionen) werden alle folgenden Argumente in der Kommandozeile als Operanden interpretiert, auch wenn sie mit einem „`-`“-Zeichen beginnen (siehe Option `--`, "Allgemeine Optionen").

datei.suffix

ist der Name einer Eingabedatei.

Der Compiler schließt aus der Endung des Dateinamens auf den Dateiinhalt und führt die jeweils erforderlichen Übersetzungsschritte aus. Der Dateiname muss daher ein Suffix enthalten, das zum Dateiinhalt passt. Die möglichen Suffixe zur Kennzeichnung von Quelldateien hängen davon ab, ob der Compiler mit den Kommandos `cc/c11/ c89` (C-Modus) oder mit `CC` (C++-Modus) aufgerufen wird.

Im Einzelnen gibt es folgende Möglichkeiten:

`c`, `C` C-Quellcode (`cc`, `c11`, `c89`) oder C++-Quellcode (`CC`) vor dem Präprozessorlauf

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

C++-Quellcode vor dem Präprozessorlauf (`CC`)

`i` C-Quellcode (`cc`, `c11`, `c89`) nach dem Präprozessorlauf

`I` C++-Quellcode nach dem Präprozessorlauf (`CC`)

`o` Objektdatei

`a` statische Bibliothek mit Objektdateien, erzeugt mit dem Dienstprogramm `ar`.

Zusätzlich zu den o.g. Suffixen können mit der Option `-Y F` (siehe "Allgemeine Optionen") benutzereigene Suffixe definiert werden, die dann ebenfalls von den einzelnen Compilerkomponenten erkannt werden.

Dateinamen ohne oder mit einem unbekannten Suffix werden ohne Warnungsmeldung an den Binder weitergereicht.

Die Angabe mindestens einer Eingabedatei *datei* `.` *suffix* oder einer Bibliothek in der Form `-l` *x* pro Compileraufruf ist erforderlich.

Wenn mehrere Eingabedateien angegeben werden, müssen diese nicht vom gleichen Typ sein: Es können in demselben Compileraufruf Quelldateien und Objektdateien angegeben werden. Bei Objektdateien und Bibliotheken ist die Reihenfolge und die Position in der Kommandozeile für den Bindevorgang wichtig.

`-L` *dvz*

`-L` *dvz* ist nur bei Aufruf des Compilers mit den Kommandos `cc`, `c11` und `CC` ein Operand. Mit *dvz* kann ein zusätzliches Dateiverzeichnis angegeben werden, in dem der Binder nach den mit der `-l` Option angegebenen Bibliotheken suchen soll (weitere Einzelheiten siehe "Binder-Optionen").

`-l` *x*

Dieser Operand veranlasst den Binder, nach Bibliotheken mit den Namen `lib` *x* `.a` zu suchen (weitere Einzelheiten siehe „Binder-Optionen“).

`-l BLSLIB`

Dieser Operand veranlasst den Binder, PLAM-Bibliotheken zu durchsuchen, die mit den Shell-Umgebungsvariablen BLSLIB*nn* (00 <= *nn* <= 99) zugewiesen wurden (weitere Einzelheiten siehe „Binder-Optionen“).

Exit-Status

0	normale Beendigung des Compilerlaufs; keine Errors, aber ggf. Notes und Warnings
1	normale Beendigung des Compilerlaufs; mit Errors
2	abnormale Beendigung des Compilerlaufs; Auftreten eines Fatal Errors

3.2 Beschreibung der Optionen

In den folgenden Abschnitten sind die Optionen der Kommandos `cc`, `c11`, `c89` und `CC` nach den folgenden inhaltlichen Gesichtspunkten zusammengestellt und beschrieben.

- Optionen zur Auswahl des Sprachmodus
- Allgemeine Optionen
- Optionen zur Auswahl von Übersetzungsphasen
- Präprozessor-Optionen
- Gemeinsame Frontend-Optionen in C und C++
- C++-spezifische Frontend-Optionen
  - Allgemeine C++-Optionen
  - Template-Optionen
- Optimierungsoptionen
- Optionen zur Objektgenerierung
- Testhilfe-Option
- Laufzeit-Optionen
- Binder-Optionen
- Optionen zur Steuerung der Meldungsausgabe
- Optionen zur Ausgabe von Listen und CIF-Informationen

Im Abschnitt „Aufruf-Syntax und allgemeine Regeln“ erhalten Sie Hinweise, was bei der Eingabe von Optionen allgemein zu beachten ist.

Eine alphabetische Auflistung aller Optionen mit den entsprechenden Seitenverweisen finden Sie im Anhang.

3.2.1 Optionen zur Auswahl des Sprachmodus

`-X cc`

Diese Option dient zur Auswahl des C-Modus. Sie muss nur angegeben werden, wenn der Compiler im C-Modus laufen soll, aber nicht durch `cc`, `c11` oder `c89` aufgerufen wird.

`-X CC`

Diese Option dient zur Auswahl des C++-Modus. Sie muss nur angegeben werden, wenn der Compiler im C++-Modus laufen soll, aber nicht durch `CC` aufgerufen wird.

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

C89-Modus (Voreinstellung beim Compileraufruf mit `c89`)

Diese Option ist nur im C-Modus erlaubt. Der Compiler unterstützt C-Code gemäß dem ANSI-/ISO-C-Standard von 1990. Dieser Standard ist auch als ANSI C89-Standard bekannt. Die Angabe entspricht der Angabe `-X a` bzw. `-X c` des C/C++ V3-Compilers.
`__STDC_VERSION__` hat den Wert 199409L.

`-X 11`
`-X 2011`

C11-Modus (Voreinstellung beim Compileraufruf mit `cc` bzw. `c11` oder Angabe von `-X cc`)

Diese Option ist nur im C-Modus erlaubt. Der Compiler unterstützt C-Code gemäß dem C-Standard von 2011.
`__STDC_VERSION__` hat den Wert 201112L.

`-X kr`
`-X KR`

K&R-C-Modus

Diese Option ist nur im C-Modus erlaubt. Dieser Modus sollte nicht für Neuentwicklungen verwendet werden. Er ist beispielsweise dazu geeignet, „alte“ K&R-C-Quellen zu portieren und/oder sukzessive auf ANSI-C umzustellen.
Der Compiler akzeptiert C-Code gemäß der Definition von Kernighan/Ritchie („Programmieren in C“, 1. Ausgabe). Darüberhinaus unterstützt er C-Sprachmittel des ANSI-C-Standards, die in der Semantik nicht von der Kernighan/Ritchie-Definition abweichen (z.B. Funktions-Prototypen, `const`, `volatile`). Dies erleichtert die Umstellung einer K&R-C-Quelle auf ANSI-C. Es stehen alle C-Bibliotheksfunktionen des Systems zur Verfügung (ANSI-Funktionen, POSIX- und X/OPEN-Funktionen, UNIX-Erweiterungen). Bezüglich des Präprozessor-Verhaltens ist ANSI-/ISO-C voreingestellt. Mit der Option `-K kr_cpp` kann das Präprozessor-Verhalten auf K&R-C umgestellt werden (ggf. bei der Portierung von alten C-Quellen notwendig).
`__STDC_VERSION__` ist undefiniert. Die Option -X strict hat keinen Effekt, d.h. es gilt immer -X nostrict.

`-X 17`
`-X 2017`

C++ 2017-Modus

Diese Option ist nur im C++-Modus erlaubt. Der Compiler unterstützt C++-Code gemäß dem C++-Standard von 2017. Es steht die C++-Standard 2017-Bibliothek zur Verfügung.
`__cplusplus` hat den Wert 201703L und `__STDC_VERSION__` den Wert 199409L.

`-X 20`
`-X 2020`

C++ 2020-Modus (Voreinstellung beim Compileraufruf mit `CC` oder Angabe von `-X CC`)

Diese Option ist nur im C++-Modus erlaubt. Der Compiler unterstützt C++-Code gemäß dem C++-Standard von 2020. Eine volle Unterstützung der C++ 2020-Bibliothek ist aktuell nicht verfügbar, statt dessen wird die existierende C++ 2017-Bibliothek verwendet.
`__cplusplus` hat den Wert 202002L und `__STDC_VERSION__` den Wert 199409L.

`-X V2-COMPATIBLE`

`-X v2-compatible`

Cfront-C++ Modus
Diese Option ist nur im C++-Modus erlaubt. Dieser Modus wird aus Gründen der Kompatibilität angeboten und sollte nicht für Neuentwicklungen verwendet werden. Es werden die C++-Sprachmittel des Cfront V3.0.3 unterstützt. Cfront V3.0.3 wurde erstmals mit dem C++-Compiler V2.1 freigegeben. Es steht die Cfront-kompatible C++-Bibliothek für komplexe Mathematik und stromorientierte Ein-/Ausgabe zur Verfügung.
Zur Cfront-C++-Bibliothek siehe auch C/C++-Benutzerhandbuch [4].
C++-Quellen müssen mit `-X v2-compatible` übersetzt und gebunden werden, wenn die Objekte mit C++-V2.1/V2.2-Objekten verknüpfbar sein sollen.
Diese Angabe entspricht der Angabe `-X d` des C/C++ V3-Compilers.
`_cplusplus` hat den Wert 1 und `__STDC_VERSION` den Wert 199409L. Die Option -X strict hat keinen Effekt, d.h. es gilt immer -X nostrict.

`-X V3-COMPATIBLE`

`-X v3-compatible`

C++ V3- Modus

Diese Option ist nur im C++-Modus erlaubt. Der Compiler unterstützt C++-Code entsprechend dem C/C++ V3-Compiler. Die Angabe entspricht der Angabe `-X w` bzw. `-X e` des C/C++ V3-Compilers.

Folgende C++-Bibliotheken stehen zur Verfügung:

- die Standard-C++-Bibliothek (Strings, Containers, Iterators, Algorithms, Numerics) inklusive der Cfront-kompatiblen Ein-/Ausgabe-Klassen
- die C++-V3-Bibliothek Tools.h++

Zu den C++-Bibliotheken siehe auch C/C++-Benutzerhandbuch [4].

`__cplusplus` hat den Wert 2 (für `-X nostrict`) bzw. 199612L (für `-X strict`) und `__STDC_VERSION` den Wert 199409L.

`-X strict`

Strikter C- bzw. C++-Modus

Der Namensraum ist auf die im Standard definierten Namen beschränkt, und es stehen nur die im Standard definierten C- bzw. C++-Bibliotheksfunktionen zur Verfügung. Bestimmte Erweiterungen (wie das Schlüsselwort `asm`) und einige erwartete Prototyp-Deklarationen aus den Standard-Includes (stdio.h, stdlib.h etc.) sind nicht verfügbar.

Abweichungen vom Standard führen zu Compilermeldungen (zumeist Warnings). Durch Angabe der Option `-R strict_errors` (Siehe "Optionen zur Steuerung der Meldungsausgabe") kann im Falle von Standard-Abweichungen die Ausgabe von Errors erzwungen werden.
`__STDC__` hat den Wert 1, `_STRICT_STDC` ist definiert.

`-X nostrict`

Erweiterter C- bzw. C++-Modus

Einige erforderliche Compilermeldungen entfallen, der Namensraum ist nicht auf Namen beschränkt, die vom Standard spezifiziert sind, und einige Erweiterungen sind enthalten.
`__STDC__` hat den Wert 0, `_STRICT_STDC` ist nicht definiert.

`-K library_version=`*n*

`-K library_version=high`

Diese Option legt die Bibliotheks-Version für die Sprachmodi C++ 2017 und C++ 2020 fest.

Die Bibliotheks-Version beeinflusst die Quelle von Include-Dateien und Resolve-Bibliotheken. Bei Bibliotheks-Version 1 sind es `/usr/include/CXX01` und `SYSLNK.CRTE.CXX01`, bei Bibliotheks-Version 2 sind es `/usr/include/CXX02` und `SYSLNK.CRTE.CXX02`.

Zu Details siehe das C/C++ Benutzerhandbuch [4].

3.2.2 Allgemeine Optionen

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

Allgemeine Eingaberegeln zur `-K`-Option finden Sie im Abschnitt "Aufruf-Syntax und allgemeine Regeln".
Als Argumente *arg* zur allgemeinen Steuerung des Übersetzungsablaufs sind folgende Angaben möglich:

```
verbose
no_verbose
```

Derzeit ist die Angabe von `-K verbose` nur beim `CC`-Kommando sinnvoll. Sie bewirkt, dass zusätzliche Informationen zur Template-Instanziierung auf die Standard-Fehlerausgabe `stderr` geschrieben werden. `-K no_verbose` ist voreingestellt.

`-o` *ausgabeziel*

Ohne Angabe dieser Option legt der Compiler die erzeugten Ausgabedateien mit Standardnamen im aktuellen Dateiverzeichnis ab.
Mit der `-o`-Option können die diversen Ausgabedateien eines Compilerlaufs umbenannt und/oder in ein anderes Dateiverzeichnis geschrieben werden.

ausgabeziel kann sein: nur der Name eines Dateiverzeichnisses, nur ein Dateiname oder ein Dateiname inklusive Dateiverzeichnisbestandteilen. Die angegebenen Dateiverzeichnisse müssen bereits existieren.

ausgabeziel = Dateiverzeichnisname *dvz*

Die Ausgabedateien werden mit Standardnamen im angegebenen Dateiverzeichnis *dvz* abgelegt:

- Wenn eine ausführbare Datei erzeugt wird, erhält die Datei den Namen *dvz*`/a.out`.
- Bei Angabe der Option `-c` erhält die Objektdatei den Namen *dvz/quelldatei*`.o`.
- Bei Angabe der Option `-E` wird die Präprozessor-Ausgabe statt auf die Standardausgabe `stdout` in die Datei *dvz/quelldatei* `.i` (`cc`/`c11`/`c89`-Kommando) oder *dvz/quelldatei*`.I` (`CC`-Kommando) geschrieben.
- Bei Angabe der Option `-M` wird die Präprozessor-Ausgabe (Dateiabhängigkeitsliste zur Weiterverarbeitung mit `make`) statt auf die Standardausgabe `stdout` in die Datei *dvz/quelldatei*`.mk` geschrieben.
- Bei Angabe der Option `-P` wird die Präprozessor-Ausgabe in die Datei *dvz/quelldatei*`.i` (`cc`/`c11`/`c89`-Kommando) oder *dvz/quelldatei*`.I` (`CC`-Kommando) geschrieben.

Mit Ausnahme der vom Binder erzeugten ausführbaren Datei werden bei der Übersetzung von mehreren Quelldateien die Ausgabedateien pro übersetzte Quelldatei angelegt.

ausgabeziel = Dateiname *dateiname* oder
ausgabeziel = Dateiverzeichnisname und Dateiname *dvz/dateiname*

Wenn eine ausführbare Datei erzeugt wird oder wenn gleichzeitig mit der Option `-o` eine der Optionen `-c`, `-E`, `-M`, `-P` angegeben wird, schreibt der Compiler das Ergebnis in eine Datei namens *dateiname* und legt diese entweder im aktuellen Verzeichnis oder in dem mit dvz angegebenen Dateiverzeichnis ab. Mit Ausnahme der vom Binder erzeugten ausführbaren Datei kann für alle übrigen Ausgabedateien nur dann ein Dateiname vereinbart werden, wenn pro Compileraufruf nur eine Quelldatei übersetzt wird. Wird mehr als eine Eingabedatei, jedoch lediglich **eine** Ausgabedatei angegeben, wird eine Warnung ausgegeben und ausgabeziel wird auf den Standardwert zurückgesetzt. Wenn eine ausführbare Datei erzeugt wird, darf der mit `-o` angegebene Dateiname nicht identisch sein mit dem Namen einer vom Compiler generierten oder explizit in der Kommandozeile angegebenen Objektdatei. Beispielsweise werden folgende Kommandos mit Fehler abgewiesen:

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

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

`-V`

Während der Ausführung von `cc/c11/c89/CC` wird die Version und der Copyright-Vermerk des Compilers ausgegeben. Außerdem wird für jede übersetzte Quelldatei eine zusammenfassende Meldung ausgegeben, in der die Anzahl und das Gewicht der Fehlermeldungen sowie die verbrauchte CPU-Zeit angegeben sind. Beim Bindevorgang werden zusätzlich die BINDER-Version sowie eine Liste der verwendeten CRTE-Bibliotheken ausgegeben.

`-Y F,` *dateityp* `,` *benutzer_suffix*

Mit dieser Option können zusätzlich zu den Standard-Suffixen (siehe "Aufruf-Syntax und allgemeine Regeln") benutzereigene Suffixe *benutzer_suffix* für Eingabedateien vom Typ *dateityp* vereinbart werden.

Für *dateityp* sind folgende Angaben möglich:

`c++`	C++-Quelldatei
`c`	C-Quelldatei
`prec++`	C++-Präprozessor-Ausgabedatei
`prec`	C-Präprozessor-Ausgabedatei
`obj`	Objektdatei
`lib`	statische Bibliothek

Beispiel

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

Eine Eingabedatei namens *datei* `.llm` wird vom Compiler als Objektdatei erkannt.

`--`

Diese Option beendet die Eingabe der Optionen. Mit Ausnahme der zur Kategorie der „Operanden“ zählenden Binder-Optionen werden alle folgenden Argumente in der Kommandozeile als Dateinamen interpretiert, auch wenn sie mit einem Bindestrich beginnen. Somit ist es möglich, Dateinamen anzugeben, die mit einem Bindestrich beginnen (z.B. `-hallo.c`).
Folgende Binder-Optionen sind nach der Option `--` zulässig:
`-l` *x*

`-l BLSLIB`

`-L` *dvz* (nur bei den Kommandos `cc`, `c11` und `CC`; beim `c89`-Kommando würde diese Angabe als Dateiname interpretiert!)

3.2.3 Optionen zur Auswahl von Übersetzungsphasen

Bei Angabe einer der folgenden Optionen wird generell der Bindelauf unterdrückt. Ggf. angegebene Binder-Optionen und -Operanden werden ignoriert.

`-c`

Der Compilerlauf wird beendet, nachdem für jede übersetzte Quelldatei ein LLM erzeugt und in eine Objektdatei *datei*`.o` abgelegt wurde. Die Objektdatei wird standardmäßig in das aktuelle Dateiverzeichnis geschrieben. Mit der Option `-o` (siehe "Allgemeine Optionen") kann ein anderer Dateiname und/oder ein anderes Dateiverzeichnis vereinbart werden.

`-E`

Der Compilerlauf wird nach der Präprozessorphase beendet. Das Ergebnis wird auf die Standardausgabe `stdout` geschrieben. Dabei werden Leerzeilen zusammengefasst, und es werden entsprechende `#line`-Anweisungen generiert. Standardmäßig werden die C- bzw. C++-Kommentare in der Präprozessorausgabe entfernt (siehe Option `-C`, "Präprozessor-Optionen"). Bei Angabe der Option `-o` (siehe "Allgemeine Optionen") wird das Präprozessorergebnis statt auf die Standardausgabe `stdout` in eine Datei geschrieben.

`-M`

Der Compilerlauf wird nach der Präprozessorphase beendet. Anstelle einer normalen Präprozessorausgabe (vgl. `-E`, `-P`) wird eine Liste von Dateiabhängigkeitszeilen generiert und auf die Standardausgabe `stdout` geschrieben. Diese Liste ist für die Weiterverarbeitung mit dem POSIX-Programm `make` geeignet. Bei Angabe der Option `-o` (siehe "Allgemeine Optionen") wird die Dateiabhängigkeitsliste statt auf die Standardausgabe `stdout` in eine Datei geschrieben.
Hinweis
Templates in C++ 2017, C++ 2020 und C++ V3-Quellen werden nicht implizit inkludiert.

`-P`

Der Compilerlauf wird nach der Präprozessorphase beendet. Das Ergebnis wird aber nicht wie bei der Option `-E` auf die Standardausgabe `stdout`, sondern in eine Datei *datei*`.i` (`cc`/`c11`/`c89`-Kommando) bzw. *datei*`.I` (`CC`-Kommando) geschrieben und im aktuellen Dateiverzeichnis abgelegt. Die Ausgabe enthält keine zusätzlichen `#line`-Anweisungen. Standardmäßig werden die C- bzw. C++-Kommentare in der Präprozessorausgabe entfernt (siehe Option `-C`, "Präprozessor-Optionen"). *datei* `.i` kann später mit den `cc`/`c11`/`c89`/`CC`-Kommandos weiterübersetzt werden, *datei*`.I` nur mit dem `CC`-Kommando. Mit der Option `-o` (siehe "Allgemeine Optionen") kann ein anderer Dateiname und/oder ein anderes Dateiverzeichnis vereinbart werden.

`-y`

Diese Option kann nur beim `CC`-Kommando in den Modi C++ 2017, C++ 2020 und C++ V3 angegeben werden. Der Compilerlauf wird nach der Prälinker-Phase (automatische Template-Instanziierung) beendet. Pro übersetzte Quelldatei wird eine Objektdatei *quelldatei*`.o` generiert, in der die Templates instanziiert sind. Dies ist bei Objekten sinnvoll, die später in eine Bibliothek (`.a`- Bibliothek) oder in eine vorgebundene Objektdatei (`-r`) aufgenommen werden sollen; für Templates innerhalb von Bibliotheken oder vorgebundenen Objektdateien wird keine automatische Instanziierung durchgeführt. Die Verwendung der Option `-y` ist nur im voreingestellten automatischen Instanziierungsmodus (`-T auto`) zweckmäßig.

Beispiel

Inhalt der Quelldateien (Auszüge):

```
// a.h:

class A {int i;};

// f.h:

template <class T> void f(T)
{
   /* beliebiger Code */
}

// b.c:

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

// main.c:

extern void foo();

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

Kommandos:

`CC -c b.c`

Bei der ersten Übersetzung werden eine Objektdatei `b.o` und eine Template-Informationsdatei `b.o.ii` generiert, jeweils mit dem Eintrag, dass die Funktion `f(A)` nicht instanziiert ist.

`CC -y b.o`

Es werden die bei der ersten Übersetzung generierten Dateien `b.o` und `b.o.ii` aktualisiert und die Funktion `f(A)` instanziiert.

`ar -r x.a b.o`

Das Modul in `b.o` wird in die Bibliothek `x.a` aufgenommen.

`CC main.c x.a`

Es wird eine ausführbare Datei `a.out` generiert.

Die folgende Kommandofolge würde dagegen nicht zum gewünschten Ziel führen:

```
rm *.o *.ii *.a a.out   /* Bereinigung des aktuellen Verzeichnisses */
CC -c b.c
ar -r x.a b.o
CC main.c x.a
```

Diese Kommandofolge führt zu einer Fehlermeldung, da für die Templates in der Bibliothek `x.a` keine automatische Instanziierung durchgeführt wird und die Funktion `f(A)` deshalb nicht gefunden werden kann.

3.2.4 Präprozessor-Optionen

`-A "`*name*`(`*token-Folge*`)"`

Mit dieser Option kann ein Prädikat (Assertion) definiert werden, analog zur Präprozessor-Anweisung `#assert` (siehe Abschnitt „Erweiterungen gegenüber ANSI-/ISO-C“ im C/C++-Benutzerhandbuch [4]). Die Anführungszeichen werden wegen der Sonderbedeutung der runden Klammern in der POSIX-Shell benötigt. Alternativ können die runden Klammern auch mit dem Gegenschrägstrich \ entwertet werden:
`-A` *name*`\(`*token-Folge*`\)`

`-C`

Diese Option wird nur ausgewertet, wenn gleichzeitig die Option `-E` oder `-P` angegeben wird (siehe "Optionen zur Auswahl von Übersetzungsphasen"). Sie bewirkt, dass C- bzw. C++-Kommentare in der Präprozessorausgabe nicht entfernt werden. Standardmäßig werden die Kommentare entfernt.

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

Mit dieser Option lassen sich Namen, symbolische Konstanten und Makros definieren (analog zur Präprozessoranweisung `#define`).
`-D` *name* wirkt wie eine Anweisung `#define` *name* `1`,
`-D` *name*=*wert* entspricht der `#define`-Anweisung für Textersatz `#define` *name wert.*

`-H`

Es wird während des Übersetzungslaufs eine Liste aller benutzten Include-Dateien auf die Standard-Fehlerausgabe `stderr` ausgegeben.

`-i` *header*

Mit dieser Option wird eine Include-Datei *header* spezifiziert, die vor dem Quellprogrammtext inkludiert wird (Pre-Include).

Für *header* kann wahlweise angegeben werden:

- vollqualifizierter Pfadname der Include-Datei
- relativer Pfadname der Include-Datei auf Basis der Option `-I`

Die durch *header* spezifizierte Include-Datei wird analog einer Include-Datei behandelt, die in einer `#include`-Anweisung am Anfang der Quellprogramm-Datei angegeben ist. Sollen mehrere Include-Dateien pre-inkludiert werden, dann müssen die zugehörigen `#include`-Anweisungen in einer einzigen Include-Datei zusammengefasst werden, die dann via Option `-i` zu spezifizieren ist.

`-I` *dvz*

dvz wird in die Liste der Dateiverzeichnisse aufgenommen, in denen der Präprozessor nach Include-Dateien sucht. Wird diese Option mehrfach angegeben, so bestimmt die Reihenfolge der Angabe auch die Reihenfolge der Suche nach Include-Dateien.

Wenn in der `#include`-Anweisung der relative Pfadname der Include-Datei (beginnt nicht mit Schrägstrich /) in Anführungszeichen "..." eingeschlossen ist, durchsucht der Präprozessor die Dateiverzeichnisse in folgender Reihenfolge:

1. das Dateiverzeichnis der Quell- oder Include-Datei, die die `#include`-Anweisung enthält

2. die Dateiverzeichnisse, die mit der Präprozessor-Option `-I` angegeben wurden
3. entweder die mit der Option `-Y I` angegebenen Dateiverzeichnisse oder die Standard-Dateiverzeichnisse (siehe "Benutzen der POSIX-Bibliotheksfunktionen").

Wenn in der `#include`-Anweisung der relative Pfadname der Include-Datei in spitzen Klammern <...> eingeschlossen ist, durchsucht der Präprozessor nur die oben unter 2. und 3. angegebenen Dateiverzeichnisse.

Wenn der Präprozessor statt der Standard-Dateiverzeichnisse andere Dateiverzeichnisse zuletzt durchsuchen soll, können diese mit der Option `-Y I` angegeben werden.

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

Allgemeine Eingaberegeln zur `-K`-Option finden Sie im Abschnitt "Aufruf-Syntax und allgemeine Regeln".
Als Argumente *arg* zur Steuerung des Präprozessor-Verhaltens sind folgende Angaben möglich:

`ansi_cpp`
`kr_cpp`

`-K ansi_cpp` ist in allen C- und C++-Sprachmodi des Compilers voreingestellt. Das heißt, dass auch im K&R-C-Modus das Präprozessor-Verhalten entsprechend dem ANSI-/ISO-C-Standard unterstützt wird. Mit `-K kr_cpp` kann das veraltete Präprozessor-Verhalten gemäß Reiser cpp und Johnson pcc eingeschaltet werden.

`-U` *name*

Die Definition für ein Makro oder eine symbolische Konstante *name* wird gelöscht (analog zur Präprozessoranweisung `#undef`). *name* ist ein vordefinierter Präprozessorname (siehe "Vordefinierte Präprozessornamen") oder ein Name, der mit der Option `-D` in der Kommandozeile vor oder nach der Option `-U` definiert wurde.
Auf `#define`-Anweisungen im Quellprogramm hat die Option keine Wirkung.

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

Der Präprozessor sucht zuletzt in den mit *dvz* angegebenen Verzeichnissen nach Include-Dateien.
Ohne Angabe dieser Option werden die Standard-Dateiverzeichnisse zuletzt durchsucht (siehe "Benutzen der POSIX-Bibliotheksfunktionen").

3.2.5 Gemeinsame Frontend-Optionen in C und C++

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

Allgemeine Eingaberegeln zur `-K`-Option finden Sie im Abschnitt "Aufruf-Syntax und allgemeine Regeln".
Als Argumente *arg* zur Steuerung des Compiler-Frontends in den C- und C++-Sprachmodi sind folgende Angaben möglich:

`uchar`
`schar`

Der Datentyp `char` ist standardmäßig vom Typ `unsigned char`. Bei Angabe von `-K schar` wird `char` in Ausdrücken und Konversionen als `signed char` behandelt. Bei Verwendung dieser Option können Portabilitätsprobleme auftreten!

`at`
`no_at`

Bei Angabe von `-K no_at` ist in Bezeichnern das at-Zeichen '@' nicht erlaubt.
`-K at` ist voreingestellt. Das at-Zeichen in Bezeichnern ist eine Erweiterung gegenüber dem Standard.

> ¡ Bei Nutzung der Cfront-C++-Bibliothek darf die Option `-K no_at` nicht verwendet werden.

`dollar`
`no_dollar`

Bei Angabe von `-K no_dollar` ist in Bezeichnern das Dollar-Zeichen '$' nicht erlaubt.
`-K dollar` ist voreingestellt. Das Dollar-Zeichen in Bezeichnern ist eine Erweiterung gegenüber dem Standard.

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

Diese Option legt fest, ob der C/C++-Compiler Code für Zeichen und Zeichenketten im EBCDIC- oder im ASCII-Format (ISO 8859-1) erzeugt.
In C/C++ können Zeichenketten binär codierte Zeichen als oktale oder sedezimale Escape-Sequenzen mit folgender Syntax enthalten.

- oktale Escape-Sequenzen: `'\[0-7] [0-7] [0-7]'`
- sedezimale Escape-Sequenzen: `'\x[0-9A-F] [0-9A-F]'`

Ob der C/C++-Compiler Escape-Sequenzen in das ASCII-Format konvertiert oder nicht, hängt von der verwendeten `literal_encoding_`...-Option ab.

`literal_encoding_native`
Der C/C++-Compiler belässt Code für Zeichen und Zeichenketten im EBCDIC-Format, d.h. er übernimmt die Zeichen(ketten) unkonvertiert in den Objektcode. `literal_encoding_native` ist Voreinstellung.

`literal_encoding_ascii`
Der C/C++-Compiler erzeugt Code für Zeichen und Zeichenketten im ASCII-Format. In den Zeichenketten enthaltene Escape-Sequenzen werden *nicht* in das ASCII-Format konvertiert.

`literal_encoding_ascii_full`
Der C/C++-Compiler erzeugt Code für Zeichen und Zeichenketten im ASCII-Format. In den Zeichenketten enthaltene Escape-Sequenzen werden ebenfalls in das ASCII-Format konvertiert.

`literal_encoding_ebcdic`
Der C/C++-Compiler belässt Code für Zeichen und Zeichenketten im EBCDIC-Format, d.h. er übernimmt die Zeichen(ketten) unkonvertiert in den Objektcode. `literal_encoding_ebcdic` hat somit dieselbe Wirkung wie `literal_encoding_ebcdic_full` oder `literal_encoding_native`.

`literal_encoding_ebcdic_full`
Der C/C++-Compiler belässt Code für Zeichen und Zeichenketten im EBCDIC-Format, d.h. er übernimmt die Zeichen(ketten) unkonvertiert in den Objektcode. `literal_encoding_ebcdic_full` hat somit dieselbe Wirkung wie `literal_encoding_ebcdic` oder `literal_encoding_native`.

Voraussetzungen für die ASCII-Unterstützung:

- Sie dürfen die C-Bibliotheksfunktionen in Ihrem Quellprogramm nicht explizit deklarieren, sondern nur indirekt via Inkludieren des entsprechenden CRTE-Headers. Andernfalls kann es zum Übersetzungsfehler ‘CFE1079[ERROR]..: Typangabe erwartet / expected a type specifier‘ kommen.
- Inkludieren Sie für jede in Ihrem Programm verwendete CRTE-Funktion (C-Bibliotheksfunktion), die mit Zeichenketten arbeitet, die passende bzw. dazugehörige Include-Datei. Andernfalls können diese Funktionen Zeichenketten nicht korrekt verarbeiten. Insbesondere müssen Sie für die Funktion *printf()* die Include-Datei <stdio.h> mit `#include <stdio.h>` inkludieren.
- Für die Verwendung von CRTE-Funktionen müssen zusätzlich die folgenden Optionen angegeben werden:

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

`signed_fields_signed`
`signed_fields_unsigned`

Bei Angabe von `-K signed_fields_unsigned` sind `signed` Bitfelder immer vom Typ `unsigned`. Diese Option wird aus Kompatibilitätsgründen zu älteren C-Versionen angeboten und ist nur im K&R-C-Modus sinnvoll.
`-K signed_fields_signed` ist voreingestellt.

`plain_fields_signed`
`plain_fields_unsigned`

Diese Argumente steuern, ob Integer-Bitfelder (`short`, `int`, `long`, `long long`) standardmäßig vom Typ `signed` oder `unsigned` sind.
`-K plain_fields_signed` ist voreingestellt.

`long_preserving`
`unsigned_preserving`

Diese Argumente steuern, ob das Ergebnis von arithmetischen Operationen mit Operanden vom Typ `long` und `unsigned int`, gemäß K&R (erste Ausgabe, Anhang 6.6) vom Typ `long` ist (`long_preserving`) oder gemäß ANSI-/ISO-C vom Typ `unsigned long` (`unsigned_preserving`).
`-K unsigned_preserving` ist voreingestellt.

`alternative_tokens`
`no_alternative_tokens`

Diese Argumente steuern, ob der Compiler alternative Tokens erkennen soll:

in den C- und C++-Sprachmodi Digraph-Sequenzen (z.B. `<:` für `[`),
nur in den C++-Sprachmodi zusätzliche Schlüsselwort-Operatoren (z.B. `and` für `&&`, `bitand` für `&`).

In den Modi C11, C++ V3, C++ 2017 und C++ 2020 ist `-K alternative_tokens` voreingestellt, in allen anderen Sprachmodi `-K no_alternative_tokens`.

`longlong`
`no_longlong`

Diese Argumente steuern, ob der Datentyp `long long` vom Compiler erkannt wird. `-K longlong` ist voreingestellt. In diesem Fall wird das Präprozessor-Define `_LONGLONG` gesetzt.
Bei Angabe von `-K no_longlong` führt der Gebrauch des Datentyps `long long` zu einem Fehler.
Diese Angabe ist nur im strikten C89-Modus und im strikten C++ V3-Modus zulässig.

`end_of_line_comments`
`no_end_of_line_comments`

Diese Argumente steuern, ob der Compiler C++-Kommantare (//...) im erweiterten C89-Modus akzeptiert.
`-K no_end_of_line_comments` ist im erweiterten C89-Modus voreingestellt.

In den anderen Sprachmodi ist jeweils nur eine Angabe erlaubt, die andere wird mit einer Warnung ignoriert. Im strikten C89-Modus und im K&R-C-Modus sind C++-Kommentare nicht erlaubt, in den C++-Modi und im C11-Modus sind sie immer erlaubt.

3.2.6 C++-spezifische Frontend-Optionen

Die Optionen in den folgenden Abschnitten „Allgemeine C++-Optionen“ und „Template-Optionen“ gelten nur für das CC-Kommando.

3.2.6.1 Allgemeine C++-Optionen

Mit den allgemeinen C++-Optionen steuern Sie folgende C++-Spracheigenschaften:

- ob die Schlüsselwörter `wchar_t` und `bool` erkannt werden
- welchen Gültigkeitsbereich die Initialisierungsanweisungen in `for`- und `while`-Schleifen haben
- ob die veraltete Template-Spezialisierungs-Syntax akzeptiert wird

Die aufgezählten Spracheigenschaften werden im Cfront-C++-Modus nicht unterstützt.

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

Allgemeine Eingaberegeln zur `-K`-Option finden Sie im Abschnitt "Aufruf-Syntax und allgemeine Regeln". Als Argumente *arg* zur Steuerung des C++-Frontends sind folgende Angaben möglich:

`using_std`
`no_using_std`

Diese Argumente betreffen den Gebrauch der C++-Bibliotheksfunktionen, deren Namen alle im Standard-Namensraum `std` definiert sind.

Bei Angabe von `-K using_std` ist das Verhalten so, als ob am Beginn einer Übersetzungseinheit die folgenden Zeilen stehen würden:

```
namespace std{}
using namespace std;
```

`-K using_std` ist die Voreinstellung im erweiterten C++ V3-Modus (`-X v3-compatible -X nostrict`).

`-K no_using_std` ist die Voreinstellung im strikten C++ V3-Modus (`-X v3-compatible -X strict`), im C++ 2017-Modus (`-X 2017`) und im C++ 2020-Modus (`-X 2020`) und das einzig mögliche Verhalten im Cfront-C++-Modus (`-X v2-compatible`).
Wenn im C++ V3-Modus, im C++ 2017-Modus oder im C++ 2020-Modus `-K no_using_std` gesetzt ist, muss das Quellprogramm vor dem ersten Gebrauch einer C++-Bibliotheksfunktion die Anweisung `using namespace std;` enthalten oder die Namen entsprechend qualifizieren.

`wchar_t_keyword`
`no_wchar_t_keyword`

Mit diesen Argumenten legen Sie fest, ob `wchar_t` als Schlüsselwort erkannt wird.

`-K wchar_t_keyword` ist die Voreinstellung im C++ V3-Modus und das einzig mögliche Verhalten im C++ 2017- und C++ 2020-Modus. In diesem Fall wird das Präprozessor-Makro `_WCHAR_T` definiert.

`-K no_wchar_t_keyword` ist die Voreinstellung und das einzig mögliche Verhalten im Cfront-C++-Modus.

`bool`
`no_bool`

Mit diesen Argumenten legen Sie fest, ob `bool` als Schlüsselwort erkannt wird.

`-K bool` ist die Voreinstellung im C++ V3-Modus und das einzig mögliche Verhalten im C++ 2017- und C++ 2020-Modus. In diesem Fall wird das Präprozessor-Makro `_BOOL` definiert.

`-K no_bool` ist die Voreinstellung und das einzig mögliche Verhalten im Cfront-C++-Modus.

`old_for_init`
`new_for_init`

Mit diesen Argumenten legen Sie fest, wie eine Initialisierungsanweisung in `for`- und `while`-Schleifen behandelt werden soll.

`-K old_for_init`
Spezifiziert, dass eine Initialisierungsanweisung zum selben Gültigkeitsbereich wie die gesamte Schleife gehört. Dies ist die Voreinstellung im Cfront-C++-Modus.

`-K new_for_init`
Spezifiziert die neue ANSI-C++-konforme Gültigkeitsbereichsregel, die die gesamte Schleife mit ihrem eigenen implizit generierten Gültigkeitsbereich umgibt. Dies ist die Voreinstellung im C++ V3-Modus und das einzig mögliche Verhalten im C++ 2017- und C++ 2020-Modus.

`no_old_specialization`
`old_specialization`

Diese Argumente sind nur in den Sprachmodi C++ V3, C++ 2017 und C++ 2020 relevant. Sie legen fest, ob die neue Syntax für Template-Spezialisierungen `template<>` verpflichtend ist.

`-K no_old_specialization` ist Voreinstellung im C++ V3-Modus und das einzig mögliche Verhalten im C++ 2017- und C++ 2020-Modus. In diesem Fall wird das Makro `__OLD_SPECIALIZATION_SYNTAX` vom Compiler nicht definiert.

Die Angabe von `-K old_specialization` definiert der Compiler das Makro `__OLD_SPECIALIZATION_SYNTAX` implizit mit dem Wert 1.

3.2.6.2 Template-Optionen

Die folgenden Optionen sind nur in den Modi C++ 2017, C++ 2020 und C++ V3 relevant, da im Cfront-C++-Modus keine Templates unterstützt werden.

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

Diese Optionen steuern die Art der Instanziierung von Templates mit externer Linkage. Dazu zählen Funktions-Templates sowie Funktionen (nicht-statische und nicht-inline) und statische Variablen, die Elemente von Klassen-Templates sind. Im Folgenden werden diese Arten von Templates unter dem Begriff „Template-Einheiten“ zusammengefasst.

In allen Instanziierungsmodi generiert der Compiler pro Übersetzungseinheit alle Instanzen, die mit der expliziten Instanziierungsanweisung `template` *declaration* oder mit dem Instanziierungspragma `#pragma instantiate` *template-einheit* angefordert werden.

Die restlichen Template-Einheiten werden wie folgt instanziiert:

`-T none`
Außer den explizit angeforderten Instanzen werden sonst keine Instanzen generiert.

`-T auto` (Voreinstellung)
Die Instanziierung erfolgt über alle Übersetzungseinheiten hinweg durch einen Prälinker. Der Prälinker wird erst aktiviert, wenn mit dem `CC`-Kommando eine ausführbare Datei erzeugt wird oder wenn die Option `-y` (siehe "Optionen zur Auswahl von Übersetzungsphasen") angegeben wird. Beim Erzeugen einer vorgebundenen Objektdatei (Option `-r`) erfolgen keine Instanziierungen durch den Prälinker. Das Prinzip der automatischen Instanziierung ist ausführlich im Abschnitt „Automatische Instanziierung“ dargestellt.

`-T local`
Die Instanziierungen werden pro Übersetzungseinheit durchgeführt. Es werden alle Template-Einheiten instanziiert, die in einer Übersetzungseinheit benutzt werden. Dabei generierte Funktionen haben interne Linkage. Dadurch wird ein sehr einfacher Mechanismus für den Einstieg in die Template-Programmierung zur Verfügung gestellt. Der Compiler instanziiert die Funktionen, die in jeder Übersetzungseinheit benötigt werden, als lokale Funktionen. Das Programm bindet sie und läuft korrekt ab. Durch diese Methode entsteht jedoch eine Vielzahl von Kopien der instanziierten Funktionen und ist daher für die Produktion nicht empfehlenswert. Dieser Modus ist aus den gleichen Gründen nicht geeignet, wenn eines der Templates `static`-Variablen enthält.

Achtung:

Das `basic_string`-Template enthält eine `static`-Variable, um die leere Zeichenkette darzustellen. Wenn Sie die Option `-T local` und aus der Bibliothek den Typ `string` verwenden, wird die leere Zeichenkette nicht mehr erkannt. Bitte vermeiden Sie diese Kombination, weil sie zu ernsthaften Problemen führen kann.

`-T all`
Die Instanziierungen werden pro Übersetzungseinheit durchgeführt. Es werden alle Template-Einheiten instanziiert, die in einer Übersetzungseinheit benutzt oder deklariert werden. Alle Elementfunktionen und statischen Variablen eines Klassen-Templates werden unabhängig davon instanziiert, ob sie benutzt werden oder nicht. Funktions-Templates werden auch dann instanziiert, wenn sie lediglich deklariert werden.

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

Mithilfe dieser Option können Objekte und Bibliotheken angegeben werden, die bei der Bestimmung der zu generierenden Instanzen vom Prälinker in folgender Weise berücksichtigt werden:

pl_filei ist der Name einer Objektdatei (`.o`-Datei) oder einer statischen Bibliothek(`.a`-Datei).

- Wenn eine Objektdatei oder Bibliothek *pl_filei* die Definition einer Funktion oder eines statischen Datenelements enthält, wird keine Instanz einer Template-Einheit generiert, die hierzu ein Duplikat ist.
- Wenn eine Objektdatei oder Bibliothek *pl_filei* Instanzen für Template-Einheiten benötigt, werden diese Instanzen nicht generiert.

Problemstellung

Die Bibliotheken `libX.a` und `libY.a` enthalten Referenzen auf dieselben Template-Instanzen. Wenn die Objekte der beiden Bibliotheken jeweils mit der Option `-y` vorinstanziiert werden, entstehen Duplikate.

Dem Prälinker muss in solchen Fällen ein Hinweis gegeben werden, dass Symbole anderswo definiert sind und er deshalb keine Instanz generieren soll. Hierzu steht die Option `-T add_prelink_files` zur Verfügung.

Lösung

Zunächst werden die Objekte der Bibliothek `libX.a` mit der Option `-y` vorinstanziiert.

Anschließend werden die Objekte der Bibliothek `libY.a` vorinstanziiert. Dabei wird mit der Option `-T add_prelink_files,libX.a` bekanntgegeben, dass der Prälinker die Bibliothek `libX.a` berücksichtigen muss und keine Duplikate zu `libX.a` generiert.

`-T max_iterations,`*n*

Spezifiziert im automatischen Instanziierungsmodus (`-T auto`) die maximale Anzahl *n* der Prälinker-Durchläufe. Voreingestellt ist *n* = 30. Wenn für *n* der Wert 0 angegeben wird, ist die Anzahl der Prälinker-Durchläufe nicht limitiert.

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

Diese Optionen steuern die Erstellung einer ETR-Datei `datei.etr` (ETR=Explicit Template Request) für die Anwendung der expliziten Template-Instanziierung (siehe dazu Abschnitt „Generieren von expliziten Template-Instanziierungsanweisungen (ETR-Dateien)“)

Achtung:

Die Optionen `etr_file_all` und `etr_file_assigned` werden ignoriert, falls sie zusammen mit den Präprozessor-Optionen `-P / -E / -M` benutzt werden.

`-T etr_file_none`
Diese Angabe ist Voreinstellung und unterdrückt die Ausgabe der Instanziierungs-Informationen.

`-T etr_file_all`
Hiermit werden alle möglichen Template-Informationen ausgegeben.

`-T etr_file_assigned`
Es werden nur die vom Prälinker zugewiesenen instanziierten Templates ausgegeben.

`-T [no_]definition_list bzw. -T [no_]dl`

Diese Option ermöglicht eine interne Kommunikation zwischen dem Frontend und dem Prälinker während der Rekompilierungsphase der automatischen Template-Instanziierung. Weitere Einzelheiten finden Sie im Abschnitt „Automatische Instanziierung“.

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

Allgemeine Eingaberegeln zur `-K`-Option finden Sie auf "Aufruf-Syntax und allgemeine Regeln". Als Argumente *arg* zur Steuerung der Template-Instanziierung sind folgende Angaben möglich:

`assign_local_only`
`no_assign_local_only`

Diese Argumente legen fest, ob die Zuweisung von Instanziierungen nur lokal unterstützt wird. Ist `-K assign_local_only` gesetzt, gilt im Detail Folgendes:

- Instanziierungen können nur Objektdateien zugewiesen werden, die sich im aktuellen Dateiverzeichnis befinden (lokale Dateien).
- Instanziierungen können nur einer Objektdatei zugewiesen werden, wenn das aktuelle Dateiverzeichnis zum Zeitpunkt der Instanziierung dem aktuellen Dateiverzeichnis zum Übersetzungszeitpunkt entspricht.

Beispiel

```
cd dir1          # Das aktuelle Dateiverzeichnis zum
CC -c test1.c    # Übersetzungszeitpunkt von test1.c ist dir1
cd ../dir2       # Das aktuelle Dateiverzeichnis zum
CC -c test2.c    # Übersetzungszeitpunkt von test2.c ist dir2
cd ../dir1       # Das aktuelle Dateiverzeichnis für den Prälinker
                 # ist dir1
CC -K assign_local_only -o test test1.c ../dir2/test2.o
```

In diesem Beispiel ist die Zuweisung von Instanziierungen nur zur lokalen Objektdatei `test1.o` möglich.

`-K no_assign_local_only` ist voreingestellt.

`implicit_include`
`no_implicit_include`

Diese Argumente legen fest, ob die Definition eines Templates implizit inkludiert wird (siehe Abschnitt „Implizites Inkludieren“).

`-K implicit_include` ist voreingestellt.

`instantiation_flags`
`no_instantiation_flags`

`-K instantiation_flags` ist voreingestellt und bewirkt, dass spezielle Symbole generiert werden, die vom Prälinker bei der automatischen Instanziierung genutzt werden.
Bei Angabe von `-K no_instantiation_flags` werden diese Symbole nicht generiert, wodurch sich die Objektgröße reduziert. Eine automatische Instanziierung mit `-T auto` ist dann nicht möglich.

3.2.7 Optimierungsoptionen

Wenn keine der folgenden Optionen zur Optimierung angegeben wird, führt der Compiler keine Optimierungen durch. Dieses Verhalten entspricht der SDF-Option `LEVEL=*LOW`.

Die einzelnen Optimierungsmaßnahmen und ihre Auswirkungen sind ausführlich im C/C++-Benutzerhandbuch [4] im Abschnitt „Verlauf der Optimierung“ dargestellt.

`-O`
`-F O2`

Diese Optionen schalten die Standardoptimierung des Compilers ein. Der Unterschied zwischen diesen beiden Optionen besteht darin, dass bei `-O` intern jede Optimierungsstrategie nur einmal durchgeführt wird, bei `-F O2` mehrmals. Entsprechend wird in der Optimierungsstufe `-O` deutlich weniger Übersetzungszeit benötigt als in der „hochoptimierenden“ Stufe `-F O2`.

Der Compiler führt folgende Standardoptimierungen durch:

- Berechnung konstanter Ausdrücke zur Übersetzungszeit
- Optimierung der Indexrechnung in Schleifen
- Eliminierung unnötiger Zuweisungen
- Propagation konstanter Ausdrücke
- Eliminierung redundanter Ausdrücke
- Optimierung von Sprüngen auf unbedingte Sprungbefehle

Außerdem wird eine Registeroptimierung durchgeführt.

Im Unterschied zur SDF-Option (Optimierungsstufe *HIGH bzw. *VERY-HIGH ohne Parameter) ist die Schleifenexpansion ausgeschaltet.

Wenn die Standardoptimierung nicht explizit mit `-O` oder `-F O2` eingeschaltet ist, wird sie automatisch in der Stufe `-O` aktiviert, wenn die Optionen `-F loopunroll` (Schleifenexpansion) oder `-F i`, `-F inline_by_source` (Inline-Generierung von Benutzer-Funktionen) angegeben werden.

`-F I`[*name*]

Mit dieser Option kann angegeben werden, für welche C-Bibliotheksfunktionen die Implementierung im CRTE angenommen werden kann. Dies erlaubt eine bessere Optimierung des Programms.
Bei Angabe von -F I ohne *name* werden alle Aufrufe zu bekannten C-Bibliotheksfunktionen gesondert behandelt.
Wird die Option -F I nicht angegeben, so wird kein Aufruf gesondert behandelt.
Bei Angabe von -F I*name* (ohne trennendes Leerzeichen) wird nur die Funktion *name* gesondert behandelt.
Sollen mehrere Funktionen gesondert behandelt werden, muss die Option -F I*name* mehrfach angegeben werden.

Die Option -F I kann unabhängig von der normalen Optimierung angegeben werden.

Den größten Effekt erreicht der Compiler durch Inline-Generierung einer Funktion. Dabei wird der Funktionscode direkt in die Aufrufstelle eingesetzt. Zeitaufwendige Verwaltungsaktivitäten des Laufzeitsystems (z.B. Register retten und restaurieren, Rücksprung aus der Funktion) fallen weg. Die Progamm-Ablaufzeit wird damit verkürzt.

Folgende C-Bibliotheksfunktionen können inline generiert werden:

abs	strcat
fabs	strlen
labs	strcmp
memcmp	strncmp
memcpy	strcpy
memset	

Inline generierte Funktionen können weder zum Bindezeitpunkt durch andere Funktionen ersetzt noch beim Testen mit AID als Testpunkte benutzt werden.

Für die Inline-Generierung von C-Bibliotheksfunktionen braucht die Standardoptimierung des Compilers nicht eingeschaltet zu sein.

Der Compiler kennt die Semantik der CRTE-Bibliotheksfunktionen. Mit der Option `-F I` *name* teilt man dem Compiler mit, optimierte Funktionen zu generieren, die semantisch den CRTE-Bibliotheksfunktionen entsprechen. Ist kein Name angegeben, sollte der Compiler sein Wissen um alle CRTE-Funktionen nutzen (dem Compiler sind etwa 150 Funktionen bekannt).

Nicht inline generierte Funktionen bleiben als Aufruf erhalten. Es sind jedoch Optimierungen möglich, die bei Benutzer-Funktionen nicht machbar sind. Zum Beispiel kann der Compiler die Information nutzen, dass die Funktion `isdigit()` keine Seiteneffekte hat.

Einige Funktionen sind sehr speziell, da sie komplett inline generiert werden. Für diese Funktionen erzeugt der Compiler den Code direkt, ohne an CRTE zu übergeben. Diese Funktionen sind in der obigen Tabelle aufgeführt.

In einigen Fällen ist diese Optimierung unerwünscht. Falls das Programm fehlerbereinigt werden soll, möchten Sie evtl. einen Breakpoint in einer solchen Funktion setzen. Dies ist bei komplett inline generierten Funktionen nicht möglich. (Genauer: Sie können zwar einen Breakpoint setzen, doch wird dieser nicht erreicht. Benutzt wird der vom Compiler generierte Code, nicht die Funktion, bei der der Breakpoint gesetzt wurde.)

Ein anderer Fall liegt vor, wenn eine Funktion mit einem dem Compiler bekannten Namen selbst definiert wurde. In den meisten Fällen wird diese Funktion eine von CRTE unterschiedliche Semantik besitzen. Kommt es zu einem Konflikt zwischen einer solchen selbstdefinierten Funktion und dieser Option, vermuten alle Aufrufe die CRTE-Semantik. Die Warnung CFE2067 wird in diesem Fall ausgegeben.

Beachten Sie, dass die CRTE-Semantik in jeder Übersetzungseinheit benutzt wird. Die Warnung wird nur in der Übersetzungseinheit ausgegeben, die die private Definition enthält.

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

Diese alternativ zu verwendenden Optionen steuern die Inline-Generierung von benutzereigenen Funktionen. Wie auch bei der Inline-Generierung von einigen C-Bibliotheksfunktionen aus der Standardbibliothek (siehe `-F I`) wird jeder Aufruf einer Inline-Funktion durch den entsprechenden Funktionscode ersetzt; durch die Einsparung der Aufruf- und Rücksprung-Codefolge wird eine bessere Ablaufzeit erzielt. Bei Angabe der Optionen `-F i`, `-F i` *name* oder `-F inline_by_source` wird gleichzeitig die Standardoptimierung (`-O`) aktiviert, es sei denn `-F O2` wurde explizit gesetzt.

`-F i` und `-F i` *name*
Bei Angabe von `-F i` mit oder ohne *name* wählt der Compiler Funktionen für die Inline-Generierung nach eigenen Kriterien aus. Ggf. im Quellprogramm vorhandene inline-Pragmas und C++-sprachspezifische Inline-Funktionen werden vom Compiler bei der Suche nach geeigneten Kandidaten automatisch mitberücksichtigt (siehe auch `-F inline_by_source`).

Bei Angabe von *name* (ohne trennendes Leerzeichen!) wird zusätzlich die Funktion *name* inline generiert. Sollen mehrere selbstgewählte Funktionen vom Compiler bei der Inline-Generierung berücksichtigt werden, muss die Option `-F i` *name* mehrmals angegeben werden.
Die Angabe von `-F i` *name* wird bei C++-Übersetzungen, d.h. beim `CC`-Kommando ignoriert.

`-F inline_by_source`
Bei Angabe dieser Option werden ausschließlich folgende benutzereigene Funktionen inline generiert:

- Bei C-Übersetzungen (`cc`, `c11`, `c89`) C-Funktionen, die mit der Anweisung `#pragma inline` *name* deklariert sind (siehe auch Abschnitt „inline-Pragma“ im C/C++-Benutzerhandbuch [4]). Das inline-Pragma wird in C++ nicht unterstützt.
- Bei C++-Übersetzungen (`CC`) die C++-sprachspezifischen Inline-Funktionen. Dies sind innerhalb von Klassen definierte Funktionen sowie Funktionen mit dem Attribut `inline`.

Hinweis zu Inline-Funktionen in C++
Die Inline-Generierung der C++-sprachspezifischen Inline-Funktionen wird auch dann durchgeführt, wenn die Optimierung nicht eingeschaltet ist bzw. wenn die Optionen `-F i` oder `-F inline_by_source` nicht gesetzt sind. Dies kann mit der Option `-F no_inlining` unterdrückt werden.

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

Diese Option steuert die Expansion von Schleifen. Durch eine mehrmalige Expansion des Schleifenrumpfes wird eine geringere Ausführungszeit für die Schleifendurchläufe erzielt. Standardmäßig unterbleibt diese Optimierungsmaßnahme. Bei Angabe dieser Option wird automatisch die Standardoptimierung (`-O`) aktiviert, es sei denn `-F O2` wurde explizit gesetzt.

Bei Angabe von `-F loopunroll` ohne *n* werden vom Compiler Schleifenrümpfe 4 mal expandiert. Mit *n* kann ein eigener Expansionsfaktor ausgewählt werden, wobei *n* ein Wert zwischen 1 und 100 sein kann.

Die Angabe von `-F loopunroll[,`*n*`]` garantiert nicht, dass der Optimierer in jedem Fall eine Schleifenexpansion ausführt: Der Optimierer entscheidet dies in Abhängigkeit von der Schleifenstruktur sowie vom angegebenen Faktor *n*.

`-F no_inlining`

Mit dieser Option kann die Inline-Generierung der C++-sprachspezifischen Inline-Funktionen unterdückt werden, die standarmäßig auch dann durchgeführt wird, wenn die Optionen `-F i` oder `-F inline_by_source` nicht angegeben werden.
Wenn `-F no_inlining` gleichzeitig mit der Option `-F i` oder `-F inline_by_source` angegeben wird, gilt die jeweils letzte Angabe in der Kommandozeile.
Ist `-F no_inlining` als Letztes angegeben, wird die ursprünglich angeforderte Inline-Generierung auch von benutzereigenen C-Funktionen nicht durchgeführt (die implizit gesetzte Optimierung `-O` bleibt allerdings eingeschaltet).
Die mit der Option `-F I` eingeschaltete Inline-Generierung von C-Bibliotheksfunktionen wird durch `-F no_inlining` nicht beeinflusst.

3.2.8 Optionen zur Objektgenerierung

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

Allgemeine Eingaberegeln zur `-K`-Option finden Sie im Abschnitt "Aufruf-Syntax und allgemeine Regeln".
Als Argumente *arg* zur Objektgenerierung sind folgende Angaben möglich:

Assembler-Befehle für Unterprogrammeinsprünge

```
subcall_basr
subcall_lab
```

`-K subcall_basr` (Voreinstellung)
Standardmäßig wird der Befehl BASR generiert.

`-K subcall_lab`
Es werden die prozessorunabhängigen Assembler-Befehle LA und B generiert. Programme mit dieser Befehlsfolge sind auf allen 7500-Anlagen ablauffähig.

Achtung: Diese Option ist in den Modi C++V3, C++ 2017 und C++ 2020 nicht erlaubt.

Generierung des ETPND-Bereichs

Mit den folgenden Optionen wird die `#pragma`-Anweisung zur Erzeugung eines ETPND-Bereichs (siehe Abschnitt „ETPND-Pragma“ im C/C++-Benutzerhandbuch [4]) ignoriert bzw. das Datum-Format des ETPND-Bereichs festgelegt.

```
no_etpnd
calendar_etpnd
julian_etpnd
```

`-K no_etpnd` (Voreinstellung)
Standardmäßig wird kein ETPND-Bereich generiert.

`-K calendar_etpnd`
Das Format des Datums im ETPND-Bereich wird wie folgt festgelegt: 8 Byte Kalenderdatum - 4 Byte Ladeadresse.

`-K julian_etpnd`
Das Format des Datums im ETPND-Bereich wird wie folgt festgelegt: 6 Byte Kalenderdatum - 3 Byte julianisches Datum - 4 Byte Ladeadresse.

Generierung des Entry-Codes für Funktionsaufrufe

```
ilcs_opt
ilcs_out
```

`-K ilcs_opt` (Voreinstellung)
Der ILCS-Entry-Code wird inline generiert. Die Laufzeit des erzeugten Objektes wird beschleunigt.

`-K ilcs_out`
Der ILCS-Entry-Code für Funktionsaufrufe wird im Laufzeitsystem angesprungen. Dadurch reduziert sich das Code-Volumen des Moduls.

Behandlung von enum-Daten

`enum_value`
`enum_long`

`-K enum_value` (Voreinstellung)
Standardmäßig werden die `enum`-Daten abhängig vom Wertebereich auf `char`, `short` oder `long` abgebildet.

`-K enum_long`
`enum`-Daten werden immer wie Objekte vom Typ `long` behandelt.

Generierung der Entry-Namen bei LLMs

`llm_convert`
`llm_keep`

`-K llm_convert` (Voreinstellung)
Standardmäßig werden bei der Generierung von Entry-Namen Unterstriche in Dollarzeichen umgewandelt.

`-K llm_keep`
Bei der Generierung von Entry-Namen werden die Unterstriche beibehalten.

Die Umsetzung des Unterstrichs betrifft in den C-Sprachmodi alle externen Symbole, in den C++-Sprachmodi nur die mit `extern "C"` deklarierten Symbole (nicht die Entry-Namen der C-Bibliotheksfunktionen). Bei der Kodierung von externen C++-Symbolen werden generell Unterstriche beibehalten.

`no_llm_case_lower`
`llm_case_lower`

`-K no_llm_case_lower` (Voreinstellung)
Standardmäßig werden bei der Generierung von Entry-Namen Klein- in Großbuchstaben umgewandelt.

`-K llm_case_lower`
Bei der Generierung von Entry-Namen wird die Kleinschreibung beibehalten.

Die Umwandlung von Klein- in Großbuchstaben betrifft in den C-Sprachmodi und im Cfront-C++-Modus alle externen Symbole, in den Sprachmodi C++ 2017, C++ 2020 und C++ V3 nur die mit `extern "C"` deklarierten Symbole. Bei der Codierung von externen C++-Symbolen in den Sprachmodi C++ 2017, C++ 2020 und C++ V3 werden generell Kleinbuchstaben beibehalten.

Achtung:

Die C-Bibliotheksfunktionen werden nur dann vollständig unterstützt, wenn eine der beiden folgenden Option-Kombinationen spezifiziert wurde:

- `-K llm_convert` und `-K no_llm_case_lower`
- `-K llm_keep` und `-K llm_case_lower`

`csect_suffix=` *suffix*
`csect_hashpath`

Diese Optionen spezifizieren die Art, in der CSECT-Namen gebildet werden. Standardmäßig wird der CSECT-Name vom Modulnamen abgeleitet und der Modulname wird – sofern nicht ausdrücklich spezifiziert – vom Quellnamen abgeleitet.
Die Optionen können genutzt werden, um unterschiedliche CSECT-Namen zu generieren, falls die Objektnamen dieselben sind.

Mithilfe beider Optionen entsteht eine 30 Zeichen lange Zeichenkette als Basis für die realen CSECT-Namen. Diese Basis kann mittels `'-K verbose / -v'` angezeigt werden.

Die Basis wird wie üblich geändert durch:

- umwandeln aller Kleinbuchstaben in Großbuchstaben
- ändern aller Sonderzeichen wie z. B. '_' oder '.' nach '$'
- hinzufügen von '&@' oder '&#', um reale CSECT-Namen zu generieren.

Mithilfe dieser Optionen selektieren Sie unterschiedliche Suffixes, die an den Objektnamen angehängt werden. Wird der Objektname länger als 30 Zeichen (abzüglich der Suffixlänge), so wird er abgeschnitten.

`-K csect_suffix=`
Mit dieser Option spezifizieren Sie einen benutzerdefinierten Suffix. Es werden max. 10 Stellen benutzt.

`-K csect_hashpath`
Mit dieser Option wird aus dem vollen Objektpfadnamen (inklusive '..'; Links werden nicht expandiert) eine Zeichenfolge von 7 Zeichen generiert. Diese Zeichenfolge wird als Suffix benutzt.

Ablage von const-Objekten

`no_roconst`
`roconst`

`-K no_roconst` (Voreinstellung)
Standardmäßig werden Objekte vom Typ `const` im Datenmodul abgelegt (WRITEABLE).

`-K roconst`
Objekte vom Typ `const` werden im Codemodul (READ-ONLY) abgelegt. Die Konstanten können nicht überschrieben werden, auch wenn mit einem `cast`-Operator das `const`-Attribut entfernt wird.

Achtung: Nur globale oder lokale `static` -Konstanten sind betroffen. Lokale `auto`-Variablen mit dem Typattribut `const` können nicht im READ-ONLY Bereich abgelegt werden.

Ablage von Zeichenketten-Konstanten

`no_rostr`
`rostr`

`-K no_rostr` (Voreinstellung)
Standardmäßig werden Zeichenketten-Konstanten im Datenmodul abgelegt (WRITEABLE), so dass die Werte überschrieben werden können, sobald das `const`-Attribut mit einem `cast`-Operator gelöscht wurde.

`-K rostr`
Zeichenketten-Konstanten und Aggregat-Initialisierungskonstanten werden im Codemodul (READ-ONLY) abgelegt.

Gleitpunkt-Arithmetik im /390- und IEEE-Format

```
no_ieee_floats
ieee_floats
```

`-K no_ieee_floats` (Voreinstellung)
Standardmäßig werden Gleitpunkt-Datentypen und -Operationen im /390-Format verwendet.

`-K ieee_floats`
Gleitpunkt-Datentypen und -Operationen werden im IEEE-Format verwendet. Dies betrifft alle Variablen und Konstanten der Datentypen `float`, `double` und `long double` innerhalb eines C/C++-Programms.

Achtung:

Je nachdem, ob für Gleitpunkt-Datentypen und -Operationen das IEEE-Format oder das /390-Format verwendet wird, kann dasselbe C/C++-Programm aus folgenden Gründen unterschiedliche Ergebnisse liefern:

- IEEE-Gleitpunkt-Zahlen verwenden eine andere interne Darstellung als /390-Gleitpunkt-Zahlen.
- C++-Bibliotheksfunktionen unterstützen nicht das IEEE-Format und müssen deshalb gegebenenfalls durch C-Funktionen ersetzt werden.
- IEEE-Gleitpunkt-Operationen unterscheiden sich semantisch von den entsprechenden /390-Gleitpunkt-Operationen, z.B. beim Runden. So wird im IEEE-Format standardmäßig „Round to Nearest“ angewendet anstatt „Round to Zero“ wie beim /390-Format.

Voraussetzungen für die Verwendung des IEEE-Formats:

- Inkludieren Sie für jede in Ihrem Programm verwendete CRTE-Funktion (C-Bibliotheksfunktion), die mit Gleitpunkzahlen arbeitet, die zugehörige Include-Datei. Andernfalls können diese Funktionen die Gleitpunktzahlen nicht korrekt verarbeiten. Insbesondere müssen Sie für die Funktion *printf()* die Include-Datei <stdio.h> mit `#include <stdio.h>` inkludieren.
- CRTE enthält einige C-Bibliotheksfunktionen, die das IEEE-Format für Gleitpunktzahlen verwenden. Für die korrekte Verwendung der IEEE-Funktionsnamen müssen Sie zusätzlich zur Option `ieee_floats` die beiden folgenden Optionen angeben:
  `-K llm_keep`
  `-K llm_case_lower`

Generierung von gemeinsam benutzbarem Code

```
no_share
share
```

`-K no_share` (Voreinstellung)
Standardmäßig erzeugt der Compiler keinen gemeinsam benutzbaren Code.

`-K share`
Der Compiler erzeugt gemeinsam benutzbaren Code, bestehend aus einer gemeinsam benutzbaren Code-CSECT und einer nicht gemeinsam benutzbaren Daten-CSECT. Module mit gemeinsam benutzbarem Code können nur in BS2000-Umgebung (SDF) sinnvoll weiterverarbeitet werden.

Ablage von Hilfsvariablen

```
workspace_static
workspace_stack
```

`-K workspace_static` (Voreinstellung)
Standardmäßig werden Hilfsvariablen im statischen Datenbereich angelegt.

`-K workspace_stack`
Die für Hilfsvariablen benötigten Daten werden auf dem Stack abgelegt.

Mehrfachdefinition von extern sichtbaren Variablen

`external_multiple`
`external_unique`

`-K external_multiple`
Eine extern sichtbare Variable, die in mehreren Modulen definiert ist, wird nur genau einem Speicherbereich zugeordnet.
Um dies zu erreichen, darf die Variable bei keiner Definition statisch initialisiert werden. Der Compiler legt den Speicher für diese Variable im COMMON-Bereich an. Wenn die Variable bei der Definition statisch initialisiert ist, wird der Speicher im Datenbereich angelegt. Die Zuordnung zu genau einem Speicherbereich ist dann nicht möglich.
Dieses Verhalten ist im K&R-C-Modus voreingestellt.

`-K external_unique`
Extern sichtbare Variablen dürfen nur in genau einem Modul definiert werden und müssen in allen anderen Modulen als `extern` deklariert werden. Der Speicherplatz für solche Variablen wird im Datenmodul desjenigen Objekts angelegt, in dem die Variable definiert wurde. Dieses Verhalten ist in den Sprachmodi C89, C11 und in allen C++-Sprachmodi voreingestellt. In den C++-Sprachmodi darf diese Voreinstellung nicht geändert werden.

Länge von externen C-Namen

Die folgenden Optionen legen die Länge von externen C-Namen fest und betreffen in den C-Sprachmodi alle externen Symbole, in den C++-Sprachmodi nur die mit `extern "C"` deklarierten Symbole (nicht die Entry-Namen der C-Bibliotheksfunktionen).

`c_names_std`
`c_names_unlimited`
`c_names_short`

`-K c_names_std` (Voreinstellung)
Standardmäßig sind externe C-Namen maximal 32 Zeichen lang. Längere Namen werden vom Compiler auf 32 Zeichen verkürzt. Bei der Generierung von gemeinsam nutzbarem Code (`-K share`) können nur 30 Zeichen genutzt werden.

`-K c_names_unlimited`
Es findet keine Namensverkürzung statt. Der Compiler generiert in diesem Fall Entry-Namen im EEN-Format. EEN-Namen können eine Länge von maximal 32000 Zeichen haben. Module, die EEN-Namen enthalten, werden vom Compiler im LLM-Format 4 abgelegt. Ausführliche Informationen zur Weiterverarbeitung von LLMs im Format 4 finden Sie im Abschnitt "Binder-Optionen" (`-B extended_external_names`). EEN-Namen werden im Cfront-C++-Modus nicht unterstützt.

`-K c_names_short`
Externe C-Namen werden auf 8 Zeichen gekürzt.

Hinweis

Optionen, die die Länge von externen Namen beeinflussen, wirken auch auf die Namen von Static-Funktionen, da der Compiler Namen von Static-Funktionen wie die Namen von externen Funktionen behandelt.

3.2.9 Testhilfe-Option

`-g`

Der Compiler erzeugt zusätzliche Informationen (LSD) für die Testhilfe AID. Standardmäßig werden keine Testhilfeinformationen erzeugt.

Ein Programm, d.h. eine vom Binder erzeugte ausführbare Datei, kann in der POSIX-Shell mit dem Kommando `debug` getestet werden. Nach Eingabe dieses Kommandos befindet man sich im BS2000-Systemmodus (angezeigt durch %DEBUG/).
Die Eingabe der AID-Kommandos erfolgt dann wie im Handbuch „AID Testen von C/C++-Programmen“ [11] beschrieben. Nach Beendigung des Programms ist wieder die POSIX-Shell die aktuelle Umgebung.

Die Beschreibung des `debug`-Kommandos finden Sie im Handbuch „POSIX Kommandos“ [3].

3.2.10 Laufzeit-Optionen

Mit den folgenden Optionen kann bei der Übersetzung des Moduls, das die `main`-Funktion enthält, das Laufzeitverhalten des Programms beeinflusst werden. Bei der Übersetzung anderer Module haben diese Optionen keinen Effekt.

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

Allgemeine Eingaberegeln zur `-K`-Option finden Sie im Abschnitt "Aufruf-Syntax und allgemeine Regeln".
Als Argumente *arg* zur Steuerung des Laufzeitverhaltens sind folgende Angaben möglich:

`integer_overflow`
`no_integer_overflow`

`-K integer_overflow` (Voreinstellung)
Standardmäßig ist die Programmaske gemäß ILCS-Konvention auf X‘0C‘ gesetzt.

`-K no_integer_overflow`
Die Programmaske wird auf X’00’ eingestellt.

Die beiden Programm-Masken haben folgende Wirkung:

	X‘0C‘	X‘00‘
Festpunkt-Überlauf	zugelassen	unterdrückt
Dezimal-Überlauf	zugelassen	unterdrückt
Exponenten-Unterlauf	unterdrückt	unterdrückt
Mantisse Null	unterdrückt	unterdrückt

Hinweise
Bei Sprachmix darf die ILCS-Programmaske nicht verändert werden!

Die Auswahl der generierten Befehle ist nicht von der Option `-K integer_overflow` beeinflusst. Daher bedeutet das Zulassen von INTEGER-OVERFLOW nicht, dass in jedem Fall ein Überlauf ausgelöst wird.

`prompting`
`no_prompting`

`-K prompting` (Voreinstellung)
Bei Aufruf des Programms aus der BS2000-Umgebung (SDF) wird eine Prompting-Zeile ausgegeben, in der Parameter an die `main`-Funktion oder Umlenkungen der Standard-Ein-/Ausgabeströme angegeben werden können.

`-K no_prompting`
Die Ausgabe der Prompting-Zeile wird unterdrückt.

Bei Programmstart aus der POSIX-Shell hat diese Option keine Bedeutung, da in diesem Fall die Parameter immer in der Aufrufzeile angegeben werden.

`statistics`
`no_statistics`

`-K statistics` (Voreinstellung)
Bei Beendigung eines mit dieser Option generierten Programms wird die verbrauchte CPU-Zeit ausgegeben. Dies geschieht jedoch nur, wenn das Programm ins BS2000 transferiert und dort gestartet wird.

`-K no_statistics`
Die Ausgabe der verbrauchten CPU-Zeit wird unterdrückt.

`stacksize=`*n*

Mit der Option `-K stacksize` kann durch Angabe einer Zahl *n* (8 bis 99999) festgelegt werden, wie viele Kilobytes für das erste Segment des C-Laufzeitstacks reserviert werden sollen. Voreingestellt sind 64 Kilobytes.

`environment_encoding_std`
`environment_encoding_ebcdic`

Durch diese Optionen kann die Kodierung von externen Zeichenketten, wie Argumenten von `main` und Umgebungsvariablen, gesteuert werden.

`–K environment_encoding_std` (Voreinstellung).
Die externen Zeichenketten werden so kodiert, wie es bei den Optionen `–K literal_encoding_ascii`, `–K literal_encoding_ascii_full`, `–K literal_encoding_ebcdic` bzw. `–K literal_encoding_ebcdic_full` angegeben wurde (siehe "Gemeinsame Frontend-Optionen in C und C++").

`–K environment_encoding_ebcdic`
Diese Option wird aus Kompatibilitätsgründen angeboten. Trotz der Angabe von `–K literal_encoding_ascii` bzw. `–K literal_encoding_ascii_full` werden externe Zeichenketten in EBCDIC kodiert.

Folgende Tabelle verdeutlicht die Optionskombinationen und die Kodierung der externen Zeichenketten:

	environment_encoding_std	**environment_encoding_ebcdic**
literal_encoding_ebcdic*	EBCDIC	EBCDIC
literal_encoding_ascii*	ASCII	EBCDIC

3.2.11 Binder-Optionen

Folgende Optionen an den Binder werden nicht ausgewertet, wenn gleichzeitig eine der Optionen `-c`, `-E` , `-M` oder `-P` (Beendigung des Compilerlaufs nach der Übersetzung bzw. nach dem Präprozessorlauf, siehe "Optionen zur Auswahl von Übersetzungsphasen") angegeben wird.

`-B extended_external_names`
`-B short_external_names`

Diese Option wird benötigt, wenn das zu bindende Programm auf sehr alten Anlagen ablaufen soll. Auf Anlagen mit BS2000/OSD Version 1 bis 3 oder dem Produkt BLSSERV mit Version kleiner als 2.0 können keine LLMs mit langen Namen (EEN) verarbeitet werden. Für solche Anlagen muss das generierte Element ein LLM-Format 1 haben.

EEN-Namen, d.h. ungekürzte externe C++-Symbole, sind generell in Modulen enthalten, die mit dem Compiler im C++ 2017, C++ 2020 bzw. C++ V3-Modus erzeugt werden.
Ungekürzte externe C-Symbole werden nur dann generiert, wenn bei der Übersetzung die Option `-K c_names_unlimited` angegeben wird (siehe "Optionen zur Objektgenerierung").
In diesem Fall werden auch externe C-Symbole vom Compiler nicht auf 32 Bytes verkürzt.
Module mit EEN-Namen werden vom Compiler im LLM-Format 4 abgelegt. Die Module der im C++ 2017, C++ 2020 bzw. C++ V3-Modus verwendeten C++-Bibliotheken und -Laufzeitsysteme des CRTE liegen ebenfalls im LLM-Format 4 vor.

Wenn die vom Compiler erzeugten Module keine EEN-Namen enthalten, d.h. im LLM-Format 1 vorliegen, spielt diese Option keine Rolle, da der Binder in diesem Fall generell das dem Eingabeformat entsprechende LLM-Format 1 erzeugt.

Standardmäßig generiert der BINDER das LLM-Format 4. Die EEN-Namen bleiben im Ergebnismodul ungekürzt erhalten. LLMs im Format 4 können unvollständig, d.h. mit offenen Externbezügen auf EEN-Namen gebunden und beliebig mit dem Binder weiterverarbeitet werden.

`-B extended_external_names`
Diese Angabe wird nur aus Kompatibilitätsgründen unterstützt.

`-B short_external_names`
Diese Angabe wird benötigt, wenn der Binder das LLM-Format 1 generieren soll.

In diesem Fall müssen alle Symbole mit langen Namen (EEN) befriedigt werden. Es bleibt kein EEN im generierten LLM bestehen. Bleibt hier eine offene Referenz, so bleibt sie auf Dauer offen und kann nicht nachträglich befriedigt werden.

Zusammenfassung der generierten LLM-Formate

Eingabeformat	**Option -B**	**Ausgabeformat**
LLM 1	Keine Angabe / extended_external_names / short_external_names	LLM 1
LLM 4 (EEN)	Keine Angabe / extended_external_names	LLM 4
	short_external_names	LLM 1

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

Diese Option hat Auswirkungen auf das Einbinden des C-Laufzeitsystems.

Standardmäßig, d. h. ohne Angabe der Option oder bei Angabe von -d y, wird für die C-Standardbibliothek libc.a ein RESOLVE auf die Bibliothek SYSLNK.CRTE.PARTIAL-BIND abgesetzt. Anstelle des kompletten C-Laufzeitsystems wird nur ein Verbindungsmodul eingebunden, das alle offenen Externverweise auf das C-Laufzeitsystem befriedigt. Das C-Laufzeitsystem selbst wird zum Ablaufzeitpunkt dynamisch nachgeladen, und zwar entweder aus dem Klasse-4-Speicher, falls das C-Laufzeitsystem vorgeladen ist, oder aus der Bibliothek SYSLNK.CRTE.

Bei Angabe von `-d n` wird das C-Laufzeitsystem komplett aus der Bibliothek SYSLNK.CRTE eingebunden.

Mit `-d compl` wird die „Complete-Partial-Bind“-Technik des CRTE unterstützt. Dazu wird die Bibliothek SYSLNK.CRTE.COMPL eingebunden.
Eine ausführliche Beschreibung der „Complete-Partial-Bind“-Technik finden Sie im Handbuch „CRTE“ [5].

Im C++ V3-Modus wird anstelle der Standardbibliotheken SYSLNK.CRTE.RTSCPP und SYSLNK.CRTE.STDCPP die spezielle Bibliothek SYSLNK.CRTE.CPP-COMPL eingebunden. Diese Bibliothek wird ebenfalls anstelle von SYSLNK.CRTE.TOOLS verwendet.

Hinweis
Im CFRONT-C++-Modus wird die „Complete-Partial-Bind“-Technik nicht unterstützt. Die Option `-d compl` wird in diesem Fall auf `-dy` zurückgesetzt.

Im C++ 2017- und C++ 2020-Modus wird die „Complete-Partial-Bind“-Technik nicht unterstützt. Die Option `-d compl` wird in diesem Fall mit einem Fehler abgewiesen.

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

Allgemeine Eingaberegeln zur `-K`-Option finden Sie im Abschnitt "Aufruf-Syntax und allgemeine Regeln".
Als Argumente *arg* zur Steuerung des Binders sind folgende Angaben möglich:

```
link_stdlibs
no_link_stdlibs
```

`-K link_stdlibs` ist voreingestellt und bewirkt, dass bestimmte Standardbibliotheken automatisch eingebunden werden (siehe auch Option `-l`). Das heißt, für diese Bibliotheken werden intern die entsprechenden `-l`-Optionen automatisch abgesetzt:

1. nur beim `CC`-Kommando
   - `-l Cxx` im C++ 2017- und C++ 2020-Modus
   - `-l Cstd` im C++ V3-Modus
   - `-l C` im Cfront-C++-Modus
2. immer
   - `-l c`

Bei Angabe von `-K no_link_stdlibs` werden die o.g. Bibliotheken nicht automatisch eingebunden. `-K no_link_stdlibs` wird automatisch gesetzt, wenn mit der Option `-r` eine vorgebundene Objektdatei erzeugt wird.

`-l x`

Diese Option veranlasst den Binder, beim Auflösen von Externverweisen per Autolink die Bibliothek mit dem Namen `lib` *x*.a zu durchsuchen. Standardmäßig durchsucht der Binder folgende Dateiverzeichnisse in der angegeben Reihenfolge nach der Bibliothek:

1. die mit `-L` angegebenen Dateiverzeichnisse
2. entweder die mit der Option `-Y P` angegebenen Dateiverzeichnisse oder das Standard-Dateiverzeichnis `/usr/lib`.

`-l` *x* zählt zur Kategorie der Operanden und kann auch nach Beendigung der Optioneneingabe mit `--` angegeben werden (siehe auch „Operanden“ (Aufruf-Syntax und allgemeine Regeln)).

Die Standardbibliotheken des C- und C++-Laufzeitsystems sind nicht im Dateiverzeichnis `/usr/lib` des POSIX-Dateisystems installiert, sondern als PLAM-Bibliotheken im BS2000.

Zuordnung der Standardkürzel *x* zu den BS2000-PLAM-Bibliotheken:

x	**Bibliotheksname**	**Inhalt**
`c`	`SYSLNK.CRTE.PARTIAL-BIND`	Verbindungsmodul zum dynamischen Nachladen des C-Laufzeitsystems (Standardfall)
	`SYSLNK.CRTE`	Einzelmodule zum kompletten Einbinden des C-Laufzeitsystems (bei `-d n`)
`m`	siehe c	
`C`	siehe c	
	`SYSLNK.CRTE.CFCPP`	Cfront-C++-Laufzeitsystem
	`SYSLNK.CRTE.CPP`	Cfront-C++-Bibliothek für Ein-/Ausgabe und komplexe Mathematik
`Cstd`	siehe c	
	`SYSLNK.CRTE.RTSCPP`	C++ V3-Laufzeitsystem
	`SYSLNK.CRTE.STDCPP`	C++ V3-Standard-Bibliothek
`Cxx`	siehe c	
	siehe Cxx1 oder Cxx2	C++ 2017 oder C++ 2020 Bibliothek, je nach Option `-K library_version`
`Cxx1`	siehe c	
	`SYSLNK.CRTE.CXX01`	C++ 2017 Bibliothek in der Bibliotheks-Version 1
`Cxx2`	siehe c	

	SYSLNK.CRTE.CXX02	C++ 2017 oder C++ 2020 Bibliothek in der Bibliotheks-Version 2
RWtools	SYSLNK.CRTE.TOOLS	C++-V3-Bibliothek Tools.h++

Der Binder befriedigt die offenen Externverweise aus diesen PLAM-Bibliotheken nur, wenn die Option `-l x` verwendet wird, nicht bei Angabe des expliziten Pfadnamens (z.B. `/usr/lib/libRWtools.a`) mithilfe des Operanden *datei*.*suffix* (siehe "Aufruf-Syntax und allgemeine Regeln")!

Beim Aufruf des Compilers in den C-Sprachmodi wird als letzte `-l`-Option implizit `-l c` hinzugefügt, beim Aufruf im Cfront-C++-Modus `-l C`, beim Aufruf im C++ V3-Modus `-l Cstd` und beim Aufruf im C++ 2017 oder C++ 2020 Modus `-l Cxx` (gilt nicht bei `-K no_link_stdlibs`).

Die Reihenfolge und die Position, in der die `-l`-Optionen und ggf. Objektdateien (bzw. Quelldateien, aus denen der Compiler Objektdateien generiert) in der Kommandozeile angegeben werden, sind für den Bindevorgang signifikant.
Beispielsweise würde mit dem Kommando `CC -X v3-compatible test.c -l RWtools` das Programm ordnungsgemäß gebunden, das Kommando `CC -X v3-compatible -l RWtools test.c` jedoch zu einem Fehler führen.

`-l BLSLIB`

Diese Option veranlasst den Binder, PLAM-Bibliotheken zu durchsuchen, die mit den Shell-Umgebungsvariablen BLSLIB*nn* (00 <= *nn* <= 99) zugewiesen wurden.
Die Umgebungsvariablen müssen vor Aufruf des Compilers mit den Bibliotheksnamen versorgt und mit dem POSIX-Kommando `export` exportiert werden. Die Bibliotheken werden in aufsteigender Reihenfolge *nn* durchsucht.

Alle mit BLSLIBxx angegebenen Bibliotheken werden zyklisch durchsucht. Der BINDER behandelt die Bibliotheken so, als ob sie in **einer** RESOLVE-Anweisung als Liste angegeben worden wären.

`-l BLSLIB` zählt zur Kategorie der Operanden und kann auch nach Beendigung der Optioneneingabe mit `--` angegeben werden (siehe auch „Operanden“ (Aufruf-Syntax und allgemeine Regeln)).

Beispiel

Die mit BLSLIB00 zugewiesene Bibliothek enthält offene Externverweise auf die mit BLSLIB01 zugewiesene Bibliothek, diese wiederum enthält offene Externverweise auf die BLSLIB00-Bibliothek (sog. Rückbezüge).

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

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

`-L dvz`

Mit *dvz* wird ein zusätzliches Dateiverzeichnis angegeben, in dem der Binder nach den mit `-l`-Optionen angegebenen Bibliotheken suchen soll. Standardmäßig wird nur das Dateiverzeichnis `/usr/lib` nach den Bibliotheken durchsucht. Ein mit `-L` angegebenes Dateiverzeichnis wird vor dem Standard-Dateiverzeichnis `/usr/lib` bzw. vor den mit der Option `-Y P` angegebenen Verzeichnissen durchsucht. Die Reihenfolge, in der die `-L`-Optionen in der Kommandozeile angegeben werden, bestimmt die Suchreihenfolge des Binders. Diese Option zählt nur bei den Kommandos `cc`, `c11` und `CC` zur Kategorie der Operanden und kann deshalb nur bei diesen Kommandos auch nach Beendigung der Optioneneingabe mit `--` angegeben werden (siehe auch „Operanden“ (Aufruf-Syntax und allgemeine Regeln)).

`-r`

Mit dieser Option können mehrere Objektdateien zu einer einzigen Objektdatei vorgebunden werden. Eine vorgebundene Objektdatei ist nicht ausführbar und enthält Relocation-Informationen, die für einen erneuten Bindelauf benötigt werden.
Beim Vorbinden mit -r sind implizit die folgenden Optionen gesetzt:
-K no_link_stdlibs und -B extended_external_names. Das heißt, die C/C++-Standardbibliotheken werden nicht eingebunden und im Falle von langen C- und C++-Namen (EENs) wird das LLM-Format 4 generiert. Die ggf. angegebenen Optionen `-K link_stdlibs` und -B short_external_names werden ignoriert. Beim Erzeugen einer vorgebundenen Objektdatei erfolgen keine Instanziierungen durch den Prälinker.
Unaufgelöste Referenzen führen zu keiner Fehlermeldung.
Die vorgebundene Objektdatei erhält den Namen a.out bzw. den mit der -o-Option angegebenen Namen. Die Objektdatei kann nur sinnvoll weiterverarbeitet (gebunden) werden, wenn der Name der vorgebundenen Objektdatei das Suffix .o oder ein mit der Option -Y F vereinbartes Suffix (siehe "Allgemeine Optionen") enthält.

`-s`

Aus der Ausgabedatei werden Symboltabellen-Informationen entfernt. Die Abschnitte mit den Zusatzinformationen zur Fehlersuche und mit den Zeilennummern sowie die dazugehörenden Relokations-Informationen werden entfernt.

Die Option wird ignoriert, wenn gleichzeitig Testhilfe-Informationen für AID angefordert werden (Optionen `-g`). Außerdem wird sie in allen C++-Modi ignoriert, da die Symboltabellen zur Ablaufzeit für globale Intitialisierungen benötigt werden. Die Option entspricht der BINDER-Anweisung SAVE-LLM SYMBOL-DICTIONARY=*NO.

`-Y P,` *dvz1*`[:` *dvz2*`...]`

Der Binder sucht zuletzt in den mit *dvz* angegebenen Verzeichnissen nach Bibliotheken. Ohne Angabe dieser Option wird das Standard-Dateiverzeichnisse `/usr/lib` zuletzt durchsucht.

`-z nodefs`

Diese Option wird nur beim Binden von C-Programmen (cc, c11, c89) unterstützt. Bei Angabe dieser Option kann ein C-Programm gebunden werden, in dem alle Externverweise auf die C-Standardbibliothek libc.a offen bleiben, d.h. es wird kein RESOLVE auf die Bibliotheken SYSLNK.CRTE.PARTIAL-BIND oder SYSLNK.CRTE abgesetzt. Die offenen Externverweise werden zum Ablaufzeitpunkt dynamisch aus dem in den Klasse-4-Speicher vorgeladenen C-Laufzeitsystem befriedigt.

Bei Verwendung dieser Option werden auch „unresolved externals“ auf Benutzermodule ignoriert und nicht gemeldet. Erst beim Laden des Programms erhält man Hinweise auf unbefriedigte Externverweise.

```
-z dup_ignore
-z dup_warning
-z dup_error
```

Diese Optionen steuern das Verhalten von doppelten Entry-Namen während des Bindevorgangs.

`-z dup_ignore`
Doppelte Entry-Namen werden während des Bindevorgangs ignoriert. Dies ist der Standardwert.

`-z dup_warning`
Doppelte Entry-Namen führen während des Bindevorgangs zu einer Warnung.

`-z dup_error`
Doppelte Entry-Namen führen während des Bindevorgangs zu einem Error.

Ein Programm, das doppelte Entry-Namen (duplicates) enthält, kann im POSIX nicht ausgeführt werden. Es beendet sich sofort mit dem Return-Code 127.
Der Compiler kann eventuell gefundene Duplikate nicht namentlich nennen. Wenn das Programm in BS2000-Umgebung (SDF) mit `/LOAD-EXECUTABLE-PROGRAM` geladen wird, werden Duplikate mit der Meldung BLS0339 angezeigt.
In welchen Modulen die doppelten Entrys enthalten sind, kann mit Hilfe der Binderliste (siehe -N binder, "Optionen zur Ausgabe von Listen und CIF-Informationen") und ggf. des Hilfsprogramms `nm` herausgefunden werden.

3.2.12 Optionen zur Steuerung der Meldungsausgabe

Detailliertere Informationen zur Meldungsaugabe des Compilers finden Sie im C/C++-Benutzerhandbuch [4] im Abschnitt „Aufbau der Compilermeldungen“.

`-R diagnose_to_listing`

Diese Option ermöglicht es, Diagnoseinformationen (normalerweise ausgegeben auf `stderr`) als spezielles „Ergebnislisting“ zu sortieren und an das Ende der Listingdatei zu kopieren. Die Meldungen werden nach ihrem Fehlergewicht sortiert!

`-R limit,`*n*

Diese Option legt die Anzahl von Errors fest, ab der der Compiler mit der Übersetzung nicht mehr fortfahren soll (Notes und Warnings werden eigens gezählt). Voreingestellt ist *n*=50. Bei *n*=0 versucht der Compiler unabhängig von der Anzahl auftretender Errors solange wie möglich mit der Übersetzung fortzufahren.

`-R min_weight,`*min_weight*

Diese Option legt fest, ab welchem Gewicht die Diagnosemeldungen des Compilers auf die Standard-Fehlerausgabe `stderr` ausgegeben werden sollen.
`-R min_weight,warnings` ist voreingestellt. Für *min_weight* sind folgende Angaben möglich:

`notes`	Alle Meldungen, d.h. auch die Notes werden ausgegeben.
`warnings`	Die Ausgabe von Notes wird unterdrückt (Voreinstellung).
`errors`	Die Ausgabe von Notes und Warnings wird unterdrückt.
`fatals`	Die Ausgabe von Notes, Warnings und Errors wird unterdrückt.

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

Mit diesen Optionen kann das voreingestellte Fehlergewicht von Diagnosemeldungen geändert werden. *msgid* ist die entsprechende Meldungsnummer. Das Fehlergewicht von Fatal Errors kann nicht verändert werden und das von Errors nur dann, wenn sie in der Originalmeldung mit einem Stern gekennzeichnet sind: `[*ERROR]`. Abhängig vom Sprachmodus oder von der Situation im Code kann die gleiche Meldungsnummer *msgid* ein unterschiedliches Fehlergewicht haben (Warning oder Error).

`-R show_column`
`-R no_show_column`

Diese Option legt fest, ob die Diagnosemeldungen des Compilers in kurzer oder in ausführlicher Form generiert werden.
`-R show_column` ist voreingestellt. Zusätzlich zur Diagnosemeldung wird die Original-Quellprogrammzeile ausgegeben, in der die Fehlerstelle markiert ist (mit ^).
Bei Angabe von `-R no_show_column` unterbleibt die Ausgabe der markierten Quellprogrammzeile.

`-R strict_errors`
`-R strict_warnings`

Diese Option wird in den nicht-strikten Sprachmodi (`-X nostrict`) ignoriert. `-R strict_warnings` ist voreingestellt und bewirkt die Ausgabe von Warnings, wenn Sprachkonstrukte benutzt werden, die zwar eine Abweichung vom ANSI-/ISO-Standard, jedoch keine schwere Verletzung der dort festgelegten Sprachregeln darstellen (z.B. implementierungsspezifische Spracherweiterungen, siehe C/C++-Benutzerhandbuch [4]).
Bei Angabe von `-R strict_errors` werden in solchen Fällen Errors ausgegeben. Schwere Verletzungen führen automatisch zu Errors.

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

Die Ausgabe der Meldung mit der Meldungsnummer *msgid* wird unterdrückt. Es gibt Meldungen, deren Ausgabe nicht unterdrückt werden kann (z.B. Fatal Errors). Die Meldung wird trotzdem für die Summary-Meldung gezählt. Auch führt ein Fehler immer dazu, dass kein Modul generiert wird. Dies gilt auch wenn die Ausgabe des Fehlers unterdrückt wird.

`-R use_before_set`
`-R no_use_before_set`

`-R use_before_set` ist voreingestellt und bewirkt die Ausgabe von Warnings, wenn im Programm lokale `auto`-Variablen benutzt werden, bevor ihnen ein Wert zugewiesen wurde.
Bei Angabe der Option `-R no_use_before_set` wird die Ausgabe solcher Warnings unterdrückt.

`-v`

Die Meldungsausgabe erfolgt wie bei der Optionenkombination
`-R min_weight,notes` und `-K verbose`.

`-w`

Diese Option ist ein Synonym für `-R min_weight,errors`.

3.2.13 Optionen zur Ausgabe von Listen und CIF-Informationen

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

Mit dieser Option können analog zum MAP-Operanden der BINDER-Anweisung SAVE-LLM Standardlisten des BINDER angefordert werden. Binderlisten werden nur erzeugt, wenn eine ausführbare Datei oder eine vorgebundene Objektdatei (`-r`) erzeugt wird. Ohne Angabe von *file* werden die Binderlisten in eine Ausgabedatei *datei*`.lst` geschrieben, wobei *datei* der Name der ausführbaren Datei oder der vorgebundenen Objektdatei ist (`a.out` bzw. der mit der Option `-o` vereinbarte Name). Mit *file* kann ein anderer Ausgabedateiname vereinbart werden. Die Option `-N binder` wird ignoriert, wenn gleichzeitig eine der Optionen `-c`, `-E`, `-M`, `-P` oder `-y` angegeben wird.

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

(*output-spec* ist eine Datei oder ein Verzeichnis).
Der Compiler generiert ein CIF (Compilation Information File), das Informationen für die angegebenen *consumers* enthält. Ohne Angabe von *output-spec* wird das CIF pro übersetzte Quelldatei in eine Datei namens *quelldatei*`.cif` geschrieben. Mit *output-spec* kann ein anderer Ausgabedateiname vereinbart werden; in diesem Fall kann nur eine Quelldatei übersetzt werden. Zur Weiterverarbeitung der erzeugten CIF-Informationen steht der globale Listengenerator `cclistgen` zur Verfügung (siehe "Globaler Listengenerator (cclistgen)").
Für *consumer* sind folgende Angaben möglich:

`option` oder `lo` (Optionen)

`prepro` oder `lp` (Ergebnis des Präprozessors)

`source_error` oder `ls` (Fehler im Quellprogramm)

`data_allocation_map` oder `lm` (Adressen)

`cross_reference` oder `lx` oder `xref` (Querverweise)

`object` oder `la` (Objektcode)

`project` oder `lP` (Projektinformation, nur beim CC-Kommando)

`summary` oder `lS` (Statistik)

`ALL`

Bei Angabe von `ALL` werden alle CIF-Informationen generiert, die möglich sind, z.B. bei Beendigung des Compilerlaufs nach der Präprozessorphase (Optionen `-E`, `-P`) CIF-Informationen zu einer Optionen-, Präprozessor- und Statistikliste. Das CIF kann bei Angabe von ALL u.U. sehr groß werden!

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

Die mit dieser Option angeforderten Listen schreibt der Compiler entweder pro übersetzte Quelldatei in eine Listendatei *quelldatei*`.lst` oder für alle übersetzten Quelldateien in die mit der Option `-N output` angegebene Listendatei *file*.
Nach Erreichen der maximalen Angaben von Errors (steuerbar durch `-R limit`) wird keine Quellprogramm-Information in der Quellprogramm-/Fehlerliste ausgegeben. In einem solchen Fall kann über dieses Listing kein Bezug zu Fehlerstellen mehr festgestellt werden.
Für *listing* sind folgende Angaben möglich:

`option` oder `lo` (Optionenliste)

`prepro` oder `lp` (Präprozessorliste)

`source_error` oder `ls` (Quellprogramm-/Fehlerliste)

`data_allocation_map` oder `lm` (Adressliste)

`cross_reference` oder `lx` (Querverweisliste, siehe auch Option `-N xref`)

`object` oder `la` (Objektcodeliste)

`project` oder `lP` (Projektliste, nur beim `CC`-Kommando)

`summary` oder `lS` (Statistikliste)

`ALL`

Bei Angabe von `ALL` werden alle Listen generiert, die möglich sind, z.B. bei Beendigung des Compilerlaufs nach der Präprozessorphase (Optionen `-E`, `-P`) eine Optionen-, Präprozessor- und Statistikliste.

`-N map_structlevel,`*n*

Mit dieser Option lässt sich steuern, bis zu welcher Strukturtiefe Elemente einer Struktur in der mit der Option `-N data_allocation_map` angeforderten Liste enthalten sind. Für *n* können Werte von 0 bis 256 einschließlich angegeben werden.
Es werden Strukturelemente bis zu der mit *n* angegebenen Schachtelungstiefe in der Adressliste abgebildet. Bei Angabe der Schachtelungstiefe 0 werden keine Strukturelemente ausgegeben.

Strukturelemente werden durch Einrückung und Klammerung {} dargestellt. Elemente der Schachtelungstiefe 16 oder höher werden nicht mehr eingerückt.

Beispiele für den Aufbau der Übersetzungslisten finden Sie im C/C++-Benutzerhandbuch [4], Abschnitt „Beschreibung der Listenbilder“.

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

Mit dieser Option kann der Name (*output-spec*) einer Ausgabedatei oder eines Ausgabeverzeichnisses angegeben werden, in die die Compilerlisten für alle Quelldateien geschrieben werden sollen.
Ohne Angabe von *output-spec* wird pro übersetzte Quelldatei eine Listendatei *quelldatei*`.lst` erzeugt.
Bezeichnet *output-spec* ein bereits existierendes Ausgabeverzeichnis, so wird dafür standardmäßig der Name *output-spec/quelldatei.lst* vergeben. Andernfalls wird *output-spec* als Dateiname interpretiert.

Für *layout* sind folgende Angaben möglich:

`normal` oder `for_normal_print` (Voreinstellung)
Standardmäßig beträgt die Seitenhöhe 64 Zeilen und die Zeilenbreite 132 Zeichen.

`rotation` oder `for_rotation_print`
Die Seitenhöhe für die Compilerliste wird auf 84 Zeilen, die Zeilenbreite auf 120 Zeichen festgelegt.

Mit *lpp* lässt sich eine Seitenhöhe von 11 bis 255 Zeilen pro Seite vereinbaren.

Mit *cpl* lässt sich eine Zeilenbreite von 120 bis 255 Zeichen pro Zeile vereinbaren.

Hinweis
Da die Ausgabedatei für den Druck im POSIX aufbereitet ist, sind in der Datei am Anfang einiger Zeilen bis zu 3 Drucksteuerzeichen enthalten. Ferner endet jede Zeile mit dem Druckersteuerzeichen für “carriage return”. Wird die Ausgabedatei gedruckt, ist die Zeilenlänge `cpl-1`.

`-N title,`*text*

Mit dieser Option kann angegeben werden, ob im Listenkopf eine zusätzliche Zeile erscheinen soll und welcher Text dort stehen soll. Die Option `-N title` bezieht sich, im Unterschied zu den Pragmas (nur Quellprogramm- und Präprozessorliste), auf alle Compilerlisten. Es empfiehlt sich, den gewünschten Text in Anführungszeichen "*text*" einzuschließen, da so in jedem Fall eine 1:1 Übergabe gewährleistet ist.
Bei der Quellprogramm- und Präprozessorliste haben ggf. vorhandene TITLE- und PAGE-Pragmas Vorrang vor der Angabe `-N title`. Siehe auch Abschnitt „Pragmas zum Steuern des Listenbildes“ im C/C++-Benutzerhandbuch [4].

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

Mit dieser Option lässt sich steuern, welche Teile die mit der Option `-N cross_reference` angeforderte Querverweisliste enthält.
Ohne Angabe der Option `-N xref` enthält die Querverweisliste eine Liste der Variablen, Funktionen und Labels (entspricht `-N xref,v,f,l`).
In jedem Fall enthält die Querverweisliste einen FILETABLE-Teil mit den Namen aller Dateien, Bibliotheken und Elementen, die der Compiler als Quellen verwendet.

Bei Angabe der Option `-N xref` enthält die Querverweisliste neben dem FILETABLE-Teil nur die mit den Argumenten *xrefopt* angeforderten Teile. Für *xrefopt* sind folgende Angaben möglich:

`p`	Liste der vom Präprozessor bearbeiteten Namen in `#include`- und `#define`-Anweisungen
`y`	Liste der benutzerdefinierten Typen (typedefs, Struktur-, Union-, Klassen-und Aufzählungstypen)
`v`	Liste der Variablen
`f`	Liste der Funktionen
`l`	Liste der Labels
`t`	Liste der Templates (nur bei C++-Übersetzungen)
`o=`*str*	Reihenfolge, in der die einzelnen Teile in der Querverweisliste aufgeführt werden. *str* ist eine Folge von maximal 6 Zeichen (Buchstaben für die entsprechenden Listen s.o.). Voreingestellt ist die oben angeführte Reihenfolge (also `o=pyvflt`). Wenn mit `o=`*str* nicht alle Buchstaben für die mit `-N xref` angeforderten Listen angegeben werden, so werden die fehlenden Buchstaben implizit an das Ende von *str* hinzugefügt, jeweils in der o.g. Standardreihenfolge.

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

Allgemeine Eingaberegeln zur `-K`-Option finden Sie auf "Aufruf-Syntax und allgemeine Regeln".
Als Argumente *arg* zur Steuerung der Listenausgabe sind folgende Angaben möglich:

`include_user`
`include_all`
`include_none`

Diese Argumente steuern, ob und welche Include-Dateien in der Quellprogramm-, Präprozessor- und Querverweisliste abgebildet werden.

`-K include_user` ist voreingestellt und bewirkt, dass nur die benutzereigenen Include-Dateien abgebildet werden.

Bei Angabe von `-K include_all` werden alle Include-Dateien, d.h. die Standard-Include-Dateien und die benutzereigenen Include-Dateien abgebildet.

Bei Angabe von `-K include_none` werden keine Include-Dateien abgebildet.

`cif_include_user`
`cif_include_all`
`cif_include_none`

Diese Argumente steuern, ob und aus welchen Include-Dateien CIF-Informationen für die Quellprogramm-, Präprozessor- und Querverweisliste generiert werden.

`-K cif_include_user` ist voreingestellt und bewirkt, dass nur die benutzereigenen Include-Dateien im CIF berücksichtigt werden.

Bei Angabe von `-K cif_include_all` werden alle Include-Dateien, d.h. die Standard-Include-Dateien und die benutzereigenen Include-Dateien im CIF berücksichtigt

Bei Angabe von `-K cif_include_none` werden keine Include-Dateien im CIF berücksichtigt.

`pragmas_interpreted`
`pragmas_ignored`

Diese Argumente steuern, ob `#pragma`-Anweisungen zur Steuerung des Listenbildes ausgewertet werden (siehe auch Abschnitt „Pragmas zum Steuern des Listenbildes“ im C/C++-Benutzerhandbuch [4]). `-K pragmas_interpreted` ist voreingestellt.

3.3 Dateien

datei `.c` | `.C`	C-Quelldatei (`cc`, `c11`, `c89`) oder C++-Quelldatei (`CC`) vor dem Präprozessorlauf
datei `.cpp` | `.CPP` | `.cxx` | `.CXX` | `.cc` | `.CC` | `.c++` | `.C++`	C++-Quelldatei vor dem Präprozessorlauf
datei `.i`	C-Quelldatei (`cc`, `c11`, `c89`) nach dem Präprozessorlauf
datei `.I`	C++-Quelldatei nach dem Präprozessorlauf
datei `.o`	LLM-Objektdatei
datei `.a`	statische Bibliothek mit Objektdateien, erzeugt mit dem Dienstprogramm `ar`
datei `.lst`	Datei mit Übersetzungslisten
datei `.cif`	Datei mit CIF-Informationen zur Weiterverarbeitung mit dem globalen Listengenerator `cclistgen`
datei `.etr`	Datei mit expliziten Instanziierungsanweisungen
datei `.o.ii`	Informationsdatei für die automatische Template-Instanziierung (intern verwendet)
`a.out`	ausführbare Datei
datei `.mk`	Präprozessor-Ausgabedatei zur Weiterverarbeitung mit `make`
`/var/tmp/` ...	Zwischendateien des Übersetzungslaufs

3.4 Umgebungsvariablen

Mit folgenden Umgebungsvariablen lassen sich die cc/c11/c89/CC-Kommandos beeinflussen:

LANG, LC_MESSAGES	Sprache der Meldungsausgaben
TMPDIR	Name des Verzeichnisses, in dem Zwischendateien temporär abgelegt werden
BLSLIB*nn*	Zuweisen von PLAM-Bibliotheken, die der Binder per Autolink durchsuchen soll
IO_CONVERSION	Automatisches Konvertieren (IO_CONVERSION=YES) von ASCII nach EBCDIC

3.5 Vordefinierte Präprozessornamen

Beim Aufruf des Compilers mit `cc`, `c11`, `c89` oder `CC` sind abhängig vom gewählten Kommando und von der Eingabe bestimmter Optionen Präprozessor-Makros und -Prädikate vordefiniert.

Bei ein paar Makros können die Werte nicht verändert werden. Es geht weder auf der Kommandozeile noch per `#define` oder `#undef` in der Source. Die betroffenen Makros sind: `__cplusplus`, `__STDC__`, `__STDC_VERSION__` und `__SNI__STDCplusplus`.

Vordefinierte Präprozessor-Makros (Defines)

`_BOOL`	in den Sprachmodi C++ V3, C++ 2017 und C++ 2020 bei Option `-K bool` (Voreinstellung)
`__CGLOBALS_PRAGMA`	immer gesetzt
`__cplusplus`	in allen C++-Sprachmodi: == 1 im Cfront-C++-Modus == 2 im erweiterten C++ V3-Modus == 199612L im strikten C++ V3-Modus == 201703L im C++ 2017-Modus == 202002L im C++ 2020-Modus
`c_plusplus`	in allen C++-Sprachmodi
`__CFRONT_V3`	im Cfront-C++-Modus
`__EDG_NO_IMPLICIT_INCLUSION`	in den Sprachmodi C++ V3, C++ 2017 und C++ 2020 , wenn im Rahmen der Template-Instanziierung implizites Inkludieren ausgeschaltet wurde (`-K no_implicit_include`)
`__EXISTCGLOB`	immer gesetzt
`__HALF_TAG_LOOKUP`	immer gesetzt
`_IEEE`	Option `-K ieee_floats`
`LANGUAGE_C`	immer definiert
`_LANGUAGE_C`	immer definiert
`_LIBCPP_STD_VER`	wird im Sprachmodus C++ 2020 vom Compiler auf den Wert 17 gesetzt wird im Sprachmodus C++ 2017 von Bibliotheks-Headern auf den Wert 17 gesetzt
`__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 bei der Option `-K old_specialization`
`_OSD_POSIX`	immer gesetzt
`__OSD_POSIX`	immer gesetzt
`__SHORT_NAMES`	Ist definiert, wenn `–K c_names_short` angegeben wurde
`__SIGNED_CHARS__`	Option `-K schar`
`__SMALL_VA_DCL`	immer gesetzt
`__SNI`	in allen C-Modi und im Cfront-C++-Modus
`__SNI_HOST_BS2000`	nie gesetzt (reserviert für Übersetzung in BS2000-Umgebung (SDF))
`__SNI_HOST_BS2000_POSIX`	immer gesetzt
`__SNI_PRINTF_CHECK`	immer gesetzt
`__SNI__STDCplusplus`	in allen C++-Sprachmodi:
	== 0 in den erweiterten Sprachmodi (`-X nostrict`) == 1 in den strikten Sprachmodi (`-X strict`)
`__SNI_TARG_BS2000`	nie gesetzt (reserviert für Übersetzung in BS2000-Umgebung (SDF))
`__SNI_TARG_BS2000_POSIX`	immer gesetzt
`__STDC__`	immer gesetzt:
	== 0 in den erweiterten Sprachmodi (`-X strict`) == 1 in den strikten Sprachmodi (`-X strict`)
`__STDC_HOSTED__`	immer gesetzt
`__STDC_NO_ATOMICS__`	immer gesetzt
`__STDC_NO_COMPLEX__`	immer gesetzt
`__STDC_NO_THREADS__`	immer gesetzt
`__STDCPP_DEFAULT_NEW_ALIGNMENT__`	
	immer gesetzt
	== 8U
`__STDC_UTF_16__`	immer gesetzt

`__STDC_UTF_32__`	immer gesetzt
`__STDC_VERS_CRTE__`	im K&R-C-Modus undefiniert == 199409L in den Sprachmodi C89, Cfront-C++ und C++ V3 == 201112L in den Sprachmodi C11, C++ 2017 und C++ 2020
`__STDC_VERSION__`	im K&R-C-Modus undefiniert == 199409L in dem Sprachmodus C89 und in allen C++-Sprachmodi == 201112L in dem Sprachmodus C11
`_STRICT_STDC`	in den strikten Sprachmodi (`-X strict`)
`_WCHAR_T`	Option `-K wchar_t_keyword` (Voreinstellung in den Modi C++ V3, C++ 2017 und C++ 2020) Wenn diese Option nicht gesetzt ist (z.B. in den C-Modi oder im Cfront-C++-Modus), wird `_WCHAR_T` in diversen Standard-Includes definiert, um ein `typedef` für `wchar_t` abzusetzen.
`_WCHAR_T_KEYWORD`	Option `-K wchar_t_keyword` (Voreinstellung in den Modi C++ V3, C++ 2017 und C++ 2020)
`_XPG_IV`	beim Aufrufkommando `c11` oder `c89`

Vordefinierte Präprozessor-Prädikate (#assert)

`data_model(bit32)`	immer gesetzt
`cpu(7500)`	bei Generierung von /390-Code
`machine(7500)`	bei Generierung von /390-Code
`system(bs2000)`	immer gesetzt

4 Globaler Listengenerator (cclistgen)

Der globale Listengenerator wird mit dem Kommando `cclistgen` aufgerufen. Eingabequellen für den Listengenerator sind vom Compiler generierte CIFs (Compilation Information Files), die er pro Übersetzungseinheit in eine Datei *quelldatei*`.cif` bzw. in eine explizit angegeben Datei *file* geschrieben hat (siehe Option `-N cif`, "Optionen zur Ausgabe von Listen und CIF-Informationen"). Die generierten Listen werden standardmäßig auf `stdout` geschrieben, bei Angabe der Option `-o` in die dort angegebene Ausgabedatei. Aus den modullokalen CIF-Informationen für Querverweis- und Projektlisten erzeugt der Listengenerator globale, modulübergreifende Querverweis- und Projektlisten. Die übrigen Listen werden pro Quelldatei generiert.

4.1 Aufruf-Syntax

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

Eine Mischung von Optionen und Operanden ist nicht erlaubt. Die Reihenfolge „erst Optionen, dann Operanden“ muss eingehalten werden.

Optionen

Keine *option* angegeben

> Es wird eine Quellprogramm-/Fehlerliste erzeugt und auf `stdout` ausgegeben.

option

> Mit Optionen können Art und Umfang der zu generierenden Listen gesteuert werden. Die Optionen sind im Abschnitt "Optionen" beschrieben.
>
> Wenn `cclistgen` mit unzulässigen Optionen aufgerufen wird, gibt das Programm eine Fehlermeldung aus und beendet sich mit dem exit-Status ungleich `0`.

Operanden

cif-datei

> Name der CIF-Datei, aus der eine Liste erstellt werden soll. Es können beliebig viele CIF-Dateien angegeben werden. Die Angabe mindestens einer CIF-Datei ist erforderlich. Es findet keine syntaktische Überprüfung auf das Suffix `.cif` statt, d.h. es werden auch andere Dateinamen akzeptiert (siehe auch Compileroption `-N cif` , "Optionen zur Ausgabe von Listen und CIF-Informationen").

Exit-Status

Nach einer erfolgreichen Listengenerierung wird der Exit-Wert 0 zurückgeliefert, im Fehlerfall ein Exit-Wert ungleich 0.

4.2 Optionen

`-o` *ausgabedatei*

Die globale Liste wird in die Datei *ausgabedatei* geschrieben. Enthält *ausgabedatei* keine Dateiverzeichnisbestandteile, wird die Datei in das aktuelle Dateiverzeichnis geschrieben, sonst in das mit *ausgabedatei* angegebene Dateiverzeichnis. Standardmäßig wird die Liste nach `stdout` ausgegeben. Bei Verwendung der Option `-o` richtet sich das Ausgabe-Codeset (ASCII oder EBCDIC) nach dem Codeset des Zielsystems. Es werden jedoch immer BS2000-Drucksteuerzeichen generiert.

`-V`

Es wird eine Versionsangabe und eine Copyright-Meldung auf `stderr` ausgegeben.

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

Die mit dieser Option angeforderten Listen schreibt der Listengenerator entweder nach `stdout` oder in die mit der Option `-o` *ausgabedatei* angegebene Datei *ausgabedatei*. Für *listing* sind folgende Angaben möglich:

`option` oder `lo` (Optionenliste)

`prepro` oder `lp` (Präprozessorliste)

`source_error` oder `ls` (Quellprogramm-/Fehlerliste)

`data_allocation_map` oder `lm` (Adressliste)

`cross_reference` oder `lx` (Querverweisliste)

`object` oder `la` (Objektcodeliste)

`project` oder `lP` (Projektliste, nur bei C++-Programmen sinnvoll)

`summary` oder `lS` (Statistikliste)

`ALL`

Bei Angabe von `ALL` werden alle Listen generiert.

Es können nur die Listings generiert werden, für die Information bei der Übersetzung angefordert wurde (siehe `-Ncif` im Abschnitt "Optionen zur Ausgabe von Listen und CIF-Informationen"). Wird ein Listing angefordert, aber die notwendige Information steht nicht in der CIF-Datei, so wird eine Fehlermeldung ausgegeben.

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

Mit dieser Option kann das Layout der globalen Liste beeinflusst werden.

Für *layout* sind folgende Angaben möglich:

`normal` oder `for_normal_print` (Voreinstellung)
Standardmäßig beträgt die Seitenhöhe 64 Zeilen und die Zeilenbreite 132 Zeichen.

`rotation` oder `for_rotation_print`
Die Seitenhöhe für die Liste wird auf 84 Zeilen, die Zeilenbreite auf 120 Zeichen festgelegt.

Mit *lpp* lässt sich eine Seitenhöhe von 11 bis 255 Zeilen pro Seite vereinbaren.

Mit *cpl* lässt sich eine Zeilenbreite von 120 bis 255 Zeichen pro Zeile vereinbaren.

Hinweis
Da die Ausgabedatei für den Druck im POSIX aufbereitet ist, sind in der Datei am Anfang einiger Zeilen bis zu 3 Drucksteuerzeichen enthalten. Ferner endet jede Zeile mit dem Druckersteuerzeichen für „Carriage Return". Wird die Ausgabedatei gedruckt, ist die Zeilenlänge `cpl-1`.

`-N title,` *text*

Mit dieser Option kann angegeben werden, ob im Listenkopf eine zusätzliche Zeile erscheinen soll und welcher Text dort stehen soll. Die Option `-N title` bezieht sich, im Unterschied zu den Pragmas (nur Quellprogramm- und Präprozessorliste), auf alle Compilerlisten. Es empfiehlt sich, den gewünschten Text in Anführungszeichen "*text*" einzuschließen, da so in jedem Fall eine 1:1 Übergabe gewährleistet ist.
Bei der Quellprogramm- und Präprozessorliste haben ggf. vorhandene TITLE- und PAGE-Pragmas Vorrang vor der Angabe `-N title`. Siehe auch Abschnitt „Pragmas zum Steuern des Listenbildes" im C/C++-Benutzerhandbuch [4].

`-N map_structlevel,`*n*

Siehe Compileroption `-N map_structlevel,`*n* im Abschnitt "Optionen zur Ausgabe von Listen und CIF-Informationen".

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

Mit dieser Option lässt sich steuern, welche Teile die mit der Option `-N cross_reference` angeforderte Querverweisliste enthält.
Ohne Angabe der Option `-N xref` enthält die Querverweisliste eine Liste der Variablen, Funktionen und Labels (entspricht `-N xref,v,f,l`).
In jedem Fall enthält die Querverweisliste einen FILETABLE-Teil mit den Namen aller Dateien, Bibliotheken und Elementen, die der Compiler als Quellen verwendet.

Bei Angabe der Option `-N xref` enthält die Querverweisliste neben dem FILETABLE-Teil nur die mit den Argumenten *xrefopt* angeforderten Teile. Für *xrefopt* sind folgende Angaben möglich:

`p`	Liste der vom Präprozessor bearbeiteten Namen in `#include`- und `#define`-Anweisungen
`y`	Liste der benutzerdefinierten Typen (typedefs, Struktur-, Union-, Klassen- und Aufzählungstypen)
`v`	Liste der Variablen
`f`	Liste der Funktionen
`l`	Liste der Labels
`t`	Liste der Templates (nur bei C++-Übersetzungen)
`o=`*str*	Reihenfolge, in der die einzelnen Teile in der Querverweisliste aufgeführt werden. *str* ist eine Folge von maximal 6 Zeichen (Buchstaben für die entsprechenden Listen s.o.). Voreingestellt ist die oben angeführte Reihenfolge (also `o=pyvflt`). Wenn mit `o=` *str* nicht alle Buchstaben für die mit `-N xref` angeforderten Listen angegeben werden, so werden die fehlenden Buchstaben implizit an das Ende von *str* hinzugefügt, jeweils in der o.g. Standardreihenfolge.

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

Allgemeine Eingaberegeln zur `-K`-Option finden Sie auf "Aufruf-Syntax und allgemeine Regeln".
Als Argumente *arg* zur Steuerung der Listenausgabe sind folgende Angaben möglich:

`include_user`
`include_all`
`include_none`

Diese Argumente steuern, ob und welche Include-Dateien in der Quellprogramm-, Präprozessor- und Querverweisliste abgebildet werden.

`-K include_user` ist voreingestellt und bewirkt, dass nur die benutzereigenen Include-Dateien abgebildet werden.

Bei Angabe von `-K include_all` werden alle Include-Dateien, d.h. die Standard-Include-Dateien und die benutzereigenen Include-Dateien abgebildet.

Bei Angabe von `-K include_none` werden keine Include-Dateien abgebildet.

Eine Abbildung von Include-Dateien ist nur möglich, wenn die Information auch in der CIF-Datei steht (siehe die Compiler-Option `-K cif_include_user`).

`pragmas_interpreted`
`pragmas_ignored`

Diese Argumente steuern, ob `#pragma`-Anweisungen zur Steuerung des Listenbildes ausgewertet werden (siehe auch Abschnitt „Pragmas zum Steuern des Listenbildes“ im C/C++-Benutzerhandbuch [4]).
`-K pragmas_interpreted` ist voreingestellt.

5 Anhang: Optionenübersicht (alphabetisch)

Option	Kategorie	Abschnitt
—	allgemein	Allgemeine Optionen
–A	Präprozessor	Präprozessor-Optionen
–B extended_external_names	Binden	Binder-Optionen
–B short_external_names	Binden	Binder-Optionen
–C	Präprozessor	Präprozessor-Optionen
–c	Übersetzungsphasen (Objektcode)	Optionen zur Auswahl von Übersetzungsphasen
–D *name*[=*wert*]	Präprozessor	Präprozessor-Optionen
–d compl	Binden	Binder-Optionen
–d n	Binden	Binder-Optionen
–d y	Binden	Binder-Optionen
–E *name*	Übersetzungsphasen (Präprozessor)	Optionen zur Auswahl von Übersetzungsphasen
–F I	Optimierung	Optimierungsoptionen
–F i[*name*]	Optimierung	Optimierungsoptionen
–F inline_by_source	Optimierung	Optimierungsoptionen
–F loopunroll	Optimierung	Optimierungsoptionen
–F no_inlining	Optimierung	Optimierungsoptionen
–F O2	Optimierung	Optimierungsoptionen
–g	Testhilfe	Testhilfe-Option
–H	Präprozessor	Präprozessor-Optionen
–i *header*	Präprozessor	Präprozessor-Optionen
–I *dvz*	Präprozessor	Präprozessor-Optionen
–K [no_]alternative_tokens	C- und C++-Frontend	Gemeinsame Frontend-Optionen in C und C++
–K ansi_cpp	Präprozessor	Präprozessor-Optionen
–K [no_]assign_local_only	C++-Frontend (Templates)	Template-Optionen

–K [no_]at	C- und C++-Frontend	Gemeinsame Frontend-Optionen in C und C++
–K [no_]bool	C++-Frontend (allgemein)	Allgemeine C++-Optionen
–K c_names_short	Objektgenerierung	Optionen zur Objektgenerierung
–K c_names_std	Objektgenerierung	Optionen zur Objektgenerierung
–K c_names_unlimited	Objektgenerierung	Optionen zur Objektgenerierung
–K calendar_etpnd	Objektgenerierung	Optionen zur Objektgenerierung
–K cif_include_all	CIF	Optionen zur Ausgabe von Listen und CIF-Informationen
–K cif_include_none	CIF	Optionen zur Ausgabe von Listen und CIF-Informationen
–K cif_include_user	CIF	Optionen zur Ausgabe von Listen und CIF-Informationen
–K csect_hashpath	Objektgenerierung	Optionen zur Objektgenerierung
–K csect_suffix=	Objektgenerierung	Optionen zur Objektgenerierung
–K [no_]dollar	C- und C++-Frontend	Gemeinsame Frontend-Optionen in C und C++
–K [no_] end_of_line_comments	C-Frontend	Gemeinsame Frontend-Optionen in C und C++
–K enum_long	Objektgenerierung	Optionen zur Objektgenerierung
–K enum_value	Objektgenerierung	Optionen zur Objektgenerierung
–K environment_encoding_ebcdic	Laufzeit	Laufzeit-Optionen
–K environment_encoding_std	Laufzeit	Laufzeit-Optionen
–K external_multiple	Objektgenerierung	Optionen zur Objektgenerierung
–K external_unique	Objektgenerierung	Optionen zur Objektgenerierung
–K [no_]ieee_floats	Objektgenerierung	Optionen zur Objektgenerierung
–K ilcs_opt	Objektgenerierung	Optionen zur Objektgenerierung
–K ilcs_out	Objektgenerierung	Optionen zur Objektgenerierung
–K [no_]implicit_include	C++-Frontend (Templates)	Template-Optionen
–K include_all	Listen	Optionen zur Ausgabe von Listen und CIF-Informationen

–K include_none	Listen	Optionen zur Ausgabe von Listen und CIF-Informationen
–K include_user	Listen	Optionen zur Ausgabe von Listen und CIF-Informationen
–K [no_]instantiation_flags	C++-Frontend (Templates)	Template-Optionen
–K [no_]integer_overflow	Laufzeit	Laufzeit-Optionen
–K julian_etpnd	Objektgenerierung	Optionen zur Objektgenerierung
–K kr_cpp	Präprozessor	Präprozessor-Optionen
–K library_version=	Sprachmodus	Optionen zur Auswahl des Sprachmodus
–K [no_]link_stdlibs	Binden	Binder-Optionen
–K literal_encoding_ascii	C- und C++-Frontend	Gemeinsame Frontend-Optionen in C und C++
–K literal_encoding_ascii_full	C- und C++-Frontend	Gemeinsame Frontend-Optionen in C und C++
–K literal_encoding_ebcdic	C- und C++-Frontend	Gemeinsame Frontend-Optionen in C und C++
–K literal_encoding_ebcdic_full	C- und C++-Frontend	Gemeinsame Frontend-Optionen in C und C++
–K literal_encoding_native	C- und C++-Frontend	Gemeinsame Frontend-Optionen in C und C++
–K [no_]llm_case_lower	Objektgenerierung	Optionen zur Objektgenerierung
–K llm_convert	Objektgenerierung	Optionen zur Objektgenerierung
–K llm_keep	Objektgenerierung	Optionen zur Objektgenerierung
–K [no_]longlong	C- und C++-Frontend	Gemeinsame Frontend-Optionen in C und C++
–K long_preserving	C- und C++-Frontend	Gemeinsame Frontend-Optionen in C und C++
–K new_for_init	C++-Frontend (allgemein)	Allgemeine C++-Optionen
–K no_etpnd	Objektgenerierung	Optionen zur Objektgenerierung
–K old_for_init	C++-Frontend (allgemein)	Allgemeine C++-Optionen
–K [no_]old_specialization	C++-Frontend (allgemein)	Allgemeine C++-Optionen
–K plain_fields_signed	C- und C++-Frontend	Gemeinsame Frontend-Optionen in C und C++
–K plain_fields_unsigned	C- und C++-Frontend	Gemeinsame Frontend-Optionen in C und C++
–K pragmas_ignored	Listen	Optionen zur Ausgabe von Listen und CIF-Informationen

–K pragmas_interpreted	Listen	Optionen zur Ausgabe von Listen und CIF-Informationen
–K [no_]prompting	Laufzeit	Laufzeit-Optionen
–K [no_]roconst	Objektgenerierung	Optionen zur Objektgenerierung
–K [no_]rostr	Objektgenerierung	Optionen zur Objektgenerierung
–K schar	C- und C++-Frontend	Gemeinsame Frontend-Optionen in C und C++
–K [no_]share	Objektgenerierung	Optionen zur Objektgenerierung
–K signed_fields_signed	C- und C++-Frontend	Gemeinsame Frontend-Optionen in C und C++
–K signed_fields_unsigned	C- und C++-Frontend	Gemeinsame Frontend-Optionen in C und C++
–K stacksize=*n*	Laufzeit	Laufzeit-Optionen
–K [no_]statistics	Laufzeit	Laufzeit-Optionen
–K subcall_basr	Objektgenerierung	Optionen zur Objektgenerierung
–K subcall_lab	Objektgenerierung	Optionen zur Objektgenerierung
–K uchar	C- und C++-Frontend	Gemeinsame Frontend-Optionen in C und C++
–K unsigned_preserving	C- und C++-Frontend	Gemeinsame Frontend-Optionen in C und C++
–K [no_]using_std	C++-Frontend (allgemein)	Allgemeine C++-Optionen
–K [no_]verbose	allgemein	Allgemeine Optionen
–K [no_]wchar_t_keyword	C++-Frontend (allgemein)	Allgemeine C++-Optionen
–K workspace_stack	Objektgenerierung	Optionen zur Objektgenerierung
–K workspace_static	Objektgenerierung	Optionen zur Objektgenerierung
–l BLSLIB	Binden	Binder-Optionen
–L *dvz*	Binden	Aufruf-Syntax und allgemeine Regeln, Binder-Optionen
–l *x*	Binden	Aufruf-Syntax und allgemeine Regeln, Binder-Optionen
–M	Übersetzungsphasen (Präprozessor)	Optionen zur Auswahl von Übersetzungsphasen
–N binder,...	Binden (Listen)	Optionen zur Ausgabe von Listen und CIF-Informationen

–N cif,...	CIF	Optionen zur Ausgabe von Listen und CIF-Informationen
–N *listing*,...	Listen	Optionen zur Ausgabe von Listen und CIF-Informationen
–N map_structlevel	Listen	Optionen zur Ausgabe von Listen und CIF-Informationen
–N output	Listen	Optionen zur Ausgabe von Listen und CIF-Informationen
–N title	Listen	Optionen zur Ausgabe von Listen und CIF-Informationen
–N xref	Listen	Optionen zur Ausgabe von Listen und CIF-Informationen
–O	Optimierung	Optimierungsoptionen
–o *ausgabeziel*	allgemein	Allgemeine Optionen
–P	Übersetzungsphasen (Präprozessor)	Optionen zur Auswahl von Übersetzungsphasen
–r	Binden	Binder-Optionen
–R diagnose_to_listing	Compilermeldungen	Optionen zur Steuerung der Meldungsausgabe
–R error	Compilermeldungen	Optionen zur Steuerung der Meldungsausgabe
–R limit	Compilermeldungen	Optionen zur Steuerung der Meldungsausgabe
–R min_weight,...	Compilermeldungen	Optionen zur Steuerung der Meldungsausgabe
–R note	Compilermeldungen	Optionen zur Steuerung der Meldungsausgabe
–R [no_]show_column	Compilermeldungen	Optionen zur Steuerung der Meldungsausgabe
–R strict_errors	Compilermeldungen	Optionen zur Steuerung der Meldungsausgabe
–R strict_warnings	Compilermeldungen	Optionen zur Steuerung der Meldungsausgabe
–R suppress	Compilermeldungen	Optionen zur Steuerung der Meldungsausgabe
–R [no_]use_before_set	Compilermeldungen	Optionen zur Steuerung der Meldungsausgabe
–R warning	Compilermeldungen	Optionen zur Steuerung der Meldungsausgabe
–s	Binden	Binder-Optionen
–T add_prelink_files	C++-Frontend (Templates)	Template-Optionen
–T all	C++-Frontend (Templates)	Template-Optionen

–T auto	C++-Frontend (Templates)	Template-Optionen
–T [no_]definition_list	C++-Frontend (Templates)	Template-Optionen
–T [no_]dl	C++-Frontend (Templates)	Template-Optionen
–T etr_file_all	C++-Frontend (Templates)	Template-Optionen
–T etr_file_assigned	C++-Frontend (Templates)	Template-Optionen
–T etr_file_none	C++-Frontend (Templates)	Template-Optionen
–T local	C++-Frontend (Templates)	Template-Optionen
–T max_iterations	C++-Frontend (Templates)	Template-Optionen
–T none	C++-Frontend (Templates)	Template-Optionen
–U *name*	Präprozessor	Präprozessor-Optionen
–V	allgemein	Allgemeine Optionen
–v	Compilermeldungen	Optionen zur Steuerung der Meldungsausgabe
–w	Compilermeldungen	Optionen zur Steuerung der Meldungsausgabe
–X cc	Sprachmodus (C)	Optionen zur Auswahl des Sprachmodus
–X CC	Sprachmodus (C++)	Optionen zur Auswahl des Sprachmodus
–X kr	Sprachmodus (C)	Optionen zur Auswahl des Sprachmodus
–X KR	Sprachmodus (C)	Optionen zur Auswahl des Sprachmodus
–X [no]strict	Sprachmodus	Optionen zur Auswahl des Sprachmodus
–X v2-compatible	Sprachmodus (C++)	Optionen zur Auswahl des Sprachmodus
–X V2-COMPATIBLE	Sprachmodus (C++)	Optionen zur Auswahl des Sprachmodus
–X v3-compatible	Sprachmodus (C++)	Optionen zur Auswahl des Sprachmodus
–X V3-COMPATIBLE	Sprachmodus (C++)	Optionen zur Auswahl des Sprachmodus
–X 11	Sprachmodus (C)	Optionen zur Auswahl des Sprachmodus
–X 17	Sprachmodus (C++)	Optionen zur Auswahl des Sprachmodus
–X 1990	Sprachmodus (C)	Optionen zur Auswahl des Sprachmodus
–X 20	Sprachmodus (C++)	Optionen zur Auswahl des Sprachmodus
–X 2011	Sprachmodus (C)	Optionen zur Auswahl des Sprachmodus
–X 2017	Sprachmodus (C++)	Optionen zur Auswahl des Sprachmodus

–X 2020	Sprachmodus (C++)	Optionen zur Auswahl des Sprachmodus
–X 89	Sprachmodus (C)	Optionen zur Auswahl des Sprachmodus
–X 90	Sprachmodus (C)	Optionen zur Auswahl des Sprachmodus
–y	Übersetzungsphasen (Prälinker)	Optionen zur Auswahl von Übersetzungsphasen
–Y F,...	allgemein	Allgemeine Optionen
–Y I,...	Präprozessor	Präprozessor-Optionen
–Y P,...	Binden	Binder-Optionen
–z dup_error	Binden	Binder-Optionen
–z dup_ignore	Binden	Binder-Optionen
–z dup_warning	Binden	Binder-Optionen
–z nodefs	Binden	Binder-Optionen

6 Literatur

Die Handbücher finden Sie im Internet unter https://bs2manuals.ts.fujitsu.com.

[1] **POSIX** (BS2000/OSD)
Grundlagen für Anwender und Systemverwalter
Benutzerhandbuch

[2] **C-Bibliotheksfunktionen für POSIX-Anwendungen** (BS2000/OSD)
Referenzhandbuch

[3] **POSIX** (BS2000/OSD)
Kommandos
Benutzerhandbuch

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

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

[6] **C++** (BS2000)
C++-Bibliotheksfunktionen
Referenzhandbuch

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

[8] **Tools.h++ V7.0**
User‘s Guide

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

[10] **C-Bibliotheksfunktionen** (BS2000/OSD)
Referenzhandbuch

[11] **AID** (BS2000/OSD)
Testen von C/C++ - Programmen
Benutzerhandbuch

[12] **AID** (BS2000/OSD)
Advanced Interactive Debugger
Benutzerhandbuch

Sonstige Literatur und Standards

[13] Programmieren in C
von Brian W. Kernighan und Dennis M. Ritchie

[14] Die C++-Programmiersprache
(3. Ausgabe)
von Bjarne Stroustrup

Die englische Originalausgabe „The C++ Programming Language (Third Edition)“ ist unter
der ISBN-Nr. 0-201-88954-4 erhältlich

[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++“