Deutsch

C/C++ V4.0A

C/C++-Compiler

Benutzerhandbuch

Ausgabe Juni 2020

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 bs2000services@ts.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

C/C++-Compiler

1 Einleitung

In diesem Kapitel werden folgende Themen behandelt:

- Kurzbeschreibung des Produkts
- Konzept und Zielgruppe des Handbuchs
- Änderungen gegenüber dem Vorgängerhandbuch
- Metasyntax
  - Allgemeine Metasyntax
  - SDF-Metasyntax

1.1 Kurzbeschreibung des Produkts

Mit C/C++ steht ein Compiler zur Verfügung, der die Programmiersprachen C und C++ in unterschiedlichen Sprachmodi unterstützt (siehe C-Sprachunterstützung des Compilers und C++-Sprachunterstützung des Compilers).

Bei der Entwicklung von C- und C++-Programmen wird der Programmierer außerdem durch einen globalen Listengenerator unterstützt, der u.a. modulübergreifende Listen (Querverweislisten, Projektlisten) erzeugt.

Der C/C++-Compiler und die anderen Entwicklungswerkzeuge sind sowohl über eine komfortable SDF-Benutzeroberfläche steuerbar, als auch aus der POSIX-Umgebung aufrufbar (siehe [1]).

Für das Erstellen und für den Ablauf von C- und C++-Programmen wird das **C**ommon **R**un**t**ime **E**nvironment CRTE zusätzlich benötigt, mit dem u.a. folgende Komponenten ausgeliefert werden:

- Include-Dateien und Module für die C-Bibliothek (ANSI-C-Funktionen, POSIX-Funktionen gemäß XPG4 spec1170, BS2000-spezifische Erweiterungen) und
- Include-Dateien und Module für diverse C++-Bibliotheken (zu C++ V2.1 kompatible C++-Bibliothek, Standard-C++-Bibliothek, Tools.h++).

1.2 Konzept und Zielgruppe des Handbuchs

Das vorliegende Benutzerhandbuch beschreibt, wie C/C++-Programme mit dem C/C++-Compiler und weiteren Komponenten des C/C++-Entwicklungssystems im Betriebssystem BS2000 in BS2000-Systemumgebung (SDF) verarbeitet werden.

Das Handbuch wendet sich an Benutzer, die über Kenntnisse der Programmiersprachen C und C++ sowie des Betriebssystems BS2000 verfügen.

Im Einzelnen befasst sich das Benutzerhandbuch mit folgenden Themen:

- Bereitstellen und Übersetzen von Quellprogrammen
- Erzeugen von Listen mit dem globalen Listengenerator
- Binden, Laden und Starten
- Ablauf von C/C++-Programmen (Parametereingaben, Testhilfen)
- Funktions- und Sprachverknüpfung
- 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)
- Kurzbeschreibung der mit CRTE ausgelieferten C++-Bibliotheken

Die Programmentwicklung in der POSIX-Umgebung mit den Kommandos `cc`, `c11`, `c89`, `CC` und `cclistgen` ist in einem eigenen Handbuch „POSIX-Kommandos des C/C++-Compilers“ [1] beschrieben. Dieses Handbuch ist in erster Linie ein Nachschlagewerk zu den POSIX-Kommandos. Ausführliche, über die POSIX-Steuerung hinausgehende Informationen zum Leistungs- und Funktionsumfang des Compilers sind dem vorliegenden Benutzerhandbuch zu entnehmen, und zwar

- Abschnitt „Verlauf der Optimierung“
- Abschnitt „Aufbau der Compilermeldungen“
- Kapitel „C-Sprachunterstützung des Compilers“
- Kapitel „C++-Sprachunterstützung des Compilers“
- Kapitel „Funktions- und Sprachverknüpfung“
- Kapitel „C++-Bibliotheken und C++-Laufzeitsystem“

Literaturhinweise werden im Text in Kurztiteln angegeben. Im Literaturverzeichnis ist der vollständige Titel jeder Druckschrift aufgeführt.

1.3 Änderungen gegenüber dem Vorgängerhandbuch

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:

MODIFY-SOURCE-PROPERTIES LANGUAGE=	
*C(MODE=*ANSI)	*C(MODE=*1990,STRICT=*NO)
*C(MODE=*STRICT-ANSI)	*C(MODE=*1990,STRICT=*YES)
*CPLUSPLUS(MODE=*ANSI)	*CPLUSPLUS(MODE=*V3,STRICT=*NO)
*CPLUSPLUS(MODE=*STRICT-ANSI)	*CPLUSPLUS(MODE=*V3,STRICT=*YES)
*CPLUSPLUS(MODE=*CPP)	*CPLUSPLUS(MODE=*V2)

Bei MODIFY-SOURCE-PROPERTIES wurde der Operand VIRTUAL-FUNCTION-TAB nach hinten verschoben und versteckt. Er wird auch nicht mehr beschrieben. Der letzte Operand mit unveränderter Position ist INSTANTIATION.

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 Metasyntax

Für die Darstellung von Kommandos und Anweisungen werden in diesem Handbuch metasprachliche Konventionen verwendet, die in den beiden folgenden Abschnitten erläutert werden:

- Allgemeine Metasyntax
- SDF-Metasyntax

1.4.1 Allgemeine Metasyntax

Für die Formatdarstellung von BS2000-Kommandos 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.
name	Kleinbuchstaben in *Kursivschrift* bezeichnen Variablen, die bei der Eingabe durch aktuelle Werte ersetzt werden müssen (siehe auch <name>).
<name>	In SDF-Formaten werden Variablen in spitze Klammern eingeschlossen.
YES NO	Die Unterstreichung kennzeichnet den Standardwert, der automatisch eingesetzt wird, wenn keine Angabe gemacht wird.
{ YES / NO } { OFF | ON }	Geschweifte Klammern schließen Alternativen ein. Die Alternativen werden durch einen Schrägstrich oder senkrechten Strich getrennt. Aus den angegebenen Größen muss eine ausgewählt werden. Wird der unterstrichene Standardwert gewünscht, ist keine Angabe erforderlich.
[ ]	Eckige Klammern schließen 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.
	Symbol zur Kennzeichnung von Hinweistexten

1.4.2 SDF-Metasyntax

In den folgenden Tabellen werden die metasprachlichen Konventionen für die SDF-Aufruf-Kommandos und -Anweisungen des C/C++-Compilers und des globalen Listengenerators dargestellt.

Tabelle 1: Metazeichen

In den Anweisungsformaten werden bestimmte Zeichen und Darstellungsformen verwendet, deren Bedeutung in der folgenden Tabelle erläutert wird.

Kennzeichnung	Bedeutung	Beispiele
GROSSBUCHSTABEN	Großbuchstaben bezeichnen Schlüsselwörter (Kommando-, Anweisungs-, Operandennamen, Schlüsselwortwerte). Schlüsselwortwerte beginnen mit *.	START-CPLUS-COMPILER COMPILE ELEMENT = *STD-ELEMENT
=	Das Gleichheitszeichen verbindet einen Operandennamen mit den dazugehörenden Operandenwerten.	LISTING = *NONE
< >	Spitze Klammern kennzeichnen Variablen, deren Wertevorrat durch Datentypen und ihre Zusätze beschrieben wird (siehe folgende Tabellen).	VERSION = <text 1..24>
Unterstreichung	Die Unterstreichung kennzeichnet den Standardwert eines Operanden, der automatisch eingesetzt wird, wenn keine Angabe gemacht wird.	SUMMARY = *YES / *NO
/	Der Schrägstrich trennt alternative Operandenwerte.	TEST-SUPPORT = *YES / *NO
(...)	Runde Klammern kennzeichnen Operandenwerte, die eine Struktur einleiten.	LIBRARY = *LINK(...)
[ ]	Eckige Klammern kennzeichnen struktureinleitende Operandenwerte, deren Angabe optional ist. Die nachfolgende Struktur kann ohne den einleitenden Operandenwert angegeben werden.	SOURCE = [*YES](...)
Einrückung	Die Einrückung kennzeichnet die Abhängigkeit zu dem jeweils übergeordneten Operanden.	LIBRARY = <filename> / *LINK(...) *LINK(...) | LINK-NAME =

|	Der Strich kennzeichnet zusammengehörende Operanden einer Struktur. Sein Verlauf zeigt Anfang und Ende einer Struktur an. Innerhalb einer Struktur können weitere Strukturen auftreten. Die Anzahl senkrechter Striche vor einem Operanden entspricht der Strukturtiefe.	*LIBRARY-ELEMENT(...) | LIBRARY = | ,ELEMENT = | | VERSION =
,	Das Komma steht vor weiteren Operanden der gleichen Strukturstufe.	,SOURCE = *NO ,SUMMARY = *NO
list-poss(n): list-poss:	Aus den auf list-poss folgenden Operandenwerten kann eine Liste gebildet werden. (n) bedeutet, dass maximal n Elemente in der Liste vorkommen können. Enthält die Liste mehr als ein Element, muss sie in runde Klammern eingeschlossen werden. Im Handbuch wird (n) nur angegeben, wenn es sich um eine Compilerspezifische Maximalanzahl handelt. list-poss ohne (n) bedeutet, dass für die Maximalanzahl der SDF-Defaultwert 2000 gilt.	list-poss (127): *STD / BY-SOURCE / <c-string> list-poss: <filenname> / *LINK(...)

Tabelle 2: Datentypen

Variable Operandenwerte werden in SDF durch Datentypen dargestellt. Jeder Datentyp repräsentiert einen bestimmten Wertevorrat. Die Anzahl der Datentypen ist beschränkt auf die in Tabelle 2 beschriebenen Datentypen.

Die Beschreibung der Datentypen gilt für alle Anweisungen. Deshalb werden bei den entsprechenden Operandenbeschreibungen nur noch Abweichungen von Tabelle 2 erläutert.

Datentyp	Zeichenvorrat	Bedeutung
alphanum-name	A...Z 0...9 $, #, @	
c-string	EBCDIC-Zeichen	In Hochkommata eingeschlossene Folge von EBCDIC-Zeichen. Der Buchstabe C kann vorangestellt werden. Hochkommata innerhalb des c-string müssen verdoppelt werden. Der Datentyp c-string ist mit dem Zusatz with-low versehen, d.h. es werden Groß- und Kleinbuchstaben unterschieden. Beispiele für gültige Darstellungen: 'abc' , C'_abc' , 'ABC' , 'printf("Makrotext\ n")'.
composed-name	A...Z 0...9 $,#,@ Bindestrich Punkt	Name oder Version eines PLAM-Bibliothekselements. composed-name ist mit dem Zusatz underscore (Unterstrich) versehen (vgl. Tabelle 3).
filename	A...Z 0...9 $,#,@ Bindestrich Punkt	Linkname oder vollqualifizierter Name einer katalogisierten Datei oder PLAM-Bibliothek. Die maximale Länge von Linknamen beträgt 8 Zeichen, von vollqualifizierten Namen einschließlich cat-id und user-id 54 Zeichen, ohne cat-id und user-id 41 Zeichen.
		Eingabeformat:
		linkname
		:cat:$user.{ datei / datei(nr) / gruppe / gruppe{ (*abs) / (+rel) / (-rel) } }
		linkname erstes und letztes Zeichen darf kein Bindestrich oder Punkt sein; max. 8 Zeichen; muss mindestens A...Z enthalten.

Datentyp	Zeichenvorrat	Bedeutung
filename (Fortsetzung)		:cat: wahlfreie Angabe der Katalogkennung; Zeichenvorrat auf A...Z und 0...9 eingeschränkt; max. 4 Zeichen; ist in Doppelpunkte einzuschließen; Standardwert ist die Katalogkennung, die der Benutzerkennung laut Eintrag im Benutzerkatalog zugeordnet ist.
		$user. wahlfreie Angabe der Benutzerkennung; Zeichenvorrat ist A...Z, 0...9, $, #, @; max. 8 Zeichen; darf nicht mit einer Ziffer beginnen; $ und Punkt müssen angegeben werden; Standardwert ist die eigene Benutzerkennung.
		$. (Sonderfall) System-Standardkennung
		datei Datei- oder Jobvariablenname; letztes Zeichen darf kein Bindestrich oder Punkt sein; max. 41 Zeichen; muss mindestens ein Zeichen aus A...Z enthalten.
		#datei @datei (Sonderfall) # oder @ als erstes Zeichen kennzeichnet je nach Systemgenerierung temporäre Dateien und Jobvariablen.
		datei(nr) Banddateiname nr: Versionsnummer; Zeichenvorrat ist A...Z, 0...9, $, #, @. Klammern müssen angegeben werden.

Datentyp	Zeichenvorrat	Bedeutung
filename (Fortsetzung)		gruppe Name einer Dateigenerationsgruppe (Zeichenvorrat siehe unter "datei")
		gruppe{ (*abs) / (+rel) / (-rel) }
		(*abs): absolute Generationsnummer (1-9999); * und Klammern müssen angegeben werden.
		(+rel) / (-rel): relative Generationsnummer (0-99); Vorzeichen und Klammern müssen angegeben werden.
integer	0...9,+,	+ bzw. - kann nur erstes Zeichen sein.
name	A...Z 0...9 $,#,@	In der Anweisung MODIFY-SOURCE-PROPERTIES C/C++-Name für DEFINE / UNDEFINE. Der Datentyp name ist mit dem Zusatz underscore (Unterstrich) versehen (vgl. Tabelle 3). Kleinbuchstaben können mit name nicht abgebildet werden. Hierfür muss der Datentyp c-string verwendet werden. Maximale Länge: 125 Zeichen
posix-filename	A...Z 0...9 Sonderzeichen	Zeichenfolge, die maximal 255 Zeichen lang ist. Besteht entweder aus einem oder zwei Punkten, oder aus alphanumerischen Zeichen und Sonderzeichen; Sonderzeichen sind mit dem Zeichen \ zu entwerten. Nicht erlaubt ist das Zeichen /. Zwischen Groß- und Kleinschreibung wird unterschieden.
posix-pathname	A...Z 0...9 Sonderzeichen Strukturkennzeichen: Schrägstrich	Eingabeformat: [/]$part_1$[/.../$part_n$] wobei part $_1$ ein posix-filename ist; maximal 1023 Zeichen; für Source-und Include-Dateien maximal 247 Zeichen; ist mit dem Zusatz mandatory-quotes versehen und muss deshalb generell in Hochkommata eingeschlossen werden.

x-string	Sedezimal: 00...FF	In Hochkomma eingeschlossene Folge von Sedezimalwerten. Der Buchstabe X muss vorangestellt werden.

Tabelle 3: Zusätze zu Datentypen

Zusätze zu Datentypen kennzeichnen weitere Eingabevorschriften für Datentypen. Die Zusätze schränken den Wertevorrat ein oder erweitern ihn. Im Handbuch werden folgende Zusätze in gekürzter Form dargestellt:

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

Die Beschreibung der Zusätze zu den Datentypen gilt für alle Anweisungen. Deshalb werden bei den entsprechenden Operandenbeschreibungen nur noch Abweichungen von Tabelle 3 erläutert.

Zusatz	**Bedeutung**
x..y	1. beim Datentyp integer: Intervallangabe x: Mindestwert, der für integer erlaubt ist. x ist eine ganze Zahl, die mit einem Vorzeichen versehen werden darf. y: Maximalwert, der für integer erlaubt ist. y ist eine ganze Zahl, die mit einem Vorzeichen versehen werden darf. 2. bei den übrigen Datentypen: Längenangabe x: Mindestlänge für den Operandenwert; x ist eine ganze Zahl ohne Vorzeichen. y: Maximallänge für den Operandenwert; y ist eine ganze Zahl ohne Vorzeichen.
with	Erweitert die Angabemöglichkeiten für einen Datentyp.
-low	Groß- und Kleinbuchstaben werden unterschieden.
-under	Der Unterstrich _ ist als zusätzliches Zeichen erlaubt.
without	Schränkt die Angabemöglichkeiten für einen Datentyp ein.
-cat	Die Angabe einer Katalogkennung ist nicht erlaubt.

-corr	Eingabeformat: [[C]'][V][m]m.na['] Angaben zum Datentyp product-version dürfen den Korrekturstand nicht enthalten.
-gen	Die Angabe einer Dateigeneration oder Dateigenerationsgruppe ist nicht erlaubt.
-man	Eingabeformat: [[C]'][V][m]m.n['] Angaben zum Datentyp product-version dürfen weder Freigabe- noch Korrekturstand enthalten.
-user	Die Angabe einer Benutzerkennung ist nicht erlaubt.
-vers	Die Angabe der Version (siehe datei(nr)) ist bei Banddateien nicht erlaubt.
mandatory	Bestimmte Angaben sind für einen Datentyp zwingend erforderlich
-quotes	Angaben zu den Datentypen posix-filename bzw. posix-pathname müssen in Hochkommata eingeschlossen werden.

2 Das C/C++-Entwicklungssystem im Überblick

In diesem Kapitel werden folgende Themen behandelt:

- Vom Quellprogramm zum Programmablauf
- Voraussetzungen zum Übersetzen, Binden und Programmablauf
- Allgemeine Leistungsmerkmale des C/C++-Compilers
- C/C++-spezifische Komponenten des CRTE
  - Include-Bibliotheken
  - Modulbibliotheken
- Editieren von Quellprogrammen
- POSIX-Unterstützung
  - Compiler-Ein-/Ausgaben im POSIX-Dateisystem
  - Benutzen der POSIX-Bibliotheksfunktionen
- Einführungsbeispiele
  - Beispiel 1: Übersetzen, Binden und Starten eines C-Programms
  - Beispiel 2: Übersetzen, Binden und Starten eines C++-Programms
  - Beispiel 3: Übersetzen eines C-Quellprogramms, das in einer POSIX- Datei steht und POSIX-Bibliotheksfunktionen benutzt

2.1 Vom Quellprogramm zum Programmablauf

Damit aus einem C/C++-Quellprogramm ein ablauffähiges Programm wird, sind drei Schritte nötig:

1. Bereitstellen des Quellprogramms und der Include-Dateien

   Das Quellprogramm kann in einer katalogisierten BS2000-Datei, in einem PLAM-Bibliothekselement (Typ S) oder in einer POSIX-Datei stehen.
   Include-Dateien können in PLAM-Bibliothekselementen (Typ S), in POSIX-Dateien und auch in katalogisierten Dateien stehen, falls das Quellprogramm selbst in einer katalogisierten Datei steht.
   Die Standard-Include-Dateien für die C- und C++-Bibliotheksfunktionen (nicht die POSIX-Funktionen) stehen in den CRTE-Bibliotheken $.SYSLIB.CRTE, $.SYSLIB.CRTE.CPP und $.SYSLIB.CRTE.CXX01, die für die POSIX-Bibliotheksfunktionen in der Bibliothek$.SYSLIB.POSIX-HEADER.

2. Übersetzen

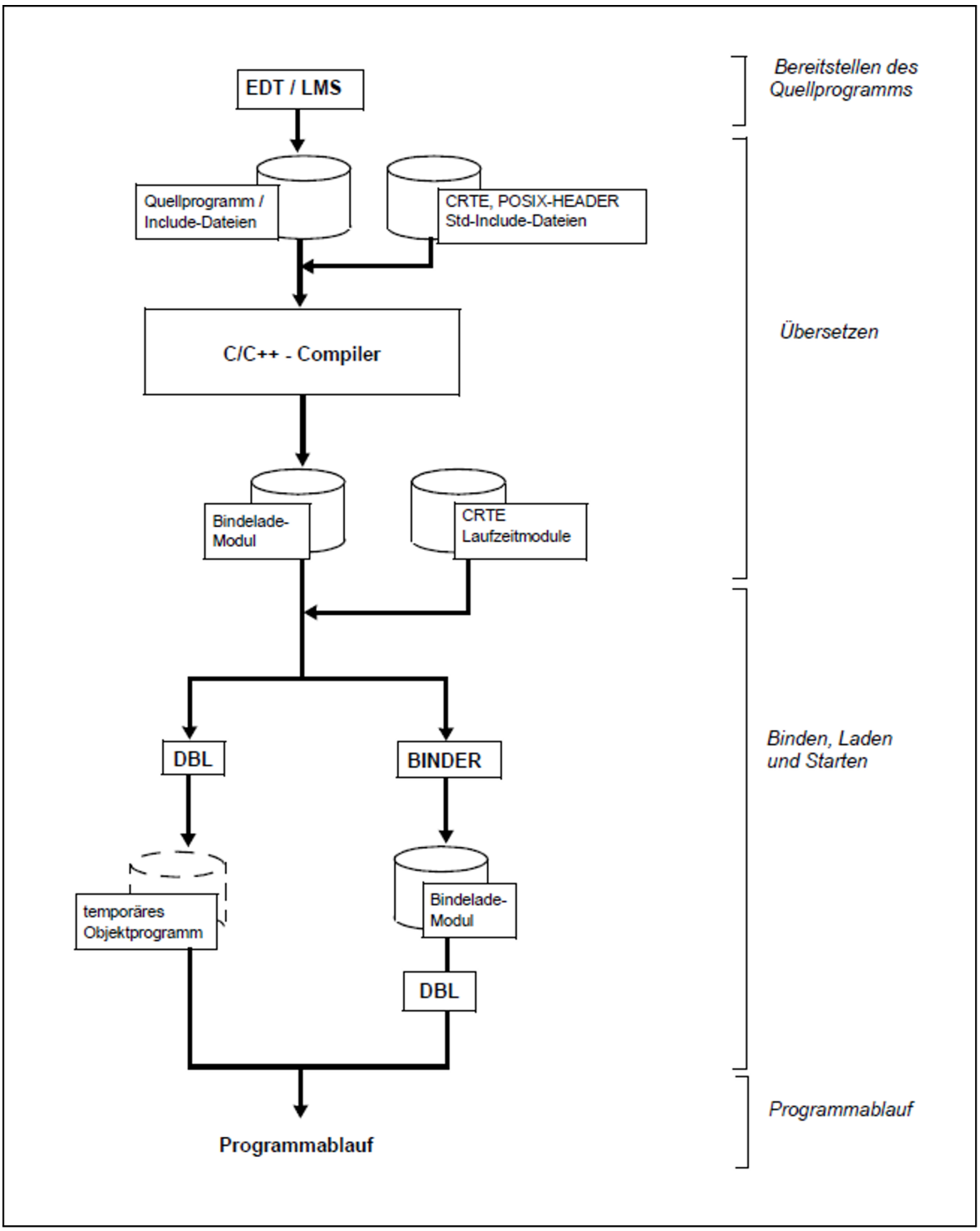
   Das Quellprogramm muss in Maschinensprache umgesetzt werden. Der Compiler erzeugt Module ausschließlich im LLM-Format und speichert sie wahlweise in PLAM-Bibliothekselemente (Typ L) oder in POSIX-Objektdateien ab.

3. Binden

   Die bei der Übersetzung erzeugten Module werden mit Modulen des C-Laufzeitsystems und bei Verwendung von C++ zusätzlich mit dem C++-Laufzeitsystem verknüpft. Es entsteht eine ablauffähige Einheit. Module, die in den Modi C++ V3 bzw. C++ 2017 erzeugt wurden, sollen nicht durch direkten Aufruf des BINDER gebunden werden, sondern ausschließlich mittels der Compiler-Anweisungen MODIFY-BIND-PROPERTIES und BIND.

Die folgende Übersicht verdeutlicht diese Grundschritte und gibt an, wo die entsprechenden Informationen im Handbuch zu finden sind.

Grundschritte für die Erstellung eines ablauffähigen C/C++-Programms


EDT / LMS
Quellprogramm / Include-Dateien
CRTE, POSIX-HEADER Std-Include-Dateien
C/C++ - Compiler
Bindelade-Modul
CRTE Laufzeitmodule
DBL
BINDER
temporäres Objektprogramm
Bindelade-Modul
DBL
Programmablauf
Bereitstellen des Quellprogramms
Übersetzen
Binden, Laden und Starten
Programmablauf

2.2 Voraussetzungen zum Übersetzen, Binden und Programmablauf

Der problemlose Ablauf des Compilers, das Binden eines Programms und der Programmablauf setzen voraus, dass folgende Komponenten installiert sind:

- die Aufrufprogramme des C/C++-Compilers und globalen Listengenerators
- die SDF-C/C++-Syntaxdatei
- die DMS-/PLAM-/BINDER-Meldungsdateien
- die C/C++-Meldungsdatei

Außerdem muss CRTE (Common Runtime Environment) ab V10.1A bzw. V11.1A vorhanden sein, mit dem u.a. folgende Komponenten ausgeliefert werden:

- die C- und C++-Laufzeitmodule
- die ILCS-Laufzeitmodule
- die Standard-Include-Dateien für die C-Bibliotheksfunktionen (nicht die POSIX-Funktionen) und C++-Bibliotheksfunktionen

Die nötigen Vorarbeiten zur Verwendung der SDF-Steuerung sowie der Meldungsdateien sind in der Freigabemitteilung zu C/C++ V4.0A beschrieben.

Hinweise zur Installation und Handhabung des CRTE finden sich in der Freigabemitteilung zu CRTE sowie im CRTE-Benutzerhandbuch [4].

Prinzipiell ist der C/C++-Compiler ab BS2000/OSD V10.0 einsetzbar.
Diverse, im vorliegenden Handbuch beschriebene Leistungen des C/C++-Compilers setzen jedoch weitere Softwareprodukte voraus:

- Nutzung der POSIX-Bibliotheksfunktionen: POSIX-HEADER ab V10.1A bzw. V11.1A

Für weitere Einzelheiten zu Software-Voraussetzungen verweisen wir auf die Freigabemitteilung zu C/C++ V4.0A.

2.3 Allgemeine Leistungsmerkmale des C/C++-Compilers

Unterstützte C- und C++-Sprachstandards

Der Compiler unterstützt die folgenden C- und C++-Sprachstandards:

- ANSI-/ISO-C mit dem ISO-C-Addendum 1 (1994), im Folgenden auch C89
- ISO/IEC C (2011), im folgenden auch C11
- ISO/IEC C++ (2017), im Folgenden auch C++ 2017

Kompatibilität

Für die Portierung bzw. Migration von älteren C- und C++-Anwendungen bietet der Compiler die folgenden Sprachmodi an:

- Kernighan&Ritchie-C
- Cfront-C++ V3.0.3, im Folgenden auch C++ V2
- die Version von C++, die von C/C++ V3.2 angeboten wurde, im Folgenden auch C++ V3
- Präprozessor-Dialekt gemäß Reiser cpp und Johnson pcc

Portable Software

Für die Entwicklung portabler Software gibt es „strikte“ C- und C++-Sprachmodi, in denen alle Abweichungen von den Sprachstandards diagnostiziert werden.

2.4 C/C++-spezifische Komponenten des CRTE

Dieser Abschnitt gibt einen Überblick über diejenigen „Behälter“ des CRTE, die für die C- und C++-Programmierung relevant sind. Für detailliertere Informationen zum Leistungsumfang und zur Benutzung der mit CRTE bereitgestellten C++-Bibliotheken beachten Sie bitte auch das Kapitel „C++-Bibliotheken und C++-Laufzeitsystem“.

Ausführliche Informationen zum Gesamtkonzept des CRTE finden Sie im CRTE-Benutzerhandbuch [4]. Außerdem sei auf die im Literaturverzeichnis aufgeführten diversen Referenzhandbücher zu den C- und C++-Bibliotheksfunktionen verwiesen.

2.4.1 Include-Bibliotheken

Bibliothek SYSLIB.CRTE

Diese Bibliothek enthält

- Standard-Include-Elemente (Typ S) für die C-Bibliotheksfunktionen (ANSI-C und BS2000-spezifische Erweiterungen)
- Standard-Include-Elemente (Typ S) für die Klassen und Funktionen der Standard-C++-Bibliothek für den Sprachmodus C++ V3
- Standard-Include-Elemente (Typ S) für die Klassen und Funktionen der C++-V3-Bibliothek Tools.h++

In dieser Bibliothek nicht enthalten sind

- die Standard-Include-Elemente für die Anwendung der POSIX-Bibliotheksfunktionen. Diese stehen nach Installation von POSIX-HEADER (Release-Unit des BS2000/OSD-BC) in der Bibliothek SYSLIB.POSIX-HEADER bereit.
- die Standard-Include-Elemente der Standard-C++-Bibliothek gemäß dem Standard C++ 2017; diese sind Bestandteil der Bibliothek SYSLIB.CRTE.CXX01.
- die Standard-Include-Elemente der Cfront-kompatiblen C++-Bibliotheksfunktionen für komplexe Mathematik und stromorientierte Ein-/Ausgabe; diese sind Bestandteil der Bibliothek SYSLIB.CRTE.CPP.

Bibliothek SYSLIB.CRTE.CXX01

Diese Bibliothek enthält die Standard-Include-Elemente für die Klassen und Funktionen der Standard-C++-Bibliothek gemäß dem Standard C++ 2017.
Siehe auch Abschnitt „Die C++-Bibliothek für den Sprachmodus C++ 2017“.

Bibliothek SYSLIB.CRTE.CPP

Diese Bibliothek enthält die Standard-Include-Elemente für die Klassen und Funktionen der Cfront-kompatiblen C++-Bibliothek für komplexe Mathematik und stromorientierte Ein-/Ausgabe, die im Cfront-C++-Modus des Compilers genutzt werden.
Siehe auch Abschnitt „Die C++-Bibliothek für den Cfront-C++ Sprachmodus V2“.

2.4.2 Modulbibliotheken

Bibliothek SYSLNK.CRTE

Diese Bibliothek enthält

- Einzelmodule des C-Laufzeitsystems (Objektmodule, Typ R)

  Diese Module können statisch oder dynamisch gebunden werden. Das C-Laufzeitsystem kann auch dynamisch nachgeladen werden. In diesem Fall muss beim Binden mit dem BINDER statt der Bibliothek SYSLNK.CRTE die Bibliothek SYSLNK.CRTE.PARTIAL-BIND (bei Bindetechnik Standard-Partial-Bind) bzw. die Bibliothek SYSLNK.CRTE.COMPL (bei Bindetechnik Complete Partial-Bind) angegeben werden.

  Das C-Laufzeitsystem wird immer für den Ablauf von C/C++-Programmen benötigt. Es enthält u.a. den Code für sämtliche C-Bibliotheksfunktionen, zentrale Ein-/Ausgaberoutinen auch zur Realisierung der C++-Ein-/Ausgabefunktionen und weitere Routinen zur Realisierung von Betriebssystemschnittstellen.

  Viele Entry-Namen des C-Laufzeitsystems beginnen mit „IC@“, „ICS“ oder „ICX“.

- Namens-Adaptermodule für neue C-Bibliotheksfunktionen (Objektmodule, Typ R und LLMs, Typ L)

  Die Adaptermodule gehören zu den nicht vorladbaren Bestandteilen des C-Laufzeitsystems und müssen deshalb in das Anwendungsprogramm eingebunden werden. Sie sind sowohl in der Bibliothek SYSLNK.CRTE als auch in der Bibliothek SYSLNK.CRTE.PARTIAL-BIND enthalten.

  Technische Detailinformationen zu diesen Adaptermodulen finden Sie im Abschnitt „Konzept der Namens-Adaptermodule im C-Laufzeitsystem“.

- C-Laufzeitsystem als dynamisch nachladbares Großmodul (LLM, Typ L)

  Dieses Großmodul wird zum Ablaufzeitpunkt dynamisch in den Benutzer-Adressraum nachgeladen, wenn beim Binden an Stelle der Bibliothek SYSLNK.CRTE die Bibliothek SYSLNK.CRTE.PARTIAL-BIND eingebunden wird, und das nachladbare C-Laufzeitsystem nicht vorgeladen ist.

Bibliotheken SYSLNK.CRTE.PARTIAL-BIND und SYSLNK.CRTE.COMPL

Diese Bibliotheken ermöglichen es, C-Programme zu binden, die keine offenen Externverweise mehr enthalten und das C-Laufzeitsystem erst zum Ablaufzeitpunkt dynamisch nachladen. Dazu enthalten die Bibliotheken Verbindungsmodule, die an Stelle der C-Laufzeitmodule eingebunden werden und alle offenen Externbezüge eines C-Programms auf das C-Laufzeitsystem befriedigen. Außerdem enthalten die Bibliotheken alle Module, die für das Binden eines C-Programms ohne offene Externbezüge auf das CRTE benötigt werden (z.B. ILCS-Module, Namens-Adaptermodule).

SYSLNK.CRTE.PARTIAL-BIND wird für die Bindetechnik Standard Partial-Bind benötigt, während SYSLNK.CRTE.COMPL von der Bindetechnik Complete Partial-Bind verwendet wird. Näheres zur Bindetechnik Partial-Bind sowie zu den Besonderheiten von Standard Partial-Bind und Complete Partial-Bind finden Sie im Handbuch „CRTE“ [4].

Das C-Laufzeitsystem selbst steht als dynamisch nachladbares Großmodul in der Bibliothek SYSLNK.CRTE zur Verfügung. Zum Ablaufzeitpunkt wird entweder das vorgeladene C-Laufzeitsystem zum Programm konnektiert oder das C-Laufzeitsystem wird - wenn es nicht vorgeladen ist - in den Benutzeradressraum nachgeladen.

Da nicht das gesamte C-Laufzeitsystem, sondern lediglich die Verbindungsmodule eingebunden werden, benötigt das fertig gebundene Programm bzw. Modul deutlich weniger Plattenspeicherplatz als beim statischen Einbinden der C-Laufzeitmodule aus der Bibliothek SYSLNK.CRTE. Außerdem wird durch die geringere Größe die Ladezeit verkürzt, falls das C-Laufzeitsystem vorgeladen ist.

Zu technischen Details siehe auch CRTE-Benutzerhandbuch [4].

Bibliotheken SYSLNK.CRTE.CPP und SYSLNK.CRTE.CFCPP

Diese Bibliotheken enthalten die Module des Cfront-kompatiblen C++-Laufzeitsystems (LLMs, Typ L).

SYSLNK.CRTE.CPP enthält die Module aller C++-Bibliotheksfunktionen für komplexe Mathematik und Standard-Ein-/Ausgabe, die im Cfront-C++-Modus des Compilers verfügbar sind. Die Entrynamen beginnen mit „ICP“.

SYSLNK.CRTE.CFCPP enthält die Module für interne Laufzeitroutinen, die im Cfront-C++-Modus des Compilers zur Realisierung der Initialisierung, Speicherverwaltung etc. benötigt werden. Die Entrynamen beginnen mit „IPP“.

Die Module können statisch mit dem BINDER oder dynamisch mit dem DBL gebunden werden. Die Module aus der Bibliothek SYSLNK.CRTE.CFCPP müssen vorrangig vor den Modulen aus der Bibliothek SYSLNK.CRTE.CPP eingebunden werden.

Siehe auch Abschnitt „Die C++-Bibliothek für den Cfront-C++ Sprachmodus V2“.

Bibliothek SYSLNK.CRTE.POSIX

Diese Bibliothek enthält einen Bindeschalter, der immer dann eingebunden werden muss, wenn die POSIX-Funktionen des C-Laufzeitsystems verwendet werden. Dieses Modul muss vorrangig vor den Modulen in der Bibliothek SYSLNK.CRTE bzw. SYSLNK.CRTE.PARTIAL-BIND bzw. SYSLNK.CRTE.COMPL eingebunden werden. Wir empfehlen, beim Binden mit dem BINDER, die Bindeschalter-Bibliothek mit einer INCLUDE-Anweisung (ohne Angabe der Modulnamen) einzubinden, da bei der Verwendung von RESOLVE-Anweisungen sonst strikt die Reihenfolge beachtet werden müsste. Analog sollte beim Binden mit der BIND-Anweisung des Compilers in der MODIFY-BIND-PROPERTIES-Anweisung die INCLUDE-Option verwendet werden.

Bibliotheken SYSLNK.CRTE.STDCPP, SYSLNK.CRTE.RTSCPP

Diese Bibliotheken enthalten die Module des C++ V3 Laufzeitsystems (LLMs, Typ L).

SYSLNK.CRTE.STDCPP enthält die Module der Standard-Bibliothek, die in dem C++ V3-Modus des Compilers genutzt werden kann.

SYSLNK.CRTE.RTSCPP enthält die Module für interne Laufzeitroutinen, die in dem C++ V3-Modus des Compilers zur Realisierung u.a. der C++-Ausnahmebehandlung und der C++-Laufzeittypinformationen (RTTI) benötigt werden.

Die Module können statisch mit der BIND-Anweisung des Compilers oder dynamisch mit dem DBL eingebunden werden.

Siehe auch Abschnitt „Die C++-Bibliothek für den Sprachmodus C++ V3“.

Bibliothek SYSLNK.CRTE.TOOLS

Diese Bibliothek enthält die Module (LLMs, Typ L) für die C++-V3-Bibliothek Tools.h++, die im C++ V3-Modus des Compilers genutzt werden kann.

Die Module können statisch mit der BIND-Anweisung des Compilers oder dynamisch mit dem DBL eingebunden werden.

Siehe auch Abschnitt „Die C++-V3-Bibliothek Tools.h++“.

Bibliothek SYSLNK.CRTE.CPP-COMPL

Diese Bibliothek enthält Adaptoren für die Bibliotheken SYSLNK.CRTE.STDCPP, SYSLNK.CRTE.RTSCPP und SYSLNK.CRTE.TOOLS.

Bibliothek SYSLNK.CRTE.CXX01

Diese Bibliothek enthält die Module des C++ 2017 Laufzeitsystems (LLMs, Typ L).

Siehe auch Abschnitt „Die C++-Bibliothek für den Sprachmodus C++ 2017“.

2.5 Editieren von Quellprogrammen

Speicherungsarten von Quellprogrammen und Include-Dateien

Der C/C++-Compiler verarbeitet Quellprogramme und Include-Dateien, die folgendermaßen abgespeichert sind:

Quellprogramme als

- katalogisierte SAM-Dateien und ISAM-Dateien (mit KEYPOS=5 und KEYLEN <= 16)
- PLAM-Bibliothekselemente vom Typ S
- POSIX-Quelldateien

Include-Dateien als

- katalogisierte Dateien, falls das Quellprogramm selbst eine katalogisierte Datei ist
- PLAM-Bibliothekselemente vom Typ S
- POSIX-Quelldateien

Dateiaufbereiter EDT

Für das Editieren eines C/C++-Quellprogramms steht im BS2000 der Dateiaufbereiter EDT zur Verfügung. Mit ihm kann man katalogisierte SAM-/ISAM-Dateien, PLAM-Bibliothekselemente und POSIX-Dateien bearbeiten.

Der EDT wandelt standardmäßig Kleinbuchstaben in Großbuchstaben um. Da C/C++-Quellprogrammtexte vorwiegend Kleinbuchstaben enthalten, muss nach Aufruf des EDT mit der Anweisung LOWER ON die Umwandlung in Großbuchstaben unterbunden werden.

Folgende Tabelle gibt einen Überblick über die wichtigsten EDT-Anweisungen zur Bearbeitung von Dateien und Bibliothekselementen. Der EDT ist detailliert im Handbuch
„EDT (BS2000/OSD)“ [15] beschrieben.

SAM-Dateien	
@READ‘datei‘	Inhalt einer SAM-Datei in die aktuelle Arbeitsdatei einlesen
@WRITE‘datei‘	Inhalt der aktuellen Arbeitsdatei in eine SAM-Datei schreiben
ISAM-Dateien	
@GET‘datei‘	Inhalt einer ISAM-Datei in die aktuelle Arbeitsdatei einlesen
@SAVE‘datei‘	Inhalt der aktuellen Arbeitsdatei in eine ISAM-Datei schreiben
@OPEN‘datei‘	ISAM-Datei real öffnen
@CLOSE	Eine mit @OPEN geöffnete ISAM-Datei schließen
PLAM-Bibliothekselemente	
@OPEN LIB=bib(ELEM=elem)	Bibliothekselement in der aktuellen Arbeitsdatei öffnen
@COPY LIB=bib(ELEM=elem)	Inhalt eines Bibliothekselements in die aktuelle Arbeitsdatei einlesen

@WRITE LIB=bib(ELEM=elem)	Inhalt der aktuellen Arbeitsdatei in ein Bibliothekselement schreiben
@CLOSE	Ein mit @OPEN geöffnetes Bibliothekselement schließen
POSIX-Dateien	
@XOPEN FILE=pfadname,MODE= ANY/UPDATE/NEW/REPLACE	Öffnen und Einlesen einer existierenden oder Anlegen einer neuen POSIX-Datei
@CLOSE	Eine mit @XOPEN geöffnete POSIX-Datei schließen
@XCOPY FILE=pfadname	Inhalt einer POSIX-Datei in die aktuelle Arbeitsdatei einlesen
@XWRITE FILE=pfadname,MODE= ANY/UPDATE/NEW/REPLACE	Inhalt der aktuellen Arbeitsdatei in eine POSIX-Datei schreiben

EDT-Anweisungen für die Bearbeitung von Dateien und Bibliothekselementen (Auszug)

Die PLAM-Bibliothekselemente speichert der EDT standardmäßig als Elemente vom Typ S mit der höchsten Versionsnummer (X'FF') ab.

Tastaturbelegung

Der Benutzer benötigt zum Programmieren in C und C++ eine Reihe von Sonderzeichen, die möglicherweise auf Tastaturen nicht vorhanden sind. Insbesondere auf Tastaturen mit deutschem Zeichensatz sind einige C/C++-spezifische Zeichen mit Umlauten belegt.
Die folgende Tabelle zeigt die unterschiedliche Tastaturbelegung und den (internationalen) Sedezimal-Code. Anhand der Tabelle kann bei Bedarf eine benutzereigene Tastaturbelegung erstellt werden.

Bezeichnung bzw. Funktionen in C und C++	**„englische" Tastatur**	**„deutsche" Tastatur**	**Sedezimal-Code**
spitze Klammer auf	<	<	4C
spitze Klammer zu	>	>	6E
logisches Oder	|	ö	4F
logisches Exklusiv-Oder	^	^	6A
Unterstrich	_	_	6D
eckige Klammer auf	[	Ä	BB
eckige Klammer zu	]	Ü	BD
Gegenschrägstrich	\	Ö	BC
geschweifte Klammer auf	{	ä	FB
geschweifte Klammer zu	}	ü	FD
Bit-Komplement	~	ß	FF

2.6 POSIX-Unterstützung

In diesem Abschnitt werden folgende Themen behandelt:

- Compiler-Ein-/Ausgaben im POSIX-Dateisystem
- Benutzen der POSIX-Bibliotheksfunktionen

2.6.1 Compiler-Ein-/Ausgaben im POSIX-Dateisystem

Der C/C++-Compiler unterstützt zusätzlich zum BS2000-Dateisystem (DVS-Dateien, PLAM-Bibliotheken etc.) das POSIX-Dateisystem. Sämtliche Ein- und Ausgaben des Compilers in einem Übersetzungs-, Präprozessor- oder Syntax-Analyse-Lauf können auch über POSIX-Dateien erfolgen.
Im Einzelnen sind dies:

- die Eingabe von Quellprogrammen (siehe SOURCE-Option in den Anweisungen COMPILE, PREPROCESS, CHECK-SYNTAX)
- die Eingabe von Include-Dateien (siehe Optionen USER-INCLUDE-LIBRARY und STD-INCLUDE-LIBRARY in der MODIFY-INCLUDE-LIBRARY-Anweisung)
- die Ausgabe von LLMs (siehe MODULE-OUTPUT-Option in der COMPILE-Anweisung)
- die Ausgabe von expandierten Quellprogrammen (siehe OUTPUT-Option in der PRE-PROCESS-Anweisung)
- die Ausgabe von Übersetzungslisten (siehe OUTPUT-Option in der MODIFY-LISTING-PROPERTIES-Anweisung)
- die Ausgabe von Meldungslisten (siehe OUTPUT-Option in der MODIFY-DIAGNOSTIC-PROPERTIES-Anweisung)
- die Ausgabe von CIF-Informationen (siehe OUTPUT-Option in der MODIFY-CIF-PROPERTIES-Anweisung)

Beliebige Mischfälle, d.h. die Ein- und Ausgabe sowohl von BS2000- als auch von POSIX-Dateien in einem Übersetzungslauf, sind möglich.

Ablage der Quellprogramm- und Include-Dateien

Die Quellprogramm- und Include-Dateien können in EBCDIC- und ASCII-Code vorliegen. Im POSIX-Dateisystem ist der EBCDIC-Code voreingestellt, in Dateisystemen auf fernen UNIX-Rechnern oder PCs der ASCII-Code. Alle Dateien eines Dateisystems (POSIX-Dateisystem oder eingehängtes fernes Dateisystem) müssen jeweils im selben (voreingestellten) Codeset vorliegen. Der Compiler fragt das Codeset eines Dateisystems zentral und nicht pro einzelne Datei ab. Dateien eines ASCII-Dateisystems werden intern für die Übersetzung nach EBCDIC konvertiert, wenn die Umgebungsvariable SYSPOSIX.IO-CONVERSION den Wert YES hat (siehe Abschnitte „Umgebungsvariablen" und „Unterstützung von Dateisystemen in ASCII" im Handbuch „C-Bibliotheksfunktionen“ [2]).

Im Unterschied zur Programmentwicklung in der POSIX-Shell (siehe Handbuch „POSIX-Kommandos des C/C++-Compilers“ [1]), müssen die Dateinamen der Quellprogramme nicht unbedingt eines der Standard-Suffixe „.c“, „.C“, „.i“ etc. enthalten.

Ausgabe von LLMs

Die vom Compiler erzeugten LLMs können in POSIX-Objektdateien („.o“-Dateien) geschrieben werden. Eine sinnvolle Weiterverarbeitung dieser Objektdateien ist nur im POSIX-Subsystem mit den Kommandos `cc`, `c11`, `c89` oder `CC` möglich (siehe Handbuch „POSIX-Kommandos des C/C++-Compilers“ [1]). In UNIX-Systemen wird das erzeugte Objektdatei-Format (LLM) nicht unterstützt.

Sonstige Compilerausgaben

- Expandierte, weiterübersetzbare Quellprogramme

  Das Ergebnis eines Präprozessorlaufes kann in eine POSIX-Quelldatei („.i“-Datei in C, „`.I`“-Datei in C++) geschrieben werden. Diese Datei kann in der POSIX-Shell mit den Kommandos `cc`, `c89` oder `CC` weiterverarbeitet werden, in BS2000-Umgebung über die SDF-Schnittstelle des C/C++-Compilers.

- Übersetzungslisten

  Die Übersetzungslisten können in eine POSIX-Listendatei („.lst“-Datei) geschrieben werden. Diese Listendatei kann in der POSIX-Shell mit dem Kommando `bs2lp` ausgedruckt werden, in BS2000-Umgebung mit dem SDF-Kommando PRINT-DOCUMENT.

- Diagnosemeldungen

  Die Fehlermeldungen des Compilers können in eine POSIX-Datei („.diag“-Datei) geschrieben werden. Diese Datei kann beispielsweise mit dem EDT in der POSIX-Shell (Kommando `edt`) und in BS2000-Umgebung gelesen werden (vgl. Hinweise zum EDT, "Editieren von Quellprogrammen").

- CIF-Informationen

  Die CIF-Informationen für die Erstellung von globalen Listen können in eine POSIX-Datei („.cif“-Datei) geschrieben werden. Diese CIF-Dateien können in der POSIX-Shell mit dem Kommando `cclistgen` weiterverarbeitet werden (siehe Handbuch „POSIX-Kommandos des C/C++-Compilers“ [1]), in BS2000-Umgebung über die SDF-Schnittstelle des globalen Listengenerators (siehe START-CPLUS-LISTING-GENERATOR, "Steuerung des globalen Listengenerators").

Ausgabe-Codeset

Das Ausgabe-Codeset der Dateien (EBCDIC oder ASCII) richtet sich nach dem Codeset des Zieldateisystems. Im POSIX-Dateisystem werden die Dateien vom Compiler im EBCDIC-Code abgelegt, in Dateisystemen auf UNIX-Rechnern im ASCII-Code, wenn die Umgebungsvariable SYSPOSIX.IO-CONVERSION den Wert YES hat (siehe Abschnitte „Umgebungsvariablen" und „Unterstützung von Dateisystemen in ASCII" im Handbuch „C-Bibliotheksfunktionen“ [2]).

Es wird nur EBCDIC-Ablaufcode erzeugt. Z.B. werden die Namen externer Entries nur im EBCDIC-Code abgelegt.

Beschreibung des Begriffes <posix-pathname>

Bei der Angabe von <posix-pathname> als Eingabe oder Ausgabe kann folgendes angegeben werden:

- ein Dateiname ohne Pfad-Angabe
- ein relativer Pfad zu einer Datei
- ein absoluter Pfad zu einer Datei
- ein relativer Pfad zu einem Dateiverzeichnis
- ein absoluter Pfad zu einem Dateiverzeichnis

Ist kein absoluter Pfad vorhanden, so wird der Name/Pfad relativ zum Home-Dateiverzeichnis des Benutzers interpretiert.

Alle Dateiverzeichnisse im Pfad müssen bereits existieren.

Wird eine Eingabe-Datei verlangt, so darf kein Dateiverzeichnis angegeben werden. Wird ein Eingabe-Dateiverzeichnis verlangt, so darf keine Datei angegeben werden.

Die Angabe einer Ausgabe-Datei ist nur sinnvoll, wenn eine einzelne Source bearbeitet wird. Werden mehrere Sourcen bearbeitet, ist die Angabe einer Ausgabe-Datei in der Regel unzulässig.
Bei der Bildung der Dateinamen ist zu beachten, dass die Ausgabe-Datei im POSIX-Subsystem nur sinnvoll weiterverarbeitet werden kann, wenn der Name einen passenden Suffix enthält.
Wird ein Dateiverzeichnis angegeben, so bestimmt der Compiler selbst den Namen der Ausgabe-Datei (siehe Abschnitt „Standardnamen für Ausgabebehälter“).

2.6.2 Benutzen der POSIX-Bibliotheksfunktionen

Mit CRTE wird ein C-Laufzeitsystem zur Verfügung gestellt, das C-Bibliotheksfunktionen mit BS2000-Funktionalität und mit POSIX-Funktionalität unterstützt.

Bibliotheksfunktionen mit BS2000-Funktionalität sind alle ANSI-definierten Funktionen sowie ca. 50 BS2000-spezifische Erweiterungen.
In folgenden Fällen können ausschließlich nur diese Funktionen genutzt werden:

- wenn kein POSIX-Subsystem verfügbar ist oder
- wenn in einem BS2000-Betriebssystem mit verfügbarem POSIX-Subsystem beim Übersetzen und Binden keine besonderen Vorkehrungen getroffen werden (siehe unten)

Die Bibliotheksfunktionen mit BS2000-Funktionalität sind in dem Handbuch „C-Bibliotheksfunktionen“ [2] beschrieben.

Bibliotheksfunktionen mit POSIX-Funktionalität sind folgende Funktionen des C-Laufzeitsystems: alle vom XPG4-Standard (spec1170) geforderten Funktionen sowie ca. 30 UNIX-spezifische Erweiterungen. Diese Funktionen und zusätzlich alle Funktionen mit BS2000-Funktionalität sind im Handbuch „C-Bibliotheksfunktionen für POSIX-Anwendungen“ [3] beschrieben.

Übersetzen und Binden von Programmen, die POSIX-Bibliotheksfunktionen benutzen

Um bei der Programmentwicklung in BS2000-Umgebung (SDF) die POSIX-Bibliotheksfunktionen nutzen zu können, sind folgende Schritte notwendig:

1. Für die Suche nach den Standard-Includes muss beim Übersetzen zusätzlich zur CRTE-Bibliothek SYSLIB.CRTE die Bibliothek SYSLIB.POSIX-HEADER angegeben werden, die die Standard-Include-Elemente für die POSIX-Funktionen enthält.

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

2. Bevor der Präprozessor auf die erste `#include`-Anweisung im Programm trifft, muss unbedingt das Define `_OSD_POSIX` gesetzt sein. Dies ist sichergestellt, wenn die Definition nicht im Quellprogramm mit der `#define`-Anweisung erfolgt, sondern global für den gesamten Übersetzungslauf mit der MODIFY-SOURCE-PROPERTIES-Anweisung.

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

3. Beim Binden muss die Bindeschalter-Bibliothek SYSLNK.CRTE.POSIX vorrangig vor der Bibliothek SYSLNK.CRTE bzw. SYSLNK.CRTE.PARTIAL-BIND bzw. SYSLNK.CRTE.COMPL eingebunden werden. Wir empfehlen, beim Binden mit dem BINDER, die Bindeschalter-Bibliothek mit einer INCLUDE-Anweisung (ohne Angabe der Modulnamen) einzubinden, da bei der Verwendung von RESOLVE-Anweisungen sonst strikt die Reihenfolge beachtet werden müsste, also z.B. mit der INCLUDE-Anweisung

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

   Analog sollte beim Binden mit der BIND-Anweisung des Compilers in der MODIFY-BIND-PROPERTIES-Anweisung die INCLUDE-Option verwendet werden:

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

Bei der Programmentwicklung in POSIX-Umgebung sind dagegen keine besonderen Vorkehrungen zu treffen, um die POSIX-Bibliotheksfunktionen nutzen zu können (siehe Handbuch „POSIX-Kommandos des C/C++-Compilers“ [1]).

2.7 Einführungsbeispiele

Dieser Abschnitt zeigt an drei einfachen Beispielen, wie C/C++-Programme im BS2000 übersetzt, gebunden und gestartet werden. Die Beispiele gehen davon aus, dass der C/C++-Compiler und die C- und C++-Laufzeitsysteme standardmäßig auf der TSOS-Kennung installiert sind.

- Beispiel 1: Übersetzen, Binden und Starten eines C-Programms
- Beispiel 2: Übersetzen, Binden und Starten eines C++-Programms
- Beispiel 3: Übersetzen eines C-Quellprogramms, das in einer POSIX- Datei steht und POSIX-Bibliotheksfunktionen benutzt

2.7.1 Beispiel 1: Übersetzen, Binden und Starten eines C-Programms

Das C-Quellprogramm steht in der katalogisierten Datei HALLO. Mit dem Compiler wird ein LLM erzeugt und in die Bibliothek PLAM.TEST geschrieben. Anschließend wird das Modul auf unterschiedliche Arten gebunden:
Variante 1: mit der BIND-Anweisung des Compilers
Variante 2: mit dem BINDER
Variante 3: mit dem DBL

Quellprogramm-Datei HALLO

```
#include <stdio.h>
int main(void)
{
  printf("Hallo, ich bin ein C-Programm\n");
  return 0;
}
```

Das Quellprogramm wurde mit dem EDT erstellt und in eine Datei namens HALLO abgespeichert.

Ablaufprotokoll zum Übersetzen, Binden und Starten

```
(IN)   bedeutet Benutzereingaben
(OUT)  bedeutet System/Programm-Meldungen
(IN)   /START-CPLUS-COMPILER ──────────────────────────────────────  (1)
(OUT)  %  BLS0523 ELEMENT 'SDFCC', VERSION '04.0A10', TYPE 'L' FROM LIBRARY
          ':P401:$TSOS.SYSLNK.CPP.040' IN PROCESS
(OUT)  %  BLS0524 LLM 'SDFCC', VERSION '04.0A10' OF '2020-04-02 13:40:11'
          LOADED
(OUT)  %  BLS0551 COPYRIGHT (C) 2020 Fujitsu Technology Solutions GmbH. ALL
          RIGHTS RESERVED
(OUT)  %  CDR9992 : BEGIN C/C++ VERSION 04.0A10
(IN)   //MODIFY-SOURCE-PROP LANGUAGE=*C ──────────────────────────  (2)
(IN)   //COMPILE SOURCE=HALLO,MODULE-OUTPUT=*LIB-ELEM(LIB=PLAM.TEST) ──  (3)
(OUT)  %  CDR9907 : NOTES: 0  WARNINGS: 0  ERRORS: 0  FATALS: 0
(OUT)  %  CDR9937 : MODULES GENERATED, CPU TIME USED = 0.0030 SEC
```

Variante 1: Binden mit der BIND-Anweisung

```
(IN)   //MOD-BIND-PROP INCLUDE=*LIB(LIB=PLAM.TEST,ELEM=HALLO),-
       //RUNTIME-LANG=*C,STDLIB=*STATIC ──────────────────────────  (4)
(IN)   //BIND OUTPUT=*LIB(LIB=PLAM.TEST1,ELEM=HALLO) ──────────────  (5)
(OUT)  %  BND3102 SOME WEAK EXTERNS UNRESOLVED
(OUT)  %  BND1501 LLM FORMAT: '1'
(OUT)  %  BND1101 BINDER NORMALLY TERMINATED. SEVERITY CLASS:
          'UNRESOLVED EXTERNAL'
(IN)  //END ──────────────────────────────────────────────────────  (6)
(OUT)  %  CDR9936 END; SUMMARY: NOTES: 0  WARNINGS: 0  ERRORS: 0  FATALS: 0
(OUT)  %  CCM0998 CPU TIME USED: 1.9967 SECONDS
```

Variante 2: Binden mit dem BINDER

```
(IN)   /START-BINDER ──────────────────────────────────────────────────  (7)
(OUT)  %  BND0500 BINDER VERSION 'V02.7A12' STARTED
(IN)   //START-LLM-CREATION INT-NAME=XY
(IN)   //INCLUDE-MODULES *LIB(LIB=PLAM.TEST,ELEM=HALLO)
(IN)   //RESOLVE-BY-AUTOLINK LIB=$.SYSLNK.CRTE
(IN)   //SAVE-LLM LIB=PLAM.TEST1,ELEM=HALLO,MAP=*NO
(OUT)  %  BND3102 SOME WEAK EXTERNS UNRESOLVED
(OUT)  %  BND1501 LLM FORMAT: '1'
(IN)   //END
(OUT)  %  BND1101 BINDER NORMALLY TERMINATED. SEVERITY CLASS:
          'UNRESOLVED EXTERNAL'
```

Variante 3: Binden, Laden und Starten mit dem DBL

```
(IN)   /ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME=$.SYSLNK.CRTE ──────  (8)
(IN)   /START-EXECUTABLE-PROGRAM FROM-FILE=*LIB-ELEM(LIBRARY=PLAM.TEST,-
       /ELEMENT-OR-SYMBOL=HALLO),DBL-PARAMETERS=*PARAM(-
       /RESOLUTION=*PARAM(ALTERNATE-LIBRARIES=*BLSLIB##))
(OUT)  %  BLS0524 LLM 'HALLO', VERSION ' ' OF '2020-04-02 13:33:56' LOADED
(OUT)  Hallo, ich bin ein C-Programm
(OUT)  %  CCM0998 CPU TIME USED: 0.0021 SECONDS
```

Starten des nach Variante 1 oder 2 gebundenen Programms

```
(IN)   /START-EXECUTABLE-PROGRAM FROM-FILE=*LIB-ELEM(LIBRARY=PLAM.TEST1,-
       /ELEMENT-OR-SYMBOL=HALLO) ──────────────────────────────────────  (9)
(OUT)  %  BLS0524 LLM 'XY', VERSION ' ' OF '2020-04-02 13:33:50' LOADED
(OUT)  Hallo, ich bin ein C-Programm
(OUT)  %  CCM0998 CPU TIME USED: 0.0028 SECONDS
```

(1) Der Compilerlauf wird gestartet.

(2) Mit der MODIFY-SOURCE-PROPERTIES-Anweisung wird der Sprachmodus C eingeschaltet (voreingestellt ist der Sprachmodus C++).

(3) Mit der COMPILE-Anweisung wird der Übersetzungslauf gestartet. Mit der SOURCE-Option wird der Name des zu übersetzenden Quellprogramms angegeben. Mit der MODULE-OUTPUT-Option wird als Ausgabeziel eine PLAM-Bibliothek angegeben. Der Name des zu erzeugenden Moduls wird aus dem Namen des Quellprogramms abgeleitet (HALLO).

(4) Mit der MODIFY-BIND-PROPERTIES-Anweisung werden die einzubindenden Module und weitere Bedingungen für den anschließenden Bindelauf mit der BIND-Anweisung des Compilers festgelegt. Mit der INCLUDE-Option (entspricht der BINDER-Anweisung INCLUDE-MODULES) wird das zuvor erzeugte LLM HALLO in der Bibliothek PLAM.TEST angegeben. Mit der RUNTIME-LANGUAGE-Option wird angegeben, dass es sich bei dem zu bindenden Programm um ein C-Programm handelt (C++ ist voreingestellt). Auf Grund dieser Option werden die für C-Programme zusätzlich benötigten Module des C-Laufzeitsystems automatisch (per Autolink) eingebunden. Die Angabe *STATIC in der STDLIB-Option bewirkt, dass das C-Laufzeitsystem nicht aus der Bibliothek $.SYSLNK.CRTE.PARTIAL-BIND (Voreinstellung), sondern aus der Bibliothek $.SYSLNK.CRTE eingebunden wird.

(5) Mit der BIND-Anweisung wird der Bindelauf gestartet. Mit der OUTPUT-Option (entspricht der BINDER-Anweisung SAVE-LLM) wird das fertig gebundene LLM unter dem Namen HALLO als Element vom Typ L in der PLAM-Bibliothek PLAM.TEST1 abgespeichert. Die BINDER-Meldung „SOME WEAK EXTERNS UNRESOLVED“ bezieht sich auf das ILCS-Modul IT0INITS. Dieses Modul enthält WEAK-EXTERN-Verweise auf alle potenziell für ILCS vorgesehenen Sprachen. Im Beispiel ist nur die Sprache C beteiligt, die anderen Verweise bleiben offen.

(6) Mit der END-Anweisung wird der Compilerlauf beendet.

(7) Das bei der Übersetzung erzeugte Modul HALLO in der PLAM-Bibliothek PLAM.TEST wird mit dem BINDER gebunden.

(8) Das bei der Übersetzung erzeugte Modul HALLO in der PLAM-Bibliothek PLAM.TEST wird mit dem DBL dynamisch gebunden, geladen und gestartet.

(9) Das mit der BIND-Anweisung des Compilers (siehe Variante 1) bzw. mit dem BINDER (siehe Variante 2) fertig gebundene Programm HALLO in der Bibliothek PLAM.TEST1 wird geladen und gestartet. Die beim dynamischen Binden mit dem DBL notwendige Angabe ALT-LIB=*BLSLIB## (siehe Variante 3) ist in diesem Fall nicht notwendig.

2.7.2 Beispiel 2: Übersetzen, Binden und Starten eines C++-Programms

Das C++-Quellprogramm besteht aus zwei PLAM-Bibliothekselementen PROG1 und PROG2. Vom Compiler werden LLMs erzeugt und in die PLAM-Bibliothek SYS.PROG.LIB geschrieben. Die Module werden anschließend mit der BIND-Anweisung des Compilers gebunden.

Quellprogramm-Element PROG1

```
#include <iostream.h>
extern void ruf(void);
int main(void)
{
   cout << "main(prog1)" << '\n';
   ruf();
   return 0;
}
```

Quellprogramm-Element PROG2

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

Die oben gezeigten Quellprogrammteile wurden mit dem EDT erstellt und jeweils mit einer WRITE-Anweisung in die PLAM-Bibliothek PLAM.BSP abgespeichert:
WRITE L=PLAM.BSP(E=elemname).

Ablaufprotokoll zum Übersetzen, Binden und Starten

```
(IN)   bedeutet Benutzereingaben
(OUT)  bedeutet System/Programm-Meldungen
(IN)   /START-CPLUS-COMPILER ──────────────────────────────────────  (1)
(OUT)  %  BLS0523 ELEMENT 'SDFCC', VERSION '04.0A10', TYPE 'L' FROM LIBRARY
          ':P401:$TSOS.SYSLNK.CPP.040' IN PROCESS
(OUT)  %  BLS0524 LLM 'SDFCC', VERSION '04.0A10' OF '2020-04-02 13:40:11'
          LOADED
(OUT)  %  BLS0551 COPYRIGHT (C) 2020 Fujitsu Technology Solutions GmbH. ALL
          RIGHTS RESERVED
(OUT)  %  CDR9992 : BEGIN C/C++ VERSION 04.0A10
(IN)   //MOD-SOURCE-PROP LANGUAGE=*CPLUS(V3-COMPATIBLE) ──────────────  (2)
(IN)   //MOD-BIND-PROP RUNTIME-LANGUAGE=*CPLUS(*V3-COMPATIBLE) ──────  (2)
(IN)   //COMPILE SOURCE=(*LIB(LIB=PLAM.BSP,ELEM=PROG1),-
       //*LIB(LIB=PLAM.BSP,ELEM=PROG2)) ──────────────────────────────  (3)
(OUT)  %  CDR9907 : NOTES: 0  WARNINGS: 0  ERRORS: 0  FATALS: 0
(OUT)  %  CDR9937 : MODULES GENERATED, CPU TIME USED = 0.0030 SEC
(OUT)  %  CDR9907 : NOTES: 0  WARNINGS: 0  ERRORS: 0  FATALS: 0
(OUT)  %  CDR9937 : MODULES GENERATED, CPU TIME USED = 0.0020 SEC
(IN)   //MOD-BIND-PROP INCLUDE=(*LIB(LIB=SYS.PROG.LIB,ELEM=PROG1),-
       //*LIB(LIB=SYS.PROG.LIB,ELEM=PROG2)) ──────────────────────────  (4)
(IN)   //BIND OUTPUT=*LIB(LIB=PLAM.BSP1,ELEM=PROG) ──────────────────  (5)
(OUT)  %  BND1501 LLM FORMAT: '4'
(OUT)  %  BND3102 SOME WEAK EXTERNS UNRESOLVED
(OUT)  %  BND1101 BINDER NORMALLY TERMINATED. SEVERITY CLASS:
          'UNRESOLVED EXTERNAL'
(IN)   //END ──────────────────────────────────────────────────────────  (6)
(OUT)  %  CDR9936 END; SUMMARY: NOTES: 0  WARNINGS: 0  ERRORS: 0  FATALS: 0
(OUT)  %  CCM0998 CPU TIME USED: 1.9967 SECONDS
(IN)   /START-EXECUTABLE-PROG *L(LIB=PLAM.BSP1,E-O-S=PROG) ──────────  (7)
(OUT)  %  BLS0523 ELEMENT 'PROG', VERSION '@', TYPE 'L' FROM LIBRARY
          ':2OS2:$USERIDXY.PLAM.BSP1' IN PROCESS
(OUT)  %  BLS0524 LLM '$LIB-ELEM$PLAM$BSP1$$PROG$$$UPPE', VERSION ' '
          OF '2020-04-02 14:22:01' LOADED
(OUT)  main(prog1)
(OUT)  ruf(prog2)
(OUT)  %  CCM0998 CPU TIME USED: 0.0018 SECONDS
```

(1) Der Compilerlauf wird gestartet.

(2) Der Sprachmodus wird festgelegt, sowohl für die Übersetzung als auch für das Binden.

(3) Mit der COMPILE-Anweisung werden die Übersetzungsläufe für die C++-Quellprogramm-Elemente PROG1 und PROG2 gestartet. In der SOURCE-Option werden in einer Liste jeweils der Name der PLAM-Bibliothek und des zu übersetzenden Bibliothekselements angegeben. Der Compiler schreibt standardmäßig die erzeugten LLMs in die Bibliothek SYS.PROG.LIB. Die Namen der LLMs werden aus den Namen der Quellprogramm-Elemente gebildet (PROG1, PROG2).

(4) Mit der MODIFY-BIND-PROPERTIES-Anweisung werden die Module angegeben, die im nachfolgenden Bindelauf eingebunden werden sollen. Mit der INCLUDE-Option (entspricht der BINDER-Anweisung INCLUDE-MODULES) wird der Name des LLM angegeben, das die `main`-Funktion enthält (PROG1) sowie der Name des zusätzlich einzubindenden LLM mit dem Unterprogramm (PROG2). Die für C++-V3-Programme zusätzlich benötigten Module der C- und C++-Laufzeitbibliotheken werden automatisch (per Autolink) eingebunden (vgl. STDLIB-Option, "MODIFY-BIND-PROPERTIES").

(5) Mit der BIND-Anweisung wird der Bindelauf gestartet. Mit der OUTPUT-Option (entspricht der BINDER-Anweisung SAVE-LLM) wird das erzeugte LLM unter dem Namen PROG als Element vom Typ L in einer PLAM-Bibliothek abgespeichert. Die BINDER-Meldung „SOME WEAK EXTERNS UNRESOLVED“ bezieht sich auf das ILCS-Modul IT0INITS. Dieses Modul enthält WEAK-EXTERN-Verweise auf alle potenziell für ILCS vorgesehenen Sprachen. Im Beispiel sind nur die Sprachen C und C++ beteiligt, die anderen Verweise bleiben offen. Zu dem vom BINDER generierten LLM-Format (in diesem Beispiel Format 4) beachten Sie bitte die OUTPUT-FORMAT-Option in der BIND-Anweisung ("BIND").

(6) Mit der END-Anweisung wird der Compilerlauf beendet.

(7) Mit dem START-EXECUTABLE-PROGRAM-Kommando wird das fertig gebundene Programm geladen und gestartet.

2.7.3 Beispiel 3: Übersetzen eines C-Quellprogramms, das in einer POSIX- Datei steht und POSIX-Bibliotheksfunktionen benutzt

Das C-Quellprogramm steht als POSIX-Quelldatei mit dem Namen `hallo.c` in dem Dateiverzeichnis `/USERIDXY/source`. Das Programm benutzt POSIX-Bibliotheksfunktionen. Der Compiler erzeugt ein LLM und schreibt dieses in eine POSIX-Objektdatei mit dem Standardnamen `hallo.o`. Diese Objektdatei wird im Dateiverzeichnis des Quellprogramms abgelegt. Die Objektdatei wird anschließend in der POSIX-Shell-Umgebung weiterverarbeitet.

Quellprogramm-Datei hallo.c

```
#include <stdio.h>
FILE *fp;
int main(void)
{
  printf("Hallo, ich bin ein C-Programm\n");
  fp = fopen("/USERIDXY/posixfiles/hallo", "w");
  fputs("hallo", fp);
  fclose(fp);
  return 0;
}
```

Das Quellprogramm wurde mit dem EDT erstellt und mit der Anweisung @XWRITE FILE=/USERIDXY/source/hallo.c abgespeichert.

Ablaufprotokoll zum Übersetzen, Binden und Starten

```
(IN)   bedeutet Benutzereingaben
(OUT)  bedeutet System/Programm-Meldungen
(IN)   /START-CPLUS-COMPILER ──────────────────────────────────────  (1)
(OUT)  %  BLS0523 ELEMENT 'SDFCC', VERSION '04.0A10', TYPE 'L' FROM LIBRARY
         ':P401:$TSOS.SYSLNK.CPP.040' IN PROCESS
(OUT)  %  BLS0524 LLM 'SDFCC', VERSION '04.0A10' OF '2020-04-02 13:40:11'
          LOADED
(OUT)  %  BLS0551 COPYRIGHT (C) 2020 Fujitsu Technology Solutions GmbH. ALL
          RIGHTS RESERVED
(OUT)  %  CDR9992 : BEGIN C/C++ VERSION 04.0A10
(IN)   //MOD-SOURCE-PROP LANG=*C,DEFINE=_OSD_POSIX ──────────────  (2)
(IN)   //MOD-INCL-LIB STD-INCL=($.SYSLIB.POSIX-HEADER,*STANDARD-LIB) ──  (3)
(IN)   //COMPILE SOURCE='/USERIDXY/source/hallo.c',-
       //MODULE-OUTPUT=*SOURCE-LOC ──────────────────────────────  (4)
(OUT)  %  CDR9907 : NOTES: 0  WARNINGS: 0  ERRORS: 0  FATALS: 0
(OUT)  %  CDR9937 : MODULES GENERATED, CPU TIME USED: 0.3829 SECONDS
(IN)   //END ──────────────────────────────────────────────────────  (5)
(OUT)  %  CDR9936 END; SUMMARY: NOTES: 0  WARNINGS: 0  ERRORS: 0  FATALS: 0
(OUT)  %  CCM0998 CPU TIME USED: 0.3829 SECONDS
(IN)   /START-POSIX-SHELL ─────────────────────────────────────────  (6)
(OUT)  POSIX Basisshell 10.0A45 created Jun 14 2016
       POSIX Shell 10.0A45 created Jul 12 2016
       Copyright (C) Fujitsu Technology Solutions 2009
                     All Rights reserved
       Last login: Thu Mar 26 10:45:44 2020 on term/002
(IN)   cd source ──────────────────────────────────────────────────  (7)
(IN)   ls hallo* ──────────────────────────────────────────────────  (8)
(OUT)  hallo.c    hallo.o
(IN)   cc hallo.o ─────────────────────────────────────────────────  (9)
(IN)   a.out ──────────────────────────────────────────────────── (10)
(OUT)  Hallo, ich bin ein C-Programm
```

(1) Der Compilerlauf wird gestartet.

(2) Mit der MODIFY-SOURCE-PROPERTIES-Anweisung wird der Sprachmodus C eingeschaltet (voreingestellt ist der Sprachmodus C++) und das für die Nutzung der POSIX-Bibliotheksfunktionen notwendige Define `_OSD_POSIX` gesetzt.

(3) Mit der MODIFY-INCLUDE-LIBRARIES-Anweisung wird zusätzlich zur CRTE-Bibliothek $.SYSLIB.CRTE (*STANDARD-LIBRARY) die Bibliothek zugewiesen, die die Standard-Include-Elemente für die POSIX-Bibliotheksfunktionen enthält ($.SYSLIB.POSIX-HEADER).

(4) Mit der COMPILE-Anweisung wird der Übersetzungslauf gestartet. Mit der SOURCE-Option wird der absolute Pfadname der zu übersetzenden POSIX-Quelldatei angegeben. POSIX-Dateinamen müssen immer in Hochkommas eingeschlossen werden. Der Operandenwert *SOURCE-LOCATION in der MODULE-OUTPUT-Option bewirkt, dass das übersetzte Modul in eine POSIX-Objektdatei mit dem Standardnamen `hallo.o` geschrieben und im Dateiverzeichnis des Quellprogramms abgelegt wird.

(5) Mit der END-Anweisung wird der Compilerlauf beendet.

(6) Da POSIX-Objektdateien nur im POSIX-Subsystem weiterverarbeitet werden können, wird mit dem POSIX-Kommando START-POSIX-SHELL aus der BS2000-Systemumgebung (SDF) in die POSIX-Umgebung (Shell) gewechselt. Nach Aufruf des Kommandos befindet man sich im Home-Dateiverzeichnis der aktuellen BS2000-Benutzerkennung (USERIDXY).

(7) Mit dem POSIX-Kommando `cd` wird in das Dateiverzeichnis `source` gewechselt, in dem sich das Quellprogramm und die vom Compiler erzeugte Objektdatei befindet.

(8) Nach Eingabe des POSIX-Kommandos `ls` werden die Quelldatei `hallo.c` und die Objektdatei `hallo.o` aufgelistet.

(9) Mit dem POSIX-Kommando `cc` wird die Objektdatei `hallo.o` zu einer ablauffähigen Einheit gebunden und in eine ausführbare POSIX-Datei mit dem Standardnamen `a.out` geschrieben. Das `cc`-Kommando ist ausführlich im Handbuch „POSIX-Kommandos des C/C++-Compilers“ [1] beschrieben.

(10) Das Programm wird ausgeführt.

3 Übersetzen

In diesem Kapitel werden folgende Themen behandelt:

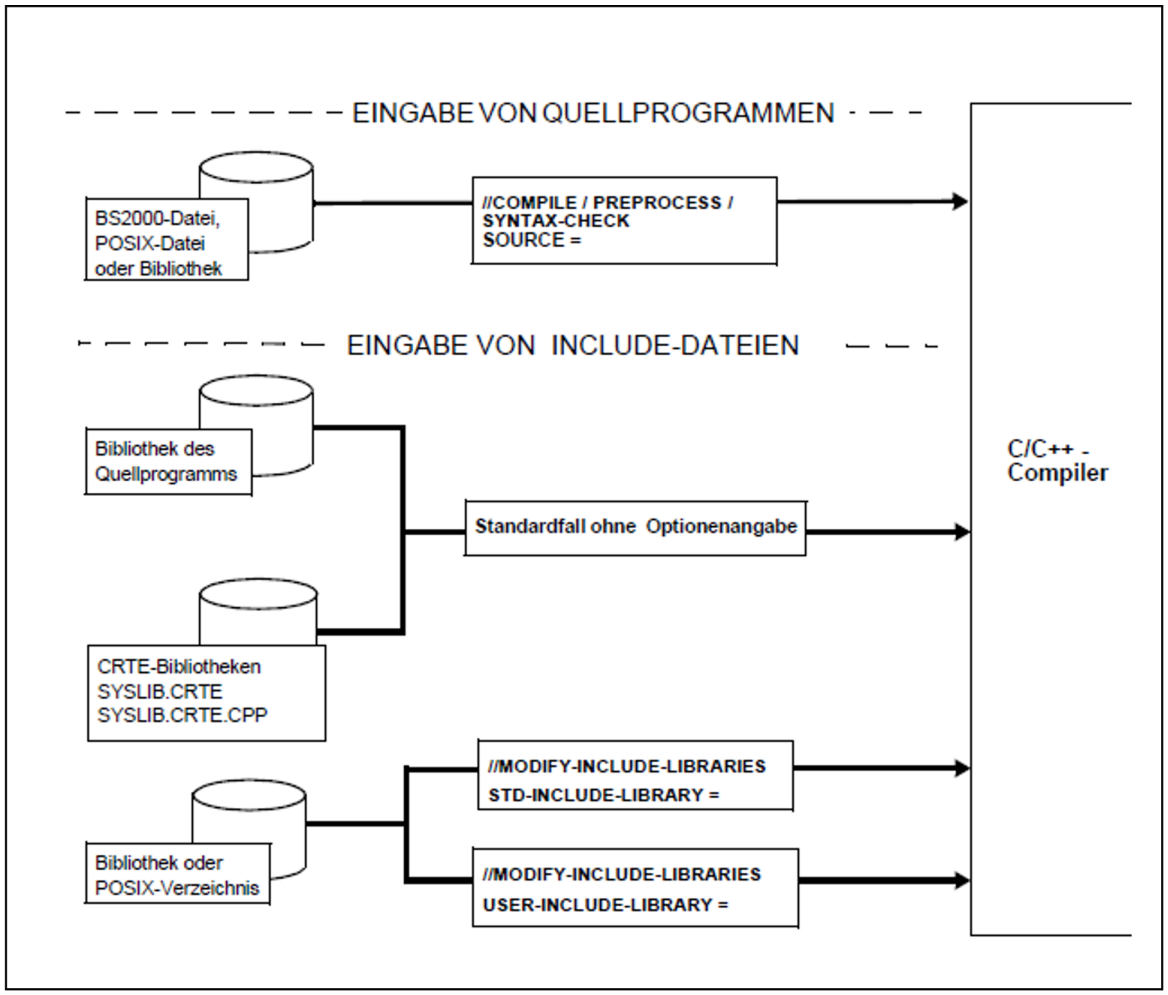
3.1 Allgemeine Aspekte des Compilerlaufs

In diesem Abschnitt werden folgende Themen behandelt:

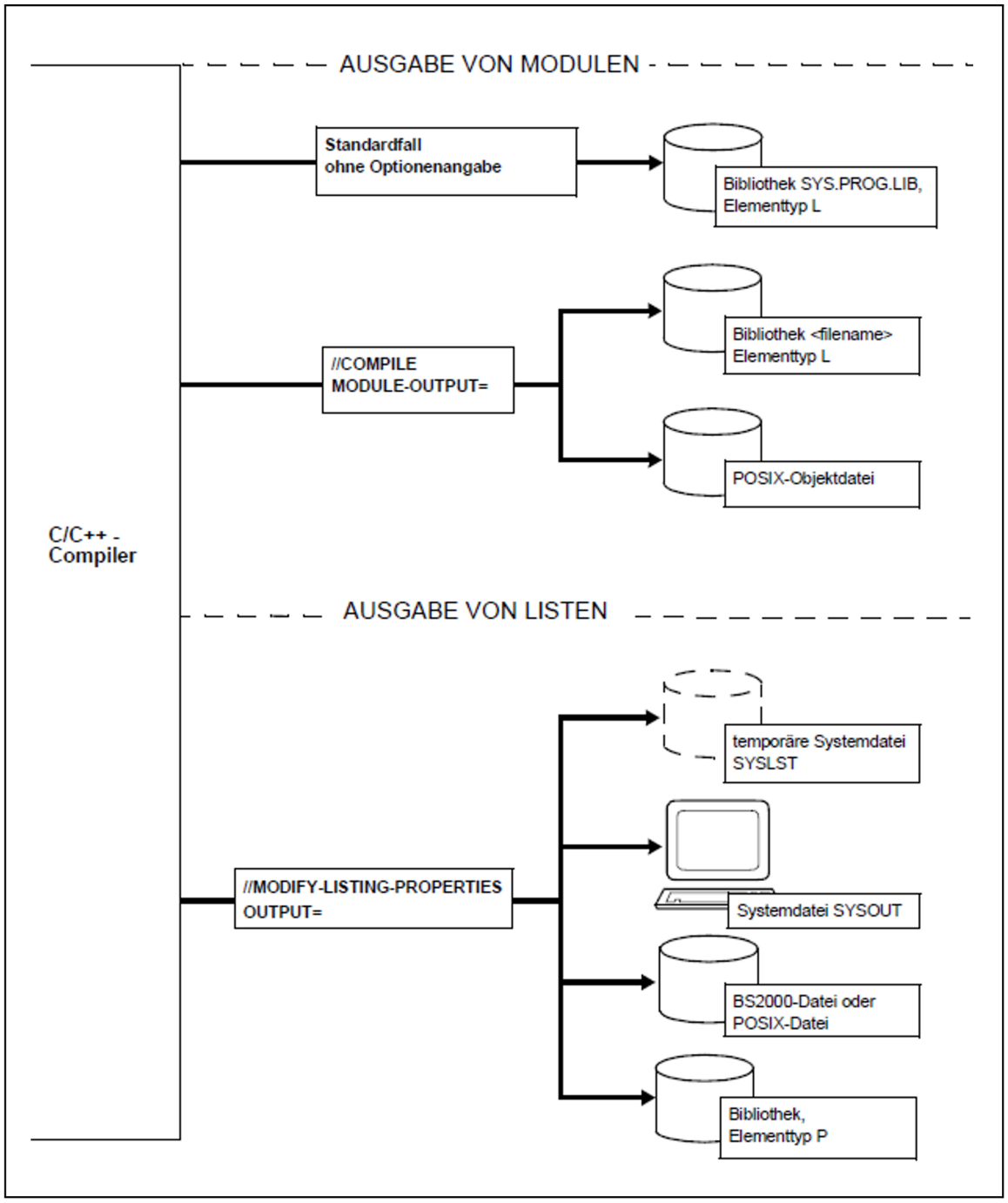
- Eingabequellen und Ausgabeziele des Compilers
- Standardnamensgenerierung
- Standardnamen für Ausgabebehälter
- Bildung von Modulnamen
- Aufbau der Compilermeldungen

3.1.1 Eingabequellen und Ausgabeziele des Compilers

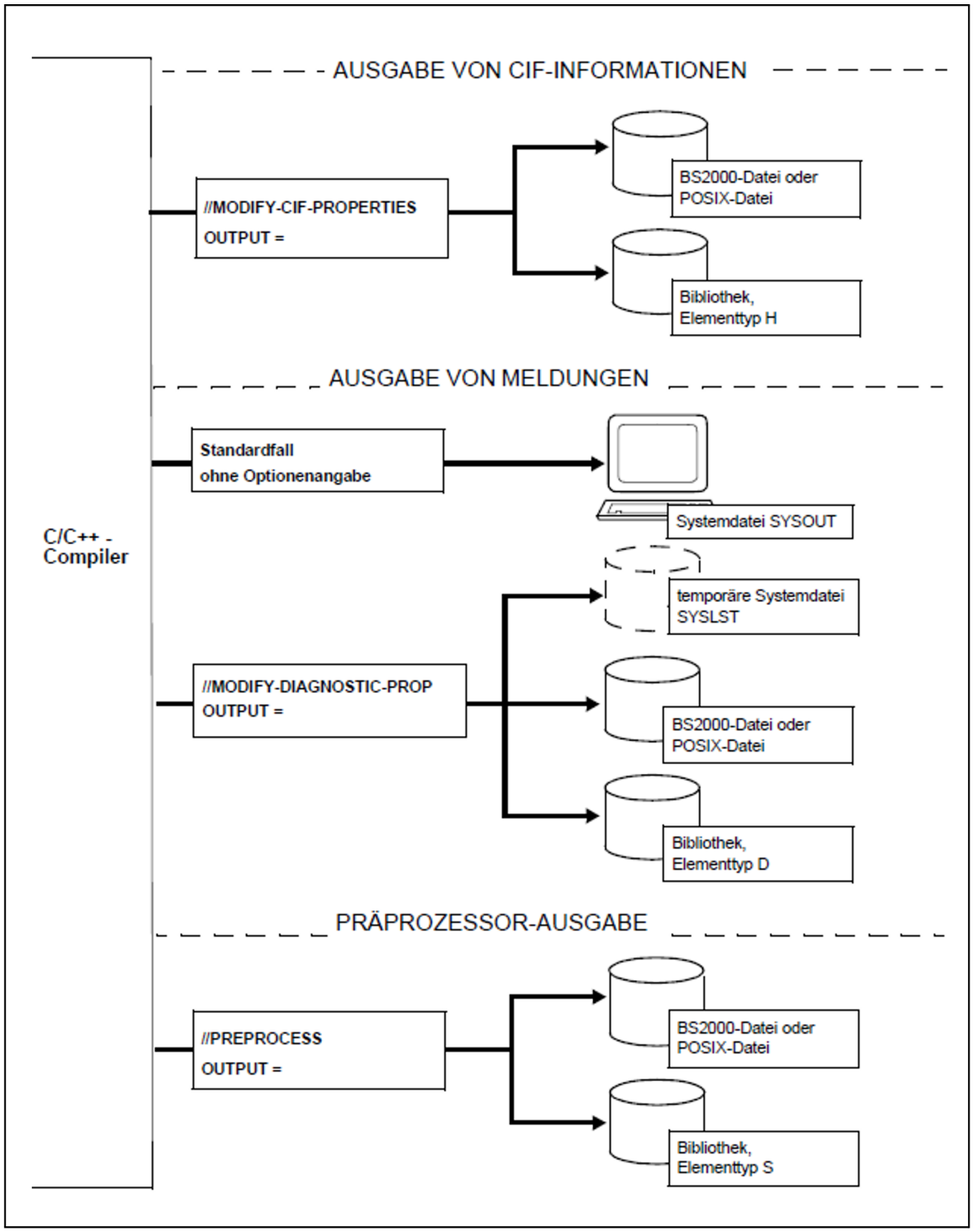
Eingabe von Quellprogrammen und Include-Dateien



Ausgabe von Modulen und Listen



Ausgabe von CIF-Informationen, Meldungen und Präprozessor



3.1.2 Standardnamensgenerierung

Wenn für die Ausgabe der Übersetzungsergebnisse die Namen der Ausgabedateien nicht explizit vereinbart werden, generiert der Compiler Standardnamen, die jeweils aus den Quellprogrammnamen abgeleitet werden.

Im Abschnitt „Standardnamen für Ausgabebehälter“ sind die Namensbildungsregeln für die Ausgabebehälter folgender Übersetzungsergebnisse zusammengefasst: Präprozessorausgabe, Meldungen, Listen und CIF-Informationen.

Die Regeln zur Modulnamensgenerierung sind im Abschnitt „Bildung von Modulnamen“ zusammengefasst.

3.1.3 Standardnamen für Ausgabebehälter

Dieser Abschnitt fasst alle Regeln zusammen, nach denen der Compiler die Standardnamen für die Ausgabebehälter folgender Übersetzungsergebnisse bildet:

- Ergebnis eines Präprozessorlaufs (PREPROCESS)
- Übersetzungslisten (MODIFY-LISTING-PROPERTIES)
- Diagnosemeldungen (MODIFY-DIAGNOSTIC-PROPERTIES)
- CIF-Informationen (MODIFY-CIF-PROPERTIES)

Standardnamen werden dann generiert, d.h. aus dem Quellprogrammnamen abgeleitet, wenn in den OUTPUT-Optionen der o.g. Anweisungen folgende Angaben gemacht werden:

OUTPUT=*STD-FILE
OUTPUT=*SOURCE-LOCATION
OUTPUT=*LIB-ELEM(LIB=...,ELEM=*STD-ELEMENT)
OUTPUT=<posix-pathname> (Name eines POSIX-Dateiverzeichnisses)

Bildung der Standardnamen von katalogisierten BS2000-Dateien

Die Ausgabe in katalogisierte BS2000-Dateien mit Standardnamen erfolgt

- generell bei der Angabe OUTPUT=*STD-FILE,
- bei der Angabe OUTPUT=*SOURCE-LOCATION, wenn das Quellprogramm aus einer BS2000-Datei oder über SYSDTA eingelesen wird.

1. Folgende ggf. vorhandene Namensteile im Quellprogrammnamen werden für die Bildung des Standard-Dateinamens nicht herangezogen und entfernt:
   - Namensteile für die Catid und Userid in BS2000-Datei- oder Bibliotheksnamen. Ausnahme: Liegt das Quellprogramm bei Angabe von OUTPUT=*SOURCE-LOCATION als katalogisierte BS2000-Datei vor, dann werden Catid und Userid beibehalten.
   - Dateiverzeichnisnamen in POSIX-Pfadnamen
   - die Suffixe `.C` , `.CPP`, `.CXX`, `.CC` und `.I`, wenn das Quellprogramm als katalogisierte BS2000-Datei oder als PLAM-Bibliothekselement vorliegt
   - die Suffixe `.c`, `.C` , `.cpp`, `.CPP`, `.cxx`, `.CXX`, `.cc`, `.CC`, `.c++`, `.C++`, `.i` und `.I`, wenn das Quellprogramm als POSIX-Datei vorliegt
2. Liegt das Quellprogramm als katalogisierte BS2000-Datei oder als POSIX-Datei vor, wird der restliche Name ggf. auf 33 Zeichen von rechts verkürzt.
3. Nur bei OUTPUT=*STD-FILE:

   Liegt das Quellprogramm als PLAM-Bibliothekselement vor, werden im Standardnamen der Bibliotheks- und Elementname verwendet und mit einem Bindestrich verbunden:

   bibliotheksname-elementname
   Ist dieser Name inklusive des Bindestrichs länger als 33 Zeichen, wird zunächst der Bibliotheksname auf 16 Zeichen von rechts verkürzt, und, falls notwendig, auch der Elementname auf 16 Zeichen.
4. Für Dateinamen unerlaubte Sonderzeichen werden generell in „$“ umgewandelt. Erlaubt sind die Sonderzeichen $, @, #, . (Punkt) und - (Bindestrich).
5. Kleinbuchstaben (in POSIX-Quelldateinamen) werden in Großbuchstaben umgewandelt.

6. Der auf 33 Zeichen verkürzte Name wird durch das entsprechende Suffix `.I` , `.LST` , `.CIF` oder `.DIAG` ergänzt.

Zusammenfassung

	Quellprogrammeingabe von			
Inhalt	***SYSDTA**	**BS2000-Datei**	**PLAM-Bibliothek**	**POSIX-Datei**
PREPROCESS	CSTDEXP.I	datei.I	bib-elem.I	datei.I
LISTING	CSTDLST.LST	datei.LST	bib-elem.LST	datei.LST
DIAGNOSTIC	CSTDDIAG.DIAG	datei.DIAG	bib-elem.DIAG	datei.DIAG
CIF	CSTDCIF.CIF	datei.CIF	bib-elem.CIF	datei.CIF

Standardnamen von katalogisierten BS2000-Dateien in Abhängigkeit von Eingabequelle und Inhalt

Bildung der Standardnamen von PLAM-Bibliothekselementen

Die Ausgabe in PLAM-Bibliothekselemente mit Standardnamen erfolgt

- bei der Angabe OUTPUT=*LIB-ELEM(ELEMENT=*STD-ELEMENT),
- bei der Angabe OUTPUT=*SOURCE-LOCATION, wenn das Quellprogramm als PLAM-Bibliothekselement vorliegt.

Bei der Angabe OUTPUT=*LIB-ELEM(LIB=*STD-LIBRARY) werden die Elemente in die PLAM-Bibliothek mit dem Standardnamen SYS.PROG.LIB geschrieben.

1. Folgende ggf. vorhandene Namensteile im Quellprogrammnamen werden für die Bildung des Standard-Elementnamens nicht herangezogen und entfernt:
   - Namensteile für die Catid und Userid in BS2000-Dateinamen
   - Dateiverzeichnisnamen in POSIX-Pfadnamen
   - die Suffixe `.C` , `.CPP`, `.CXX`, `.CC` und `.I`, wenn das Quellprogramm als katalogisierte BS2000-Datei oder als PLAM-Bibliothekselement vorliegt
   - die Suffixe `.c`, `.C` , `.cpp`, `.CPP`, `.cxx`, `.CXX`, `.cc`, `.CC`, `.c++`, `.C++`, `.i` und `.I`, wenn das Quellprogramm als POSIX-Datei vorliegt
2. Der restliche Quellprogrammname wird ggf. auf 59 Zeichen von rechts verkürzt.
3. Für Elementnamen unerlaubte Sonderzeichen werden generell in „$“ umgewandelt. Erlaubt sind die Sonderzeichen $, @, #, . (Punkt), - (Bindestrich) und _ (Unterstrich).
4. Kleinbuchstaben (in POSIX-Quelldateinamen) werden in Großbuchstaben umgewandelt.
5. Der auf 59 Zeichen verkürzte Name wird durch das entsprechende Suffix `.I` , `.LST` , `.CIF` oder `.DIAG` ergänzt.

Zusammenfassung

	Quellprogrammeingabe von			
Inhalt	***SYSDTA**	**BS2000-Datei**	**PLAM-Bibliothek**	**POSIX-Datei**

PREPROCESS	CSTDEXP.I, S	datei.I, S	elem.I, S	datei.I, S
LISTING	CSTDLST.LST, P	datei.LST, P	elem.LST, P	datei.LST, P
DIAGNOSTIC	CSTDDIAG.DIAG, D	datei.DIAG, D	elem.DIAG, D	datei.DIAG, D
CIF	CSTDCIF.CIF, H	datei.CIF, H	elem.CIF, H	datei.CIF, H

Standardnamen von PLAM-Bibliothekselementen in Abhängigkeit von Eingabequelle und Inhalt

Bildung der Standardnamen von POSIX-Dateien

Die Ausgabe in POSIX-Dateien mit Standardnamen erfolgt

- bei der Angabe OUTPUT=*SOURCE-LOCATION, wenn das Quellprogramm als POSIX-Datei vorliegt,
- bei der Angabe OUTPUT=<posix-pathname>, wenn <posix-pathname> den Namen eines POSIX-Dateiverzeichnisses (ohne Dateiname) bezeichnet.

1. In den Standardnamen nicht übernommen wird ein ggf. vorhandenes Suffix `.C` , `.CPP`, `.CXX`, `.CC` oder `.I` in BS2000-Quelldateinamen und `.c`, `.C` , `.cpp`, `.CPP`, `.cxx`, `.CXX`, `.cc`, `.CC`, `.c++`, `.C++`, `.i` oder `.I` in POSIX-Quelldateinamen.
2. Der Name wird durch das entsprechende Suffix `.i` (C-Übersetzung) , `.I` (C++-Übersetzung), `.lst`, `.cif` oder `.diag` ergänzt.
3. Kleinbuchstaben sowie beliebige Sonderzeichen werden unverändert übernommen. Eine Namensverkürzung findet nicht statt.

3.1.4 Bildung von Modulnamen

Bei der Modulnamensbildung muss man unterscheiden zwischen dem Elementnamen (Name des „Behälters“), unter dem das Modul als Bibliothekselement abgelegt wird, und dem internen Modul- und CSECT-Namen.

Wird in der COMPILE-Anweisung mit der MODULE-OUTPUT-Option der Name des Moduls nicht explizit angegeben, werden Elementnamen und interne Modul-/CSECT-Namen aus dem Namen des Quellprogramms abgeleitet.

Wird in der COMPILE-Anweisung mit der MODULE-OUTPUT-Option der Name des Moduls explizit angegeben, werden aus diesem Namen Elementnamen und interne Modul-/CSECT-Namen gebildet.

Bildung der Elementnamen von LLMs in PLAM-Bibliotheken

- Ableitung aus dem Quellprogrammnamen
  1. Folgende ggf. vorhandene Teile des Quellprogrammnamens werden für die Bildung des Elementnamens nicht herangezogen und entfernt: <cat-id> und <user-id> sowie die Suffixe `.C` , `.CPP`, `.CXX`, `.CC` oder `.I` (bei BS2000-Dateien) bzw. `.c`, `.C` , `.cpp`, `.CPP`, `.cxx`, `.CXX`, `.cc`, `.CC`, `.c++`, `.C++`, `.i` oder `.I` (bei POSIX-Dateien).
  2. Ist der übrige Quellprogrammname länger als 59 Zeichen, wird er auf 59 Zeichen von rechts verkürzt.
  3. Für Elementnamen unerlaubte Sonderzeichen werden generell in „$“ umgewandelt. Erlaubt sind die Sonderzeichen $, @, #, . (Punkt), - (Bindestrich) und _ (Unterstrich). Kleinbuchstaben in POSIX-Dateinamen werden in Großbuchstaben umgewandelt.
  4. Wird das Quellprogramm von SYSDTA gelesen, lautet der Elementname „CSTDMOD“. Dies gilt auch, wenn SYSDTA mit dem ASSIGN-SYSDTA-Kommando einer katalogisierten Datei oder einem Bibliothekselement zugewiesen wird.
- Ableitung aus dem explizit angegebenen Namen

  In diesem Fall wird der angegebene Name unverändert als Elementname genommen, d.h. ein ggf. vorhandenes Suffix `.C` , `.CPP`, `.CXX`, `.CC` oder `.I` wird nicht entfernt. Eine Namensverkürzung findet ebenfalls nicht statt. Der Name kann also bis zu 64 Zeichen lang sein.

Achtung:

Zur Weiterverarbeitung mit dem DBL können die Elementnamen von LLMs derzeit maximal 32 Zeichen lang sein. Dies gilt sowohl für das LLM, das im START-EXECUTABLE-PROGRAM-Kommando angegeben wird (Modul, das die `main`-Funktion enthält) als auch für LLMs, die dynamisch gebunden werden sollen.
LLMs, deren Elementnamen länger als 32 Zeichen lang sind, müssen daher mit dem BINDER gebunden werden.

Bildung der Namen von LLM-Objektdateien im POSIX-Dateisystem

- Ableitung aus dem Quellprogrammnamen

  Der Name der LLM-Objektdatei wird wie folgt aus dem Namen der POSIX-Quelldatei abgeleitet: Es werden lediglich ggf. vorhandene Suffixe `.c`, `.C` , `.cpp`, `.CPP`, `.cxx`, `.CXX`, `.cc`, `.CC`, `.c++`, `.C++`, `.i` oder `.I` aus dem Namen der POSIX-Quelldatei entfernt, und an den restlichen Namen wird das Suffix `.o` angehängt. Eine Namensverkürzung, die Umwandlung von diversen Sonderzeichen in das $-Zeichen sowie die Umwandlung von Klein- in Großbuchstaben findet nicht statt.
- Ableitung aus dem explizit angegebenen Namen

  Der angegebene Name wird unverändert übernommen. Es ist zu beachten, dass LLM-Objektdateien nur sinnvoll im POSIX-Subsystem weiterverarbeitet werden können, wenn sie das Suffix `.o` enthalten.

Bildung der internen Modul- und CSECT-Namen von LLMs

- Ableitung aus dem Quellprogrammnamen

  1. Folgende ggf. vorhandene Teile des Quellprogrammnamens werden für die Bildung der Modul-/CSECT-Namen nicht herangezogen und entfernt: <cat-id> und <user-id> sowie die Suffixe `.C` , .CPP, .CXX, .CC oder `.I` (bei BS2000-Dateien) bzw. `.c`, `.C` , .cpp, .CPP, .cxx, .CXX, .cc, .CC, .c++, .C++, .i oder `.I` (bei POSIX-Dateien).
  2. Ist der Rest des Quellprogrammnamens länger als 30 Zeichen, wird er auf 30 Zeichen von rechts verkürzt.
  3. Die für interne LLM-Namen unerlaubten Sonderzeichen werden generell in „$“ umgewandelt. Erlaubt sind die Sonderzeichen $, @, #, - (Bindestrich) und _ (Unterstrich). Kleinbuchstaben in POSIX-Dateinamen werden in Großbuchstaben umgewandelt.
  4. Wird das Quellprogramm von SYSDTA gelesen, wird für die Namensbildung „CSTDMOD“ zu Grunde gelegt. Dies gilt auch, wenn SYSDTA mit dem ASSIGN-SYSDTA-Kommando einer katalogisierten Datei oder einem Bibliothekselement zugewiesen wird.
  5. Aus den nach den Regeln 1. bis 4. abgeleiteten Kernnamen werden nun die endgültigen Modul- und CSECT-Namen durch Anhängen eines 2-stelligen Suffixes „&@“ bzw. „&#“ gebildet:

Modulname	<kernname>&@
Code-CSECT	<kernname>&@
Daten-CSECT	<kernname>&#

- Ableitung aus dem explizit angegebenen Namen

1. Der gesamte angegebene Name wird zur Bildung des Modul-/CSECT-Namens herangezogen, d.h. ein ggf. vorhandenes Suffix `.c`, `.C` , .cpp, .CPP, .cxx, .CXX, .cc, .CC, .c++, .C++, .i oder `.I` wird nicht entfernt.
2. Die Namensverkürzung auf 30 Zeichen von rechts, die Umwandlung unerlaubter Sonderzeichen in das $-Zeichen, die Umwandlung von Klein- in Großbuchstaben sowie das Anhängen der Suffixe erfolgt wie bei der Ableitung aus dem Quellprogrammnamen.

Bildung der Namen von ii-Dateien

Für jede Quelldatei, die ein Template benutzt, wird, falls noch nicht vorhanden, eine Instanziierungs-Informationsdatei (ii-Datei) erzeugt.
Der Name der ii-Datei wird aus dem Namen der zugehörigen Objektdatei durch anfügen des Suffix .ii gebildet.
Bei der Übersetzung von `hugo.C` würde z.B die ii-Datei `hugo.o.ii` erzeugt werden, wenn nicht explizit ein anderer Objektname als `hugo.o` angegeben wird.
Die ii-Datei steht in der Bibliothek, in der auch die Objektdatei abgelegt ist.

3.1.5 Aufbau der Compilermeldungen

Der Compiler gibt folgende Arten von Meldungen aus:

- Informationsmeldungen, die den Benutzer über den allgemeinen Ablauf der Übersetzung informieren (z.B. Start- und Beendigungsmeldung des Compilers, Meldungen wie 'MODULES GENERATED').
- Fehlermeldungen des Compiler-Frontends, die sich direkt auf das übersetzte Quellprogramm beziehen (z.B. Syntax- und Semantikfehler).

  Diese bestehen aus

  1. der eigentlichen Fehlermeldungszeile
  2. der fehlerhaften Quellprogrammzeile (optional)
  3. einer Markierung ^ der Fehlerposition (optional)
- Fehlermeldungen ohne Quellprogrammbezug (z.B. Platte voll, erfolgloses Öffnen von Dateien).

Informations- und Fehlermeldungen werden - steuerbar über die MODIFY-DIAGNOSTIC-PROPERTIES-Anweisung ("MODIFY-DIAGNOSTIC-PROPERTIES") - während der Übersetzung auf eines oder mehrere der folgenden Ausgabeziele ausgegeben:

- Datensichtstation und/oder
- wahlweise in Dateien oder Bibliotheken.

Fehlermeldungen mit Quellprogrammbezug werden auch - steuerbar über die MODIFY-LISTING-PROPERTIES-Anweisung ("MODIFY-LISTING-PROPERTIES") - in der Quellprogramm-/Fehlerliste dokumentiert.

Allgemeines Format der Compilermeldungen:

Fehlermeldungszeile:

meldungsschlüssel [*fehlergewicht*] : *dateiname* / *zeilennummer* : *meldungstext*

optional bei Fehlermeldungen mit Quellprogrammbezug:

fehlerhafte Quellprogramzeile

^ *Markierung der Fehlerposition*

Bei Fehlermeldungen ohne Quellprogrammbezug (Gewicht FATAL oder ERROR) entfallen Dateiname und Zeilennummer. Bei den Informationsmeldungen entfällt außerdem das Fehlergewicht.

meldungsschlüssel

Der 7-stellige Meldungsschlüssel setzt sich zusammen aus einer 3-stelligen Meldungsklasse, die die Compilerkomponente kennzeichnet, und einer 4-stelligen Meldungsnummer.

Meldungsklasse	**Komponente**
CFE	Frontend des Compilers: Scanner, Präprozessor, Parser, Listengenerator, Meldungswesen
CDR	Compilertreiber, II-UPDATE

UMP	Zwischensprachenkomponenten: ULS, Optimierer, Inliner
BEM	Backend
SIS	Compiler-Ein-/Ausgabe-Schnittstelle (PROSOS), Objektformat(Modul)-Generator (OFG), Diagnoseinformationen-Generator für AID (DIG)
BND	BINDER (beim Binden mit der BIND-Anweisung des Compilers)
CCM	C-Laufzeitsystem

[*fehlergewicht*]

Angabe, welches Gewicht der aufgetretene Fehler hat:

[NOTE]	Hinweise, leichte Fehler, z.B. „unschöne“ oder überflüssige Konstrukte, die auf das spätere Programmverhalten in der Regel keinen Einfluss haben. Notes werden nicht automatisch, sondern nur bei Angabe der Option `-R minweight,notes` bzw. MINIMAL-MSG-WEIGHT=*NOTE ausgegeben.
[WARNING]	Fehler, bei denen der Compiler zwar ein Modul erzeugt, jedoch ggf. Annahmen trifft, die nicht zum erwarteten Programmverhalten führen
[ERROR], [*ERROR]	Fehler, bei denen kein Modul erzeugt wird. Der Compiler versucht, bis zu der mit der Option `-R limit,`*n* bzw. MAX-ERROR-NUMBER=*n* angegebenen Anzahl Errors mit der Übersetzung fortzufahren. Mit einem Stern sind Errors des Compiler-Frontends gekennzeichnet, die entweder mit der Option `-R warning/note` bzw. CHANGE-MSG-WEIGHT=*WARNING()/*NOTE() auf das Gewicht WARNING/NOTE herabgestuft werden können oder Warnings, die mit der Option `-R error` bzw. CHANGE-MSG-WEIGHT=*ERROR() auf das Gewicht ERROR hochgestuft wurden.
[FATAL]	Schwerer Fehler, der zum Abbruch der Übersetzung führt
[INTERNAL]	Compilerfehler, der ebenfalls zum Abbruch der Übersetzung führt

dateiname

Name der Datei oder des Bibliothekselementes mit dem fehlerhaften Quellprogrammcode

zeilennummer

Angabe der Quellprogrammzeile, in der der Fehler auftritt

meldungstext

Text der Fehlermeldung, der in Deutsch oder Englisch ausgegeben werden kann

Fehlerhafte Quellprogrammzeile mit Markierung der Fehlerposition

Bei Angabe der Option `-R show_column` bzw. SHOW-COLUMN=*YES (Voreinstellung) wird zusätzlich zur Diagnosemeldung die fehlerhafte Original-Quellprogrammzeile ausgegeben, in der die Fehlerstelle markiert ist (mit ^). Die Ausgabe der markierten Quellprogrammzeile kann mit `-R no_show_column` bzw. SHOW-COLUMN=*NO unterdrückt werden.

Beispiel

```
//MODIFY-SOURCE-PROP LANG=*C(MODE=1990, STRICT=*YES)
//MODIFY-DIAGNOSTIC-PROP CHANGE-MSG-WEIGHT=*ERROR(CFE1064)
//COMPILE SOURCE=TEST.C
% CFE1064 [*ERROR]: TEST.C / 1: declaration does not declare anything
  struct { int a; };
  ^
% CFE1020 [ERROR]: TEST.C / 3: identifier "abc" is undefined
  abc def;
  ^
% CFE2095 [*ERROR]: USER.H / 3: too few arguments in invocation of macro "M"
  int i = M(3);
             ^
```

Die Fehlernummer CFE1064 ist im Original ein WARNING und wurde mit der Option CHANGE-MSG-WEIGHT=*ERROR(CFE1064) auf die Fehlerklasse ERROR hochgestuft. Die Fehlerklasse von Errors, die im Original nicht mit einem Stern ausgegeben werden (hier z.B. CFE1020), kann nicht verändert werden.
Die Fehlernummer CFE2095 ist ein Beispiel für einen im strikten C89-Modus generierten Error, der mit CHANGE-MSG-WEIGHT=*WARNING(CFE2095) herabgestuft werden kann. Die gleiche Fehlernummer hat im erweiterten C89-Modus die Fehlerklasse WARNING.

Zu etlichen Fehlermeldungen kann mit dem HELP-MSG-Kommando eine nähere Erläuterung abgerufen werden:

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

Die Compilermeldungen können in deutsch oder englisch ausgegeben werden. Der Standardwert hängt von der Systemgenerierung ab. Mit folgendem Kommando kann die taskspezifische Voreinstellung geändert werden:

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

3.2 Steuerung des Compilers

Der C/C++Compiler wird analog zu diversen BS2000-Programmen (BINDER, LMS etc.) als eigenständiges Programm aufgerufen und über eine komfortable SDF-Anweisungsschnittstelle gesteuert. Somit können alle Möglichkeiten genutzt werden, die das Produkt SDF (System Dialog Facility) für die Eingabe von Programmanweisungen bietet: Verschiedene Stufen des geführten und ungeführten Dialogs, Eingabe aus Prozedur- oder Kommando-Dateien.

Ausführliche Informationen und Beispiele zu den verschiedenen Formen der SDF-Steuerung finden Sie im Handbuch „SDF Dialogschnittstelle“ [12].

3.2.1 Aufruf des Compilers (START-CPLUS-COMPILER)

/START-CPLUS-COMPILER	Kurznamen: SRCPC, CPLUS-COMPILER, CPC
MONJV= *NONE / <filename 1..54> ,CPU-LIMIT= *JOB-REST / <integer 1..32767>	

MONJV = *NONE / <filename 1..54>
Mit dieser Option kann eine BS2000-Jobvariable zur Überwachung des Compilerlaufs zugewiesen werden. <filename> gibt den Namen einer überwachenden Jobvariablen an, in die der Compiler eine Anzeige über mögliche Ablauffehler hinterlegt.

Das Betriebssystem bildet in der Jobvariablen zwei Werte ab:

eine Zustandsanzeige von 3 Byte Länge,
eine Rückkehrcode-Anzeige von 4 Byte Länge.

Die Zustandsanzeige gibt an, ob die zuletzt ausgeführte Anweisung erfolgreich war oder nicht. Bei einer fatalen Meldung wird sie auf "$A " gesetzt, sonst auf "$T ".

Während des Compilerlaufs werden Meldungen von unterschiedlichem Gewicht ausgegeben. Der Rückkehrcode gibt an welches das schwerste Gewicht dieser Meldungen war.

Fehlergewicht	Rückkehrcode
kein Fehler	0000
[NOTE]	1001
[WARNING]	1002
[ERROR]	2003
[FATAL]	3005

Diese beiden Angaben müssen nicht synchron sein. Hier zwei Beispiele für ungewöhnliche Ergebnisse:

```
/START-CPLUS-COMPILER
//COMPALE MUELL
//END
```

Dieses Beispiel liefert "$A 0000". Das "$A " kommt von dem Schreibfehler in der SDF-Anweisung. Dieser wird direkt von SDF gemeldet. Die "0000" kommt, da der Compiler selbst nie eine Meldung ausgegeben hat.

```
/START-CPLUS-COMPILER
//COMPILE FATAL.C
//STEP
//COMPILE GOOD.C
//END
```

Dieses Beispiel liefert "$T 3005". Das "$T " kommt, weil die zweite Übersetzung gut geht. Die "3005" kommt, weil bei der ersten Übersetzung eine Meldung vom Gewicht FATAL erzeugt wurde.

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

Mit dieser Option kann die maximale CPU-Zeit für den Compilerlauf festgelegt werden. Die Option entspricht dem CPU-LIMIT-Operanden des START-EXECUTABLE-PROGRAM-Kommandos.

3.2.2 Beschreibung der Compiler-Anweisungen

In diesem Abschnitt werden folgende Themen behandelt:

3.2.2.1 Anweisungsübersicht

Zum Prinzip der „ausführenden“ und „modifizierenden“ Compiler-Anweisungen beachten Sie bitte die Hinweise im Abschnitt „Prinzip und allgemeine Eingaberegeln“.

- Ausführende Anweisungen

  BIND

  Binde- und/oder Prälinker-Lauf starten

  CHECK-SYNTAX

  Syntax-Analyse starten

  COMPILE

  Übersetzungslauf (einschließlich Modulgenerierung) starten

  PREPROCESS

  Präprozessorlauf starten

- Modifizierende und definierende Anweisungen

  In Klammern ist angegeben, bei welcher ausführenden Anweisung die jeweilige MODIFY-Anweisung ausgewertet wird.

  MODIFY-BIND-PROPERTIES

  Optionen zur Steuerung des Binde- und/oder Prälinker-Laufs (BIND)

  MODIFY-CIF-PROPERTIES

  Optionen zur Ausgabe von CIF-Informationen (CHECK-SYNTAX, COMPILE, PREPROCESS)

  MODIFY-DIAGNOSTIC-PROPERTIES

  Optionen zur Steuerung der Meldungsausgabe (CHECK-SYNTAX, COMPILE, PREPROCESS)

  MODIFY-INCLUDE-LIBRARIES

  Optionen zur Eingabe von Include-Dateien (CHECK-SYNTAX, COMPILE, PREPROCESS)

  MODIFY-LISTING-PROPERTIES

  Optionen zur Ausgabe von Listen (CHECK-SYNTAX, COMPILE, PREPROCESS)

  MODIFY-MODULE-PROPERTIES

  Optionen zur Steuerung der Objekt- und Moduleigenschaften (COMPILE)

  MODIFY-OPTIMIZATION-PROPERTIES

  Optimierungsoptionen (COMPILE)

  MODIFY-RUNTIME-PROPERTIES

  Laufzeit-Optionen (COMPILE)

  MODIFY-SOURCE-PROPERTIES

  Frontend-Optionen (CHECK-SYNTAX, COMPILE, PREPROCESS)

  MODIFY-TEST-PROPERTIES

  Testhilfe-Optionen (COMPILE)

- Informierende Anweisungen

  SHOW-DEFAULTS

  Standardeinstellungen des Compilers anzeigen

  SHOW-PROPERTIES

  aktuelle Optionswerte der MODIFY-Anweisungen anzeigen

- RESET-TO-DEFAULT

  Optionswerte der MODIFY-Anweisungen auf die Standardeinstellungen des Compilers setzen

- END

  Compilerlauf beenden

- SDF-Standardanweisungen

  Zusätzlich zu den oben aufgeführten C/C++-spezifischen Anweisungen können während eines Compilerlaufs auch die folgenden SDF-Standardanweisungen genutzt werden. Die ausführliche Beschreibung dieser SDF-Anweisungen finden Sie im Benutzerhandbuch „Einführung in die Dialogschnittstelle SDF“ [12].

  EXECUTE-SYSTEM-CMD

  Systemkommando während des Compilerlaufs ausführen

  HELP-MSG-INFORMATION

  Text einer System- oder Compilermeldung auf SYSOUT ausgeben

  HOLD-PROGRAM

  Compilerlauf anhalten, um z.B. Systemkommandos einzugeben; die Rückkehr in den Anweisungsmodus des Compilers erfolgt mit dem Kommando RESUME-PROGRAM

  MODIFY-SDF-OPTIONS

  Benutzersyntaxdatei aktivieren/deaktivieren und SDF-Einstellungen ändern

  REMARK

  Kommentare in die Anweisungsfolge schreiben

  RESET-INPUT-DEFAULTS

  taskspezifische Default-Werte zurücksetzen

  RESTORE-SDF-INPUT

  zuvor eingegebene Anweisungen oder Kommandos anzeigen

  SHOW-INPUT-DEFAULTS

  taskspezifische Default-Werte anzeigen

  SHOW-INPUT-HISTORY

  Inhalt des Eingabepuffers anzeigen

  SHOW-SDF-OPTIONS

  Informationen über alle aktivierten Syntaxdateien und SDF-Einstellungen für die aktuelle Task ausgeben

  SHOW-STMT

  SDF-Syntax einer Anweisung ausgeben

  STEP

  Wiederaufsetzpunkt innerhalb einer Kommando-/Prozedur-Datei definieren

  WRITE-TEXT

  Text ausgeben

3.2.2.2 Prinzip und allgemeine Eingaberegeln

- Ausführende und modifizierende Compiler-Anweisungen

  Der Compilerlauf wird mit dem Kommando START-CPLUS-COMPILER gestartet und mit der Anweisung END beendet. Während eines Compilerlaufs können mehrere Übersetzungs- bzw. Bindeläufe mit den entsprechenden ausführenden Anweisungen (siehe "Anweisungsübersicht") gestartet werden. Zur Steuerung der Übersetzungs- und Bindeläufe stehen MODIFY-Anweisungen zur Verfügung, die den ausführenden Anweisungen immer vorausgehen müssen. Die MODIFY-Anweisungen behalten ihre Gültigkeit über die abgearbeiteten ausführenden Anweisungen hinweg. Siehe auch Beispiel.

- Operandenwert *UNCHANGED und Standardeinstellungen des Compilers

  Beim Arbeiten mit dem Compiler im geführten Dialog wird in den SDF-Anweisungsmenüs des Compilers für viele Operanden der Operandenwert *UNCHANGED angezeigt.

  Der Wert *UNCHANGED ist der SDF-Defaultwert, der in der aktuellen Compiler-Anweisung immer dann eingesetzt (und am Bildschirm angezeigt) wird, wenn zu einem Operanden keine Angabe gemacht wird. *UNCHANGED bedeutet, dass für den Operanden jeweils der Operandenwert gilt, der mit einer vorhergehenden Compiler-Anweisung vereinbart wurde. Wenn zu einem Operanden im gesamten Compilerlauf noch keine Angabe gemacht wurde, wie z.B. auch unmittelbar nach Start des Compilers, wird für *UNCHANGED der Standardwert des Compilers eingesetzt. Dieser Standardwert gilt solange, bis mit einer Compiler-Anweisung ein anderer Operandenwert angegeben wird. Der Standardwert des Compilers ist in den Anweisungsbeschreibungen durch Unterstreichung gekennzeichnet. Für manche Operanden ist der Standardwert von dem eingestellten Sprachmodus abhängig. In diesem Fall ist kein Wert unterstrichen.
  Mit der Anweisung SHOW-PROPERTIES kann man sich über die aktuell gültigen Operandenwerte der Compiler-Anweisungen informieren. Statt *UNCHANGED wird dann der tatsächlich dafür eingesetzte Operandenwert ausgegeben.

- Alias-Namen

  Die meisten Compiler-Anweisungen haben Alias-Namen, die alternativ zu den im geführten Dialog angebotenen Namen verwendet werden können. Diese Alias-Namen sind bei den Anweisungsbeschreibungen jeweils zu Beginn aufgeführt.

- Spin-Off-Mechanismus

  Eine nicht gelungene ausführende Anweisung löst den Spin-Off aus. In diesem Fall werden alle Anweisungen bis zum nächsten STEP ignoriert.

- Input- / Output - Library - Elemente

  Die Versionsangabe VERSION=@ wird abgelehnt, da dies kein gültiger Wert für die LMS-Version ist.

Beispiel

```
/START-CPLUS-COMPILER
//MODIFY-SOURCE-PROPERTIES DEFINE=_OSD_POSIX ————————————————————  (1)
//MODIFY-LISTING-PROPERTIES SOURCE=*YES ————————————————————————  (2)
//MODIFY-INCLUDE-LIBRARIES STD-INCLUDE-LIBRARY=
  (*STANDARD-LIBRARY, $.SYSLIB.POSIX-HEADER) ————————————————————  (3)
//COMPILE SOURCE=HALLO1.CC ——————————————————————————————————  (4)
//MODIFY-SOURCE-PROPERTIES LANG=*C ——————————————————————————  (5)
//COMPILE SOURCE=HALLO2.C ———————————————————————————————————  (6)
//END
/
```

In der MODIFY-SOURCE-PROPERTIES-Anweisung (1) wird lediglich ein Define gesetzt, für die übrigen Operanden gelten die Standardeinstellungen des Compilers, z.B. der neueste C++-Sprachmodus LANGUAGE=*CPLUSPLUS(MODE=*LATEST, STRICT=*NO).

Bei der ersten Übersetzung (4) werden die vorangehenden MODIFY-Anweisungen ausgewertet und infolgedessen folgende Aktionen durchgeführt:

- Setzen des Defines _OSD_POSIX (1)
- Durchsuchen der Include-Bibliotheken $.SYSLIB.CRTE, $.SYSLIB.CRTE.CXX01 und $.SYSLIB.POSIX-HEADER (3)
- Übersetzen des Quellprogramms HALLO1.CC (4) im C++ 2017-Sprachmodus (1) und Ausgabe einer Quellprogramm-/Fehlerliste auf SYSLST (2)

Vor der zweiten Übersetzung (6) wird in einer MODIFY-SOURCE-PROPERTIES-Anweisung (5) der C-Sprachmodus eingestellt. Da dieser nicht genauer angegeben wurde, gelten für die Unteroperanden die Standardeinstellungen der Compilers, also *C(MODE=*LATEST, STRICT=*NO). Bis auf diese Änderung sind für die Übersetzung des C-Quellprogramms HALLO2.C alle anderen Angaben der vorangehenden MODIFY-Anweisungen gültig, d.h. es werden die folgenden Aktionen durchgeführt.:

- Setzen des Defines _OSD_POSIX (1)
- Durchsuchen der Include-Bibliotheken $.SYSLIB.CRTE und $.SYSLIB.POSIX-HEADER (3)
- Übersetzen des Quellprogramms HALLO2.C (6) im erweiterten C11-Sprachmodus (5) und Ausgabe einer Quellprogramm-/Fehlerliste auf SYSLST (2)

Die Angabe von *STANDARD-LIBRARY in (3) wird bei den beiden Übersetzungen unterschiedlich aufgelöst. Die Menge der durchsuchten Bibliotheken ist hier vom Sprachmodus abhängig. Die Abbildung von *STANDARD-LIBRARY auf die Liste der Bibliotheken wird bei der Übersetzung (4,6) gemacht, nicht bei der Anweisung MODIFY-INCLUDE-LIBRARIES (3).

3.2.2.3 BIND

Alias-Name: LINK

Mit dieser Anweisung wird ein Bindelauf gestartet. Im Falle von C++ V3 und C++ 2017-Objekten wird außerdem der Prälinker zur automatischen Template-Instanziierung aktiviert. Die Eingabequellen und andere Bedingungen für den Prälinker- und Bindelauf werden in vorangehenden MODIFY-BIND-PROPERTIES-Anweisungen festgelegt.

BIND

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

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

    *LIBRARY-ELEMENT(...)

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

        |       *LINK(...)

        |           |   LINK-NAME= <filename 1..8>

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

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

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

,OUTPUT-FORMAT= *UNCHANGED / *LLM(...)

    *LLM(...)

        |   EXTERNAL-NAMES= *UNCHANGED / *STD / *SHORT / *EXTENDED

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

ACTION = list-poss: *PRELINK / *MODULE-GENERATION
Die Angabe *PRELINK ist nur in den Modi C++ V3 und C++ 2017 relevant und betrifft die automatische Template-Instanziierung durch den Prälinker.

Wenn die ACTION-Option nicht angegeben wird, gilt folgende Voreinstellung:

ACTION = (*PRELINK,*MODULE-GENERATION)

In den Modi C++ V3 und C++ 2017 wird dann sowohl ein Prälinker- als auch ein Bindelauf durchgeführt. Das Ergebnis sind Einzelmodule, in denen alle Templates instanziiert sind und ein gebundenes Modul. In den C-Modi und im Cfront-C++-Modus wird die Angabe *PRELINK ignoriert und es wird nur ein Bindelauf durchgeführt.

Bei Angabe von ACTION=*PRELINK wird nur ein Prälinker-Lauf durchgeführt. Das Ergebnis sind Einzelmodule, in denen alle Templates instanziiert sind. Ggf. Angaben in den OUTPUT- und OUTPUT-FORMAT-Optionen werden ignoriert.

Bei Angabe von ACTION=*MODULE-GENERATION wird nur ein Bindelauf durchgeführt.

OUTPUT = *NONE / *LIBRARY-ELEMENT(...)
Mit dieser Option wird angegeben, in welcher PLAM-Bibliothek (LIBRARY=) und unter welchem Elementnamen (ELEMENT=) das gebundene Modul abgelegt werden soll. Diese Angaben werden in einer SAVE-LLM-Anweisung (als MODULE-CONTAINER-Operand) an den BINDER weitergereicht. Wenn mit der Option ADD-OPTION keine anderen Angaben gemacht werden, gelten für die restlichen Operanden der SAVE-LLM-Anweisung die entsprechenden Standardeinstellungen des BINDER.
Die Angabe `*NONE` ist Standard und ist nur beim Prälinken sinnvoll, dh. es wird kein Binde-Objekt erzeugt (siehe oben, ACTION =).

LIBRARY =<filename 1..54>
Das Modul wird in die PLAM-Bibliothek mit dem angegebenen Namen geschrieben.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>

Mit <filename> kann (statt eines katalogisierten Bibliotheksnamens) auch ein Linkname angegeben werden. Der Linkname muss vor Aufruf des Compilers mit dem ADD-FILE-LINK-Kommando der PLAM-Bibliothek zugeordnet worden sein.

ELEMENT = <composed-name 1..64 with-under>(...)
Das Modul wird unter dem angegebenen Namen in die mit LIBRARY= angegebene PLAM-Bibliothek geschrieben.

VERSION = *UPPER-LIMIT
Enthält die Elementangabe keine Versionsbezeichnung, verwendet der Compiler die höchstmögliche Version.

VERSION = *INCREMENT
Das Element erhält die gegenüber der höchsten vorhandenen Version um 1 inkrementierte Versionsnummer, vorausgesetzt, die höchste vorhandene Versionsbezeichnung endet mit einer inkrementierbaren Ziffer. Ist die Versionsbezeichnung nicht inkrementierbar, wird der Bindelauf mit Fehler abgebrochen.
Beispiel siehe COMPILE-Anweisung ("COMPILE").

VERSION = <composed-name 1..24 with-under>
Das Element erhält die angegebene Versionsbezeichnung.

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

Diese Option steuert die Behandlung von Symbolnamen im EEN-Format (EEN = Extended External Name) durch den BINDER. Die Angaben werden als FOR-BS2000-VERSION-Operand der SAVE-LLM-Anweisung an den BINDER weitergereicht.

EEN-Namen, d.h. ungekürzte externe C++-Symbole, sind generell in Modulen enthalten, die mit dem Compiler im C++ V3 oder C++ 2017-Modus erzeugt werden.
Ungekürzte externe C-Symbole werden nur dann generiert, wenn bei der Übersetzung folgende Option angegeben wird: MODIFY-MODULE-PROPERTIES C-NAMES=*UNLIMITED (siehe "MODIFY-MODULE-PROPERTIES").
In diesem Fall werden auch längere externe C-Symbole vom Compiler nicht auf 32 Zeichen verkürzt.

Module mit EEN-Namen werden vom Compiler im LLM-Format 4 abgelegt. Die Module der im C++ V3 und C++ 2017 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.

EXTERNAL-NAMES = *UNCHANGED
Es gilt die Angabe der letzten BIND-Anweisung.

EXTERNAL-NAMES = *STD
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 oder DBL weiterverarbeitet werden.

EXTERNAL-NAMES = *SHORT
Diese Angabe wird benötigt, wenn der BINDER das LLM-Format 1 generieren soll.

EXTERNAL-NAMES = *EXTENDED
Diese Angabe wird nur aus Kompatibilitätsgründen unterstützt.

Zusammenfassung der generierten LLM-Formate

Eingabeformat	SDF-Option EXTERNAL-NAMES =	Ausgabeformat
LLM 1	Keine Angabe / *STD / *SHORT / *EXTENDED	LLM 1
LLM 4 (EEN)	Keine Angabe / *STD / *EXTENDED	LLM 4
	*SHORT	LLM 1

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

Mit <c-string> können zusätzlich zum MODULE-CONTAINER-Operanden (siehe OUTPUT-Option) und zum FOR-BS2000-VERSION-Operanden (siehe OUTPUT-FORMAT-Option) weitere Operanden der BINDER-Anweisung SAVE-LLM angegeben werden. Mehrere Operanden müssen durch Kommas getrennt werden:
'operand1,operand2,...'
Die Operanden werden ohne SDF-Analyse direkt an den BINDER durchgereicht. Die folgenden Operanden der BINDER-Anweisung SAVE-LLM werden der MODIFY-BIND-PROPERTIES-Anweisung entnommen und dürfen deshalb nicht mit ADD-OPTION angegeben werden: TEST-SUPPORT (Abspeichern der LSD-Informationen) und MAP (Ausgeben einer Binderliste).

Automatische Template-Instanziierung

Die mit ADD-OPTION angegebenen Operanden werden derzeit nicht bei der automatischen Template-Instanziierung durch den Prälinker, sondern nur beim nachfolgenden Binden durch den BINDER berücksichtigt. Die Operanden sollten nicht die Art, Anzahl oder Reihenfolge der einzubindenden Module beeinflussen, da unterschiedliche Voraussetzungen beim Prälinker- und Bindelauf ggf. zu Duplikaten oder unbefriedigten Externverweisen führen können.

Beispiel

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

3.2.2.4 CHECK-SYNTAX

Alias-Name: DO-SYNTAX-CHECK

Der Übersetzungslauf wird nach der Syntaxanalyse eines oder mehrerer Quellprogramme beendet. Es wird kein Objektcode erzeugt.

```
CHECK-SYNTAX

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

    *LIBRARY-ELEMENT(...)

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

        |        *LINK(...)

        |            |    LINK-NAME= <filename 1..8>

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

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

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

SOURCE =
Mit dieser Option werden eines oder mehrere Quellprogramme angegeben, die einer Syntaxanalyse unterzogen werden sollen.
Ein Quellprogramm kann von der Systemdatei SYSDTA, aus einer katalogisierten BS2000-Datei, aus einer PLAM-Bibliothek oder aus einer POSIX-Datei eingelesen werden.
Bei der Quellprogrammeingabe von SYSDTA kann pro CHECK-SYNTAX-Anweisung nur ein Quellprogramm eingelesen werden.

SOURCE = *SYSDTA
Die Eingabe erfolgt von der Systemdatei SYSDTA. SYSDTA ist im Dialogbetrieb der Datensichtstation zugewiesen und kann mit dem ASSIGN-SYSDTA-Kommando auf eine katalogisierte Datei oder ein PLAM-Bibliothekselement umgewiesen werden (siehe auch "Hinweise zur Eingabe über SYSDTA").

SOURCE = <filename 1..54>
<filename> ist der Name einer katalogisierten BS2000-Datei.

SOURCE = <posix-pathname>
Als <posix-pathname> ist nur ein Dateiname zugelassen. Zur Beschreibung des Begriffs <posix-pathname> siehe "Compiler-Ein-/Ausgaben im POSIX-Dateisystem".

SOURCE = *LIBRARY-ELEMENT(...)
Es werden eine PLAM-Bibliothek und ein Element daraus angegeben.

LIBRARY = <filename 1..54>
<filename> ist der Name einer PLAM-Bibliothek.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>

<filename> ist der Linkname einer PLAM-Bibliothek. Der Linkname muss vor Aufruf des Compilers mit dem ADD-FILE-LINK-Kommando dem Bibliotheksnamen zugeordnet worden sein.

ELEMENT = <composed-name 1..64 with-under>(...)
<composed-name> ist der vollqualifizierte Name eines Elements aus der zuvor angegebenen PLAM-Bibliothek. Das Element muss vom Typ S sein.

VERSION = *HIGHEST-EXISTING
Enthält die Elementangabe keine Versionsbezeichnung, nimmt der Compiler das Element mit der höchsten Version.

VERSION = <composed-name 1..24 with-under>
Der Compiler nimmt das Element mit der angegebenen Version.

Hinweis

Das implizite Inkludieren für die Template-Instanziierung findet (im Unterschied zur COMPILE-Anweisung) **nicht** statt.

3.2.2.5 COMPILE

Es werden eines oder mehrere Quellprogramme übersetzt und es wird pro Übersetzungseinheit ein Modul generiert.

```
COMPILE

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

SOURCE =
Mit dieser Option werden eines oder mehrere Quellprogramme angegeben, die übersetzt werden sollen.
Ein Quellprogramm kann von der Systemdatei SYSDTA, aus einer katalogisierten BS2000-Datei, aus einer PLAM-Bibliothek oder aus einer POSIX-Datei eingelesen werden.
Bei der Quellprogrammeingabe von SYSDTA kann pro COMPILE-Anweisung nur ein Quellprogramm eingelesen werden.

SOURCE = *SYSDTA

Die Eingabe von der Systemdatei SYSDTA ist nur in den C-Modi und im Cfront-C++-Modus des Compilers möglich. In den Modi C++ V3 und C++2017 wird die Angabe *SYSDTA mit einer entsprechenden Fehlermeldung abgewiesen. SYSDTA ist im Dialogbetrieb der Datensichtstation zugewiesen und kann mit dem ASSIGN-SYSDTA-Kommando auf eine katalogisierte Datei oder ein PLAM-Bibliothekselement umgewiesen werden (siehe auch "Hinweise zur Eingabe über SYSDTA").

SOURCE = <filename 1..54>
<filename> ist der Name einer katalogisierten BS2000-Datei.

SOURCE = <posix-pathname>
Als <posix-pathname> ist nur ein Dateiname zugelassen. Zur Beschreibung des Begriffs <posix-pathname> siehe "Compiler-Ein-/Ausgaben im POSIX-Dateisystem".

SOURCE = *LIBRARY-ELEMENT(...)
Es werden eine PLAM-Bibliothek und ein Element daraus angegeben.

LIBRARY = <filename 1..54>
<filename> ist der Name einer PLAM-Bibliothek.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>

<filename> ist der Linkname einer PLAM-Bibliothek. Der Linkname muss vor Aufruf des Compilers mit dem ADD-FILE-LINK-Kommando dem Bibliotheksnamen zugeordnet worden sein.

ELEMENT = <composed-name 1..64 with-under>(...)
<composed-name> bezeichnet den vollqualifizierten Namen eines Elements aus der zuvor angegebenen PLAM-Bibliothek. Das Element muss vom Typ S sein.

VERSION = *HIGHEST-EXISTING
Enthält die Elementangabe keine Versionsbezeichnung, nimmt der Compiler das Element mit der höchsten Version.

VERSION = <composed-name 1..24 with-under>
Der Compiler nimmt das Element mit der angegebenen Version.

MODULE-OUTPUT =
Mit dieser Option wird gesteuert, in welcher Bibliothek und/oder unter welchem Namen die erzeugten Module abgelegt werden sollen.

MODULE-OUTPUT = *SOURCE-LOCATION
Das Modul wird dorthin geschrieben, wo auch das Quellprogramm steht. Ist das Quellprogramm ein PLAM-Bibliothekselement, wird das Modul in die Bibliothek des Quellprogramms geschrieben. Ist das Quellprogramm eine POSIX-Datei, wird das Modul als Objektdatei (.o-Datei) in das Dateiverzeichnis des Quellprogramms geschrieben.

Der Element- bzw. Objektdatei-Name wird aus dem Quellprogrammnamen abgeleitet (siehe Abschnitt „Bildung von Modulnamen“).
Das Modul kann nicht in eine `write-only` Datei geschrieben werden. Die Bibliothek bzw. die Objektdatei in Posix muss immer auch eine Leseberechtigung haben.
Die Angabe *SOURCE-LOCATION ist unzulässig, wenn das Quellprogramm aus einer katalogisierten BS2000-Datei oder über SYSDTA eingelesen wird (siehe "Hinweise zur Eingabe über SYSDTA").

MODULE-OUTPUT = <posix-pathname>
Das Modul wird als LLM-Objektdatei in das POSIX-Dateisystem geschrieben.

Als <posix-pathname> ist sowohl ein Dateiname als auch ein Dateiverzeichnis zugelassen. Zur Beschreibung des Begriffs <posix-pathname> siehe "Compiler-Ein-/Ausgaben im POSIX-Dateisystem".

Bei der Angabe eines Dateinamens wird die Objektdatei unter diesem Namen abgelegt. Die Angabe eines Dateinamens ist unzulässig, wenn mehrere Quellprogramme mit einer COMPILE-Anweisung übersetzt werden.

Bei der Angabe eines Dateiverzeichnisnamens *dvz* wird eine Objektdatei für jedes übersetzte Quellprogramm unter dem Standardnamen *quelldatei.o* in das Dateiverzeichnis *dvz* geschrieben (siehe auch Abschnitt „Bildung von Modulnamen“).

Die mit <posix-pathname> angegebenen Dateiverzeichnisse müssen bereits existieren.Bei der Bildung der Dateinamen ist zu beachten, dass Objektdateien im POSIX-Subsystem nur sinnvoll weiterverarbeitet (gebunden) werden können, wenn der Name das Suffix `.o` enthält oder ein Suffix, das mit der Option `-Y F` der `cc/c11/c89/CC`-Kommandos vereinbart wurde (siehe Handbuch „POSIX-Kommandos des C/C++-Compilers“ [1]).

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

Es wird angegeben, in welcher PLAM-Bibliothek (LIBRARY=) und unter welchem Elementnamen (ELEMENT=) das Modul abgelegt werden soll.

LIBRARY = *STD-LIBRARY
Standardmäßig werden Module in die Bibliothek SYS.PROG.LIB geschrieben.

LIBRARY = *SOURCE-LIBRARY
Das Modul wird in die PLAM-Bibliothek geschrieben, in der das Quellprogramm steht. Die Angabe *SOURCE-LIBRARY ist unzulässig, wenn das Quellprogramm aus einer katalogisierten BS2000-Datei, einer POSIX-Datei oder über SYSDTA eingelesen wird.

LIBRARY =<filename 1..54>
Das Modul wird in die PLAM-Bibliothek mit dem angegebenen Namen geschrieben.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>
Mit <filename> kann (statt eines katalogisierten Bibliotheksnamens) auch ein Linkname angegeben werden. Der Linkname muss vor Aufruf des Compilers mit dem ADD-FILE-LINK-Kommando der PLAM-Bibliothek zugeordnet worden sein.

ELEMENT = *STD-ELEMENT(...)
Der Elementname wird aus dem Quellprogrammnamen abgeleitet (siehe Abschnitt „Bildung von Modulnamen“).

VERSION = *UPPER-LIMIT
Enthält die Elementangabe keine Versionsbezeichnung, verwendet der Compiler die höchstmögliche Version.

VERSION = *INCREMENT
Das Element erhält die gegenüber der höchsten vorhandenen Version um 1 inkrementierte Versionsnummer, vorausgesetzt, die höchste vorhandene Versionsbezeichnung endet mit einer inkrementierbaren Zahl. Ist die Versionsbezeichnung nicht inkrementierbar, wird der Übersetzungslauf mit Fehler abgebrochen.

Achtung: In den Modi C++ V3 und C++ 2017 ist die Angabe von ***INCREMENT** nicht erlaubt.

Beispiel

höchste vorhandene Version	durch *INCREMENT erzeugte Version
ABC1	ABC2
ABC9	Fehler
ABC09	ABC10
003	004
keine	001

VERSION = <composed-name 1..24 with-under>
Das Element erhält die angegebene Versionsbezeichnung.

ELEMENT = <composed-name 1..64 with-under>(...)
Das Modul wird unter dem angegebenen Namen in die mit LIBRARY= angegebene PLAM-Bibliothek geschrieben. Diese Angabe ist unzulässig, wenn mehrere Quellprogramme mit einer COMPILE-Anweisung übersetzt werden.

Zur Weiterverarbeitung mit dem DBL können Elementnamen von LLMs maximal 32 Zeichen lang sein (siehe auch Abschnitt „Bildung von Modulnamen“).

VERSION = *UPPER-LIMIT / *INCREMENT /
<composed-name 1..24 with-under>
Die Version kann in gleicher Weise angegeben werden, wie oben unter ELEMENT=*STD-ELEMENT(...) beschrieben.

Achtung: In den Modi C++ V3 und C++ 2017 ist die Angabe von ***INCREMENT** nicht erlaubt.

3.2.2.6 Hinweise zur Eingabe über SYSDTA

In den C-Sprachmodi und im Cfront-C++-Sprachmodus des Compilers wird aus Kompatibilitätsgründen die Eingabe einer Quelle über die Systemdatei SYSDTA ermöglicht. Hierzu muss folgende Option angegeben werden:

SOURCE=*SYSDTA

In den Modi C++ V3 und C++ 2017 ist die Eingabe über SYSDTA nicht möglich.

SYSDTA ist im Dialogbetrieb standardmäßig der Datensichtstation zugeordnet. Quellprogrammeingaben an der Datensichtstation sind zwar prinzipiell möglich, aber nicht sinnvoll, da die Quelle anschließend nicht mehr verfügbar ist.

Wenn deshalb mit der SOURCE-Option *SYSDTA angegeben wird, sollte SYSDTA einer katalogisierten Datei oder einem PLAM-Bibliothekselement zugewiesen werden. Das Kommando hierfür lautet

```
/ASSIGN-SYSDTA TO-FILE = {dateiname | *LIB-ELEM(LIB=bibliothek,ELEM=element)}
```

Die Zuweisung kann auf folgende Arten erfolgen:

1. Zuweisung von SYSDTA vor Aufruf des Compilers

   Beispiel

   /ASSIGN-SYSDTA TO-FILE=*quelle*

   /START-CPLUS-COMPILER

   Die zugewiesene Quelle muss vor dem eigentlichen Quellcode sämtliche benötigten MODIFY-Anweisungen und die COMPILE-Anweisung enthalten:

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

2. Zuweisung von SYSDTA nach Aufruf des Compilers

   Beispiel

   /START-CPLUS-COMPILER

   //MODIFY-SOURCE-PROP LANG=*C

   //MODIFY-...

   //EXECUTE-SYSTEM-CMD (ASSIGN-SYSDTA TO-FILE=*quelle*)

   Die benötigten MODIFY-Anweisungen werden vor der SYSDTA-Umweisung angegeben und müssen deshalb nicht in der zugewiesenen Quelle stehen. Nach der EXECUTE-Anweisung können keine weiteren SDF-Anweisungen angegeben werden. Die zugewiesene Quelle muss vor dem eigentlichen Quellcode mindestens die COMPILE-Anweisung enthalten:

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

Wenn das Ende der Quelldatei erreicht ist, wird der Compilerlauf beendet. Eine weitere Umweisung von SYSDTA während eines Compilerlaufs ist nicht möglich.

Auch wenn das Quellprogramm mit dem ASSIGN-SYSDTA-Kommando zugewiesen wird, gilt nach wie vor als Eingabequelle SYSDTA. Dies hat zur Folge, dass bei der Generierung von Standardnamen der Quellprogrammname „CSTD“ zu Grunde gelegt wird (vgl. die Ausgabeparameter *SOURCE-LOCATION oder ELEMENT=*STD-ELEMENT). Außerdem ist die Angabe des Parameters *SOURCE-LIBRARY wirkungslos, wenn ein Bibliothekselement nicht mit der SOURCE-Option, sondern mit dem ASSIGN-SYSDTA-Kommando zugewiesen wird.

3.2.2.7 END

Diese Anweisung beendet den Compilerlauf.

```
END
```

3.2.2.8 MODIFY-BIND-PROPERTIES

Alias-Name: SET-BIND-PROPERTIES
MODIFY-LINK-PROPERTIES
SET-LINK-PROPERTIES

Diese Anweisung legt die Eingabequellen und andere Bedingungen für den Prälinker- und Bindelauf fest, der mit einer anschließenden BIND-Anweisung gestartet wird.

MODIFY-BIND-PROPERTIES

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

```
        |           |   LINK-NAME= <filename 1..8>
        |   ,ELEMENT= *ALL(...) / <composed-name 1..64 with-under>(...)
        |       *ALL(...)
        |           |   VERSION= *HIGHEST-EXISTING / <composed-name 1..24 with-under>
        |       <composed-name 1..64 with-under>(...)
        |           |   VERSION= *HIGHEST-EXISTING / <composed-name 1..24 with-under>
    *LIBRARY(...)
        |   LIBRARY-NAME= <filename 1..54> / *LINK(...)
        |       *LINK(...)
        |           |   LINK-NAME= <filename 1..8>
,MAX-INSTANTIATE-ITER= 30 / *UNCHANGED / <integer 0..100>
,TEMPLATE-DEF-LIST= *UNCHANGED / *YES / *NO
,ADD-STATEMENT= *UNCHANGED / *NONE / list-poss: <c-string 1..1800 with-low>
,RUNTIME-LANGUAGE= *UNCHANGED / *C / *CPLUSPLUS(...)
    *CPLUSPLUS(...)
        |   MODE= *UNCHANGED / *LATEST / *2017 / *V2-COMPATIBLE / *V3-COMPATIBLE
,STDLIB= *UNCHANGED / *DYNAMIC / *DYNAMIC-COMPLETE / *STATIC / *NONE
,TOOLSLIB= *UNCHANGED / *YES / *NO
,TEST-SUPPORT= *UNCHANGED / *YES / *NO
,LISTING= *UNCHANGED / *NONE / *SYSLST / <filename 1..54>
```

START-LLM-CREATION = *YES / *NO
Die Angabe START-LLM-CREATION=*YES ist analog zur BINDER-Anweisung START-LLM-CREATION, mit der ein neuer Bindeprozess gestartet und ein neues LLM im Arbeitsbereich erzeugt wird. *YES ist die Standardeinstellung des Compilers und automatisch gesetzt, wenn es sich um die erste MODIFY-BIND-PROPERTIES-Anweisung seit Aufruf des Compilers handelt.

Ab der zweiten MODIFY-BIND-PROPERTIES-Anweisung wird START-LLM-CREATION automatisch auf den SDF-Defaultwert *NO gesetzt. Dieser Wert gilt für alle nachfolgenden MODIFY-BIND-PROPERTIES-Anweisungen auch über BIND-Anweisungen hinweg, bis er explizit auf *YES geändert wird.
Solange der Wert *NO gesetzt ist, behalten die Angaben zu den INCLUDE- und RESOLVE-Bibliotheken ihre Gültigkeit für alle nachfolgenden Bindeläufe.
Zum Zusammenspiel der Anweisungen MODIFY-BIND-PROPERTIES und BIND siehe auch Abschnitt „Zusammenspiel der Anweisungen MODIFY-BIND-PROPERTIES und BIND“.

INCLUDE =
Mit dieser Option werden analog zum MODULE-CONTAINER-Operanden der BINDER-Anweisung INCLUDE-MODULES Module angegeben, die eingebunden werden sollen.

Automatische Template-Instanziierung

Die mit der INCLUDE-Option angegebenen Module werden vom Prälinker immer instanziiert. Für den Fall, dass dieselbe Bibliothek auch in der RESOLVE-Option angegeben wird, ist jedoch darauf zu achten, dass die Template-Instanziierung dort nicht ausgeschaltet ist (siehe RESOLVE-Option, INSTANTIATE = *NO / *IGNORE, "MODIFY-BIND-PROPERTIES").

INCLUDE = *UNCHANGED
Es gelten die Angaben der MODIFY-BIND-PROPERTIES-Anweisungen seit dem letzten START-LLM-CREATION=*YES.

INCLUDE = *NONE
Der Optionswert *NONE löscht nur die jeweilige INCLUDE-Kette, veranlasst jedoch nicht die komplette Aktion wie bei START-LLM-CREATION=YES. NONE ist der Compiler-Default, d.h. ein Wert nach dem Start des Compilers oder nach dem RESET-TO-DEFAULT.

INCLUDE = list-poss: *LIBRARY-ELEMENT(...)
Es wird angegeben, in welcher PLAM-Bibliothek (LIBRARY=) und unter welchem Elementnamen (ELEMENT=) das einzubindende Modul abgelegt ist.
Mehrere *LIBRARY-ELEMENT-Angaben in einer Liste werden intern in mehrere INCLUDE-MODULES-Anweisungen des BINDER umgesetzt, und zwar in der Reihenfolge, in der die *LIBRARY-ELEMENT-Angaben in der Liste stehen.

LIBRARY = <filename 1..54>
<filename> ist der Name einer PLAM-Bibliothek.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>

<filename> ist der Linkname einer PLAM-Bibliothek. Der Linkname muss vor Aufruf des Compilers mit dem ADD-FILE-LINK-Kommando dem Bibliotheksnamen zugeordnet worden sein.

ELEMENT = *ALL(...)
Es werden alle Module aus der mit LIBRARY= angegebenen PLAM-Bibliothek eingebunden.

VERSION = *HIGHEST-EXISTING

Enthält die Elementangabe keine Versionsbezeichnung, nimmt der Compiler die Elemente mit den jeweils höchsten Versionen.

VERSION = <composed-name 1..24 with-under>
Der Compiler nimmt die Elemente mit der angegebenen Version.

ELEMENT = <composed-name 1..64 with-under>(...)
<composed-name> bezeichnet den vollqualifizierten Namen eines Moduls aus der mit LIBRARY= angegebenen PLAM-Bibliothek.

VERSION = *HIGHEST-EXISTING
Enthält die Elementangabe keine Versionsbezeichnung, nimmt der Compiler das Element mit der höchsten Version.

VERSION = <composed-name 1..24 with-under>
Der Compiler nimmt das Element mit der angegebenen Version.

ADD-OPTION = *UNCHANGED / *NONE / <c-string 1..1800 with-low>
Mit <c-string> können zusätzlich zum MODULE-CONTAINER-Operanden weitere Operanden der BINDER-Anweisung INCLUDE-MODULES angegeben werden. Mehrere Operanden müssen durch Kommas getrennt werden:
' operand1 , operand2 ,...'
Die Operanden werden ohne SDF-Analyse direkt an den BINDER durchgereicht. Eingabebeispiel siehe BIND-Anweisung ("BIND").

Automatische Template-Instanziierung

Die mit ADD-OPTION angegebenen Operanden werden derzeit nicht bei der automatischen Template-Instanziierung durch den Prälinker, sondern nur beim nachfolgenden Binden durch den BINDER berücksichtigt. Die Operanden sollten nicht die Art, Anzahl oder Reihenfolge der einzubindenden Module beeinflussen, da unterschiedliche Voraussetzungen beim Prälinker- und Bindelauf ggf. zu Duplikaten oder unbefriedigten Externverweisen führen können.

RESOLVE =
Mit dieser Option werden analog zur BINDER-Anweisung RESOLVE-BY-AUTOLINK Bibliotheken angegeben, aus denen der BINDER unbefriedigte Externverweise befriedigen soll (Autolink-Verfahren).

Die C-/C++-Laufzeitbibliotheken des CRTE werden nicht mit der RESOLVE-Option angegeben. Diese werden implizit auf Grund der Angaben in den Optionen RUNTIME-LANGUAGE, STDLIB, TOOLSLIB und RUNTIME-ENVIRONMENT in der richtigen Reihenfolge und Ausprägung eingebunden.

Automatische Template-Instanziierung

Im Gegensatz zur INCLUDE-Option kann mit der RESOLVE-Option die Template-Instanziierung gesteuert werden. Standardmäßig werden Module innerhalb von RESOLVE-Bibliotheken vom Prälinker nicht instanziiert. Siehe die Angabe INSTANTIATE.

RESOLVE = *UNCHANGED
Es gelten die Angaben der MODIFY-BIND-PROPERTIES-Anweisungen seit dem letzten START-LLM-CREATION=*YES.

RESOLVE = [*AUTOLINK](...)

LIBRARY = list-poss(40): <filename 1..54> / *LINK(LINK-NAME = <filename 1..8>)

Es wird der katalogisierte Dateiname oder der Linkname einer PLAM-Bibliothek angegeben, die per Autolink-Verfahren durchsucht werden soll. Werden mehrere Bibliotheken in einer Liste angegeben, so entspricht dies einer RESOLVE-BY-AUTOLINK-Anweisung mit einer Liste von RESOLVE-Bibliotheken. Die Liste darf maximal 40 Bibliotheken enthalten (Binder-Einschränkung).

INSTANTIATE = *NO / *YES / *IGNORE
Diese Option ist nur bei automatischer Template-Instanziierung durch den Prälinker relevant und steuert, ob die mit der RESOLVE-Option angegebenen Bibliotheken nicht instanziiert (Voreinstellung *NO), instanziiert (*YES) oder vom Prälinker überhaupt nicht berücksichtigt (*IGNORE) werden sollen. Während eine Bibliothek bei den Angaben *NO und *YES beim Instanziieren berücksichtigt, d.h. nach bereits vorhandenen Definitionen durchsucht wird, wird sie bei Angabe von *IGNORE vom Prälinker inhaltlich nicht betrachtet und lediglich im späteren Bindeprozess berücksichtigt.

> **i** Wenn dieselbe Bibliothek sowohl in der RESOLVE- als auch in der INCLUDE-Option angegeben wird, ist Folgendes zu beachten: Sobald für eine Bibliothek in einer RESOLVE-Option INSTANTIATE=*NO (Voreinstellung) oder *IGNORE angegeben wird, führt der Prälinker für sämtliche Elemente dieser Bibliothek keine Template-Instanziierungen durch, auch wenn die Elemente explizit mit der INCLUDE-Option eingebunden werden.

ADD-OPTION = *UNCHANGED / *NONE / <c-string 1..1800 with-low>
Mit <c-string> können zusätzlich zum LIBRARY-Operanden weitere Operanden der BINDER-Anweisung RESOLVE-BY-AUTOLINK angegeben werden. Mehrere Operanden müssen durch Kommas getrennt werden:

'operand1,operand2,...'
Die Operanden werden ohne SDF-Analyse direkt an den BINDER durchgereicht. Eingabebeispiel siehe BIND-Anweisung ("BIND").

Automatische Template-Instanziierung

Die mit ADD-OPTION angegebenen Operanden werden derzeit nicht bei der automatischen Template-Instanziierung durch den Prälinker, sondern nur beim nachfolgenden Binden durch den BINDER berücksichtigt. Die Operanden sollten nicht die Art, Anzahl oder Reihenfolge der einzubindenden Module beeinflussen, da unterschiedliche Voraussetzungen beim Prälinker- und Bindelauf ggf. zu Duplikaten oder unbefriedigten Externverweisen führen können.

RESOLVE = *NONE
Der Optionswert *NONE löscht nur die jeweilige RESOLVE-Kette, veranlasst jedoch nicht die komplette Aktion wie bei START-LLM-CREATION=YES. NONE ist der Compiler-Default, d.h. ein Wert nach dem Start des Compilers oder nach dem RESET-TO-DEFAULT.

ADD-PRELINK-FILES =
Diese Option ist nur bei automatischer Template-Instanziierung durch den Prälinker relevant.
Mithilfe dieser Option können PLAM-Bibliotheken insgesamt bzw. einzelne Module daraus angegeben werden, die bei der Bestimmung der zu generierenden Instanzen vom Prälinker in folgender Weise berücksichtigt werden:

- Wenn die angegebene Bibliothek die Definition einer Template-Einheit (Funktion oder statisches Datenelement) enthält, wird keine Instanz generiert, die hierzu ein Duplikat ist.
- In der angegebenen Bibliothek selbst werden keine Instanziierungen durchgeführt. Das heißt, wenn die Bibliothek Instanzen für Template-Einheiten benötigt (externe Referenzen), werden diese nicht generiert.

Wenn mit der BIND-Anweisung sowohl ein Prälinker- als auch ein Bindelauf gestartet wird, werden die mit ADD-PRELINK-FILES angegebenen Bibliotheken bzw. Module beim Binden nicht berücksichtigt.

Problemstellung

Die Module in den Bibliotheken PLAM.X und PLAM.Y enthalten Definitionen derselben Template-Instanzen. Wenn die Module der beiden Bibliotheken jeweils mit der Option ACTION=*PRELINK in der BIND-Anweisung 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 ADD-PRELINK-FILES zur Verfügung.

Lösung

Zunächst werden die Module der Bibliothek PLAM.X vorinstanziiert:

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

Anschließend werden die Module der Bibliothek PLAM.Y vorinstanziiert. Dabei wird mit der Option ADD-PRELINK-FILES bekanntgegeben, dass der Prälinker die Bibliothek PLAM.X berücksichtigen muss und keine Duplikate zu PLAM.X generiert.

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

ADD-PRELINK-FILES = *UNCHANGED
Es gelten die Angaben der MODIFY-BIND-PROPERTIES-Anweisungen seit dem letzten START-LLM-CREATION=*YES.

ADD-PRELINK-FILES = *NONE
Der Optionswert *NONE löscht nur die jeweilige ADD-PRELINK-FILES-Kette, veranlasst jedoch nicht die komplette Aktion wie bei START-LLM-CREATION=YES. NONE ist der Compiler-Default, d.h. ein Wert nach dem Start des Compilers oder nach dem RESET-TO-DEFAULT.

ADD-PRELINK-FILES = list-poss: *LIBRARY-ELEMENT(...)
Es werden analog zur INCLUDE-Option (bzw. zur INCLUDE-MODULES-Anweisung des BINDER) Bibliothekselemente angegeben, die vom Prälinker beim Vorinstanziieren berücksichtigt werden sollen.

LIBRARY = <filename 1..54>
<filename> ist der Name einer PLAM-Bibliothek.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>

<filename> ist der Linkname einer PLAM-Bibliothek. Der Linkname muss vor Aufruf des Compilers mit dem ADD-FILE-LINK-Kommando dem Bibliotheksnamen zugeordnet worden sein.

ELEMENT = *ALL(...)
Es werden alle Module aus der mit LIBRARY= angegebenen PLAM-Bibliothek beim Vorinstanziieren berücksichtigt.

VERSION = *HIGHEST-EXISTING
Enthält die Elementangabe keine Versionsbezeichnung, nimmt der Compiler die Elemente mit den jeweils höchsten Versionen.

VERSION = <composed-name 1..24 with-under>
Der Compiler nimmt das Element mit der angegebenen Version.

ELEMENT = <composed-name 1..64 with-under>(...)
<composed-name> bezeichnet den vollqualifizierten Namen eines Moduls aus der mit LIBRARY= angegebenen PLAM-Bibliothek.

VERSION = *HIGHEST-EXISTING

Enthält die Elementangabe keine Versionsbezeichnung, nimmt der Compiler das Element mit der höchsten Version.

VERSION = <composed-name 1..24 with-under>
Der Compiler nimmt das Element mit der angegebenen Version.

ADD-PRELINK-FILES = list-poss: *LIBRARY(...)
Die angegebenen Bibliotheken werden vom Prälinker wie RESOLVE-Bibliotheken behandelt, d.h. es werden nur diejenigen Module aus den angegebenen PLAM-Bibliotheken beim Vorinstanziieren berücksichtigt, aus denen beim eigentlichen Binden unbefriedigte Externverweise befriedigt werden könnten.

LIBRARY-NAME = <filename 1..54>
<filename> ist der Name einer PLAM-Bibliothek.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>

<filename> ist der Linkname einer PLAM-Bibliothek. Der Linkname muss vor Aufruf des Compilers mit dem ADD-FILE-LINK-Kommando dem Bibliotheksnamen zugeordnet worden sein.

MAX-INSTANTIATE-ITER = 30 / *UNCHANGED / <integer 0..100>
Diese Option ist nur bei automatischer Template-Instanziierung durch den Prälinker relevant und legt die maximale Anzahl der Prälinker-Durchläufe fest. Voreingestellt ist 30. Wenn für <integer> der Wert 0 angegeben wird, ist die Anzahl der Prälinker-Durchläufe nicht limitiert.

TEMPLATE-DEF-LIST = *UNCHANGED / *YES / *NO
Diese Option für die automatische Template-Instanziierung schaltet während der internen Nachübersetzungsphase eine Kommunikationstechnik zwischen Frontend und Prälinker über eine Definitionsliste ein bzw. aus (siehe dazu auch "Automatische Instanziierung").

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

Mit <c-string> kann eine zusätzliche BINDER-Anweisung angegeben werden. Mehrere BINDER-Anweisungen werden als eine Liste von c-strings angegeben:
(*'anweisung1','anweisung2','...'*).
Die Anweisungen werden ohne SDF-Analyse direkt an den BINDER durchgereicht.

Automatische Template-Instanziierung

Die mit ADD-STATEMENT angegebenen BINDER-Anweisungen werden derzeit nicht bei der automatischen Template-Instanziierung durch den Prälinker, sondern nur beim nachfolgenden Binden durch den BINDER berücksichtigt. Die Anweisungen sollten nicht die Art, Anzahl oder Reihenfolge der einzubindenden Module beeinflussen, da unterschiedliche Voraussetzungen beim Prälinker- und Bindelauf ggf. zu Duplikaten oder unbefriedigten Externverweisen führen können.

RUNTIME-LANGUAGE =

Auf Grund dieser Option werden die richtigen, den C- bzw. C++-Sprachmodi entsprechenden C-/C++-Laufzeitsysteme des CRTE automatisch eingebunden.

RUNTIME-LANGUAGE = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-BIND-PROPERTIES-Anweisung.

RUNTIME-LANGUAGE = *C
Es wird das C-Laufzeitsystem eingebunden.
Mit der Option STDLIB wird die Art festgelegt, in der das C-Laufzeitsystem eingebunden wird.

RUNTIME-LANGUAGE = *CPLUSPLUS(...)
Neben dem C-Laufzeitsystem, das immer benötigt wird, werden zusätzlich C++-Bibliotheken und -Laufzeitsysteme des CRTE eingebunden. Mit der Option STDLIB wird die Art festgelegt, in der diese CRTE-Bibliotheken eingebunden werden.

MODE = *UNCHANGED / *LATEST / *2017 / *V2-COMPATIBLE / *V3-COMPATIBLE
Abhängig davon, in welchem Modus des Compilers die Benutzermodule erzeugt wurden, werden entsprechende CRTE-Bibliotheken beim Binden benötigt.

*UNCHANGED:

Es gilt die Angabe der letzten MODIFY-BIND-PROPERTIES-Anweisung.

*LATEST:

Diese Angabe entspricht der Angabe des letzten unterstützten Standards und kann sich in zukünftigen Compiler-Versionen ändern. In dieser Version des Compilers ist die Angabe identisch zu *2017.

*2017:

Benötigt wird die C++ 2017-Bibliothek (SYSLNK.CRTE.CXX01).

*V2-COMPATIBLE:

Benötigt werden das Cfront-C++-Laufzeitsystem (SYSLNK.CRTE.CFCPP) und und die Cfront-C++-Bibliothek für komplexe Mathematik und stromorientierte Ein-/Ausgabe (SYSLNK.CRTE.CPP).
Diese Angabe entspricht der Angabe *CPP des C/C++ V3-Compilers.

*V3-COMPATIBLE:

Benötigt werden das C++ V3-Laufzeitsystem (SYSLNK.CRTE.RTSCPP) und die C++ V3-Bibliothek (SYSLNK.CRTE.STDCPP). Das Binden der im C++ V3-Modus nutzbaren Bibliothek Tools.h++ wird mit einer eigenen Option TOOLSLIB gesteuert.
Diese Angabe entspricht der Angabe *ANSI bzw. *STRICT-ANSI des C/C++ V3-Compilers.

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

Mit dieser Option wird festgelegt, auf welche Art die Externverweise auf die den Optionen RUNTIME-LANGUAGE entsprechenden C/C++-Bibliotheken des CRTE aufgelöst werden. Die Angaben *DYNAMIC UND *STATIC haben derzeit nur eine unterschiedliche Auswirkung auf das Einbinden des C-Laufzeitssystems.
Die C++-Bibliotheken werden bei den Angaben *DYNAMIC und *STATIC identisch behandelt, nämlich immer komplett („statisch“) eingebunden.

*UNCHANGED:

Es gilt die Angabe der letzten MODIFY-BIND-PROPERTIES-Anweisung.

*DYNAMIC:

Es werden alle Externverweise auf die CRTE-Bibliotheken befriedigt. Im Falle des C-Laufzeitsystems wird nur ein Adaptermodul aus der Bibliothek SYSLNK.CRTE.PARTIAL-BIND fest eingebunden und erst zum Ablaufzeitpunkt das vorgeladene C-Laufzeitsystem zum Programm konnektiert.

*DYNAMIC-COMPLETE:

Diese Option stellt eine Variante des dynamischen Bindens dar, die die Complete-Partial-Bind-Bibliotheken von CRTE verwendet.
Im C++ 2017-Modus ist diese Angabe nicht erlaubt.
Die externen Verweise werden aus der Bibliothek SYSLNK.CRTE.COMPL befriedigt. Im C++ V3-Modus, d.h. bei Angabe von RUNTIME-LANGUAGE=*CPLUSPLUS(MODE=*V3-COMPATIBLE), wird statt der Bibliotheken SYSLNK.CRTE.RTSCPP und SYSLNK.CRTE.STDCPP die Bibliothek SYSLNK.CRTE.CPP-COMPL verwendet.

> **i** Im CFRONT-C++-Modus wird die Option STDLIB=DYNAMIC-COMPLETE auf STDLIB=DYNAMIC zurückgesetzt. Insbesondere wird die Technik Complete-Partial-Bind für den CFRONT-C++-Modus nicht unterstützt. Eine ausführliche Beschreibung der Complete-Partial-Bind-Bibliotheken finden Sie im Handbuch „CRTE” [4].

*STATIC:

Es werden alle Externverweise auf die CRTE-Bibliotheken befriedigt.
Im Falle des C-Laufzeitsystems werden alle Einzelmodule aus der Bibliothek SYSLNK.CRTE fest eingebunden.

*NONE:

Die Externverweise auf die CRTE-Bibliotheken bleiben offen und werden entweder in einem weiteren Bindelauf durch festes Einbinden (BINDER bzw. BIND-Anweisung) befriedigt oder erst zum Ablaufzeitpunkt durch dynamisches Binden mit dem DBL.

> **i** Beim Offenlassen der Externbezüge (STDLIB=*NONE) ist Folgendes zu beachten: Bei der automatischen Template-Instanziierung durch den Prälinker wird die Standard-C++-Bibliothek (SYSLNK.CRTE.STDCPP bzw. SYSLNK.CRTE.CXX01) nicht mitbetrachtet, womit die Gefahr von Duplikat-Definitionen besteht. Ausführlichere Informationen zu dieser Problematik finden Sie im Abschnitt „Automatische Instanziierung“.

TOOLSLIB = *UNCHANGED / *YES / *NO
Bei Angabe von *YES wird die nur in C++ V3 nutzbare Bibliothek Tools.h++ eingebunden (SYSLNK.CRTE.TOOLS). Bei Angabe der Option STDLIB=DYNAMIC-COMPLETE wird statt SYSLNK.CRTE.TOOLS die Bibliothek SYSLNK.CRTE.CPP-COMPL verwendet. Bei Angabe von *NO bleiben die Externverweise auf die Bibliothek Tools.h++ offen.

TEST-SUPPORT = *UNCHANGED / *YES / *NO
Mit dieser Option wird gesteuert, ob die bei der Übersetzung erzeugten LSD-Informationen für die Dialogtesthilfe AID im gebundenen Modul abgespeichert werden sollen. Die Angabe wird als TEST-SUPPORT-Operand der BINDER-Anweisung SAVE-LLM abgesetzt.

LISTING = *UNCHANGED / *NONE / *SYSLST / <filename 1..54>
Mit dieser Option können analog zum MAP-Operanden der BINDER-Anweisung SAVE-LLM Standardlisten des BINDER angefordert werden. Die Binderlisten werden über die Systemdatei SYSLST ausgegeben oder in die mit <filename> angegebene katalogisierte Datei.

Bei START-LLM-CREATION=*YES wird das Listing auf *NONE zurückgesetzt.

3.2.2.9 Zusammenspiel der Anweisungen MODIFY-BIND-PROPERTIES und BIND

Es können mehrere MODIFY-BIND-PROPERTIES-Anweisungen in einen Bindeprozess einbezogen werden. Die mit den RESOLVE-, INCLUDE- und ADD-PRELINK-FILES-Optionen angegebenen Bibliotheken werden solange gesammelt und sind auch über nachfolgende BIND-Anweisungen hinaus gültig, bis die Option START-LLM-CREATION auf *YES gesetzt wird.

Bei START-LLM-CREATION=*YES wird auch das Listing auf *NONE zurückgesetzt.

START-LLM-CREATION ist automatisch nur dann auf *YES gesetzt, wenn die MODIFY-BIND-PROPERTIES-Anweisung zum ersten Mal seit Aufruf des Compilers oder zum ersten Mal nach einer RESET-TO-DEFAULT-Anweisung angegeben wird. Ab der zweiten MODIFY-Anweisung gilt der Wert *NO.

Eine Liste von Bibliotheken innerhalb einer RESOLVE-Option hat die gleiche Bedeutung wie eine Liste von Bibliotheken innerhalb einer RESOLVE-BY-AUTOLINK-Anweisung des BINDER. Eine weitere MODIFY-BIND-PROPERTIES-Anweisung mit einer Option RESOLVE wird als eine eigene RESOLVE-BY-AUTOLINK-Anweisung des BINDER abgesetzt und separat abgearbeitet.

3.2.2.10 MODIFY-CIF-PROPERTIES

Alias-Namen: SET-CIF-PROPERTIES
MODIFY-INFO-FILE-PROPERTIES
SET-INFO-FILE-PROPERTIES

Mit dieser Anweisung kann der Compiler veranlasst werden, ein CIF (Compilation Information File) zu erzeugen. Das CIF kann Informationen zu allen Übersetzungslisten enthalten. Das CIF wird pro übersetztes Quellprogramm in eine eigene Datei geschrieben, die dann mit dem globalen Listengenerator weiterverarbeitet werden kann (siehe "Steuerung des globalen Listengenerators").

```
MODIFY-CIF-PROPERTIES

 CONSUMER= *UNCHANGED / *NONE / *ALL / list-poss(9): *BY-LISTING-PROPERTIES / *OPTIONS /
                          *SOURCE / *PREPROCESSING-RESULT / *DATA-ALLOCATION-MAP /
                          *CROSS-REFERENCE / *PROJECT-INFORMATION / *ASSEMBLER-CODE /
                          *SUMMARY
,INCLUDE-INFORMATION= *UNCHANGED / *NONE / *ALL / *USER-INCLUDES-ONLY
,OUTPUT= *UNCHANGED / *NONE / *STD-FILE / *SOURCE-LOCATION / <filename 1..54> /
          <posix-pathname> / *LIBRARY-ELEMENT(...)
    *LIBRARY-ELEMENT(...)
        |       LIBRARY= *STD-LIBRARY / *SOURCE-LIBRARY / <filename 1..54>
        |       ,ELEMENT= *STD-ELEMENT(...) / <composed-name 1..64 with-under>(...)
        |           *STD-ELEMENT(...)
        |               |       VERSION= *UPPER-LIMIT / <composed-name 1..24 with-under>
        |           <composed-name 1..64 with-under>(...)
        |               |       VERSION= *UPPER-LIMIT / <composed-name 1..24 with-under>
```

CONSUMER = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-CIF-PROPERTIES-Anweisung.

CONSUMER = *NONE
Es wird kein permanentes CIF erzeugt.
Wenn lokale Listen angefordert werden (mit MODIFY-LISTING-PROPERTIES), wird für deren Erstellung ein temporäres CIF erzeugt (Präfix #T), das bei TASK-Ende gelöscht wird.

CONSUMER = *ALL
Es wird ein CIF erzeugt, das Informationen für alle Listen enthält, die abhängig von den jeweils durchlaufenen Compilerkomponenten (PREPROCESS, CHECK-SYNTAX, COMPILE) generiert werden können.

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

Es wird ein CIF erzeugt, das Informationen zu den angegebenen Listen enthält.

*BY-LISTING-PROPERTIES: Es werden CIF-Informationen für alle Listen erzeugt, die mit der MODIFY-LISTING-PROPERTIES-Anweisung angefordert werden.

INCLUDE-INFORMATION = *UNCHANGED / *NONE / *ALL / *USER-INCLUDES-ONLY
Mit dieser Option lässt sich steuern, ob und aus welchen Include-Dateien CIF-Informationen für die Quellprogramm-, Präprozessor- und Querverweisliste generiert werden. Standardmäßig werden die benutzereigenen Include-Dateien und nicht die Standard-Include-Dateien im CIF berücksichtigt.

OUTPUT = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-CIF-PROPERTIES-Anweisung.

OUTPUT = *NONE
Es wird kein permanentes CIF erzeugt.

OUTPUT = *STD-FILE
Das CIF wird in eine katalogisierte BS2000-Datei geschrieben. Der Name dieser Datei wird aus dem Namen des Quellprogramms abgeleitet:

Quelle	*SYSDTA	BS2000-Datei	PLAM-Bibliothek	POSIX-Datei
Standardname	CSTDCIF.CIF	datei.CIF	bib-elem.CIF	datei.CIF

Wenn das Quellprogramm in einer PLAM-Bibliothek steht, werden im Standard-Dateinamen der Bibliotheks- und Elementname der Quelle verwendet und mit einem Bindestrich verbunden (bib-elem). Die Regeln zur Bildung der Standardnamen durch den Compiler sind ausführlich im Abschnitt „Standardnamen für Ausgabebehälter“ dargestellt.

OUTPUT = *SOURCE-LOCATION
Ausgabeort und Name des CIF werden wie folgt aus dem Ort und Namen des Quellprogramms abgeleitet:

Quelle	*SYSDTA	BS2000-Datei	PLAM-Bibliothek	POSIX-Datei
Ausgabeort	BS2000-Datei	BS2000-Datei	Bibliothek der Quelle	Dateiverzeichnis der Quelle
Standardname	CSTDCIF.CIF	datei.CIF	elem.CIF (Typ H)	datei.cif

Die Regeln zur Bildung der Standardnamen durch den Compiler sind ausführlich im Abschnitt „Standardnamen für Ausgabebehälter“ dargestellt.

OUTPUT = <filename 1..54>

Das CIF wird in eine katalogisierte BS2000-Datei mit dem angegebenen Namen geschrieben. Diese Angabe ist bei der Übersetzung mehrerer Quellprogramme unzulässig.

OUTPUT = <posix-pathname>
Das CIF wird in das POSIX-Dateisystem geschrieben.

Als <posix-pathname> ist sowohl ein Dateiname als auch ein Dateiverzeichnis zugelassen. Zur Beschreibung des Begriffs <posix-pathname> siehe "Compiler-Ein-/Ausgaben im POSIX-Dateisystem".

Bei der Angabe eines Dateinamens wird das CIF unter diesem Namen abgelegt. Die Angabe eines Dateinamens ist unzulässig, wenn mehrere Quellprogramme mit einer Anweisung übersetzt werden.

Bei der Angabe eines Dateiverzeichnisnamens *dvz* wird das CIF für jedes übersetzte Quellprogramm unter dem Standardnamen *quelldatei.cif* in das Dateiverzeichnis *dvz* geschrieben (siehe auch Abschnitt „Standardnamen für Ausgabebehälter“).

Die mit <posix-pathname> angegebenen Dateiverzeichnisse müssen bereits existieren.CIF-Dateien können im POSIX-Subsystem mit dem globalen Listengenerator `cclistgen` weiterverarbeitet werden.

OUTPUT = *LIBRARY-ELEMENT(...)
Es wird angegeben, in welcher PLAM-Bibliothek (LIBRARY=) und unter welchem Elementnamen (ELEMENT=) das CIF abgelegt werden soll. Die Elemente werden unter dem Typ H abgespeichert.

LIBRARY = *STD-LIBRARY
Das CIF wird standardmäßig in die Bibliothek SYS.PROG.LIB geschrieben.

LIBRARY = *SOURCE-LIBRARY
Das CIF wird in die PLAM-Bibliothek geschrieben, in der das Quellprogramm steht. Die Angabe *SOURCE-LIBRARY ist unzulässig, wenn das Quellprogramm aus einer katalogisierten BS2000-Datei, einer POSIX-Datei oder über SYSDTA eingelesen wird.

LIBRARY = <filename 1..54>
Das CIF wird in einer PLAM-Bibliothek mit dem angegebenen Namen abgelegt.

ELEMENT = *STD-ELEMENT(...)

Standardmäßig wird der Elementname des CIF aus dem Namen des Quellprogramms abgeleitet:

Quelle	*SYSDTA	BS2000-Datei	PLAM-Bibliothek	POSIX-Datei
Standardname	CSTDCIF.CIF	datei.CIF	elem.CIF	datei.CIF

Die Regeln zur Bildung der Standardnamen durch den Compiler sind ausführlich im Abschnitt „Standardnamen für Ausgabebehälter“ dargestellt.

VERSION = *UPPER-LIMIT
Enthält die Elementangabe keine Versionsbezeichnung, erhält das Element die höchstmögliche Versionsnummer.

VERSION = <composed-name 1..24 with-under>
Das CIF wird in das Element mit der angegebenen Versionsbezeichnung geschrieben.

ELEMENT = <composed-name 1..64 with-under>(...)
Das CIF wird in ein Bibliothekselement (Typ H) mit dem angegebenen Namen geschrieben. Diese Angabe ist bei der Übersetzung mehrerer Quellprogramme unzulässig.

VERSION = *UPPER-LIMIT / <composed-name 1..24 with-under>
Die Version kann in gleicher Weise angegeben werden, wie oben unter ELEMENT=*STD-ELEMENT(...) beschrieben.

3.2.2.11 MODIFY-DIAGNOSTIC-PROPERTIES

Alias-Name: SET-DIAGNOSTIC-PROPERTIES

Mit dieser Anweisung kann man eine eigene Liste der Compilermeldungen erzeugen und für diese Liste diverse Ausgabeziele festlegen. Außerdem lassen sich Meldungsgewichte festlegen und Fehlerbedingungen für den Abbruch einer Übersetzung vereinbaren.

MODIFY-DIAGNOSTIC-PROPERTIES

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

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

*NOTE(...)

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

*WARNING(...)

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

*ERROR(...)

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

*DEFAULT(...)

| MSGID= *ALL / list-poss: <alphanum-name 7..7>

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

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

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

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

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

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

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

```
,OUTPUT= *UNCHANGED / list-poss(10): *SYSOUT / *SYSLST / *STD-FILE / *SOURCE-LOCATION /
                *TO-LISTING /<filename 1..54> / <posix-pathname> / *LIBRARY-ELEMENT(...)
    *LIBRARY-ELEMENT(...)
        |    LIBRARY= *STD-LIBRARY / *SOURCE-LIBRARY / <filename 1..54> / *LINK(...)
        |        *LINK(...)
        |            |    LINK-NAME= <filename 1..8>
        |    ,ELEMENT= *STD-ELEMENT(...) / <composed-name 1..64 with-under>(...)
        |        *STD-ELEMENT(...)
        |            |    VERSION= *UPPER-LIMIT / <composed-name 1..24 with-under>
        |        <composed-name 1..64 with-under>(...)
        |            |    VERSION= *UPPER-LIMIT / <composed-name 1..24 with-under>
```

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

Mit dieser Option kann angegeben werden, ab welchem Gewicht die Compilermeldungen in die Meldungsliste aufgenommen werden sollen.

CHANGE-MSG-WEIGHT =

Mit dieser Option kann das vorgegebene Gewicht von Meldungen verändert werden, die das Compiler-Frontend ausgibt (beginnen mit CFE).
Alle Angaben werden gesammelt, auch über mehrere Anweisungen hinweg.

CHANGE-MSG-WEIGHT = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-DIAGNOSTIC-PROPERTIES-Anweisung.

CHANGE-MSG-WEIGHT = list-poss: *NOTE(...) / *WARNING(...) /*ERROR(...)
Das vorgegebene Gewicht von Meldungen wird auf das angegebene Gewicht geändert. Notes können bis zum Gewicht ERROR hochgestuft werden. Warnings können auf das Gewicht NOTE herabgestuft und auf das Gewicht ERROR hochgestuft werden. Errors können bis auf das Gewicht NOTE herabgestuft werden, dies allerdings nur dann, wenn sie in der Originalmeldung mit einem Stern gekennzeichnet sind: [*ERROR]. Das Fehlergewicht von Fatal Errors kann nicht verändert werden.

MSGID = list-poss: <alphanum-name 7..7>
<alphanum-name> ist der 7-stellige Meldungsschlüssel der Compilermeldung, deren Gewicht geändert werden soll.

CHANGE-MSG-WEIGHT = *DEFAULT(...)
Das Gewicht einer Meldung wird auf den ursprünglichen Wert zurückgesetzt.

MSGID = list-poss: <alphanum-name 7..7>
<alphanum-name> ist der 7-stellige Meldungsschlüssel der Compilermeldung, deren Gewicht auf das ursprüngliche Gewicht zurückgesetzt werden soll.

MSGID = *ALL
Es werden alle Meldungen auf das ursprüngliche Gewicht zurückgesetzt.

SUPPRESS-MSG = *UNCHANGED / *NONE / list-poss: *USE-BEFORE-SET / <alphanum-name 7..7>
Mit dieser Option können Compilermeldungen vom Gewicht NOTE oder WARNING angegeben werden, die in der Meldungsliste nicht aufgeführt werden sollen. Damit lässt sich die Meldungsliste auf die dem Benutzer wichtigen Meldungen beschränken.

*UNCHANGED

Es gilt die Angabe der letzten MODIFY-DIAGNOSTIC-PROPERTIES-Anweisung.

*NONE

Alle Compiler-Meldungen werden dem Wert von MINIMAL-MSG-WEIGHT entsprechend ausgegeben.

*USE-BEFORE-SET

Die Ausgabe von Warnings wird unterdrückt, wenn im Programm lokale `auto`-Variablen benutzt werden, bevor ihnen ein Wert zugewiesen wurde.

<alphanum-name 7..7>

Ist der Meldungsschlüssel einer Compilermeldung vom Gewicht NOTE oder WARNING, die nicht ausgegeben werden soll.

Beispiel

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

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

*UNCHANGED

Es gilt die Angabe der letzten MODIFY-DIAGNOSTIC-PROPERTIES-Anweisung.

50

Der Compiler bricht jede Übersetzung ab, die mehr als 49 Errors aufweist.

<integer 1..255>

Gibt die Fehlerzahl an, ab der der Übersetzungslauf abgebrochen werden soll.

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

Diese Option ist nur zusammen mit der Angabe LANGUAGE=...(STRICT=YES) (siehe MODIFY-SOURCE-PROPERTIES) sinnvoll verwendbar.

*WARNING ist voreingestellt und bewirkt die Ausgabe von Warnings, wenn Sprachkonstrukte benutzt werden, die zwar eine Abweichung vom relevanten C- bzw. C++-Standard, jedoch keine schwere Verletzung der dort festgelegten Sprachregeln darstellen (z.B. implementierungsspezifische Spracherweiterungen, siehe Abschnitt „Erweiterungen gegenüber ANSI-/ISO-C“ und Abschnitt „Erweiterungen gegenüber ANSI-/ISO-C++“).

Bei Angabe von *ERROR werden in solchen Fällen Errors ausgegeben. Schwerere Verletzungen führen automatisch zu Errors.

SHOW-COLUMN = *UNCHANGED / *YES / *NO
Diese Option legt fest, ob die Diagnosemeldungen des Compilers in kurzer oder in ausführlicher Form generiert werden.

*UNCHANGED

Es gilt die Angabe der letzten MODIFY-DIAGNOSTIC-PROPERTIES-Anweisung.

*YES

Zusätzlich zur Diagnosemeldung wird die Original-Quellprogrammzeile ausgegeben, in der die Fehlerstelle markiert ist (mit ^).

*NO

Es unterbleibt die Ausgabe der markierten Quellprogrammzeile.

SHOW-INCLUDES = *UNCHANGED / *YES / *NO
Bei Angabe von *YES werden die Namen der vom Präprozessor verwendeten Include-Dateien ausgegeben.

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

*UNCHANGED

Es gilt die Angabe der letzten MODIFY-DIAGNOSTIC-PROPERTIES-Anweisung.

*VERSION

Für jede bearbeitete Datei wird die Meldung CDR9402 ausgegeben. Beim Bindevorgang werden zusätzlich die genutzte CRTE-Version sowie die Namen der CRTE-Bibliotheken ausgegeben.

*MESSAGES

Derzeit ist diese Angabe nur in den Modi C++ V3 und C++ 2017 sinnvoll. Sie bewirkt, dass zusätzliche Informationen zur automatischen Template-Instanziierung durch den Prälinker auf SYSOUT ausgegeben werden.

GENERATE-ETR-FILE = *UNCHANGED / *NO / *ALL-INSTANTIATIONS / *ASSIGNED-INSTANTIATIONS
Mithilfe dieser Option kann eine ETR-Datei (ETR=Explicit Template Request) erstellt werden, die die Instanziierungs-Anweisungen für die verwendeten Templates enthält (siehe dazu auch Abschnitt „Generieren von expliziten Template-Instanziierungsanweisungen(ETR-Dateien)“). Der Dateiname wird vom Namen der Objekt-Datei abgeleitet und hat den Suffix .etr.

Der Standardwert der Option ist *NO, dh. es wird keine ETR-Datei erzeugt. Bei der Angabe *ALL-INSTANTIATIONS werden alle verwendeten Instanzen protokolliert, bei *ASSIGNED-INSTANTIATIONS nur diejenigen, die vom Prälinker dieser Datei zugeordnet werden und somit in der ii-Datei stehen.

OUTPUT = *UNCHANGED / list-poss(10): *SYSOUT / *SYSLST / *STD-FILE / *SOURCE-LOCATION / <filename 1..54> / <posix-pathname> / *LIBRARY-ELEMENT(...)
Mit der OUTPUT-Option können für die Meldungsliste diverse Ausgabeziele parallel vereinbart werden.

> ¡ Informationsmeldungen des Compilers (Meldungen ohne Meldungsgewicht) werden zusätzlich zu den mit dieser Option vereinbarten Ausgabezielen immer auch nach SYSOUT ausgegeben. Alle anderen Meldungen werden ausschließlich auf die angegeben Ausgabeziele ausgegeben.

OUTPUT = *SYSOUT
Standardmäßig werden die Übersetzungsmeldungen am Bildschirm (Systemdatei SYSOUT) ausgegeben.

OUTPUT = *SYSLST
Die Meldungsliste wird in die temporäre Systemdatei SYSLST geschrieben, von wo aus sie nach Ende der Task (LOGOFF) auf den Drucker ausgegeben wird.

OUTPUT = *STD-FILE
Die Meldungsliste wird in eine katalogisierte BS2000-Datei geschrieben. Der Name dieser Datei wird aus dem Namen des Quellprogramms abgeleitet:

Quelle	*SYSDTA	BS2000-Datei	PLAM-Bibliothek	POSIX-Datei
Standardname	CSTDDIAG.DIAG	datei.DIAG	bib-elem.DIAG	datei.DIAG

Wenn das Quellprogramm in einer PLAM-Bibliothek steht, werden im Standard-Dateinamen der Bibliotheks- und Elementname der Quelle verwendet und mit einem Bindestrich verbunden (bib-elem). Die Regeln zur Bildung der Standardnamen durch den Compiler sind ausführlich im Abschnitt „Standardnamen für Ausgabebehälter“ dargestellt.

OUTPUT = *SOURCE-LOCATION
Ausgabeort und Name der Meldungsliste werden wie folgt aus dem Ort und Namen des Quellprogramms abgeleitet:

Quelle	*SYSDTA	BS2000-Datei	PLAM-Bibliothek	POSIX-Datei
Ausgabeort	BS2000-Datei	BS2000-Datei	Bibliothek der Quelle	Dateiverzeichnis der Quelle
Standardname	CSTDDIAG.DIAG	datei.DIAG	elem.DIAG (Typ D)	datei.diag

Die Regeln zur Bildung der Standardnamen durch den Compiler sind ausführlich im Abschnitt „Standardnamen für Ausgabebehälter“ dargestellt.

OUTPUT = *TO-LISTING
Der Diagnose-Output wird an das Ende der Listing-Datei als eigenes Teil-Listing geschrieben. Die Meldungsliste wird nach dem Meldungsgewicht sortiert, wobei Meldungen ohne Gewicht (Informationsmeldungen des Compilers) nicht mit übernommen werden. Der Name der Ausgabedatei wird bestimmt durch den Wert der Option OUTPUT bei MODIFY-LISTING-PROPERTIES. Die dort beschriebenen Restriktionen bezüglich der Modi C++ V3 und C++ 2017 müssen beachtet werden.

OUTPUT = <filename 1..54>
Die Meldungsliste wird in eine katalogisierte BS2000-Datei mit dem angegebenen Namen geschrieben. Bei der Übersetzung mehrerer Quellprogramme ist diese Angabe nicht sinnvoll, weil die Datei jedes Mal überschrieben wird.

OUTPUT = <posix-pathname>
Die Meldungsliste wird in das POSIX-Dateisystem geschrieben.

Als <posix-pathname> ist sowohl ein Dateiname als auch ein Dateiverzeichnis zugelassen. Zur Beschreibung des Begriffs <posix-pathname> siehe "Compiler-Ein-/Ausgaben im POSIX-Dateisystem".

Bei der Angabe eines Dateinamens wird die Meldungsliste unter diesem Namen abgelegt.

Bei der Angabe eines Dateiverzeichnisnamens *dvz* wird die Meldungsliste für jedes übersetzte Quellprogramm unter dem Standardnamen *quelldatei.diag* in das Dateiverzeichnis *dvz* geschrieben (siehe auch Abschnitt „Standardnamen für Ausgabebehälter“).

Die mit <posix-pathname> angegebenen Dateiverzeichnisse müssen bereits existieren.Bei der Übersetzung mehrerer Quellprogramme ist die Angabe eines Dateinamens nicht sinnvoll, weil die Datei jedes Mal überschrieben wird.

OUTPUT = *LIBRARY-ELEMENT(...)
Es wird angegeben, in welcher PLAM-Bibliothek (LIBRARY=) und unter welchem Elementnamen (ELEMENT=) die Meldungsliste abgelegt werden soll. Die Elemente werden unter dem Typ D abgespeichert.

LIBRARY = *STD-LIBRARY
Die Meldungsliste wird standardmäßig in die Bibliothek SYS.PROG.LIB geschrieben.

LIBRARY = *SOURCE-LIBRARY
Die Meldungsliste wird in die PLAM-Bibliothek geschrieben, in der das Quellprogramm steht.
Die Angabe *SOURCE-LIBRARY ist unzulässig, wenn das Quellprogramm aus einer katalogisierten BS2000-Datei, einer POSIX-Datei oder über SYSDTA eingelesen wird.

LIBRARY = <filename 1..54>

Die Meldungsliste wird in die PLAM-Bibliothek mit dem angegebenen Namen geschrieben.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>

Statt eines katalogisierten Bibliotheksnamens kann auch ein Linkname angegeben werden. Der Linkname muss vor Aufruf des Compilers mit dem ADD-FILE-LINK-Kommando der PLAM-Bibliothek zugeordnet worden sein.

ELEMENT = *STD-ELEMENT(...)
Standardmäßig wird der Elementname der Meldungsliste aus dem Namen des Quellprogramms abgeleitet:

Quelle	*SYSDTA	BS2000-Datei	PLAM-Bibliothek	POSIX-Datei
Standardname	CSTDDIAG.DIAG	datei.DIAG	elem.DIAG	datei.DIAG

Die Regeln zur Bildung der Standardnamen durch den Compiler sind ausführlich im Abschnitt „Standardnamen für Ausgabebehälter“ dargestellt.

VERSION = *UPPER-LIMIT
Enthält die Elementangabe keine Versionsbezeichnung, erhält das Element die höchstmögliche Versionsnummer.

VERSION = <composed-name 1..24 with-under>
Die Meldungsliste wird in das Element mit der angegebenen Versionsbezeichnung geschrieben.

ELEMENT = <composed-name 1..64 with-under>(...)
Die Meldungsliste wird in ein Bibliothekselement (Typ D) mit dem angegebenen Namen geschrieben. Bei der Übersetzung mehrerer Quellprogramme ist die Angabe eines Elementnamens nicht sinnvoll, weil das Element jedes Mal überschrieben wird.

VERSION = *UPPER-LIMIT / <composed-name 1..24 with-under>
Die Version kann in gleicher Weise angegeben werden, wie oben unter ELEMENT=*STD-ELEMENT(...) beschrieben.

3.2.2.12 MODIFY-INCLUDE-LIBRARIES

Alias-Namen: MODIFY-INCLUDE-SEARCH
SET-INCLUDE-LIBRARIES
SET-INCLUDE-SEARCH

Mit dieser Anweisung wird festgelegt, welche Include-Bibliotheken und -Dateiverzeichnisse vom Compiler durchsucht werden sollen. Außerdem wird die Reihenfolge bestimmt, in der diese Bibliotheken und Dateiverzeichnisse durchsucht werden.

```
MODIFY-INCLUDE-LIBRARIES

 USER-INCLUDE-LIBRARY= *UNCHANGED / list-poss: *SOURCE-LIBRARY / *STANDARD-LIBRARY /
                                <filename 1..54> / <posix-pathname> / *LINK(...)
      *LINK(...)
            LINK-NAME= <filename 1..8>
 ,STD-INCLUDE-LIBRARY= *UNCHANGED / list-poss: *USER-INCLUDE-LIBRARY /
                          *STANDARD-LIBRARY / *SOURCE-LIBRARY / <filename 1..54> /
                          <posix-pathname> / *LINK(...)
      *LINK(...)
          |   LINK-NAME= <filename 1..8>
 ,CURRENT-LIBRARY= *UNCHANGED / *YES / *NO
```

USER-INCLUDE-LIBRARY =
Mit dieser Option können PLAM-Bibliotheken und POSIX-Dateiverzeichnisse angegeben werden, in denen die benutzereigenen Include-Dateien (angefordert mit `#include "..."`) stehen. Abhängig von der Option CURRENT-LIBRARY wird vor den mit USER-INCLUDE-LIBRARY angegebenen Bibliotheken und Dateiverzeichnissen zunächst die Bibliothek oder das Dateiverzeichnis derjenigen Quell- oder Include-Datei durchsucht, die die Include-Anweisung `#include "..."` enthält.

Wird die Option USER-INCLUDE-LIBRARY nicht angegeben, gilt folgende Voreinstellung:

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

Include-Dateien, die in der `#include`-Anweisung in Anführungszeichen angegeben werden (`#include "`*name*"), sucht der Compiler in der Bibliothek bzw. im Dateiverzeichnis des Quellprogramms und anschließend in den CRTE-Bibliotheken, die die Standard-Include-Dateien enthalten. In allen Sprachmodi wird die CRTE-Bibliothek $.SYSLIB.CRTE für die Suche zugewiesen. Im Sprachmodus Cfront-C++ wird vorher die CRTE-Bibliothek $.SYSLIB.CRTE.CPP zugewiesen. Im Sprachmodus C++ 2017 wird vorher die CRTE-Bibliothek $.SYSLIB.CRTE.CXX01 zugewiesen.

Wird die Option angegeben, ist die o.g. Voreinstellung ausgeschaltet und der Compiler durchsucht nur die explizit angegebenen Bibliotheken/Dateiverzeichnisse. Das heißt:
Wenn die Include-Dateien in der Bibliothek bzw. im Dateiverzeichnis des Quellprogramms und in den CRTE-Bibliotheken gesucht werden sollen, müssen die Optionswerte *SOURCE-LIBRARY und *STANDARD-LIBRARY explizit angegeben werden.
Die Bibliotheken/Dateiverzeichnisse werden in der Reihenfolge durchsucht, in der sie angegeben wurden.

USER-INCLUDE-LIBRARY = *UNCHANGED
Es gelten die Angaben der letzten MODIFY-INCLUDE-LIBRARY-Anweisung. Wenn seit dem Start des Compilers noch keine Werte verändert wurden, gelten die Standardeinstellungen des Compilers (*SOURCE-LIBRARY, *STANDARD-LIBRARY).

USER-INCLUDE-LIBRARY = *SOURCE-LIBRARY
Die angeforderte Include-Datei wird in der Bibliothek bzw. in dem Dateiverzeichnis gesucht, wo auch das Quellprogramm steht. Die Angabe *SOURCE-LIBRARY ist wirkungslos, wenn das Quellprogramm in einer katalogisierten BS2000-Datei steht und generell bei Quellprogrammeingabe über SYSDTA (siehe auch "Hinweise zur Eingabe über SYSDTA").

USER-INCLUDE-LIBRARY = *STANDARD-LIBRARY
In allen Sprachmodi wird die CRTE-Bibliothek $.SYSLIB.CRTE für die Suche zugewiesen. Im Sprachmodus Cfront-C++ wird vorher die CRTE-Bibliothek $.SYSLIB.CRTE.CPP zugewiesen. Im Sprachmodus C++ 2017 wird vorher die CRTE-Bibliothek $.SYSLIB.CRTE.CXX01 zugewiesen.

USER-INCLUDE-LIBRARY = <filename 1..54>
<filename> ist der Name der PLAM-Bibliothek, in der die angeforderten benutzereigenen Include-Dateien gesucht werden sollen.

USER-INCLUDE-LIBRARY = <posix-pathname>
<posix-pathname> ist der Name des POSIX-Dateiverzeichnisses, in dem die angeforderten benutzereigenen Include-Dateien gesucht werden sollen. Zur Beschreibung des Begriffs <posix-pathname> siehe "Compiler-Ein-/Ausgaben im POSIX-Dateisystem".

USER-INCLUDE-LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>

Statt eines katalogisierten PLAM-Bibliotheksnamens kann auch ein Linkname angegeben werden. <filename> ist der Linkname der zugewiesenen Include-Bibliothek. Er muss vor Aufruf des Compilers mit dem ADD-FILE-LINK-Kommando der PLAM-Bibliothek zugewiesen worden sein.

STD-INCLUDE-LIBRARY =
Mit dieser Option können PLAM-Bibliotheken und POSIX-Dateiverzeichnisse angegeben werden, in denen die Standard-Include-Dateien (angefordert mit `#include <...>`) stehen.

STD-INCLUDE-LIBRARY = *UNCHANGED

Es gelten die Angaben der letzten MODIFY-INCLUDE-LIBRARY-Anweisung. Wenn seit dem Start des Compilers noch keine Werte verändert wurden, gelten die Standardeinstellungen des Compilers (*USER-INCLUDE-LIBRARY, *STANDARD-LIBRARY).

STD-INCLUDE-LIBRARY = *USER-INCLUDE-LIBRARY
Die Suchreihenfolge für die Standard-Include-Dateien wird aus den Angaben in der USER-INCLUDE-LIBRARY-Option abgeleitet, wobei von dort nur die explizit angegebenen Bibliotheken/Dateiverzeichnisse berücksichtigt werden und nicht die Angaben *SOURCE-LIBRARY und *STANDARD-LIBRARY.

STD-INCLUDE-LIBRARY = *STANDARD-LIBRARY
In allen Sprachmodi wird die CRTE-Bibliothek $.SYSLIB.CRTE für die Suche zugewiesen. Im Sprachmodus Cfront-C++ wird vorher die CRTE-Bibliothek $.SYSLIB.CRTE.CPP zugewiesen. Im Sprachmodus C++ 2017 wird vorher die CRTE-Bibliothek $.SYSLIB.CRTE.CXX01 zugewiesen.
Beachten Sie die Hinweise zu den Standard-Include-Dateien bei Verwendung der POSIX-Bibliotheksfunktionen.

STD-INCLUDE-LIBRARY = <filename 1..54>
<filename> ist der Name der PLAM-Bibliothek, in der die angeforderten Standard-Include-Dateien gesucht werden sollen.

STD-INCLUDE-LIBRARY = <posix-pathname>
<posix-pathname> ist der Name des POSIX-Dateiverzeichnisses, in dem die angeforderten Standard-Include-Dateien gesucht werden sollen. Zur Beschreibung des Begriffs <posix-pathname> siehe "Compiler-Ein-/Ausgaben im POSIX-Dateisystem".
Diese Angabe ist für die Standard-Includes des CRTE nicht sinnvoll, da diese aus den CRTE-Bibliotheken genommen werden sollen.

STD-INCLUDE-LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>

Statt eines katalogisierten PLAM-Bibliotheksnamens kann auch ein Linkname angegeben werden. <filename> ist der Linkname der zugewiesenen Include-Bibliothek. Er muss vor Aufruf des Compilers mit dem ADD-FILE-LINK-Kommando der PLAM-Bibliothek zugewiesen worden sein.

CURRENT-LIBRARY = *UNCHANGED / *YES / *NO
Diese Option betrifft die Suche nach Include-Dateien, die mit `#include "..."` angefordert werden.
*YES: Standardmäßig werden die Include-Dateien zunächst in der Bibliothek oder dem Dateiverzeichnis derjenigen Quell- oder Include-Datei gesucht, die die `#include`-Anweisung enthält und anschließend erst in den mit der Option USER-INCLUDE-LIBRARY angegebenen Dateiverzeichnissen/Bibliotheken. Dies entspricht dem Verhalten des Compilers im POSIX.

Steht das Quellprogramm in einer katalogisierten BS2000-Datei, werden Include-Dateien unter den katalogisierten BS2000-Dateien gesucht.

*NO: Bei Angabe von *NO werden nur die mit USER-INCLUDE-LIBRARY angegebenen Dateiverzeichnissen /Bibliotheken durchsucht. Dies entspricht dem Verhalten der Vorgänger-Compiler C und C++ V2.2.

Standard-Include-Dateien für die POSIX-Bibliotheksfunktionen

Die für die Verwendung der POSIX-Bibliotheksfunktionen notwendigen Standard-Include-Dateien sind nicht in der Bibliothek $.SYSLIB.CRTE enthalten, sondern in der mit BS2000/OSD-BC ausgelieferten Bibliothek $.SYSLIB.POSIX-HEADER. Diese Bibliothek muss immer zusätzlich zur Bibliothek $.SYSLIB.CRTE angegeben werden, wenn das Programm POSIX-Funktionen verwendet. Außerdem muss vor Auftreten der ersten `#include`-Anweisung im Programm das Define `_OSD_POSIX` gesetzt sein. Dies ist z.B. sichergestellt, wenn die Definition bei der Übersetzung mit der DEFINE-Option in der MODIFY-SOURCE-PROPERTIES-Anweisung erfolgt.

Beispiele zur MODIFY-INCLUDE-LIBRARY-Anweisung

Beispiel 1

Das Quellprogramm steht in einer PLAM-Bibliothek namens PLAM.SOURCE. Als Sprachmodus ist erweitertes C11 eingeschaltet.

Das Quellprogramm enthält die Anweisung

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

Der Benutzer macht folgende Angaben:

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

Da die Option CURRENT-LIBRARY nicht angegeben wird, gilt CURRENT-LIBRARY=*YES.

Suchablauf:

Die Include-Datei `"incl.h"` wird der Reihe nach in folgenden Bibliotheken/Dateiverzeichnissen gesucht:

PLAM.SOURCE	(Auf Grund der Option CURRENT-LIBRARY=*YES die Bibliothek des Quellprogramms, das die `#include`-Anweisung enthält)
$.SYSLIB.CRTE	(Standard-Include-Bibliothek)
LIB1	(Bibliothek mit dem Namen LIB1)
LIB2	(Bibliothek mit dem Linknamen LIB2)
PLAM.SOURCE	(Bibliothek des Quellprogramms)
/home/user-incl	(POSIX-Dateiverzeichnis mit dem Namen /home/user-incl)

Beispiel 2

Das Quellprogramm steht in einer PLAM-Bibliothek namens PLAM.SRC. Als Sprachmodus ist Cfront-C++ eingeschaltet.

Der Benutzer macht folgende Angaben:

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

Die Suche nach den Include-Dateien gestaltet sich wie folgt:

`#include "...`	**`#include <...>`**

$XYZ.LIB PLAM.SRC LIB1	$XYZ.LIB LIB1 $.SYSLIB.CRTE.CPP $.SYSLIB.CRTE

Beispiel 3

Das Quellprogramm steht in dem POSIX-Dateiverzeichnis `/home/xy/src`. Als Sprachmodus ist erweitertes C11 eingeschaltet.

Der Benutzer macht folgende Angaben:

```
//MODIFY-INCLUDE-LIBRARIES -
//USER-INCLUDE-LIBRARY=(*SOURCE-LIBRARY, '/home/xy/incl1', *STANDARD-LIBRARY),-
//STD-INCLUDE-LIBRARY=(*STANDARD-LIBRARY, $.SYSLIB.POSIX-HEADER,-
//*USER-INCLUDE-LIBRARY,'/home/xy/incl2'),CURRENT-LIBRARY=*NO
```

Die Suche nach den Include-Dateien gestaltet sich wie folgt:

`#include "..."`	**`#include <...>`**
/home/xy/src /home/xy/incl1 $.SYSLIB.CRTE	$.SYSLIB.CRTE $.SYSLIB.POSIX-HEADER /home/xy/incl1 /home/xy/incl2

3.2.2.13 MODIFY-LISTING-PROPERTIES

Alias-Name: SET-LISTING-PROPERTIES

Mit dieser Anweisung kann man auswählen, welche Übersetzungslisten der Compiler erzeugen soll. Außerdem kann man das Layout beeinflussen und den Ausgabeort der Listen festlegen. Der Aufbau der Compilerlisten wird im Abschnitt „Beschreibung der Listenbilder“ anhand von Beispielen erläutert.

```
MODIFY-LISTING-PROPERTIES

OPTIONS= *UNCHANGED / *YES / *NO

,SOURCE= *UNCHANGED / *NO / [*YES](...)

    *YES(...)

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

,PREPROCESSING-RESULT= *UNCHANGED / *NO / *YES

,DATA-ALLOCATION-MAP= *UNCHANGED / *NO / [*YES](...)

    *YES(...)

        |    STRUCTURE-LEVEL= *UNCHANGED / *NONE / *MAX / <integer 0..256>

,CROSS-REFERENCE= *UNCHANGED / *NO / [*YES](...)

    *YES(...)

        |    PREPROCESSING-INFO= *UNCHANGED / *YES / *NO

        |    ,TYPES= *UNCHANGED / *YES / *NO

        |    ,VARIABLES= *UNCHANGED / *YES / *NO

        |    ,FUNCTIONS= *UNCHANGED / *YES / *NO

        |    ,LABELS= *UNCHANGED / *YES / *NO

        |    ,TEMPLATES= *UNCHANGED / *YES / *NO

        |    ,ORDER= *UNCHANGED / *STD / list-poss(6): *PREPROCESSING-INFO / *TYPES /

        |        *VARIABLES / *FUNCTIONS / *LABELS / *TEMPLATES

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

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

,SUMMARY= *UNCHANGED / *YES / *NO

,LAYOUT= *UNCHANGED / *FOR-NORMAL-PRINT(...) / *FOR-ROTATION-PRINT(...)

    *FOR-NORMAL-PRINT(...)

        |    LINE-SIZE= *UNCHANGED / *STD / <integer 120..255>
```

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

OPTIONS = *UNCHANGED / *YES / *NO

*YES: Der Compiler erzeugt eine Liste aller voreingestellten und vom Benutzer angegebenen Compiler-Optionen.

SOURCE = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.

SOURCE = *NO
Es wird keine Quellprogramm-/Fehlerliste erzeugt.

SOURCE = *YES(...)
Es wird eine Quellprogramm-/Fehlerliste erzeugt (nicht bei PREPROCESS).

MINIMAL-MSG-WEIGHT = *NOTE / *WARNING / *ERROR / *FATAL
Mit diesem Operanden lässt sich bestimmen, ab welchem Gewicht die Fehlermeldungen in der Quellprogrammliste stehen sollen.

Achtung:
Mit dieser Suboption kann die Menge der ausgegebenen Meldungen gegenüber der analogen Suboption von MODIFY-DIAGNOSTIC-PROPERTIES nur eingeschränkt werden, z. B. von WARNING auf ERROR.
Falls bei MODIFY-DIAGNOSTIC-PROPERTIES für MINIMAL-MSG-WEIGHT = *WARNING (Defaulteinstellung) angegeben wurde, kann mit MODIFY-LISTING-PROPERTIES nicht erreicht werden, dass Notes ausgegeben werden.

Beispiele

1. In der Quellprogrammliste sollen Fehlermeldungen ab dem Gewicht NOTE ausgegeben werden:

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

2. Auf die Systemdatei SYSOUT sollen Fehlermeldungen ab dem Gewicht NOTE ausgegeben werden, in der Quellprogrammliste ab dem Gewicht WARNING:

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

(Defaulteinstellung)

> Nach Erreichen von MAX-ERROR-NUMBER wird keine Quellprogramm-Information in der Quellprogramm-/Fehlerliste ausgegeben. In einem solchen Fall kann über dieses Listing kein Bezug zu Fehlerstellen mehr festgestellt werden.

PREPROCESSING-RESULT = *UNCHANGED / *NO / *YES

*UNCHANGED

Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.

*NO

Der Compiler erzeugt keine Präprozessorliste.

*YES

Der Compiler erzeugt eine Präprozessorliste.

DATA-ALLOCATION-MAP = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.

DATA-ALLOCATION-MAP = *NO
Der Compiler erzeugt keine Adressliste.

DATA-ALLOCATION-MAP = *YES(...)
Der Compiler erzeugt eine Adressliste (nur bei COMPILE).

STRUCTURE-LEVEL = *UNCHANGED / *NONE / *MAX / <integer 0..256>

*UNCHANGED

Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.

*NONE

Strukturelemente werden in der Adressliste nicht abgebildet.

*MAX

Es werden Strukturelemente bis zur maximalen Schachtelungstiefe (256) in der Adressliste abgebildet.

<integer 0..256>

Es werden Strukturelemente bis zu der mit <integer> angegebenen Schachtelungstiefe in der Adressliste abgebildet. Bei Angabe der Schachtelungstiefe 0 werden keine Strukturelemente ausgegeben (entspricht STRUCTURE-LEVEL=*NONE).

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

CROSS-REFERENCE = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.

CROSS-REFERENCE = *NO
Der Compiler erzeugt keine Querverweisliste.

CROSS-REFERENCE = *YES(...)

Der Compiler erzeugt eine Querverweisliste (nicht bei PREPROCESS). Die Querverweisliste enthält in jedem Fall ein FILETABLE-Teil mit den Namen aller Dateien, Bibliotheken und Elemente, die der Compiler als Quellen verwendet.
Die Querverweisliste wird pro Übersetzungseinheit, d.h. modullokal, angelegt. Um eine modulübergreifende Querverweisliste zu erhalten, können mit der Anweisung MODIFY-CIF-PROPERTIES CIF-Informationen erzeugt und anschließend mit dem globalen Listengenerator weiterverarbeitet werden.

PREPROCESSING-INFO = *UNCHANGED / *YES / *NO
Die Querverweisliste enthält wahlweise eine Liste der vom Präprozessor bearbeiteten Namen.

TYPES = *UNCHANGED / *YES / *NO
Die Querverweisliste enthält wahlweise eine Liste der benutzerdefinierten Typen (typedefs, Struktur-, Union-, Klassen- und Aufzählungstypen).

VARIABLES = *UNCHANGED / *YES / *NO
Die Querverweisliste enthält - mit *NO unterdrückbar - eine Liste der Variablen.

FUNCTIONS = *UNCHANGED / *YES / *NO
Die Querverweisliste enthält - mit *NO unterdrückbar - eine Liste der Funktionen.

LABELS = *UNCHANGED / *YES / *NO
Die Querverweisliste enthält - mit *NO unterdrückbar - eine Liste der Labels.

TEMPLATES = *UNCHANGED / *YES / *NO
Die Querverweisliste enthält wahlweise eine Liste der Templates (nur bei Übersetzungen im Modus C++ V3 oder C++2017).

ORDER = *UNCHANGED / *STD / list-poss(6): *PREPROCESSING-INFO / *TYPES / *VARIABLES / *FUNCTIONS / *LABELS / *TEMPLATES
Mit diesem Operanden kann die Reihenfolge festgelegt werden, in der die einzelnen Teile in der Querverweisliste aufgeführt werden.
*STD

Voreingestellt ist die oben nach list-poss angegebene Reihenfolge.

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

*UNCHANGED

Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.

*YES:

Der Compiler erzeugt eine Projektliste (nur bei COMPILE). Diese Liste zeigt die im Quellprogramm original verwendeten Namen und die entsprechenden Namen, die der Compiler für den Binder intern generiert. Dies ist insbesondere bei C++-Übersetzungen wichtig, weil in diesem Fall die genererierten Namen von Funktionen auch die Typen ihrer Parameter in codierter Form enthalten.
Die Projektliste wird pro Übersetzungseinheit, d.h. modullokal, angelegt. Um eine modulübergreifende Projektliste zu erhalten, können mit der Anweisung MODIFY-CIF-PROPERTIES CIF-Informationen erzeugt und anschließend mit dem globalen Listengenerator weiterverarbeitet werden.

*No:

Der Compiler erzeugt keine solche Liste.

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

*UNCHANGED

Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.

*YES:

Der Compiler erzeugt eine Objektcodeliste (nur bei COMPILE).

*No:

Der Compiler erzeugt keine solche Liste.

SUMMARY = *UNCHANGED / *YES / *NO

*UNCHANGED

Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.

*YES:

Der Compiler erzeugt eine Liste mit statistischen Angaben über den Compilerlauf.

*No:

Der Compiler erzeugt keine solche Liste.

LAYOUT =

Mit dieser Option kann die Seitenbreite (Anzahl Zeichen pro Zeile) und die Seitenhöhe (Anzahl Zeilen pro Seite) für die Compilerlisten bestimmt werden.

Bei Auswahl einer Zeilenbreite von 120 Zeichen erhalten alle Listen einen schmaleren Listenkopf und -fuß. Die Textzeilen werden nur bei den tabellarischen Listen (Optionen-, Querverweis- und Adressliste) entsprechend umbrochen. Überlange Textzeilen in der Quellprogramm-, Präprozessor- und Objectcode-Liste werden beim Ausdruck abgeschnitten.

Bei Angabe einer BS2000-Ausgabedatei ist in jeder Zeile die erste Spalte für die Vorschubsteuerung reserviert.
Bei Ausgabe in eine POSIX-Datei werden die für POSIX passenden Kontrollzeichen für Zeilen- bzw. Seitenvorschub generiert. Dadurch wird die Zeilenlänge in der POSIX-Ausgabedatei bis zu 3 Zeichen größer als die gewählte Angabe für die Zeilenbreite.

LAYOUT = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.

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

Listen im Querformat.

LINE-SIZE = *UNCHANGED

Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.
Dies gilt auch dann, wenn diese Angabe bei *FOR-ROTATION-PRINT gemacht wurde.

LINE-SIZE = *STD
Es werden 132 Zeichen pro Zeile ausgegeben.

LINE-SIZE = <integer 120..255>
Es werden 120 bis 255 Zeichen pro Zeile ausgegeben.

LINES-PER-PAGE = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.
Dies gilt auch dann, wenn diese Angabe bei *FOR-ROTATION-PRINT gemacht wurde.

LINES-PER-PAGE = *STD
Es werden 64 Zeilen pro Seite ausgegeben.

LINES-PER-PAGE = <integer 11..255>
Pro Seite werden 11 bis 255 Zeilen ausgegeben.
Als Untergrenze sind 11 Zeilen festgelegt, damit pro Seite mindestens der Listenkopf und -fuß sowie eine Textzeile ausgegeben werden kann.

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

Listen im Hochformat.

Um solche Listen auszudrucken, muss im PRINT-DOCUMENT-Kommando der ROTATION-Parameter angegeben werden.

LINE-SIZE = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.
Dies gilt auch dann, wenn diese Angabe bei *FOR-NORMAL-PRINT gemacht wurde.

LINE-SIZE = *STD
Es werden 120 Zeichen pro Zeile ausgegeben.

LINE-SIZE = <integer 120..255>
Es werden 120 bis 255 Zeichen pro Zeile ausgegeben.

LINES-PER-PAGE = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.
Dies gilt auch dann, wenn diese Angabe bei *FOR-NORMAL-PRINT gemacht wurde.

LINES-PER-PAGE = *STD
Es werden 84 Zeilen pro Seite ausgegeben.

LINES-PER-PAGE = <integer 11..255>
Pro Seite werden 11 bis 255 Zeilen ausgegeben.
Als Untergrenze sind 11 Zeilen festgelegt, damit pro Seite mindestens der Listenkopf und -fuß sowie eine Textzeile ausgegeben werden kann.

INCLUDE-INFORMATION = *UNCHANGED / *NONE / *ALL / *USER-INCLUDES-ONLY
Mit dieser Option lässt sich steuern, ob und welche Include-Dateien in der Quellprogramm-, Präprozessor- und Querverweisliste abgebildet werden. Standardmäßig werden die benutzereigenen Include-Dateien abgebildet, die Standard-Include-Dateien nicht.

LISTING-PRAGMAS =
Mit dieser Option lässt sich steuern, ob und welche im Quelltext vorhandenen `#pragma`-Anweisungen zur Gestaltung der Quellprogramm- und Präprozessorliste berücksichtigt werden sollen.
Die Beschreibung der `#pragma`-Anweisungen finden Sie im Abschnitt „Pragmas zum Steuern des Listenbildes“.

LISTING-PRAGMAS = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.

LISTING-PRAGMAS = *INTERPRETED / *IGNORED
Es werden alle `#pragma`-Anweisungen berücksichtigt (*INTERPRETED) bzw. ignoriert (*IGNORED).

LISTING-PRAGMAS = *SELECT(...)
Eine oder mehrere der folgenden `#pragma`-Anweisungen zur Listensteuerung werden berücksichtigt (*YES) oder ignoriert (*NO).

PAGE = *UNCHANGED / *YES / *NO
Anweisung `#pragma PAGE` [*text*]:
Seitenvorschub und wahlweise Zeile im Listenkopf

TITLE = *UNCHANGED / *YES / *NO
Anweisung `#pragma TITLE` *text*:
Zeile im Listenkopf

SPACE = *UNCHANGED / *YES / *NO
Anweisung `#pragma SPACE` [*n*]:
Einfügen von Leerzeilen

LIST = *UNCHANGED / *YES / *NO
Anweisung `#pragma LIST[ING] ON` oder `#pragma LIST[ING] OFF`:
Unterdrücken der Ausgabe von Quelltextzeilen

INITIAL-TITLE-TEXT = *UNCHANGED / *NONE / <c-string 1..256>
Mit dieser Option kann angegeben werden, ob im Listenkopf eine zusätzliche Zeile erscheinen und welcher Text dort stehen soll. Die INITIAL-TITLE-TEXT-Option bezieht sich, im Unterschied zu den Pragmas (nur Quellprogramm- und Präprozessorliste), auf alle Compilerlisten.
Bei der Quellprogramm- und Präprozessorliste haben ggf. vorhandene TITLE- und PAGE-Pragmas Vorrang vor der INITIAL-TITLE-TEXT-Angabe.

OUTPUT = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.

OUTPUT = *SYSLST
Die Listen werden standardmäßig in die temporäre Systemdatei SYSLST geschrieben, von wo aus sie nach Ende der Task (LOGOFF) auf den Drucker ausgegeben werden. In den Modi C++ V3 und C++ 2017 wird die Ausgabe nach SYSLST nicht unterstützt und mit einer entsprechenden Fehlermeldung abgewiesen.

OUTPUT = *SYSOUT
Die Listen werden auf die Systemdatei SYSOUT geschrieben, die im Dialogbetrieb der Datensichtstation zugeordnet ist. In den Modi C++ V3 und C++ 2017 wird die Ausgabe nach SYSOUT nicht unterstützt und mit einer entsprechenden Fehlermeldung abgewiesen.

OUTPUT = *STD-FILE

Die Listen werden in eine katalogisierte BS2000-Datei geschrieben. Der Name dieser Datei wird aus dem Namen des Quellprogramms abgeleitet:

Quelle	*SYSDTA	BS2000-Datei	PLAM-Bibliothek	POSIX-Datei
Standardname	CSTDLST.LST	datei.LST	bib-elem.LST	datei.LST

Wenn das Quellprogramm in einer PLAM-Bibliothek steht, werden im Standard-Dateinamen der Bibliotheks- und Elementname der Quelle verwendet und mit einem Bindestrich verbunden (bib-elem). Die Regeln zur Bildung der Standardnamen durch den Compiler sind ausführlich im Abschnitt „Standardnamen für Ausgabebehälter“ dargestellt.

OUTPUT = *SOURCE-LOCATION
Ausgabeort und Name werden wie folgt aus dem Ort und Namen des Quellprogramms abgeleitet:

Quelle	*SYSDTA	BS2000-Datei	PLAM-Bibliothek	POSIX-Datei

Ausgabeort	BS2000-Datei	BS2000-Datei	Bibliothek der Quelle	Dateiverzeichnis der Quelle
Standardname	CSTDLST.LST	datei.LST	elem.LST (Typ P)	datei.lst

Die Regeln zur Bildung der Standardnamen durch den Compiler sind ausführlich im Abschnitt „Standardnamen für Ausgabebehälter“ dargestellt.

OUTPUT = <filename 1..54>
Die Listen werden in eine katalogisierte BS2000-Datei mit dem angegebenen Namen geschrieben. Bei der Übersetzung mehrerer Quellprogramme ist diese Angabe nicht sinnvoll, weil die Datei jedes Mal überschrieben wird.

OUTPUT = <posix-pathname>

Die Listen werden in das POSIX-Dateisystem geschrieben.

Als <posix-pathname> ist sowohl ein Dateiname als auch ein Dateiverzeichnis zugelassen. Zur Beschreibung des Begriffs <posix-pathname> siehe "Compiler-Ein-/Ausgaben im POSIX-Dateisystem".

Bei der Angabe eines Dateinamens werden die Listen unter diesem Namen abgelegt.

Bei der Angabe eines Dateiverzeichnisnamens *dvz* werden die Listen für jedes übersetzte Quellprogramm unter dem Standardnamen *quelldatei.lst* in das Dateiverzeichnis *dvz* geschrieben (siehe auch Abschnitt „Standardnamen für Ausgabebehälter“).

Die mit <posix-pathname> angegebenen Dateiverzeichnisse müssen bereits existieren.Bei der Übersetzung mehrerer Quellprogramme ist die Angabe eines Dateinamens nicht sinnvoll, weil die Datei jedes Mal überschrieben wird.

OUTPUT = *LIBRARY-ELEMENT(...)
Es wird angegeben, in welcher PLAM-Bibliothek (LIBRARY=) und unter welchem Elementnamen (ELEMENT=) die Listen abgelegt werden sollen. Die Elemente werden unter dem Typ P abgespeichert.

LIBRARY = *STD-LIBRARY
Die Listen werden standardmäßig in die Bibliothek SYS.PROG.LIB geschrieben.

LIBRARY = *SOURCE-LIBRARY
Die Listen werden in die PLAM-Bibliothek geschrieben, in der das Quellprogramm steht.
Die Angabe *SOURCE-LIBRARY ist unzulässig, wenn das Quellprogramm aus einer katalogisierten BS2000-Datei, einer POSIX-Datei oder über SYSDTA eingelesen wird.

LIBRARY = <filename 1..54>
Die Listen werden in die PLAM-Bibliothek mit dem angegebenen Namen geschrieben.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>

Statt des Bibliotheksnamens kann auch ein Linkname angegeben werden. <filename> ist der Linkname der zugewiesenen Bibliothek. Er muss vor Aufruf des Compilers mit dem ADD-FILE-LINK-Kommando der PLAM-Bibliothek zugewiesen worden sein.

ELEMENT = *STD-ELEMENT(...)

Standardmäßig wird der Elementname der Listen aus dem Namen des Quellprogramms abgeleitet:

Quelle	*SYSDTA	BS2000-Datei	PLAM-Bibliothek	POSIX-Datei
Standardname	CSTDLST.LST	datei.LST	elem.LST	datei.LST

Die Regeln zur Bildung der Standardnamen durch den Compiler sind ausführlich im Abschnitt „Standardnamen für Ausgabebehälter“ dargestellt.

VERSION = *UPPER-LIMIT

Enthält die Elementangabe keine Versionsbezeichnung, verwendet der Compiler die höchstmögliche Version.

VERSION = *INCREMENT
Das Element erhält die gegenüber der höchsten vorhandenen Version um 1 inkrementierte Versionsnummer, vorausgesetzt, die höchste vorhandene Versionsbezeichnung endet mit einer inkrementierbaren Zahl. Ist die Versionsbezeichnung nicht inkrementierbar, wird eine Fehlermeldung ausgegeben und der Übersetzungslauf ohne Listengenerierung fortgesetzt.
Beispiel siehe COMPILE-Anweisung ("COMPILE").

Achtung: In den Modi C++ V3 und C++ 2017 ist die Angabe von ***INCREMENT** nicht erlaubt.

VERSION = <composed-name 1..24 with-under>
Das Element erhält die angegebene Versionsbezeichnung.

ELEMENT = <composed-name 1..64 with-under>(...)
Die Listen werden in ein Bibliothekselement (Typ P) mit dem angegebenen Namen geschrieben. Bei der Übersetzung mehrerer Quellprogramme ist die Angabe eines Elementnamens nicht sinnvoll, weil das Element jedes Mal überschrieben wird.

VERSION = *UPPER-LIMIT / *INCREMENT / <composed-name 1..24 with-under>
Die Version kann in gleicher Weise angegeben werden, wie oben unter ELEMENT=*STD-ELEMENT(...) beschrieben.

Achtung: In den Modi C++ V3 und C++ 2017 ist die Angabe von ***INCREMENT** nicht erlaubt.

3.2.2.14 MODIFY-MODULE-PROPERTIES

Alias-Name: SET-MODULE-PROPERTIES

Mit dieser Anweisung werden die Eigenschaften des zu generierenden Moduls festgelegt.

```
MODIFY-MODULE-PROPERTIES

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

     *YES(...)

         |  PUBLIC-SLICING=*UNCHANGED / *YES / *NO

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

 ,WORKSPACE= *UNCHANGED / *TO-STATIC-AREA / *TO-STACK

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

 ,ETPND-GENERATION= *UNCHANGED / *NO / [*YES] (...)

     *YES(...)

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

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

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

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

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

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

 ,FP-ARITHMETICS= *UNCHANGED / *390-FORMAT / *IEEE-FORMAT
```

SHAREABLE-CODE =
Diese Option steuert die gemeinsame Benutzbarkeit des erzeugten Codes.

SHAREABLE-CODE = *UNCHANGED
Es gelten die Einstellungen der letzten MODIFY-MODULE-PROPERTIES-Anweisung.

SHAREABLE-CODE = *NO
Der Compiler erzeugt keinen gemeinsam benutzbaren Code.
*NO ist Voreinstellung.

SHAREABLE-CODE = [*YES] (PUBLIC-SLICING = ...)
Der Compiler erzeugt gemeinsam benutzbaren Code und zwar ein LLM mit einer gemeinsam benutzbaren Code-CSECT und einer nicht gemeinsam benutzbaren Daten-CSECT.

PUBLIC-SLICING = *UNCHANGED / *YES / *NO
Mit PUBLIC SLICING wird gesteuert, ob das generierte Objekt bereits nach dem Attribut PUBLIC in Slices aufgeteilt wird.

*UNCHANGED

Es gelten die Einstellungen der letzten MODIFY-MODULE-PROPERTIES-Anweisung.

*YES

Das Objekt soll bereits nach dem Attribut PUBLIC in Slices aufgeteilt werden. *YES ist Voreinstellung.

*NO

Das Objekt wird in einer Slice abgelegt.

> POSIX-Objekte werden stets mit PUBLIC-SLICING = *NO generiert.
> Bei sehr großen Programmen kann es sinnvoll sein, PUBLIC-SLICING=*NO zu setzen, da auf diese Weise lange Binde-Laufzeiten für das Einlesen von Objekten mit Slices vermieden werden.

LINKAGE = *UNCHANGED / *ILCS-OUT / *ILCS-INLINE
Die Funktionsaufrufe im erzeugten Modul werden mittels ILCS bewerkstelligt. Mit dieser Option kann angegeben werden, ob der ILCS-Entry-Code für Funktionsaufrufe direkt in die Aufrufstelle eingesetzt werden soll („inline“) oder im Laufzeitsystem angesprungen werden soll („out-of-line“).

*UNCHANGED

Es gelten die Angaben der letzten MODIFY-MODULE-PROPERTIES-Anweisung.

*ILCS-OUT:

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

*ILCS-INLINE:

Standardmäßig wird der ILCS-Entry-Code inline generiert. Die Laufzeit des erzeugten Objektes wird dadurch beschleunigt.

WORKSPACE = *UNCHANGED / *TO-STATIC-AREA / *TO-STACK
Diese Option beeinflusst einige Optimierungen des Compilers.

*UNCHANGED

Es gelten die Angaben der letzten MODIFY-MODULE-PROPERTIES-Anweisung.

*TO-STATIC-AREA

Der Compiler führt folgende Optimierungen durch:

- Für Konversionen wird ein Hilfsspeicher im statischen Datenbereich angelegt und mit konstanten Werten vorbelegt.
- Bei innersten Funktionen (d.h. solchen ohne weitere Aufrufe) werden Daten, die nur innerhalb der Funktion gültig sind (auto-Variablen), nicht auf dem Stack abgelegt, sondern zusammen mit statischen Daten im Datenmodul.

Durch diese Maßnahmen entfällt die Notwendigkeit, für innerste Funktionen einen eigenen Stack-Bereich zu verwalten, wodurch der Funktions-Entrycode und somit die Laufzeit des Objekts verkürzt wird.

*TO-STACK

Die o.g. Optimierungen unterbleiben.

Die Daten (Doppelworte für Konversionen sowie auto-Variablen und Tempos von innersten Funktionen) werden auf dem Stack abgebildet.
Enthält eine Funktion Konversionen, so muss vor manchen Konversionen jedes Mal wieder ein Doppelwort auf dem Stack dynamisch vorbesetzt werden (ein zusätzlicher Befehl erforderlich).

SUBROUTINE-CALL = *UNCHANGED / *BASR / *LAB
Diese Option steuert die Realisierung von Unterprogramm-Einsprüngen mittels Assembler-Befehlen.

*UNCHANGED

Es gelten die Angaben der letzten MODIFY-MODULE-PROPERTIES-Anweisung.

*BASR:

Standardmäßig wird der Befehl BASR generiert.

*LAB:

Es werden die maschinenunabhängigen 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 und C++ 2017 nicht erlaubt.

ETPND-GENERATION =
Mit dieser Option wird die #pragma-Anweisung zur Erzeugung eines ETPND-Bereichs (siehe "ETPND-Pragma") gelöscht bzw. das Datum-Format des ETPND-Bereichs festgelegt.

ETPND-GENERATION = *UNCHANGED
Es gelten die Angaben der letzten MODIFY-MODULE-PROPERTIES-Anweisung.

ETPND-GENERATION = *NO
Standardmäßig wird kein ETPND-Bereich erzeugt.

ETPND-GENERATION = *YES(...)

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

*UNCHANGED

Es gelten die Angaben der letzten MODIFY-MODULE-PROPERTIES-Anweisung.

*CALENDAR-DATE-ONLY:

Das Datum-Format im ETPND-Bereich erhält die Form: 8 Byte Kalenderdatum - 4 Byte Ladeadresse

*WITH-JULIAN-DATE:

Im ETPND-Bereich wird folgendes Datum-Format generiert: 6 Byte Kalenderdatum - 3 Byte julianisches Datum - 4 Byte Ladeadresse

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

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

*UNCHANGED

Es gelten die Angaben der letzten MODIFY-MODULE-PROPERTIES-Anweisung.

*NO:

Standardmäßig werden bei der Generierung von Entry-Namen Klein- in Großbuchstaben umgewandelt.

*YES:

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

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

Diese Option zur 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 Codierung von externen C++-Symbolen werden generell Unterstriche beibehalten.

*UNCHANGED

Es gelten die Angaben der letzten MODIFY-MODULE-PROPERTIES-Anweisung.

*CONVERT-TO-DOLLAR:

Standardmäßig werden bei der Generierung von Entry-Namen Unterstriche in Dollarzeichen umgewandelt.

*KEEP:

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

Hinweise zu LOWER-CASE-NAMES und SPECIAL-CHARACTERS

1. Die (voreingestellte) Umsetzung von Klein- in Großbuchstaben sowie von Unterstrichen in Dollarzeichen ist immer dann notwendig, wenn das erzeugte LLM mit Objekten verknüpft werden soll, in denen die Entry-Namen entsprechend umgesetzt wurden. Dies sind:
   - Objektmodule
   - mit dem C-V2.0-Compiler erzeugte LLMs
   - mit dem C/C++-Compiler erzeugte LLMs, in denen die Entry-Namen entsprechend umgesetzt wurden
   - Objekte, die mit anderen Sprachübersetzern erzeugt wurden (z.B. COBOL, ASSEMBLER)
2. Die C-Bibliotheksfunktionen werden nur dann vollständig unterstützt, wenn die Optionen LOWER-CASE-NAMES und SPECIAL-CHARACTERS in einer der beiden folgenden Kombinationen vorliegen:
   - SPECIAL-CHARACTERS=*CONVERT-TO-DOLLAR und LOWER-CASE-NAMES=*NO
   - SPECIAL-CHARACTERS=*KEEP und LOWER-CASE-NAMES=*YES

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

Zeichenketten-Konstanten (z.B. "abc") werden entweder im Datenmodul (*WRITEABLE) oder im Codemodul (*READ-ONLY) abgelegt.

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

`const`-qualifizierte Objekte werden entweder im Datenmodul (*WRITEABLE) oder im Codemodul (*READ-ONLY) abgelegt.

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

Diese Option legt die Länge von externen C-Namen fest und 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).
Die Option wirkt auch auf Static-Funktionen.

*UNCHANGED

Es gelten die Angaben der letzten MODIFY-MODULE-PROPERTIES-Anweisung.

*STD:

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 können nur 30 Zeichen genutzt werden.

*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 erreichen. Module, die EEN-Namen enthalten, werden vom Compiler im LLM-Format 4 abgelegt. Ausführliche Informationen zur Weiterverarbeitung von LLMs im Format 4 finden Sie bei "BIND" (BIND-Anweisung, OUTPUT-FORMAT-Option).
Die Angabe *UNLIMITED wird im Cfront-C++-Modus nicht unterstützt.

*SHORT:

Externe C-Namen werden vom Compiler auf 8 Zeichen verkürzt. Diese Namensverkürzung entspricht der Behandlung von externen Namen innerhalb von Objektmodulen. Die Option wird benötigt, wenn im Programm externe Namen verwendet werden, die länger als 8 Zeichen lang sind und das vom C/C++-Compiler erzeugte Modul mit Objektmodulen verknüpft werden soll, die von den Vorgänger-Compilern C und C++ oder von Compilern für andere ILCS-Sprachen (z.B. COBOL85) erzeugt wurden.

FP-ARITHMETICS = *UNCHANGED / *390-FORMAT / *IEEE-FORMAT
Diese Option legt fest, ob der C/C++-Compiler für Gleitpunktzahlen und -Operationen Code im /390-Format oder im IEEE-Format erzeugt. Dies betrifft alle Variablen und Konstanten der Datentypen float, double und long double innerhalb eines C/C++-Programms.

*UNCHANGED

Es gelten die Angaben der letzten MODIFY-MODULE-PROPERTIES-Anweisung.

*390-FORMAT:

Der C/C++-Compiler erzeugt Code für Konstanten und Operationen im /390-Format (/390-Gleitpunkt-Arithmetik).
*390-FORMAT ist Voreinstellung.

*IEEE-FORMAT:

Der C/C++-Compiler erzeugt Code für Konstanten und Operationen im IEEE-Format (IEEE-Gleitpunkt-Arithmetik).

Achtung:

- Es gibt keine Vorkehrungen beim Zusammenbinden von Objekten, die mit unterschiedlich erzeugten Floatingpoint-Arithmetiken übersetzt wurden. Dies kann beim Ablauf solcher Programme zu unerwarteten Ergebnissen führen.
- 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-Gleitpunktzahlen verwenden eine andere interne Darstellung als /390-Gleitpunktzahlen.
  - `long double` hat im IEEE-Format die gleiche Größe und Darstellung wie `double`. Das Layout von Strukturen mit einem `long double` hängt deshalb von dem Wert dieser Option ab.
  - 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:

- Bei Nutzung der IEEE-Gleitpunktarithmetik dürfen Sie die C-Bibliotheksfunktionen in Ihrem Quellprogramm nicht explizit deklarieren, sondern nur indirekt via Inkludieren des entsprechenden CRTE-Headers (siehe Handbuch „C-Bibliotheksfunktionen“ [2]). 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 Gleitpunkzahlen arbeitet, die dazugehörige bzw. passende 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.

  Achtung:

  C++-Bibliotheksfunktionen unterstützen das IEEE-Format *nicht* und müssen deshalb gegebenenfalls durch C-Funktionen ersetzt werden.
- Im Laufzeitsystem CRTE gibt es einige C-Bibliotheksfunktionen, die IEEE-Format für Gleitpunktzahlen verwenden. Für die Verwendung der IEEE-Gleitpunkt-Arithmetik sollte die MODIFY-MODULE-PROPERTIES-Anweisung wie folgt angegeben werden:

```
MODIFY-MODULE-PROPERTIES        -
   ...
   FP-ARITHMETICS=*IEEE-FORMAT, -
   LOWER-CASE-NAMES=*YES,       -
   SPECIAL-CHARACTERS=*KEEP,    -
   ...
```

3.2.2.15 MODIFY-OPTIMIZATION-PROPERTIES

Alias-Name: SET-OPTIMIZATION-PROPERTIES

Mit dieser Anweisung lässt sich die Optimierung des Compilers ganz oder teilweise ein- und ausschalten. Einzelheiten zur Wirkung der Optimierung finden Sie ab "Verlauf der Optimierung".

```
MODIFY-OPTIMIZATION-PROPERTIES

LEVEL= *UNCHANGED / *LOW / *HIGH(...) / *VERY-HIGH(...)

    *HIGH(...) / *VERY-HIGH(...)

        |    ,LOOP-UNROLLING= *UNCHANGED / *NO / [*YES](...)

        |            *YES(...)

        |                    |        FACTOR= 4 / <integer 1..100>

,INLINING= *UNCHANGED / *NO / [*YES](...)

    *YES(...)

        |    USER-FUNCTIONS= *UNCHANGED / list-poss(127): *STD / *BY-SOURCE /

        |                              <c-string 1..255 with-low>

,BUILTIN-FUNCTIONS= *UNCHANGED / *NONE / *ALL / list-poss(11): <c-string 1..125 with-low>
```

LEVEL = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-OPTIMIZATION-PROPERTIES-Anweisung.

LEVEL = *LOW
Es werden keine Standardoptimierungen durchgeführt. In dieser Optimierungsstufe ist Testen mit AID möglich. LEVEL=*LOW wird automatisch an Stelle der Angabe *HIGH bzw. *VERY-HIGH gesetzt, wenn gleichzeitig die Option TEST-SUPPORT=*YES gesetzt ist.

LEVEL = *HIGH(...) / *VERY-HIGH(...)
Bei Angabe von *HIGH oder *VERY-HIGH werden alle Standardoptimierungen durchgeführt (siehe „Standardoptimierungen" (Verlauf der Optimierung)“). Der Unterschied zwischen diesen beiden Stufen besteht darin, dass bei *HIGH intern jede Optimierungsstrategie nur einmal durchgeführt wird, bei *VERY-HIGH mehrmals. Entsprechend wird in der Optimierungsstufe *HIGH deutlich weniger Übersetzungszeit benötigt als in der „hochoptimierenden“ Stufe *VERY-HIGH.

Mit den Parametern der *HIGH- bzw. *VERY-HIGH-Struktur lässt sich die Expansion von Schleifen einzeln beeinflussen. In diesen Optimierungsstufen ist Testen mit AID nicht möglich.

LOOP-UNROLLING = *UNCHANGED / *NO / *YES(FACTOR = 4 / <integer 1..100>...)

Diese Option steuert die Expansion von Schleifen. Durch eine mehrmalige Expansion des Schleifenrumpfes wird eine geringere Ausführungszeit für die Schleifendurchläufe erzielt. Zusätzlich entsteht mit dem expandierten Schleifenrumpf neues Optimierungspotential. Diese Optimierungsmaßnahme ist wegen der Code-Wiederholungen mit einer Vergrößerung des erzeugten Objekts verbunden.

Standardmäßig versucht der Optimierer, Schleifenrümpfe viermal zu expandieren:

- Mit <integer> kann ein eigener Expansionsfaktor ausgewählt werden. Die Angabe eines Faktors garantiert aber nicht in jedem Fall die Durchführung einer Schleifenexpansion. Vielmehr entscheidet der Optimierer anhand von Schleifenstruktur und angegebenem Expandierungsfaktor, ob die Expansion durchgeführt wird.
- Mit *NO wird die Schleifenexpansion unterdrückt.

Weitere Einzelheiten siehe auch unter „Expansion von Schleifen“ (Verlauf der Optimierung).

INLINING =
Diese Option steuert die Inline-Generierung von benutzereigenen Funktionen. Wie auch bei der Inline-Generierung von einigen C-Bibliotheksfunktionen aus der Standardbibliothek (siehe BUILTIN-FUNCTIONS), wird jeder Aufruf einer Inline-Funktion durch den entsprechenden Funktionscode ersetzt und durch die Einsparung der Aufruf- und Rücksprung-Codefolge eine bessere Ablaufzeit erzielt.
Diese Optimierungsmaßnahme ist wegen der Code-Wiederholungen mit einer Vergrößerung des erzeugten Objekts verbunden.
Weitere Einzelheiten siehe auch unter „Inline-Generierung von benutzereigenen Funktionen“ (Verlauf der Optimierung).

Standardeinstellungen des Compilers

Wenn die INLINING-Option nicht angegeben wird, gelten abhängig vom C- oder C++-Sprachmodus folgende Standardeinstellungen:

1. C-Sprachmodi (MODIFY-SOURCE-PROPERTIES LANGUAGE=*C):

   INLINING=*NO
   Es wird keine Inline-Generierung durchgeführt.
2. C++-Sprachmodi (MODIFY-SOURCE-PROPERTIES LANGUAGE=*CPLUSPLUS):

   INLINING=*YES(USER-FUNCTIONS=*BY-SOURCE)
   Es wird die Inline-Generierung der C++-sprachspezifischen Inline-Funktionen durchgeführt (mit dem Attribut `inline` versehene Funktionen und innerhalb von Klassen definierte Elementfunktionen).

INLINING = *UNCHANGED

Es gilt die Angabe der letzten MODIFY-OPTIMIZATION-PROPERTIES-Anweisung.

INLINING = *NO
Vom Optimierer werden keine benutzereigenen Funktionen inline generiert. *NO ist in den C-Sprachmodi voreingestellt, wenn keine INLINING-Option angegeben wird. Außerdem wird *NO vom Compiler an Stelle der Angaben *YES(...) automatisch angenommen, wenn gleichzeitig die Option TEST-SUPPORT=*YES gesetzt ist.

INLINING = *YES(USER-FUNCTIONS = *UNCHANGED / list-poss: *STD / *BY-SOURCE / <c-string 1..255 with-low>)

*UNCHANGED:

Es gilt die Angabe der letzten MODIFY-OPTIMIZATION-PROPERTIES-Anweisung.

*STD:

Wenn nur *STD angegeben wird, wählt der Optimierer Funktionen für die Inline-Generierung nach eigenen Kriterien aus. *STD impliziert die Angabe *BY-SOURCE. Das heißt, dass bei der Suche nach geeigneten Kandidaten auch inline-Pragmas und C++-sprachspezifische Inline-Funktionen vom Optimierer mitberücksichtigt werden (vgl. *BY-SOURCE). *STD ist voreingestellt, wenn INLINING=*YES angegeben wird.

*BY-SOURCE:

Wenn nur *BY-SOURCE angegeben wird, werden ausschließlich folgende benutzereigene Funktionen inline generiert:

- In den C-Sprachmodi C-Funktionen, die mit folgender `#pragma`-Anweisung angegeben werden:
  `#pragma inline` *funktionsname*
  Das inline-Pragma wird in den C++-Sprachmodi nicht unterstützt.
  Siehe auch Abschnitt „inline-Pragma“.
- In den C++-Sprachmodi C++-Funktionen mit dem Attribut `inline` sowie die innerhalb von Klassen definierten C++-Funktionen.

In den C++-Sprachmodi ist *BY-SOURCE voreingestellt, wenn keine INLINING-Option angegeben wird.

<c-string>:

Mit <c-string> kann der Name einer benutzereigenen C-Funktion angegeben werden, die der Optimierer inline generieren soll. Die Angabe von selbst ausgewählten Funktionen mit <c-string> wird nur in den C-Sprachmodi unterstützt, da es in C++ eigene Sprachmittel für die Inline-Generierung von Funktionen gibt. <c-string> impliziert die Angabe *BY-SOURCE. Das heißt, dass auch inline-Pragmas vom Optimierer mitberücksichtigt werden (vgl. *BY-SOURCE in den C-Sprachmodi).

list-poss:

Die Angaben *STD, <c-string> und aus Kompatibilitätsgründen auch *BY-SOURCE können miteinander kombiniert werden. In diesem Fall versucht der Optimierer zunächst die mit *BY-SOURCE und/oder <c-string> angegebenen Funktionen inline zu generieren. Anschließend wählt er - sofern *STD angegeben wird - nach eigenen Kriterien Funktionen für die Inline-Generierung aus. Die Angabe von *BY-SOURCE ist nicht notwendig, da sie bei gleichzeitiger Angabe von *STD oder <c-string> implizit angenommen wird.

Sinnvolle Kombinationen:

*STD, <c-string>
<-c-string>, <c-string>, ...

Beispiel zur INLINING-Option

```
//MODIFY-OPTIMIZATION-PROP -
//LEVEL=*HIGH,INLINING=*YES(USER-FUNCT=(*STD,'funct1','funct2'))
```

Weitere Einzelheiten siehe auch unter „Inline-Generierung von benutzereigenen Funktionen“ (Verlauf der Optimierung).

BUILTIN-FUNCTIONS = *UNCHANGED / *NONE / *ALL /list-poss: <c-string 1..125 with-low>

Mit dieser Option kann angegeben werden, für welche C-Bibliotheksfunktionen die Implementierung im CRTE angenommen werden kann. Dies erlaubt eine bessere Optimierung des Programms.

*UNCHANGED:

Es gilt die Angabe der letzten MODIFY-OPTIMIZATION-PROPERTIES-Anweisung.

*NONE:

Es wird kein Aufruf von Bibliotheksfunktionen besonders optimiert.

*ALL

Alle Aufrufe von bekannten Bibliotheksfunktionen werden gesondert behandelt.

<c-string>:

Aufrufe auf diese Funktion werden gesondert behandelt.

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:

strcpy	memcmp
strcmp	memset
strncmp	abs
strlen	fabs
strcat	labs
memcpy	

Hinweise

- Inline generierte Funktionen können weder zum Bindezeitpunkt durch andere Funktionen ersetzt noch beim Testen mit AID als Testpunkte benutzt werden.
- 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.
- Wird eine Funktion mit einem dem Compiler bekannten Namen selbst definiert, so kann es Konflikte mit dieser Option geben. Im Allgemeinen hat die vom Benutzer geschriebene Funktion eine andere Implementierung als die Funktion im CRTE. An der Stelle der Definition wird die Warnung CFE2067 ausgegeben, um auf einen Konflikt hinzuweisen.
- Beachten Sie, dass die Eigenschaften der CRTE-Implementierung in jeder Übersetzungseinheit benutzt werden. Die Warnung wird jedoch nur in der Übersetzungseinheit ausgegeben, die die private Definition enthält.

Die folgende Tabelle fasst die Standardeinstellungen der MODIFY-OPTIMIZATION-PROPERTIES-Anweisung und möglichen Modifikationen noch einmal zusammen.

	***HIGH(...) / *VERY-HIGH(...)**	***LOW**
Standardoptimierungen	*YES	*NO
LOOP-UNROLLING	*YES (steuerbar)	*NO
BUILTIN-FUNCTIONS	*NONE (steuerbar)	
INLINING (in C)	*NO (steuerbar)	

INLINING (in C++)	*YES(USER-FUNCTIONS=*BY-SOURCE) (steuerbar)

3.2.2.16 Verlauf der Optimierung

Im Folgenden sind unter dem Begriff „Standardoptimierungen“ alle Optimierungsmaßnahmen zusammengefasst, die nur global ein- oder ausgeschaltet, d.h. nicht einzeln beeinflusst werden können.

Demgegenüber gibt es die einzeln beeinflussbaren Optimierungsmaßnahmen, wozu Schleifenexpansion und Inline-Generierung zählen. Diese Optimierungsmaßnahmen können u.a. deshalb einzeln gesteuert werden, weil den durch die Optimierung erzielten Verbesserungen (z.B. schnellere Ablaufzeit) auch u.U. gewisse Nachteile gegenüberstehen, z.B. die Vergrößerung des erzeugten Moduls oder längere Übersetzungszeiten.
Wenn die Optionen LOOP-UNROLLING und INLINING eingeschaltet sind, kann sich die Performance u.U. verschlechtern. Es empfiehlt sich, die für eine Applikation jeweils günstigste Einstellung durch Tests zu ermitteln.

Standardoptimierungen

Der C/C++-Compiler führt folgende Standardoptimierungen durch:

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

Ein wichtiger Begriff für die Optimierung ist der Begriff „Basisblock“. Als Basisblock wird eine maximale unverzweigte Befehlsfolge bezeichnet. Eine solche Befehlsfolge hat genau einen Eingangspunkt und einen Ausgangspunkt.
Die Basisblöcke sind die Einheiten, über die die meisten Optimierungsmaßnahmen des C/C++-Compilers realisiert werden.

1. Berechnung konstanter Ausdrücke zur Übersetzungszeit

   Durch die Berechnung von Ausdrücken, deren Operanden während der Übersetzung wertmäßig bekannt sind, wird die Ausführung von Befehlen vom Programmlauf in den Compilerlauf verlagert und damit eine kürzere Programm-Laufzeit erreicht.
   Die Berechnungen zur Übersetzungszeit umfassen die Integer-Arithmetik und die Vergleichsoperationen.

   Beispiel

vor der Optimierung	nach der Optimierung
`I = 1 + 2;`	`I = 3;`
`I = 2 * 4;`	`I = 8;`
`1 <= 5;`	`<TRUE>`

2. Eliminierung unnötiger Zuweisungen

   Wird nach einer Zuweisung an eine Variable v der Wert von v erneut verändert, ohne dass der Wert benutzt wurde, so wird diese Zuweisung eliminiert. Eliminiert werden auch Zuweisungen an Variablen, deren Wert im weiteren Programmablauf nicht mehr verwendet wird.

   Beispiel

vor der Optimierung	nach der Optimierung
`i = 5;`	
`i = 3;`	`i = 3;`

3. Propagation konstanter Ausdrücke

   Wird in einem Ausdruck eine Variable verwendet, deren Wert zur Übersetzungszeit bereits bekannt ist, so wird diese Variable durch den entsprechenden Wert ersetzt.

   Beispiel

vor der Optimierung	nach der Optimierung
`a = 3;`	`a = 3;`
`i = a;`	`i = 3;`

   Diese Optimierung hat auch Auswirkungen auf andere Optimierungstechniken. Nach der erfolgreichen Propagation einer Variablen kann die ursprüngliche Zuweisung eventuell gelöscht werden oder es kann ein neuer konstanter Ausdruck entstehen, der schon zur Übersetzungszeit berechnet wird.

   Beispiel

vor der Optimierung	nach der Optimierung
`a = 3;`	
`i = a + 4;`	`i = 7;`
`a = 5;`	`a = 5;`

4. Eliminierung redundanter Ausdrücke

   Tritt innerhalb eines Basisblockes ein Ausdruck auf, dessen Wert zur Ausführungszeit auf Grund einer früheren Berechnung bereits bekannt ist, so ist dieser Ausdruck redundant. Um die zweite Berechnung zu vermeiden, wird der Ausdruck einer neu eingeführten Variablen zugewiesen und an allen Stellen durch diese neue Variable ersetzt.

   Beispiel

vor der Optimierung	nach der Optimierung
`a = b * c + 20;`	`h = b * c;`
`e = b * c - 10;`	`a = h + 20;`
	`e = h - 10;`

5. Optimierung der Indexrechnung in Schleifen

   Wird ein Arrayelement in einer Schleife über eine Iterationsvariable indiziert, so wird die zur Berechnung der Adresse des Arrayelements notwendige Multiplikation auf Additionen zurückgeführt. Normalerweise errechnet sich die Adresse des Arrayelements aus

   Basisadresse + Index * Länge eines Arrayelements

   Der Optimierer versorgt vor dem Schleifeneingang eine Adressvariable mit der Adresse des beim ersten Schleifendurchlauf angesprochenen Arrayelements. Bei jedem Schleifendurchlauf wird auf diese Adressvariable eine feste Schrittweite in der Länge eines Array-Elements addiert.

   Besonders lohnend ist diese Optimierung bei mehrdimensionalen Arrays, da in günstigen Fällen pro Dimension eine Multiplikation eingespart wird.

6. Optimierung von Sprüngen auf unbedingte Sprungbefehle

In Sprungbefehlen, die als Sprungziel einen unbedingten Sprung haben, wird das Sprungziel durch das des unbedingten Sprunges ersetzt. In diesem Zusammenhang wird auch überflüssiger, weil unerreichbarer Code eliminiert.

Beispiel

vor der Optimierung

```
   goto lab1;
   ...
lab1:
   goto lab2;
   ...
lab2:
```

nach der Optimierung

```
   goto lab2;
   ...
lab1:
   goto lab2;
   ...
lab2:
```

Inline-Generierung von benutzereigenen Funktionen

Die Inline-Generierung einer Funktion macht den Aufruf einer Funktion zum Ablaufzeitpunkt überflüssig, da der Funktionscode an den Aufrufstellen in den Code der rufenden Funktion integriert wird. Damit können zeitaufwendige Codefolgen für Funktions-Aufrufe und -Rücksprünge eingespart werden (z.B. Register retten und restaurieren, Stack allokieren, Parameter in den Übergabebereich schreiben). Dadurch ergeben sich z.T. beträchtliche Laufzeiteinsparungen. Außerdem erhöht die Inline-Generierung von Funktionen die Wirkung der Standardoptimierungen durch den vergrößerten Kontext. Inline-generierte `static`-Funktionen werden gelöscht.

Durch die Inline-Generierung nimmt jedoch auch die Größe der erzeugten Module zu. Durch Inline-Generierung können so große Funktionen entstehen, dass immer mehr Variablen als potenzielle Registerkandidaten möglich werden, die jedoch wegen der in beschränkter Zahl zur Verfügung stehenden Register nicht alle bedient werden können. Dieser Sachverhalt ist gegenüber den Vorteilen der Optimierungsmaßnahme abzuwägen.

Für die Inline-Generierung eignen sich vor allem kleine Funktionen, da dort der Overhead für die Funktionsaufrufe und -rücksprünge gegenüber dem eigentlichen Funktionscode einen großen Anteil einnimmt.

Funktionen mit folgenden Eigenschaften werden in keinem Fall inline generiert:

- Funktionen mit einer variablen Anzahl von Parametern (vgl. va_...-Makros in <stdarg.h>)
- Funktionen, die setjmp-Aufrufe enthalten
- rekursive Funktionen

Expansion von Schleifen

Durch Schleifenexpansion wird die Zahl der Schleifendurchläufe verringert, indem der Schleifenrumpf (Anweisungsblock) ein oder mehrere Male wiederholt, „expandiert“ wird. Da bei jedem Schleifendurchlauf Schleifensteueranweisungen ausgeführt werden müssen, die den Wert des aktuellen Schleifendurchlaufs prüfen und entsprechend verzweigen, verbessert eine Verringerung der Schleifendurchläufe die Ausführungszeit.

Wird beispielsweise der Schleifenrumpf verdoppelt (Expansionsfaktor 2), so halbiert sich der Aufwand für Schleifensteueranweisungen. Allgemein gilt: Bei einem Expansionsfaktor *n* verringert sich der Aufwand für Schleifensteueranweisungen auf 1/*n*.
Das erzeugte Modul wird jedoch durch die Code-Wiederholungen vergrößert.
Der Optimierer verwendet standardmäßig den Expansionsfaktor 4.

Mit dem expandierten Schleifenrumpf entsteht außerdem neues Optimierungspotential. In den verlängerten Basisblöcken sind Verbesserungen z.B. durch die Propagation konstanter Ausdrücke oder durch die Eliminierung redundanter Ausdrücke möglich.

Beispiel für eine Schleifenexpansion mit Expansionsfaktor 4

Vor der Expansion:

```
i = 0;
while(i < 80) {
    a[i] = b[i+1];
    i++;
    }
```

Nach der Expansion:

```
i = 0;
while(i < 80) {
    a[i] = b[i+1];
    i++;
    if(!(i < 80)) goto ende;
    a[i] = b[i+1];
    i++;
    if(!(i < 80)) goto ende;
    a[i] = b[i+1];
    i++;
    if(!(i < 80)) goto ende;
    a[i] = b[i+1];
    i++;
    }
ende:
```

Hier sind Optimierungen im Schleifenrumpf möglich.

3.2.2.17 MODIFY-RUNTIME-PROPERTIES

Alias-Name: SET-RUNTIME-PROPERTIES

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

MODIFY-RUNTIME-PROPERTIES
PARAMETER-PROMPTING= *UNCHANGED / *YES / *NO
,STACK-SIZE= 64 / *UNCHANGED / <integer 8..99999>
,STATISTIC-MESSAGES= *UNCHANGED / *CPU-TIME / *NONE
,PROGRAM-INTERRUPT= *UNCHANGED / *INTEGER-OVERFLOW / *NONE
,ENVIRONMENT-ENCODING = *UNCHANGED / *STD / *EBCDIC

PARAMETER-PROMPTING = *UNCHANGED / *YES / *NO
*YES: Das ablauffähige Programm soll die UNIX-Umgebung simulieren, d.h.

- das Programm kann mit Parametern aufgerufen werden, wenn es mit START-EXECUTABLE-PROGRAM aufgerufen wird, bzw.
- nach Programmstart soll eine Parameterzeile erscheinen, in der Parameter für die `main`-Funktion oder Umweisungen von `stdin`, `stdout` oder `stderr` angegeben werden können (siehe auch Abschnitt „Parametereingaben nach dem Programmstart“).

*NO: Das ablauffähige Programm startet ohne Anforderung der Parameterzeile.

Bei Start des Programms aus der POSIX-Shell sind die Angaben in PARAMETER-PROMPTING-Option bedeutungslos, da in diesem Fall die Parametereingaben generell in der Kommandozeile erfolgen.

STACK-SIZE = 64 / *UNCHANGED / <integer 8..99999>
Mit dieser Option kann bestimmt werden, wieviel Platz für das erste Segment des C-Laufzeitstacks reserviert werden soll.
64: Voreingestellt sind 64 Kilobyte.
<integer>: Zwischen 8 und 99999 Kilobyte Speicherplatz können reserviert werden.

STATISTIC-MESSAGES = *UNCHANGED / *CPU-TIME / *NONE
Bei Beendigung eines Programms wird standardmäßig die verbrauchte CPU-Zeit ausgegeben. Diese Meldung kann mit *NONE unterdrückt werden.

PROGRAM-INTERRUPT = *UNCHANGED / *INTEGER-OVERFLOW / *NONE

Bei der Übersetzung von Programmen, die die `main`-Funktion enthalten, kann mit diesem Operanden die Programmaske eingestellt werden.

*INTEGER-OVERFLOW entspricht der ILCS-Programmaske X’0C’.

> ¡ Die Auswahl der generierten Befehle wird durch die Option nicht beeinflusst. Daher bedeutet das Zulassen von INTEGER-OVERFLOW nicht, dass in jedem Fall ein Überlauf ausgelöst wird.

*NONE entspricht der Programmaske X’00’.

Die beiden Programmmasken haben folgende Wirkung:

	INTEGER-OVERFLOW	NONE
Festpunkt-Überlauf	zugelassen	unterdrückt
Dezimal-Überlauf	zugelassen	unterdrückt
Exponenten-Unterlauf	unterdrückt	unterdrückt
Mantisse Null	unterdrückt	unterdrückt

ℹ Bei Sprachmix darf die ILCS-Programmmaske nicht verändert werden!

ENVIRONMENT-ENCODING = *UNCHANGED / *STD / *EBCDIC
Mit diesen Optionen kann gesteuert werden, wie externe Zeichenketten (Argumente von `main` und Umgebungsvariablen) behandelt werden.
*STD ist der Standardwert. Er bewirkt, dass die externen Zeichenketten so kodiert werden, wie es bei der Option MODIFY-SOURCE-PROPERTIES LITERAL-ENCODING angegeben wurde.
Die Option *EBCDIC wird aus Kompatibilitätsgründen angeboten und bewirkt, dass trotz der Angabe von MODIFY-SOURCE-PROPERTIES=*ASCII bzw. *ASCII-FULL externe Zeichenketten in EBCDIC kodiert werden.

3.2.2.18 MODIFY-SOURCE-PROPERTIES

Alias-Name: SET-SOURCE-PROPERTIES

Mit dieser Anweisung werden Quellprogrammeigenschaften festgelegt und das Verhalten des Präprozessors und des C- und C++-Frontends beinflusst.

MODIFY-SOURCE-PROPERTIES

```
/ *  Option zur Auswahl des Sprachmodus * /
LANGUAGE= *UNCHANGED / *C(...) / *CPLUSPLUS(...)
    *C(...)
        |   MODE= *UNCHANGED / *LATEST / *1990 / *2011 /
                        *KERNIGHAN-RITCHIE
        |   STRICT = *UNCHANGED / *NO / *YES
    *CPLUSPLUS(...)
        |   MODE= *UNCHANGED / *LATEST / *2017 /
                        *V2-COMPATIBLE / *V3-COMPATIBLE
        |   STRICT = *UNCHANGED / *NO / *YES
/ *  Präprozessor-Optionen * /
,DEFINE= *NONE / *UNCHANGED / list-poss: <c-string 1..125 with-low> / <name 1..125 with-under> /
                    *SUBSTITUTE(...)
    *SUBSTITUTE(...)
        |   IDENTIFIER= <c-string 1..125 with-low> / <name 1..125 with-under>
        |   ,TOKEN-STRING= <c-string 1..125 with-low> / <name 1..125 with-under>
,UNDEFINE= *NONE / *UNCHANGED / *ALL / list-poss: <c-string 1..125 with-low> /
                        <name 1..125 with-under>
,ASSERT= *NONE / *UNCHANGED / list-poss: *SUBSTITUTE(...)
    *SUBSTITUTE(...)
        |   IDENTIFIER= <c-string 1..125 with-low> / <name 1..125 with-under>
        |   ,TOKEN-STRING= <c-string 1..125 with-low> / <name 1..125 with-under>
,PREINCLUDE= *UNCHANGED / *NONE / <c-string 1..1024 with-low>
,COMMENTS= *UNCHANGED / *YES / *NO
,PREPROCESSING-MODE= *UNCHANGED / *ANSI / *KR
```

```
,IMPLICIT-INCLUDE= *UNCHANGED / *YES / *NO
 / *  Gemeinsame Frontend-Optionen in C und C++ * /
,SIGNED-CHARACTER= *UNCHANGED / *YES / *NO
,AT-ALLOWED= *UNCHANGED / *YES / *NO
,DOLLAR-ALLOWED= *UNCHANGED / *YES / *NO
,ENUM-TYPE= *UNCHANGED / *VALUE-DEPENDENT / *LONG
,SIGNED-FIELDS= *UNCHANGED / *SIGNED / *UNSIGNED
,PLAIN-FIELDS= *UNCHANGED / *SIGNED / *UNSIGNED
,PRESERVING= *UNCHANGED / *UNSIGNED / *LONG
,ALTERNATIVE-TOKENS= *UNCHANGED / *YES / *NO
,EXTERNAL-DEFINITION= *UNCHANGED / *BY-SOURCE-LANGUAGE / *UNIQUE /
                      *MULTIPLY-ALLOWED
,LONGLONG= *UNCHANGED / *YES / *NO
,END-OF-LINE-COMMENTS= *UNCHANGED / *YES / *NO
,LITERAL-ENCODING= *UNCHANGED / *NATIVE / *ASCII / *ASCII-FULL / *EBCDIC / *EBCDIC-FULL
/ *  C++-spezifische Optionen * /
,INSTANTIATION= *UNCHANGED / *NONE / *AUTO / *LOCAL / *ALL
,USE-STD-NAMESPACE= *UNCHANGED / *YES / *NO
,KEYWORD-BOOL= *UNCHANGED / *YES / *NO
,KEYWORD-WCHAR= *UNCHANGED / *YES / *NO
,LOOP-INIT= *UNCHANGED / *OLD / *NEW
,SPECIALIZATION= *UNCHANGED / *OLD / *NEW
```

Optionen zur Auswahl des Sprachmodus

LANGUAGE =

Mit dieser Option wird angegeben, in welcher Programmiersprache, C oder C++, die zu übersetzenden Quellen vorliegen.
Die Standardeinstellung des Compilers ist LANGUAGE=*CPLUSPLUS(MODE=*LATEST, STRICT=*NO).

LANGUAGE = *UNCHANGED

Es gilt die Angabe der letzten MODIFY-SOURCE-PROPERTIES-Anweisung.

LANGUAGE = *C(...)
Das Quellprogramm ist ein C-Programm.

MODE = *UNCHANGED / *LATEST / *1990 / *2011 / *KERNIGHAN-RITCHIE

Der Operand MODE bestimmt den C-Sprachmodus:

*UNCHANGED:

Es gilt die Angabe der letzten MODIFY-SOURCE-PROPERTIES-Anweisung mit LANGUAGE=*C.

*LATEST:

Diese Angabe entspricht der Angabe des letzten unterstützten Standards und kann sich in zukünftigen Compiler-Versionen ändern. In dieser Version des Compilers ist die Angabe identisch zu *2011.

*1990:

C89-Modus
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 *ANSI bzw. *STRICT-ANSI des C/C++ V3-Compilers.
`__STDC_VERSION__` hat den Wert 199409L.

*2011:

C11-Modus
Der Compiler unterstützt C-Code gemäß dem C-Standard von 2011.
`__STDC_VERSION__` hat den Wert 201112L.

*KERNIGHAN-RITCHIE:

K&R-C-Modus
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 CRTE zur Verfügung (ANSI-Funktionen, POSIX- und X/OPEN-Funktionen, UNIX-Erweiterungen).
Bezüglich des Präprozessor-Verhaltens ist ANSI-/ISO-C voreingestellt. Mit der Option PREPROCESSING-MODE=*KR kann das Präprozessor-Verhalten auf K&R-C umgestellt werden (ggf. bei der Portierung von alten C-Quellen aus einem UNIX-System notwendig).
`__STDC_VERSION__` ist undefiniert. Der Operand STRICT hat keinen Effekt, d.h. es gilt immer STRICT=*NO.

STRICT = *UNCHANGED / *NO / *YES

*NO:

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.

*YES:

Der Namensraum ist auf die im Standard definierten Namen beschränkt, und es stehen nur die im Standard definierten 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 ANSI-VIOLATIONS=*ERROR (siehe "MODIFY-DIAGNOSTIC-PROPERTIES") kann im Falle von Standard-Abweichungen die Ausgabe von Errors erzwungen werden.
__STDC__ hat den Wert 1, _STRICT_STDC ist definiert.

LANGUAGE = *CPLUSPLUS(...)
Das Quellprogramm ist ein C++-Programm. Dies ist auch die Standardeinstellung des Compilers, bevor die Programmiersprache zum ersten mal mit der LANGUAGE-Option definiert wird.

MODE = *UNCHANGED / *LATEST / *2017 / *V2-COMPATIBLE / *V3-COMPATIBLE

Der Operand MODE bestimmt den C++-Sprachmodus:

*UNCHANGED:

Es gilt die Angabe der letzten MODIFY-SOURCE-PROPERTIES-Anweisung mit LANGUAGE=*CPLUSPLUS.

*LATEST:

Diese Angabe entspricht der Angabe des letzten unterstützten Standards und kann sich in zukünftigen Compiler-Versionen ändern. In dieser Version des Compilers ist die Angabe identisch zu *2017.

*2017:

C++ 2017-Modus
Der Compiler unterstützt C++-Code gemäß dem C++-Standard von 2017.

__cplusplus hat den Wert 201703L und __STDC_VERSION__ den Wert 199409L.

*V2-COMPATIBLE:

Cfront-C++-Modus
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 Abschnitt „Die Cfront-C++-Bibliothek“.
C++-Quellen müssen mit MODE=*V2-COMPATIBLE übersetzt werden, wenn die Module mit C++-V2.1 /V2.2-Modulen verknüpfbar sein sollen.
Diese Angabe entspricht der Angabe *CPP des C/C++ V3-Compilers.

__cplusplus hat den Wert 1 und __STDC_VERSION__ den Wert 199409L. Der Operand STRICT hat keinen Effekt, d.h. es gilt immer STRICT=*NO.

*V3-COMPATIBLE:

C++ V3-Modus
Der Compiler unterstützt C++-Code entsprechend dem C/C++ V3-Compiler.
Diese Angabe entspricht der Angabe *ANSI bzw. *STRICT-ANSI des C/C++ V3-Compilers.

__cplusplus den Wert 2 (falls STRICT=*NO) bzw. 199612L (falls STRICT=*YES) und __STDC_VERSION__ den Wert 199409L.

STRICT = *UNCHANGED / *NO / *YES
*NO:

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.

*YES:

Der Namensraum ist auf die im Standard definierten Namen beschränkt, und es stehen nur die im Standard definierten 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 ANSI-VIOLATIONS=*ERROR (siehe "MODIFY-DIAGNOSTIC-PROPERTIES") kann im Falle von Standard-Abweichungen die Ausgabe von Errors erzwungen werden.
__STDC__ hat den Wert 1, _STRICT_STDC ist definiert.

Präprozessor-Optionen

DEFINE = *NONE
Für den aktuellen Übersetzungslauf gelten ausschließlich die im Programm durch `#define`-Anweisungen definierten bzw. die vom Compiler vordefinierten Namen und Werte.

DEFINE = *UNCHANGED
Es gelten die Angaben der letzten MODIFY-SOURCE-PROPERTIES-Anweisung.

DEFINE = <c-string 1..125 with-low> / <name 1..125 with-under>
Mit <c-string> / <name> wird ein Name definiert. Diese Definition hat dieselbe Wirkung wie folgende Anweisung in einem Programm:

`#define` *name* `1`

Solche Namen werden z.B. im Programm mit den Präprozessoranweisungen `#ifdef`, `#ifndef` bzw. `#if defined()`, `#if ! defined()` abgefragt. Siehe auch Beispiel ("MODIFY-SOURCE-PROPERTIES").

Bei Verwendung der POSIX-Bibliotheksfunktionen muss vor Auftreten der ersten `#include`-Anweisung im Programm das Define _OSD_POSIX gesetzt sein. Dies wird am einfachsten mit der Definition bei der Übersetzung erreicht.

DEFINE = *SUBSTITUTE(...)
Mit dieser Struktur lassen sich Makros und symbolische Konstanten definieren (analog zu einer `#define`-Anweisung für Textersatz). Siehe auch Beispiel unten.

IDENTIFIER = <c-string 1..125 with-low> / <name 1..125 with-under>
<c-string> / <name> bezeichnet den Namen, der im Quellprogramm durch den mit TOKEN-STRING angegebenen Wert bzw. Text ersetzt werden soll.

TOKEN-STRING = <c-string 1..125 with-low> / <name 1..125 with-under>
<c-string> / <name> gibt den Wert bzw. Text an, durch den der mit IDENTIFIER bezeichnete Name im Quellprogramm ersetzt werden soll.

Hinweis

Widersprechen Angaben in der DEFINE-Option irgendwelchen #define-Anweisungen im Quellprogramm, haben die Angaben im Quellprogramm immer Vorrang!

Beispiel: DEFINE-Option

```
MODIFY-SOURCE-PROP DEFINE=('mch_file',DEBUG,_OSD_POSIX,*SUB('host',BS2000),*SUB
(LAN,'C++'))
```

Mit DEFINE definierte Werte müssen in Hochkommatas eingeschlossen werden, wenn sie andere Zeichen als die Großbuchstaben A bis Z, die Ziffern 0 bis 9 oder die Sonderzeichen $, #, @ und _ enthalten (vgl. Tabelle 2 "SDF-Metasyntax").

Die obigen Angaben in der DEFINE-Option entsprechen folgenden #define-Anweisungen im Quellprogramm:

```
#define mch_file 1
#define DEBUG 1
#define _OSD_POSIX 1
#define host BS2000
#define LAN C++
```

UNDEFINE = *NONE
Die DEFINE-Angaben (s.o.) bleiben standardmäßig unverändert.

UNDEFINE = *UNCHANGED
Es gelten die Angaben der letzten MODIFY-SOURCE-PROPERTIES-Anweisung.

UNDEFINE = *ALL
Alle DEFINE-Angaben werden gelöscht.

UNDEFINE = <c-string 1..125 with-low> / <name 1..125 with-under>
Die mit DEFINE angegebenen Namen <c-string> / <name> werden gelöscht.

ASSERT = *NONE / *UNCHANGED / list-poss: *SUBSTITUTE(...)

Mit dieser Option kann ein Prädikat (Assertion) definiert werden, analog zur Präprozessor-Anweisung #assert (siehe "Erweiterungen gegenüber ANSI-/ISO-C").

ASSERT= *SUBSTITUTE(...)

IDENTIFIER = <c-string 1..125 with-low> / <name 1..125 with-under>
<c-string> / <name> bezeichnet den Namen des Prädikats.

TOKEN-STRING = <c-string 1..125 with-low> / <name 1..125 with-under>
<c-string> / <name> gibt den Wert bzw. Text an, für den das mit IDENTIFIER bezeichnete Prädikat gilt.

PREINCLUDE = *UNCHANGED / *NONE / <c-string 1..1024 with-low>
Mit dieser Option wird eine Include-Datei spezifiziert, die via (imaginärer) #include-Anweisung am Anfang des Quellprogramms inkludiert wird (Pre-Include). Der Präprozessor sucht diese Include-Datei in den USER-INCLUDE-Pfaden.

Die via PREINCLUDE-Option spezifizierte Include-Datei wird wie eine Include-Datei behandelt, die innerhalb einer #include-Anweisung am Anfang des Quellprogramms angegeben ist.

Sollen mehr als eine Include-Datei pre-inkludiert werden, dann müssen mehrere #include-Anweisungen in einer einzigen Include-Datei zusammengefasst werden, die dann via PREINCLUDE-Option zu spezifizieren ist.

COMMENTS = *UNCHANGED / *YES / *NO
Mit dieser Option lässt sich festlegen, ob das vom Präprozessor erzeugte expandierte Quellprogramm auch die Kommentare enthalten soll.

PREPROCESSING-MODE = *UNCHANGED / *ANSI / *KR

*UNCHANGED

Es gelten die Angaben der letzten MODIFY-SOURCE-PROPERTIES-Anweisung.

*ANSI

Dies ist die Voreinstellung in allen C- und C++-Sprachmodi des Compilers. Das heißt, dass standardmäßig auch im K&R-C-Modus das Präprozessor-Verhalten entsprechend dem ANSI-/ISO-C-Standard unterstützt wird.

*KR

Mit *KR kann das veraltete Präprozessor-Verhalten gemäß Reiser cpp und Johnson pcc eingeschaltet werden.

IMPLICIT-INCLUDE = *UNCHANGED / *YES / *NO
Diese Option betrifft nur C++-Templates. Es wird festgelegt, ob die Definition eines Templates implizit inkludiert wird (siehe Abschnitt „Implizites Inkludieren“).

Gemeinsame Frontend-Optionen in C und C++

SIGNED-CHARACTER = *UNCHANGED / *YES / *NO

*UNCHANGED

Es gelten die Angaben der letzten MODIFY-SOURCE-PROPERTIES-Anweisung.

*NO

Der Datentyp `char` ist standardmäßig vom Typ `unsigned`.

*YES

`char` wird in Ausdrücken und Konversionen als `signed char` behandelt. Bei Verwendung dieser Option können Portabilitätsprobleme auftreten!

AT-ALLOWED = *UNCHANGED / *YES / *NO
Das at-Zeichen ’@’ ist in Bezeichnern erlaubt (*YES) bzw. nicht erlaubt (*NO).

> ¡ Die Cfront-C++-Bibliothek enthält Deklarationen mit dem „at“-Zeichen (@) (siehe "Die Cfront-C++-Bibliothek").

DOLLAR-ALLOWED = *UNCHANGED / *YES / *NO
Das Dollar-Zeichen ’$’ ist in Bezeichnern erlaubt (*YES) bzw. nicht erlaubt (*NO).

ENUM-TYPE = *UNCHANGED / *VALUE-DEPENDENT / *LONG

Diese Option steuert die Behandlung von `enum`-Daten.

*UNCHANGED

Es gelten die Angaben der letzten MODIFY-SOURCE-PROPERTIES-Anweisung.

*VALUE-DEPENDENT

Standardmäßig werden `enum`-Daten abhängig vom Wertebereich auf `char`, `short` oder `long` abgebildet.

*LONG

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

SIGNED-FIELDS = *UNCHANGED / *SIGNED / *UNSIGNED
*SIGNED: Standardmäßig werden `signed` Bitfelder wie `signed` behandelt.

*UNSIGNED: `signed` Bitfelder sind immer vom Typ `unsigned`. Diese Angabe wird aus Kompatibilitätsgründen zu älteren C-Versionen angeboten und ist nur im K&R-C-Modus sinnvoll.

PLAIN-FIELDS = *UNCHANGED / *SIGNED / *UNSIGNED
Diese Option steuert, ob Integer-Bitfelder (`short`, `int`, `long`) standardmäßig vom Typ `signed` (*SIGNED) oder `unsigned` (*UNSIGNED) sind.

PRESERVING = *UNCHANGED / *UNSIGNED / *LONG
Diese Option steuert, 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) oder gemäß ANSI-/ISO-C vom Typ `unsigned long` (*UNSIGNED).

ALTERNATIVE-TOKENS = *UNCHANGED / *YES / *NO

Diese Option steuert, 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 `&`).

*YES ist die Voreinstellung in den Sprachmodi C 2011, C++ V3 und C++ 2017.

*NO ist die Voreinstellung in allen anderen Sprachmodi.

EXTERNAL-DEFINITION =
Diese Option steuert, wie der Compiler den Speicher für die extern sichtbaren Variablen eines Moduls anlegt. Dies ist dann von Bedeutung, wenn das Programm aus mehreren Modulen besteht, die anschließend zu einem Programm gebunden werden.

EXTERNAL-DEFINITION = *UNCHANGED
Es gelten die Angaben der letzten MODIFY-SOURCE-PROPERTIES-Anweisung.

EXTERNAL-DEFINITION = *BY-SOURCE-LANGUAGE
Der Wert der Option EXTERNAL-DEFINITION richtet sich nach den Angaben in den Sprachmodus-Optionen:

LANGUAGE=*C(MODE=*KERNIGHAN-RITCHIE): MULTIPLY-ALLOWED

LANGUAGE=*C(MODE=*1990/*2011): UNIQUE

LANGUAGE=*CPLUSPLUS(): UNIQUE

EXTERNAL-DEFINITION = *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. Ist die Variable in mehr als einem Modul definiert, so erhält man beim Binden eine entsprechende Fehlermeldung.

EXTERNAL-DEFINITION = *MULTIPLY-ALLOWED
Diese Angabe wird für Programme verwendet, in denen eine extern sichtbare Variable in mehreren Modulen definiert wird, aber nur genau einem Speicherbereich zugeordnet werden soll. 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, so dass später nach dem Binden nur genau ein Speicherbereich der mehrfach definierten Variable zugeordnet ist.
Wenn die Variable bei der Definition statisch initialisiert wird, wird der Speicherbereich nicht im COMMON-Bereich, sondern im Datenbereich angelegt. Die Zuordnung zu genau einem Speicherbereich ist dann nicht möglich!

Diese Angabe ist in den C++-Sprachmodi nicht erlaubt.

LONGLONG = *UNCHANGED / *YES / *NO

Diese Option steuert, ob der Datentyp `long long` vom Compiler erkannt wird.

*UNCHANGED

Es gelten die Angaben der letzten MODIFY-SOURCE-PROPERTIES-Anweisung.

*YES

Standardmäßig wird der Datentyp `long long` erkannt. In diesem Fall wird das Präprozessor-Define `_LONGLONG` gesetzt.

*NO

Der Gebrauch des Datentyps `long long` führt zu einem Fehler. Dieser Wert ist nur im strikten C89-Modus und im strikten C++ V3-Modus zulässig.

END-OF-LINE-COMMENTS = *UNCHANGED / *YES / *NO

Diese Option steuert, ob der Compiler C++-Kommentare (//...) auch in C-Programmen akzeptiert. C++-Kommentare können nur im erweiterten C89-Modus (MODE=*1990, STRICT=*NO) zugelassen werden. 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.

*UNCHANGED

Es gelten die Angaben der letzten MODIFY-SOURCE-PROPERTIES-Anweisung.

*YES

Der Compiler akzeptiert C++-Kommentare im erweiterten C89-Modus.

*NO

Der Compiler akzeptiert keine C++-Kommentare im erweiterten C89-Modus (Voreinstellung).

LITERAL-ENCODING = *UNCHANGED / *NATIVE / *ASCII / *ASCII-FULL / *EBCDIC / *EBCDIC-FULL

Diese Option legt fest, ob der C/C++-Compiler Objekt-Code für EBCDIC-Zeichen und EBCDIC-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 dem bei LITERAL-ENCODING = ... spezifizierten Wert ab.

LITERAL-ENCODING = *UNCHANGED
Es gelten die Einstellungen der letzten MODIFY-SOURCE-PROPERTIES-Anweisung.

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.
*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 Objekt-Code.

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 Zeichen (ketten) unkonvertiert in den Objekt-Code.

LITERAL-ENCODING=*EBCDIC-FULL hat somit dieselbe Wirkung wie LITERAL-ENCODING=*EBCDIC oder LITERAL-ENCODING=*NATIVE

Voraussetzungen:

- Bei Nutzung der ASCII-Darstellung für Zeichen und Zeichenketten dürfen Sie 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.

- Falls die Option ASCII oder ASCII_FULL gewählt wird: Inkludieren Sie für *jede* in Ihrem Programm verwendete CRTE-Funktion (C-Bibliotheksfunktion), die mit Zeichen(ketten) 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.

  Im Laufzeitsystem CRTE gibt es einige C-Bibliotheksfunktionen, die mit ASCII-Zeichenketten arbeiten.

  Für die Verwendung von ASCII-Zeichenketten sollte die MODIFY-SOURCE-PROPERTIES-Anweisung wie folgt angegeben werden:

```
MODIFY-SOURCE-PROPERTIES           -
   ...
   LITERAL-ENCODING=ASCII[-FULL]  -
   ...
```

  Außerdem muss die MODIFY-MODULE-PROPERTIES-Anweisung mit folgenden Angaben spezifiziert werden:

```
MODIFY-MODULE-PROPERTIES           -
   ...
   LOWER-CASE-NAMES=*YES,          -
   SPECIAL-CHARACTERS=*KEEP,       -
   ...
```

Achtung:

C++ Bibliotheksfunktionen unterstützen das ASCII-Format nicht und müssen deshalb gegebenenfalls durch C-Funktionen ersetzt werden.

C++-spezifische Frontend-Optionen

INSTANTIATION = *UNCHANGED / *NONE / *AUTO / *LOCAL / *ALL

Diese Option ist nur für die Modi C++ V3 und C++ 2017 relevant. Sie steuert die Art der Instanziierung von Templates mit externer Linkage. Dazu zählen Funktions-Templates sowie Funktionen 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:

*NONE: Außer den explizit angeforderten Instanzen werden sonst keine Instanzen generiert.

*AUTO (Voreinstellung): Die Instanziierung erfolgt über alle Übersetzungseinheiten hinweg durch einen Prälinker. Der Prälinker wird erst mit der BIND-Anweisung aktiviert (siehe "BIND"). Das Prinzip der automatischen Instanziierung ist ausführlich im Abschnitt „Automatische Instanziierung“ dargestellt.

*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 `*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.

*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.

USE-STD-NAMESPACE = *UNCHANGED / *YES / *NO

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

*YES ist die Voreinstellung im erweiterten C++ V3-Modus. Das Verhalten ist dann so, als ob am Beginn einer Übersetzungseinheit die folgenden Zeilen stehen würden:

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

*NO ist die Voreinstellung im strikten C++ V3-Modus und im C++ 2017-Modus und das einzig mögliche Verhalten im Cfront-C++-Modus.

Wenn im C++ V3-Modus oder im C++ 2017-Modus USE-STD-NAMESPACE=*NO gesetzt ist, muss das Quellprogramm vor dem ersten Gebrauch einer C++-Bibliotheksfunktion die Anweisung `using namespace std;` enthalten oder die Namen entsprechend qualifizieren.

KEYWORD-BOOL = *UNCHANGED / *YES / *NO

Mit dieser Option wird festgelegt, ob `bool` als Schlüsselwort erkannt wird.

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

*NO ist die Voreinstellung und das einzig mögliche Verhalten im Cfront-C++-Modus.

KEYWORD-WCHAR = *UNCHANGED / *YES / *NO

Mit dieser Option wird festgelegt, ob `wchar_t` als Schlüsselwort erkannt wird.

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

*NO ist die Voreinstellung und das einzig mögliche Verhalten im Cfront-C++-Modus.

LOOP-INIT = *UNCHANGED / *OLD / *NEW

Mit dieser Option wird festgelegt, wie eine Initialisierungsanweisung in `for`- und `while`-Schleifen behandelt werden soll.

*OLD ist die Voreinstellung im Cfront-C++-Modus und bewirkt, dass eine Initialisierungsanweisung zum selben Gültigkeitsbereich wie die gesamte Schleife gehört.

*NEW ist die Voreinstellung im C++ V3-Modus und das einzig mögliche Verhalten im C++ 2017-Modus. Die Angabe spezifiziert die neue ANSI-C++konforme Gültigkeitsbereichsregel, die die gesamte Schleife mit ihrem eigenen implizit generierten Gültigkeitsbereich umgibt.

SPECIALIZATION = *UNCHANGED / *OLD / *NEW

Diese Option ist nur im C++ V3-Modus und im C++ 2017-Modus relevant und legt fest, ob für Template-Spezialisierungen die neue Syntax `template<>` verpflichtend ist.

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

Bei der Angabe von *OLD definiert der Compiler das Makro `__OLD_SPECIALIZATION_SYNTAX` implizit mit dem Wert 1.

3.2.2.19 MODIFY-TEST-PROPERTIES

Alias-Name: SET-TEST-PROPERTIES

Mit dieser Anweisung wird gesteuert, ob Informationen für die Testhilfe AID erzeugt werden.

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

TEST-SUPPORT = *YES
Es werden Testhilfe-Informationen für AID erstellt.
Für uneingeschränktes Testen mit AID muss die Optimierung und Inline-Generierung von benutzereigenen Funktionen unterdrückt werden (siehe MODIFY-OPTIMIZATION-PROPERTIES). Der Compiler nimmt INLINING=*NO an. Das Optimierungslevel wird auf *LOW zurückgesetzt.

TEST-SUPPORT = NO
Es werden keine Testhilfe-Informationen für AID erstellt.
Eine Rückverfolgung der Aufrufhierarchien ist dennoch möglich (z.B. nach Programmabbruch die Angabe % SDUMP %NEST).

3.2.2.20 PREPROCESS

Alias-Name: DO-PREPROCESSING

Die Übersetzung eines oder mehrerer Quellprogramme wird nach der Präprozessorphase beendet. Dabei kann pro Übersetzungseinheit ein expandiertes, weiterübersetzbares Quellprogramm erzeugt werden.

```
PREPROCESS

SOURCE= *SYSDTA / list-poss: <filename 1..54> / <posix-pathname> / *LIBRARY-ELEMENT(...)
    *LIBRARY-ELEMENT(...)
        |  LIBRARY= <filename 1..54> / *LINK(...)
        |      *LINK(...)
        |          |  LINK-NAME= <filename 1..8>
        |  ,ELEMENT= <composed-name 1..64 with-under>(...)
        |      <composed-name 1..64 with-under>(...)
        |          |  VERSION= *HIGHEST-EXISTING / <composed-name 1..24 with-under>
,OUTPUT= *NONE / *STD-FILE / *SOURCE-LOCATION / <filename 1..54>/
                <posix-pathname> / *LIBRARY-ELEMENT(...)
    *LIBRARY-ELEMENT(...)
        |  LIBRARY= *STD-LIBRARY / *SOURCE-LIBRARY / <filename 1..54> / *LINK(...)
        |      *LINK(...)
        |          |  LINK-NAME= <filename 1..8>
        |  ,ELEMENT= *STD-ELEMENT(...) / <composed-name 1..64 with-under>(...)
        |      *STD-ELEMENT(...)
        |          |  VERSION = *UPPER-LIMIT / *INCREMENT / <composed-name 1..24 with-under>
        |      <composed-name 1..64 with-under>(...)
        |          |  VERSION= *UPPER-LIMIT / *INCREMENT / <composed-name 1..24 with-under>
```

SOURCE =
Mit dieser Option werden eines oder mehrere Quellprogramme angegeben, die übersetzt werden sollen.
Ein Quellprogramm kann von der Systemdatei SYSDTA, aus einer katalogisierten BS2000-Datei, aus einer PLAM-Bibliothek oder aus einer POSIX-Datei eingelesen werden.
Bei der Quellprogrammeingabe von SYSDTA kann pro PREPROCESS-Anweisung nur ein Quellprogramm eingelesen werden.

SOURCE = *SYSDTA
Die Eingabe erfolgt von der Systemdatei SYSDTA. SYSDTA ist im Dialogbetrieb der Datensichtstation zugewiesen und kann mit dem ASSIGN-SYSDTA-Kommando auf eine katalogisierte Datei oder ein PLAM-Bibliothekselement umgewiesen werden (siehe auch "Hinweise zur Eingabe über SYSDTA").

SOURCE = <filename 1..54>
<filename> ist der Name einer katalogisierten BS2000-Datei.

SOURCE = <posix-pathname>
Als <posix-pathname> ist nur ein Dateiname zugelassen. Zur Beschreibung des Begriffs <posix-pathname> siehe "Compiler-Ein-/Ausgaben im POSIX-Dateisystem".

SOURCE = *LIBRARY-ELEMENT(...)
Es werden eine PLAM-Bibliothek und ein Element daraus angegeben.

LIBRARY = <filename 1..54>
<filename> ist der Name einer PLAM-Bibliothek.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>
<filename> ist der Linkname einer PLAM-Bibliothek. Der Linkname muss vor Aufruf des Compilers mit dem ADD-FILE-LINK-Kommando dem Bibliotheksnamen zugeordnet worden sein.

ELEMENT = <composed-name 1..64 with-under>(...)
<composed-name> bezeichnet den vollqualifizierten Namen eines Elements aus der zuvor angegebenen PLAM-Bibliothek. Das Element muss vom Typ S sein.

VERSION = *HIGHEST-EXISTING
Enthält die Elementangabe keine Versionsbezeichnung, nimmt der Compiler das Element mit der höchsten Version.

VERSION = <composed-name 1..24 with-under>
Der Compiler nimmt das Element mit der angegebenen Version.

OUTPUT =
Mit dieser Option kann gesteuert werden, ob und wohin das Ergebnis des Präprozessorlaufs abgelegt werden soll.

OUTPUT = *NONE
Es wird kein expandiertes, weiterübersetzbares Quellprogramm erzeugt. Das Ergebnis des Präprozessorlaufs (Auflösungen und ggf. Fehlermeldungen) kann nur in einer angeforderten Präprozessorliste überprüft werden.

OUTPUT = *STD-FILE

Standardmäßig wird das expandierte Programm in eine katalogisierte BS2000-Datei geschrieben. Der Name dieser Datei wird aus dem Namen des Quellprogramms abgeleitet:

Quelle	*SYSDTA	BS2000-Datei	PLAM-Bibliothek	POSIX-Datei
Standardname	CSTDEXP.I	datei.I	bib-elem.I	datei.I

Wenn das Quellprogramm in einer PLAM-Bibliothek steht, werden im Standard-Dateinamen der Bibliotheks- und Elementname der Quelle verwendet und mit einem Bindestrich verbunden (bib-elem). Die Regeln zur Bildung der Standardnamen durch den Compiler sind ausführlich im Abschnitt „Standardnamen für Ausgabebehälter“ dargestellt.

OUTPUT = *SOURCE-LOCATION
Ausgabeort und Name des expandierten Programms werden wie folgt aus dem Ort und Namen des Quellprogramms abgeleitet:

Quelle	*SYSDTA	BS2000-Datei	PLAM-Bibliothek	POSIX-Datei
Ausgabeort	BS2000-Datei	BS2000-Datei	Bibliothek der Quelle	Dateiverzeichnis der Quelle
Standardname	CSTDEXP.`I`	datei.`I`	elem.`I` (Typ S)	datei.i (C-Quelle) datei.`I` (C++-Quelle

Die Regeln zur Bildung der Standardnamen durch den Compiler sind ausführlich im Abschnitt „Standardnamen für Ausgabebehälter“ dargestellt.

OUTPUT = <filename 1..54>
Das expandierte Programm wird in eine katalogisierte BS2000-Datei mit dem angegebenen Namen geschrieben. Diese Angabe ist unzulässig, wenn mehrere Quellprogramme übersetzt werden.

OUTPUT = <posix-pathname>
Das expandierte Programm wird in das POSIX-Dateisystem geschrieben.

Als <posix-pathname> ist sowohl ein Dateiname als auch ein Dateiverzeichnis zugelassen. Zur Beschreibung des Begriffs <posix-pathname> siehe "Compiler-Ein-/Ausgaben im POSIX-Dateisystem".

Bei der Angabe eines Dateinamens wird das expandierte Programm unter diesem Namen abgelegt. Die Angabe eines Dateinamens ist unzulässig, wenn mehrere Quellprogramme mit einer Anweisung übersetzt werden.

Bei der Angabe eines Dateiverzeichnisnamens *dvz* wird für jedes Quellprogramm das expandierte Programm unter dem Standardnamen *quelldatei.i* (C-Quelle) bzw. *quelldatei.I* (C++-Quelle) in das Dateiverzeichnis *dvz* geschrieben (siehe Abschnitt „Standardnamen fürAusgabebehälter“).

Die mit <posix-pathname> angegebenen Dateiverzeichnisse müssen bereits existieren. Bei der Bildung der Dateinamen ist zu beachten, dass expandierte Quellprogramme im POSIX-Subsystem nur sinnvoll weiterverarbeitet werden können, wenn der Name das Suffix `.i` bzw. `.I` enthält oder ein Suffix, das mit der Option `-Y F` der `cc`/`c11`/`c89`/CC-Kommandos vereinbart wurde (siehe Handbuch „POSIX-Kommandos des C/C++-Compilers“ [1]).

OUTPUT = *LIBRARY-ELEMENT(...)
Es wird angegeben, in welcher PLAM-Bibliothek (LIBRARY=) und unter welchem Elementnamen (ELEMENT=) das expandierte Programm abgelegt werden soll. Die Elemente werden unter dem Typ S abgespeichert.

LIBRARY = *STD-LIBRARY
Das expandierte Programm wird standardmäßig in die Bibliothek SYS.PROG.LIB geschrieben.

LIBRARY = *SOURCE-LIBRARY
Das expandierte Programm wird in die PLAM-Bibliothek geschrieben, in der das Quellprogramm steht.
Die Angabe *SOURCE-LIBRARY ist unzulässig, wenn das Quellprogramm aus einer katalogisierten BS2000-Datei, einer POSIX-Datei oder über SYSDTA eingelesen wird.

LIBRARY = <filename 1..54>
Das expandierte Programm wird in die PLAM-Bibliothek mit dem angegebenen Namen geschrieben.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>
Mit <filename> kann ein gültiger Linkname für die PLAM-Bibliothek angegeben werden. Der Linkname muss vor Aufruf des Compilers mit dem ADD-FILE-LINK-Kommando dem Bibliotheksnamen zugeordnet worden sein.

ELEMENT = *STD-ELEMENT(...)
Standardmäßig wird der Elementname des expandierten Programms aus dem Namen des Quellprogramms abgeleitet:

Quelle	*SYSDTA	BS2000-Datei	PLAM-Bibliothek	POSIX-Datei
Standardname	CSTDEXP.I	datei.I	elem.I	datei.I

Die Regeln zur Bildung der Standardnamen durch den Compiler sind ausführlich im Abschnitt „Standardnamen für Ausgabebehälter“ dargestellt.

VERSION = *UPPER-LIMIT
Enthält die Elementangabe keine Versionsbezeichnung, verwendet der Compiler die höchstmögliche Version.

VERSION = *INCREMENT
Das Element erhält die gegenüber der höchsten vorhandenen Version um 1 inkrementierte Versionsnummer, vorausgesetzt, die höchste vorhandene Versionsbezeichnung endet mit einer inkrementierbaren Zahl. Ist die Versionbezeichnung nicht inkrementierbar, wird der Übersetzungslauf mit Fehler abgebrochen. Beispiel siehe COMPILE-Anweisung ("COMPILE").

VERSION = <composed-name 1..24 with-under>
Der Compiler verwendet die angegebene Version.

ELEMENT = <composed-name 1..64 with-under>(...)
<composed-name> bezeichnet den vollqualifizierten Elementnamen des expandierten Programms. Diese Angabe ist unzulässig, wenn mehrere Quellprogramme übersetzt werden.

VERSION = *UPPER-LIMIT / *INCREMENT / <composed-name 1..24 with-under>
Die Version kann in gleicher Weise angegeben werden, wie oben unter ELEMENT=*STD-ELEMENT(...) beschrieben.

3.2.2.21 RESET-TO-DEFAULT

Diese Anweisung setzt die aktuellen Optionswerte in den entsprechenden MODIFY-Anweisungen auf die Standardwerte des Compilers, die unmittelbar nach dem START-CPLUS-COMPILER-Kommando gelten.

RESET-TO-DEFAULT
SELECT= *UNCHANGED / *ALL / list-poss(10): *INCLUDE / *SOURCE / *MODULE / *OPTIMIZATION / *RUNTIME / *TEST / *DIAGNOSTIC / *LISTING / *CIF / *BIND

SELECT = *UNCHANGED
Es gilt die Angabe der letzten RESET-TO-DEFAULT-Anweisung.

SELECT = *ALL
Es werden die Optionswerte aller MODIFY-Anweisungen auf die Standardwerte des Compilers gesetzt.

SELECT = list-poss(10): *INCLUDE / *SOURCE / *MODULE / *OPTIMIZATION /*RUNTIME / *TEST / *DIAGNOSTIC / *LISTING / *CIF / *BIND
Es werden nur die Optionswerte der angegebenen MODIFY-Anweisungen auf die Standardwerte des Compilers gesetzt. *INCLUDE steht z.B. für MODFY-INCLUDE-LIBRARIES, *SOURCE für MODIFY-SOURCE-PROPERTIES etc.

*BIND bewirkt, dass zusätzlich zu den Optionen der MODIFY-BIND-PROPERTIES-Anweisung, die ACTION- und OUTPUT-FORMAT-Optionen der BIND-Anweisung auf den Standardwert des Compilers gesetzt werden.

Über die Standardwerte des Compilers kann man sich mit der Anweisung SHOW-DEFAULTS informieren (siehe "SHOW-DEFAULTS"). Im Handbuch sind die Standardwerte des Compilers durch Unterstreichung gekennzeichnet.

3.2.2.22 SHOW-DEFAULTS

Diese Anweisung zeigt die Standardwerte des Compilers an, die in den entsprechenden MODIFY-Anweisungen unmittelbar nach dem START-CPLUS-COMPILER-Kommando gelten und solange eingesetzt werden, bis ein Wert explizit geändert wird. Diese Werte können auch mit der Anweiung RESET-TO-DEFAULT gesetzt werden.

SHOW-DEFAULTS
SELECT= *UNCHANGED / *ALL / list-poss(10): *INCLUDE / *SOURCE / *MODULE / *OPTIMIZATION / *RUNTIME / *TEST / *DIAGNOSTIC / *LISTING / *CIF / *BIND ,OUTPUT= *UNCHANGED / *SYSOUT / *SYSLST

SELECT = *UNCHANGED
Es gilt die Angabe der letzten SHOW-DEFAULTS-Anweisung.

SELECT = *ALL
Es werden die Informationen zu allen MODIFY-Anweisungen ausgegeben.

SELECT = list-poss(10): *INCLUDE / *SOURCE / *MODULE / *OPTIMIZATION /*RUNTIME / *TEST / *DIAGNOSTIC / *LISTING / *CIF / *BIND
Es werden nur zu den angegebenen MODIFY-Anweisungen Informationen ausgegeben. *INCLUDE steht z.B. für MODIFY-INCLUDE-LIBRARIES, *SOURCE für MODIFY-SOURCE-PROPERTIES etc.

OUTPUT = *UNCHANGED / *SYSOUT / *SYSLST
Die Informationen werden über SYSOUT (Voreinstellung) oder über SYSLST ausgegeben.

Hinweis

Für einige Optionen der Anweisung MODIFY-SOURCE-PROPERTIES wird der Optionswert __unset__ angezeigt. Dabei handelt es sich um Optionen, die abhängig vom Sprachmodus unterschiedliche Standardwerte haben können (sog. „alternative Defaults“).

3.2.2.23 SHOW-PROPERTIES

Diese Anweisung zeigt die Optionswerte an, die aktuell in den entsprechenden MODIFY-Anweisungen eingestellt sind. In der Ausgabe werden indirekte Werte soweit möglich durch reale Werte ersetzt.

SHOW-PROPERTIES
SELECT= *UNCHANGED / *ALL / list-poss(10): *INCLUDE / *SOURCE / *MODULE / *OPTIMIZATION / *RUNTIME / *TEST / *DIAGNOSTIC / *LISTING / *CIF / *BIND ,OUTPUT= *UNCHANGED / *SYSOUT / *SYSLST

SELECT = *UNCHANGED
Es gilt die Angabe der letzten SHOW-PROPERTIES-Anweisung.

SELECT = *ALL
Es werden die Informationen zu allen MODIFY-Anweisungen ausgegeben.

SELECT = list-poss(10): *INCLUDE / *SOURCE / *MODULE / *OPTIMIZATION /*RUNTIME / *TEST / *DIAGNOSTIC / *LISTING / *CIF / *BIND
Es werden nur zu den angegebenen MODIFY-Anweisungen Informationen ausgegeben. *INCLUDE steht z.B. für MODFY-INCLUDE-LIBRARIES, *SOURCE für MODIFY-SOURCE-PROPERTIES etc.

OUTPUT = *UNCHANGED / *SYSOUT / *SYSLST
Die Informationen werden über SYSOUT (Voreinstellung) oder über SYSLST ausgegeben.

Hinweis

Unmittelbar nach dem Start des Compilers bzw. nach einer RESET-TO-DEFAULT-Anweisung unterscheiden sich die Ausgaben von SHOW-PROPERTIES und SHOW-DEFAULTS nur dadurch, dass bei SHOW-PROPERTIES statt des Optionswertes __unset__ der tatsächliche Wert angezeigt wird (entsprechend dem voreingestellten Sprachmodus).

3.3 Steuerung des globalen Listengenerators

Der globale Listengenerator wird mit dem Kommando START-CPLUS-LISTING-GENERATOR aufgerufen. Eingabequellen für den Listengenerator sind vom Compiler generierte CIF-Informationen, die er pro Übersetzungseinheit in PLAM-Bibliothekselemente (Typ H), in katalogisierte BS2000-Dateien oder in POSIX-Dateien abspeichert (siehe "MODIFY-CIF-PROPERTIES"). Die vom globalen Listengenerator erzeugten Listen werden in eine einzige Ausgabedatei geschrieben, und zwar standardmäßig in die Systemdatei SYSLST, bei Angabe der Option OUTPUT 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.

3.3.1 Aufruf des Listengenerators (START-CPLUS-LISTING-GENERATOR)

/START-CPLUS-LISTING-GENERATOR	Kurznamen: S-CP-L-G, CPLUS-LISTING-GENERATOR, CPLG
MONJV= *NONE / <filename 1..54> ,CPU-LIMIT = *JOB-REST / <integer 1..32767>	

MONJV = *NONE / <filename 1..54>
<filename> gibt den Namen einer überwachenden Jobvariablen an, in die der Listengenerator eine Anzeige über mögliche Ablauffehler hinterlegt. Die Anzeigen sind identisch mit denen des C/C++-Compilers (siehe "Aufruf des Compilers (START-CPLUS-COMPILER)").

3.3.2 Beschreibung der Anweisungen

In diesem Abschnitt werden folgende Themen behandelt:

- Anweisungsübersicht und Eingaberegeln
- END
- GENERATE-LISTING
- MODIFY-LISTING-PROPERTIES

3.3.2.1 Anweisungsübersicht und Eingaberegeln

Zur Steuerung des globalen Listengenerators gibt es folgende Anweisungen:

- die modifizierende Anweisung MODIFY-LISTING-PROPERTIES legt die Art, das Layout und das Ausgabeziel der zu generierenden Listen fest
- die ausführende Anweisung GENERATE-LISTING legt die Eingabequellen fest und startet die eigentliche Listengenerierung
- die END-Anweisung beendet den globalen Listengenerator
- die SDF-Standardanweisungen (analog zur Steuerung des Compilers, siehe "Anweisungsübersicht")

Die Anweisung MODIFY-LISTING-PROPERTIES muss der Anweisung GENERATE-LISTING vorausgehen. Ohne Angabe von MODIFY-LISTING-PROPERTIES werden gemäß Voreinstellung keine Listen erzeugt. Die allgemeine Regeln zur SDF-Anweisungsschnittstelle des Compilers treffen auch auf den globalen Listengenerator zu (siehe „Prinzip und allgemeine Eingaberegeln“).

3.3.2.2 END

Diese Anweisung beendet den Ablauf des Listengenerators.

```
END
```

3.3.2.3 GENERATE-LISTING

Alias-Namen: DO-LISTING-GENERATION, LIST

Diese Anweisung legt die Eingabequellen für den globalen Listengenerator fest und startet die Listengenerierung. Umfang, Layout und Ausgabeziel der Listen richten sich nach den Angaben in einer vorangehenden MODIFY-LISTING-PROPERTIES-Anweisung.

```
GENERATE-LISTING

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

    *LIBRARY-ELEMENT(...)

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

        |       *LINK(...)

        |           |   LINK-NAME= <filename 1..8>

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

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

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

CIF-FILE = <filename 1..54>
<filename> ist der Name einer katalogisierten BS2000-Datei, die CIF-Informationen enthält.

CIF-FILE = <posix-pathname>
<posix-pathname> ist der Name einer POSIX-Datei, die CIF-Informationen enthält. Zur Beschreibung des Begriffs <posix-pathname> siehe "Compiler-Ein-/Ausgaben im POSIX-Dateisystem".

CIF-FILE = *LIBRARY-ELEMENT(...)
Es werden eine PLAM-Bibliothek und eines oder mehrere Elemente (Typ H) daraus angegeben, die CIF-Informationen enthalten.

LIBRARY = *STD-LIBRARY
Standardmäßig verarbeitet der Listengenerator CIF-Elemente, die in der Bibliothek SYS.PROG.LIB stehen.

LIBRARY = <filename 1..54>
<filename> ist der Name einer PLAM-Bibliothek.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>

<filename> ist der Linkname einer PLAM-Bibliothek. Der Linkname muss vor Aufruf des Listengenerators mit dem ADD-FILE-LINK-Kommando dem Bibliotheksnamen zugeordnet worden sein.

ELEMENT = <composed-name 1..64 with-under>(...)
<composed-name> bezeichnet den vollqualifizierten Namen eines CIF-Elementes aus der zuvor angegebenen PLAM-Bibliothek. Das Element muss vom Typ H sein.

VERSION = *HIGHEST-EXISTING
Enthält die Elementangabe keine Versionsbezeichnung, nimmt der Listengenerator das Element mit der höchsten Version.

VERSION = <composed-name 1..24 with-under>
Der Listengenerator nimmt das Element mit der angegebenen Version.

3.3.2.4 MODIFY-LISTING-PROPERTIES

Alias-Name: SET-LISTING-PROPERTIES

Mit dieser Anweisung kann man auswählen, welche Listen der Listengenerator erzeugen soll. Außerdem kann man das Layout beinflussen und den Ausgabeort der Listen festlegen. Die Syntax der Anweisung ist bis auf Abweichungen in der OUTPUT-Option identisch mit der gleichnamigen Anweisung des Compilers.

MODIFY-LISTING-PROPERTIES

OPTIONS = *UNCHANGED / *YES / *NO

,SOURCE = *UNCHANGED / *NO / [*YES](...)

*YES(...)

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

,PREPROCESSING-RESULT = *UNCHANGED / *NO / *YES

,DATA-ALLOCATION-MAP = *UNCHANGED / *NO / [*YES](...)

*YES(...)

| STRUCTURE-LEVEL = *UNCHANGED / *NONE / *MAX / <integer 0..256>

,CROSS-REFERENCE = *UNCHANGED / *NO / [*YES](...)

*YES(...)

| PREPROCESSING-INFO = *UNCHANGED / *YES / *NO

| ,TYPES = *UNCHANGED / *YES / *NO

| ,VARIABLES = *UNCHANGED / *YES / *NO

| ,FUNCTIONS = *UNCHANGED / *YES / *NO

| ,LABELS = *UNCHANGED / *YES / *NO

| ,TEMPLATES = *UNCHANGED / *YES / *NO

| ,ORDER = *UNCHANGED / *STD / list-poss(6): *PREPROCESSING-INFO / *TYPES /

| *VARIABLES / *FUNCTIONS / *LABELS / *TEMPLATES

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

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

,SUMMARY = *UNCHANGED / *YES / *NO

,LAYOUT = *UNCHANGED / *FOR-NORMAL-PRINT(...) / *FOR-ROTATION-PRINT(...)

*FOR-NORMAL-PRINT(...)

| LINE-SIZE = *UNCHANGED / *STD / <integer 120..255>

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

OPTIONS = *UNCHANGED / *YES / *NO
*YES: Der Listengenerator erzeugt eine Liste aller voreingestellten und vom Benutzer angegebenen Compiler-Optionen.

SOURCE = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.

SOURCE = *NO
Es wird keine Quellprogramm-/Fehlerliste erzeugt.

SOURCE = *YES(...)
Es wird eine Quellprogramm-/Fehlerliste erzeugt.

MINIMAL-MSG-WEIGHT = *NOTE / *WARNING / *ERROR / *FATAL
Mit diesem Operanden lässt sich bestimmen, ab welchem Gewicht die Fehlermeldungen in der Quellprogrammliste stehen sollen.

Achtung:

Bei der Generierung der CIF-Informationen wurden nur die Meldungen gespeichert, die auch ausgegeben wurden. Dies folgte der Compiler-Anweisung MODIFY-DIAGNOSTIC-PROPERTIES während der Übersetzung. Mit dieser Suboption kann die Menge der ausgegebenen Meldungen gegenüber den CIF-Informationen nur eingeschränkt werden, z. B. von WARNING auf ERROR.
Falls während der Übersetzung bei MODIFY-DIAGNOSTIC-PROPERTIES für MINIMAL-MSG-WEIGHT = *WARNING (Defaulteinstellung) angegeben wurde, kann mit MODIFY-LISTING-PROPERTIES nicht erreicht werden, dass Notes ausgegeben werden.

Beispiele siehe Compiler-Anweisung MODIFY-DIAGNOSTIC-PROPERTIES.

> **i** Nach Erreichen von MAX-ERROR-NUMBER wird keine Quellprogramm-Information in der Quellprogramm-/Fehlerliste ausgegeben. In einem solchen Fall kann über dieses Listing kein Bezug zu Fehlerstellen mehr festgestellt werden.

PREPROCESSING-RESULT = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.

PREPROCESSING-RESULT = *NO
Der Listengenerator erzeugt keine Präprozessorliste.

PREPROCESSING-RESULT = *YES
Der Compiler erzeugt eine Präprozessorliste.

DATA-ALLOCATION-MAP = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.

DATA-ALLOCATION-MAP = *NO
Der Listengenerator erzeugt keine Adressliste.

DATA-ALLOCATION-MAP = *YES(...)
Der Listengenerator erzeugt eine Adressliste.

STRUCTURE-LEVEL = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.

STRUCTURE-LEVEL = *NONE
Strukturelemente werden in der Adressliste nicht abgebildet.

STRUCTURE-LEVEL = *MAX
Es werden Strukturelemente bis zu maximalen Schachtelungstiefe (256) in der Adressliste abgebildet.

STRUCTURE-LEVEL = <integer 0..256>
Es werden Strukturelemente bis zu der mit <integer> angegebenen Schachtelungstiefe in der Adressliste abgebildet. Bei Angabe der Schachtelungstiefe 0 werden keine Strukturelemente ausgegeben (entspricht STRUCTURE-LEVEL=*NONE).

CROSS-REFERENCE = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.

CROSS-REFERENCE = *NO
Der Listengenerator erzeugt keine Querverweisliste.

CROSS-REFERENCE = *YES(...)
Der Listengenerator erzeugt eine globale, d.h. modulübergreifende Querverweisliste. Die Querverweisliste enthält in jedem Fall ein FILETABLE-Teil mit den Namen aller Dateien, Bibliotheken und Elemente, die der Compiler als Quellen verwendet.

PREPROCESSING-INFO = *UNCHANGED / *YES / *NO
Die Querverweisliste enthält wahlweise eine Liste der vom Präprozessor bearbeiteten Namen.

TYPES = *UNCHANGED / *YES / *NO
Die Querverweisliste enthält wahlweise eine Liste der benutzerdefinierten Typen (typedefs, Struktur-, Union-, Klassen- und Aufzählungstypen).

VARIABLES = *UNCHANGED / *YES / *NO
Die Querverweisliste enthält - mit *NO unterdrückbar - eine Liste der Variablen.

FUNCTIONS = *UNCHANGED / *YES / *NO
Die Querverweisliste enthält - mit *NO unterdrückbar - eine Liste der Funktionen.

LABELS = *UNCHANGED / *YES / *NO
Die Querverweisliste enthält - mit *NO unterdrückbar - eine Liste der Labels.

TEMPLATES = *UNCHANGED / *YES / *NO
Die Querverweisliste enthält wahlweise eine Liste der Templates.

ORDER = *UNCHANGED / *STD / list-poss(6): *PREPROCESSING-INFO / *TYPES /*VARIABLES / *FUNCTIONS / *LABELS / *TEMPLATES
Mit diesem Operanden kann die Reihenfolge festgelegt werden, in der die einzelnen Teile in der Querverweisliste aufgeführt werden.
*STD: Voreingestellt ist die oben nach list-poss angegebene Reihenfolge.

PROJECT-INFORMATION = *UNCHANGED / *YES / *NO
*YES: Der Listengenerator erzeugt eine globale, d.h. modulübergreifende Projektliste. Sie enthält eine Gegenüberstellung aller im Quellprogramm original verwendeten externen Namen und den Namen, die der Compiler für den Binder intern generiert.

ASSEMBLER-CODE = *UNCHANGED / *YES / *NO
*YES: Der Listengenerator erzeugt eine Objektcodeliste.

SUMMARY = *UNCHANGED / *YES / *NO
*YES: Der Listengenerator erzeugt eine Liste mit statistischen Angaben über den Compilerlauf.

LAYOUT =

Mit dieser Option kann die Seitenbreite (Anzahl Zeichen pro Zeile) und die Seitenhöhe (Anzahl Zeilen pro Seite) für die Compilerlisten bestimmt werden.

Bei Auswahl einer Zeilenbreite von 120 Zeichen erhalten alle Listen einen schmaleren Listenkopf und -fuß. Die Textzeilen werden nur bei den tabellarischen Listen (Optionen-, Querverweis- und Adressliste) entsprechend umbrochen. Überlange Textzeilen in der Quellprogramm-, Präprozessor- und Objectcode-Liste werden beim Ausdruck abgeschnitten.

Bei Angabe einer BS2000-Ausgabedatei ist in jeder Zeile die erste Spalte für die Vorschubsteuerung reserviert.
Bei Ausgabe in eine POSIX-Datei werden die für POSIX passenden Kontrollzeichen für Zeilen- bzw. Seitenvorschub generiert. Dadurch wird die Zeilenlänge in der POSIX-Ausgabedatei bis zu 3 Zeichen größer als die gewählte Angabe für die Zeilenbreite.

LAYOUT = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.

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

Listen im Querformat.

LINE-SIZE = *UNCHANGED

Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.
Dies gilt auch dann, wenn diese Angabe bei *FOR-ROTATION-PRINT gemacht wurde.

LINE-SIZE = *STD
Es werden 132 Zeichen pro Zeile ausgegeben.

LINE-SIZE = <integer 120..255>
Es werden 120 bis 255 Zeichen pro Zeile ausgegeben.

LINES-PER-PAGE = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.
Dies gilt auch dann, wenn diese Angabe bei *FOR-ROTATION-PRINT gemacht wurde.

LINES-PER-PAGE = *STD
Es werden 64 Zeilen pro Seite ausgegeben.

LINES-PER-PAGE = <integer 11..255>
Pro Seite werden 11 bis 255 Zeilen ausgegeben.
Als Untergrenze sind 11 Zeilen festgelegt, damit pro Seite mindestens der Listenkopf und -fuß sowie eine Textzeile ausgegeben werden kann.

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

Listen im Hochformat.

Um solche Listen auszudrucken, muss im PRINT-DOCUMENT-Kommando der ROTATION-Parameter angegeben werden.

LINE-SIZE = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.
Dies gilt auch dann, wenn diese Angabe bei *FOR-NORMAL-PRINT gemacht wurde.

LINE-SIZE = *STD
Es werden 120 Zeichen pro Zeile ausgegeben.

LINE-SIZE = <integer 120..255>
Es werden 120 bis 255 Zeichen pro Zeile ausgegeben.

LINES-PER-PAGE = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.
Dies gilt auch dann, wenn diese Angabe bei *FOR-NORMAL-PRINT gemacht wurde.

LINES-PER-PAGE = *STD
Es werden 84 Zeilen pro Seite ausgegeben.

LINES-PER-PAGE = <integer 11..255>
Pro Seite werden 11 bis 255 Zeilen ausgegeben.
Als Untergrenze sind 11 Zeilen festgelegt, damit pro Seite mindestens der Listenkopf und -fuß sowie eine Textzeile ausgegeben werden kann.

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

Mit dieser Option lässt sich steuern, ob und welche Include-Dateien in der Quellprogramm-, Präprozessor- und Querverweisliste abgebildet werden. Standardmäßig werden die benutzereigenen Include-Dateien und nicht die Standard-Include-Dateien abgebildet. Bitte beachten Sie, dass Sie mit dieser Option die bei der Übersetzung erzeugte Information über Include-Dateien, die explizit oder implizit durch den gleichnamigen Operanden der MODIFY-CIF-PROPERTIES-Anweisung bestimmt ist, nur einschränken können.

LISTING-PRAGMAS =
Mit dieser Option lässt sich steuern, ob und welche im Quelltext vorhandenen `#pragma`-Anweisungen zur Gestaltung der Quellprogramm- und Präprozessorliste berücksichtigt werden sollen.
Die Beschreibung der `#pragma`-Anweisungen finden Sie im Abschnitt „Pragmas zum Steuern des Listenbildes“ .

LISTING-PRAGMAS = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.

LISTING-PRAGMAS = *INTERPRETED / *IGNORED
Es werden alle `#pragma`-Anweisungen berücksichtigt (*INTERPRETED) bzw. ignoriert (*IGNORED).

LISTING-PRAGMAS = *SELECT(...)
Eine oder mehrere der folgenden `#pragma`-Anweisungen zur Listensteuerung werden berücksichtigt (*YES) oder ignoriert (*NO).

PAGE = *UNCHANGED / *YES / *NO
Anweisung `#pragma PAGE` [*text*]:
Seitenvorschub und wahlweise Zeile im Listenkopf

TITLE = *UNCHANGED / *YES / *NO
Anweisung `#pragma TITLE` *text*:
Zeile im Listenkopf

SPACE = *UNCHANGED / *YES / *NO
Anweisung `#pragma SPACE [n]`:
Einfügen von Leerzeilen

LIST = *UNCHANGED / *YES / *NO
Anweisung `#pragma LIST[ING] ON` oder `#pragma LIST[ING] OFF`:
Unterdrücken der Ausgabe von Quelltextzeilen

INITIAL-TITLE-TEXT = *UNCHANGED / *NONE / <c-string 1..256>
Mit dieser Option kann angegeben werden, ob im Listenkopf eine zusätzliche Zeile erscheinen und welcher Text dort stehen soll. Die INITIAL-TITLE-TEXT-Option bezieht sich, im Unterschied zu den Pragmas (nur Quellprogramm- und Präprozessorliste), auf alle Compilerlisten. Wenn der Text länger als die mit der LINE-SIZE-Option (siehe Compiler-Anweisung "MODIFY-LISTING-PROPERTIES") definierte Zeilenlänge ist, wird er in mehrere Zeilen entsprechender Länge aufgeteilt.
Bei der Quellprogramm- und Präprozessorliste haben ggf. vorhandene TITLE- und PAGE-Pragmas Vorrang vor der INITIAL-TITLE-TEXT-Angabe.

OUTPUT = *UNCHANGED
Es gilt die Angabe der letzten MODIFY-LISTING-PROPERTIES-Anweisung.

OUTPUT = *SYSLST
Die Listen werden standardmäßig in die temporäre Systemdatei SYSLST geschrieben, von wo aus sie nach Ende der Task (LOGOFF) auf den Drucker ausgegeben werden.

OUTPUT = *SYSOUT
Die Listen werden auf die Systemdatei SYSOUT geschrieben, die im Dialogbetrieb der Datensichtstation zugeordnet ist.

OUTPUT = <filename 1..54>
Die Listen werden in eine katalogisierte BS2000-Datei mit dem angegebenen Namen geschrieben.

OUTPUT = <posix-pathname>
Die Listen werden in eine POSIX-Datei geschrieben.
Als <posix-pathname> ist nur ein Dateiname zugelassen. Zur Beschreibung des Begriffs <posix-pathname> siehe "Compiler-Ein-/Ausgaben im POSIX-Dateisystem".

OUTPUT = *LIBRARY-ELEMENT(...)
Es wird angegeben, in welcher PLAM-Bibliothek (LIBRARY=) und unter welchem Elementnamen (ELEMENT=) die Listen abgelegt werden sollen. Das Element wird unter dem Typ P abgespeichert.

LIBRARY = *STD-LIBRARY
Die Listen werden standardmäßig in die Bibliothek SYS.PROG.LIB geschrieben.

LIBRARY = <filename 1..54>
Die Listen werden in die PLAM-Bibliothek mit dem angegebenen Namen geschrieben.

LIBRARY = *LINK(...)

LINK-NAME = <filename 1..8>
Statt des Bibliotheksnamens kann auch ein Linkname angegeben werden.<filename> ist der Linkname der zugewiesenen Bibliothek. Er muss vor Aufruf des Listengenerators mit dem ADD-FILE-LINK-Kommando der PLAM-Bibliothek zugewiesen worden sein.

ELEMENT = <composed-name 1..64 with-under>(...)
Die Listen werden in ein Bibliothekselement (Typ P) mit dem angegebenen Namen geschrieben.

VERSION = *UPPER-LIMIT
Enthält die Elementangabe keine Versionsbezeichnung, verwendet der Listengenerator die höchstmögliche Version.

VERSION = *INCREMENT
Das Element erhält die gegenüber der höchsten vorhandenen Version um 1 inkrementierte Versionsnummer, vorausgesetzt, die höchste vorhandene Versionsbezeichnung endet mit einer inkrementierbaren Zahl. Ist die Versionsbezeichnung nicht inkrementierbar, wird die Listenerzeugung mit Fehler abgebrochen.
Beispiel siehe COMPILE-Anweisung ("COMPILE").
VERSION = <composed-name 1..24 with-under>
Das Element erhält die angegebene Versionsbezeichnung.

4 Binden und Programmablauf

In diesem Kapitel werden folgende Themen behandelt:

4.1 Binden

Als Ergebnis der Übersetzung eines Quellprogramms liefert der C/C++-Compiler ein Bindelademodul (LLM). Dieses Modul besteht bereits aus Maschinencode. Damit dieser Code jedoch im Rechner ablaufen kann, müssen die erzeugten Module mit anderen Modulen zu einer ablauffähigen Einheit zusammengefügt (= gebunden) werden. Die zusätzlich benötigten Module sind z.B. Module des Laufzeitsystems.
Ferner können weitere Module (z.B. von Unterprogrammen in anderen Sprachen oder von getrennt übersetzten C /C++-Programmteilen) eingebunden werden. Diese weiteren Module können dabei zu unterschiedlichen Zeiten und von verschiedenen Compilern übersetzt worden sein.

Die wichtigste Funktion des **Binders** besteht darin, die für die ladefähige Einheit erforderlichen Module aus den verschiedenen Quellen (Dateien, Bibliotheken) abzurufen und miteinander zu verknüpfen. Die Verknüpfung besteht darin, dass der Binder jedes Modul um diejenigen Adressen ergänzt, die sich auf Bereiche außerhalb des Moduls beziehen (Externverweise).

Damit die beim Binden erzeugte Einheit ablaufen kann, muss ein **Lader** sie in den Speicher bringen, so dass der Rechner auf den Code zugreifen und ihn ausführen kann.

Für die Aufgaben des Bindens und Ladens stehen im **Binder-Lader-Starter-System** des BS2000 folgende Funktionseinheiten zur Verfügung:

- Dynamischer Bindelader DBL

  Der dynamische Bindelader DBL fügt in einem Arbeitsgang Module (Objektmodule, LLMs) zu einer temporär ladbaren Einheit zusammen, lädt diese sofort in den Speicher und startet sie.

- Binder BINDER

  Der BINDER bindet Module (Objektmodule, LLMs) zu einer logisch und physikalisch strukturierten ladbaren Einheit zusammen. Diese Einheit bezeichnet man als „Bindelademodul“ (Link and Load Module, LLM). Abgespeichert wird ein LLM vom BINDER als Element vom Typ L in einer PLAM-Bibliothek.

Es gibt zwei Möglichkeiten, Programme mit dem BINDER zu binden:

1. Direkt durch Aufruf des BINDER mit dem Kommando START-BINDER
2. Implizit mit den Compiler-Anweisungen BIND und MODIFY-BIND-PROPERTIES

Auf die Compiler-Anweisungen wird in diesem Kapitel nicht mehr näher eingegangen, da sie ausführlich im Kapitel Übersetzen in den Abschnitten "BIND" und "MODIFY-BIND-PROPERTIES" beschrieben sind.

Module, die in den C-Modi und im Cfront-C++-Modus des Compilers erzeugt werden, können dynamisch mit dem DBL, statisch mit dem BINDER sowie mit der BIND-Anweisung des Compilers (d.h. implizit mit dem BINDER) gebunden werden.

Das Binden von C++ V3 bzw. C++2017-Modulen muss mit der BIND-Anweisung des Compilers erfolgen. Siehe auch Abschnitt „Einschränkung beim Binden von C++-Programmen“.

Die zum Binden von C/C++-Programmen benötigten C- und C++-Laufzeitmodule sind Bestandteil des CRTE. Einen Überblick über alle C/C++-spezifischen CRTE-Bibliotheken finden Sie im Abschnitt „C/C++-spezifische Komponenten des CRTE“. Ausführlichere Informationen mit entsprechenden Hinweisen zum Binden speziell der C++-Laufzeitbibliotheken können Sie dem Kapitel „C++-Bibliotheken und C++-Laufzeitsystem“ entnehmen.

4.1.1 Dynamisch Binden und Laden mit dem DBL

Mit dem dynamischen Bindelader DBL können in einem Arbeitsgang Module temporär zu einer ladbaren Einheit gebunden, dann in den Speicher geladen und gestartet werden. Die erzeugte Ladeeinheit wird automatisch nach Programmablauf gelöscht.

Die Arbeitsweise des DBL ist im Handbuch „Bindelader-Starter“ [13] ausführlich beschrieben.

Da die mit dem C/C++-Compiler erzeugten Module ausschließlich im LLM-Format vorliegen, muss beim Binden und Laden mit dem DBL generell das Kommando START-EXECUTABLE-PROGRAM verwendet werden.

Mit diesem Kommando können Objektmodule und LLMs verarbeitet werden. Alternativ zu durchsuchende Bibliotheken (Laufzeitbibliotheken und ggf. weitere) werden mit dem Linknamen BLSLIBnn (00 <= nn <= 99) zugewiesen. Dies geschieht vor Aufruf des DBL mit dem ADD-FILE-LINK-Kommando, z.B.:

```
/ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME=PLAM.USER
/ADD-FILE-LINK LINK-NAME=BLSLIB02,FILE-NAME=$.SYSLNK.CRTE
```

Damit der DBL alternative Bibliotheken durchsucht, ist im RESOLUTION-Parameter des START-EXECUTABLE-PROGRAM-Kommandos folgende Angabe erforderlich:

```
..ALTERNATE-LIBRARIES=*BLSLIB##
```

Der Bindelauf mit dem DBL wird mit dem START-EXECUTABLE-PROGRAM- oder LOAD-EXECUTABLE-PROGRAM-Kommando gestartet.
Nach dem START-EXECUTABLE-PROGRAM-Kommando wird das Programm sofort ausgeführt. Nach dem LOAD-EXECUTABLE-PROGRAM-Kommando besteht die Möglichkeit, weitere Kommandos (z.B. Testhilfe-Kommandos) einzugeben.

```
 /{START-EXECUTABLE-PROGRAM / LOAD-EXECUTABLE-PROGRAM}
        FROM-FILE=*LIBRARY-ELEMENT(LIBRARY=bibliothek,ELEMENT-OR-SYMBOL=mainmod),
        DBL-PARAMETERS=*PARAMETERS(RESOLUTION=*PARAMETERS(ALTERNATE-
LIBRARIES=*BLSLIB##))
```

LIBRARY=*bibliothek*, ELEMENT-OR-SYMBOL=*mainmod*

> Der DBL greift auf die angegebene PLAM-Bibliothek zu. Als Elementname ist der Name des Moduls anzugeben, das die `main`-Funktion enthält.

RESOLUTION=*PARAMETERS (ALTERNATE-LIBRARIES=*BLSLIB##)

> Diese Angabe ist immer erforderlich, wenn LLMs dynamisch gebunden werden sollen.

Lade- und Start-Funktion des DBL

Ein mit dem BINDER fertig gebundenes LLM (d.h. alle Externverweise sind befriedigt), wird mit dem DBL ohne Zuweisung alternativer Bibliotheken geladen und gestartet:

START-EXECUTABLE-PROGRAM FROM-FILE=*LIBRARY-ELEMENT(*LIBRARY= *bibliothek* ,

ELEMENT-OR-SYMBOL= *modul*)

Empfohlene BLSLIBnn-Reihenfolge für das Zuweisen der CRTE-Bibliotheken

1. C-Programme

```
/ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME= benutzerbibliothek
/ADD-FILE-LINK LINK-NAME=BLSLIB02,FILE-NAME=$.SYSLNK.CRTE
```

2. Cfront-C++-Programme

```
/ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME= benutzerbibliothek
/ADD-FILE-LINK LINK-NAME=BLSLIB02,FILE-NAME=$.SYSLNK.CRTE.CFCPP
/ADD-FILE-LINK LINK-NAME=BLSLIB03,FILE-NAME=$.SYSLNK.CRTE.CPP
/ADD-FILE-LINK LINK-NAME=BLSLIB04,FILE-NAME=$.SYSLNK.CRTE
```

3. C++ V3-Programme (mit Nutzung der Bibliothek Tools.h++)

```
/ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME= benutzerbibliothek
/ADD-FILE-LINK LINK-NAME=BLSLIB02,FILE-NAME=$.SYSLNK.CRTE.TOOLS
/ADD-FILE-LINK LINK-NAME=BLSLIB03,FILE-NAME=$.SYSLNK.CRTE.STDCPP
/ADD-FILE-LINK LINK-NAME=BLSLIB04,FILE-NAME=$.SYSLNK.CRTE.RTSCPP
/ADD-FILE-LINK LINK-NAME=BLSLIB05,FILE-NAME=$.SYSLNK.CRTE
```

4. C++ 2017-Programme

```
/ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME= benutzerbibliothek
/ADD-FILE-LINK LINK-NAME=BLSLIB02,FILE-NAME=$.SYSLNK.CRTE.CXX01
/ADD-FILE-LINK LINK-NAME=BLSLIB03,FILE-NAME=$.SYSLNK.CRTE
```

Achtung!

Beim Binden und Laden von C++ V3- bzw. C++ 2017-Programmen mit dem DBL findet keine Instanziierung von Templates statt. Alle Instanzen müssen falls erforderlich mittels der Anweisung BIND ACTION=*PRELINK vorinstanziiert werden.

4.1.2 Binden mit dem BINDER

Mit dem BINDER können Objektmodule und LLMs zu einem LLM gebunden und als Element vom Typ L in einer PLAM-Bibliothek abgespeichert werden. Der BINDER ist ausführlich im Handbuch „BINDER“ [14] beschrieben.

Module, die im Modus C++ V3 bzw. C++ 2017 erzeugt wurden, können nicht durch direkten Aufruf des BINDER, sondern nur mittels der Compiler-Anweisung BIND gebunden werden (siehe auch Abschnitt „Einschränkung beim Binden von C++-Programmen“).

Steueranweisungen für den BINDER (Auswahl)

```
/START-BINDER ──────────────────────────────────────────────────────────  (1)

//START-LLM-CREATION INT-NAME= name ──────────────────────────────────────  (2)

//INCLUDE-MODULES -

//    MOD-CONTAINER=*LIB(LIB= bibliothek ,ELEM={ mainmod / *ALL } ──────  (3)

[//INCLUDE-MODULES MOD-CONTAINER=*LIB(LIB=..., ELEM=...)] ──────────────  (4)

[//INCLUDE-MODULES -

//    MOD-CONTAINER=*LIB(LIB=$.SYSLNK.CRTE.POSIX,ELEM=*ALL)] ──────────  (5)

[//RESOLVE-BY-AUTOLINK LIB=..., [SYMBOL-NAME= externverweis ]] ──────────  (6)

[//RESOLVE-BY-AUTOLINK LIB=

  {([$.SYSLNK.CRTE.CFCPP,$.SYSLNK.CRTE.CPP,]$.SYSLNK.CRTE) /

   ([$.SYSLNK.CRTE.CFCPP,$.SYSLNK.CRTE.CPP,]$.SYSLNK.CRTE.PARTIAL-BIND)]}(7)

[//MODIFY-SYMBOL-VISIBILITY ..., VISIBLE=*NO] ──────────────────────────  (8)

//SAVE-LLM MOD-CONTAINER=*LIB(LIB= bibliothek , ELEM= element ) ──────────  (9)

//END ──────────────────────────────────────────────────────────────────── (10)
```

(1) Der BINDER wird aufgerufen.

(2) Diese Anweisung erzeugt ein neues LLM im Arbeitsbereich mit dem internen Namen *name*. Das erzeugte LLM wird mit der Anweisung SAVE-LLM (siehe 9) als Element vom Typ L in einer PLAM-Bibliothek gespeichert.

(3) *mainmod* ist der Name des LLM, das die `main`-Funktion enthält. Mit *bibliothek* wird der Name der PLAM-Bibliothek angegeben, in der sich die Module befinden. Bei Angabe von *ALL werden alle Module aus der angegebenen Eingabequelle eingebunden.

(4) Mit weiteren INCLUDE-MODULE-Anweisungen können zusätzliche Module aus verschiedenen Bibliotheken eingebunden werden.

(5) Die Bibliothek SYSLNK.CRTE.POSIX muss immer eingebunden werden, wenn POSIX-Bibliotheksfunktionen benutzt werden. Da diese „Bindeschalter“-Bibliothek vorrangig vor dem C-Laufzeitsystem eingebunden werden muss, sollte zum Einbinden generell die INCLUDE-MODULES-Anweisung verwendet werden.

(6) Mit RESOLVE-BY-AUTOLINK-Anweisungen werden dem BINDER die Externverweise (= Modulnamen) und die entsprechenden Bibliotheken bzw. nur die Bibliotheken mitgeteilt, die mit dem Autolink-Verfahren nach bisher unbefriedigten Externverweisen durchsucht werden sollen. RESOLVE-BY-AUTOLINK-Anweisungen für benutzereigene Bibliotheken/Module sollten generell vor denjenigen für die Laufzeitbibliotheken (siehe 7) angegeben werden.

(7) Die jeweils einzubindenden CRTE-Laufzeitbibliotheken werden in einer Liste angegeben.

Bei einem RESOLVE auf die Bibliothek SYSLNK.CRTE werden alle vom Programm benötigten Module des C-Laufzeitsystems fest eingebunden.

Bei einem RESOLVE auf die Bibliothek SYSLNK.CRTE.PARTIAL-BIND wird an Stelle des C-Laufzeitsystems ein Verbindungsmodul eingebunden, das alle Externverweise auf das C-Laufzeitsystem befriedigt. Das C-Laufzeitsystem selbst wird zum Ablaufzeitpunkt dynamisch nachgeladen. Das fertig gebundene Modul benötigt deutlich weniger Plattenspeicherplatz als beim statischen Einbinden der C-Laufzeitmodule aus der Bibliothek SYSLNK.CRTE. Außerdem wird die Ladezeit beschleunigt.

Wenn keine RESOLVE-BY-AUTOLINK-Anweisung angegeben wird, bleiben die Externverweise auf die Laufzeitsysteme offen. Die Laufzeitmodule werden dann zur Ablaufzeit dynamisch gebunden und geladen (siehe Abschnitt „Dynamisch Binden und Laden mit dem DBL“).

(8) Mit der MODIFY-SYMBOL-VISIBILITY-Anweisung können externe Symbole für weitere Binderläufe maskiert werden. Standardmäßig bleiben die Symbole sichtbar. Siehe auch „Maskierung von Symbolen“.

(9) Diese Anweisung speichert das aktuelle LLM, das mit START-LLM-CREATION erzeugt wurde, als Element vom Typ L in eine PLAM-Bibliothek.

(10) Mit der END-Anweisung wird der BINDER-Lauf beendet.

Bei den Anweisungen INCLUDE-MODULES und RESOLVE-BY-AUTOLINK kann an Stelle des Bibliotheksnamens (LIB=*bibliothek*) auch LIB=*BLSLINK angegeben werden. In diesem Fall müssen die zu durchsuchenden Bibliotheken mit dem Linknamen BLSLIB*nn* (00 <= *nn* <= 99) zugewiesen werden. Dies geschieht vor Aufruf des BINDERS mit dem ADD-FILE-LINK-Kommando, z.B.:

```
/ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME=PLAM.USER1
/ADD-FILE-LINK LINK-NAME=BLSLIB02,FILE-NAME=PLAM.USER2
```

Ein mit dem BINDER erzeugtes LLM kann - sofern alle Externverweise befriedigt sind - mit dem DBL ohne Zuweisung alternativer Bibliotheken geladen und gestartet werden:

`START-EXECUTABLE-PROGRAM *LIBRARY-ELEMENT(LIB=` *bibliothek* `,ELEM=` *modul* `)`

Maskierung von Symbolen

Beim Binden mit dem BINDER werden Symbole (CSECTs, ENTRYs) standardmäßig nicht maskiert und bleiben für spätere Bindeläufe mit dem BINDER oder DBL sichtbar.

Beim dynamischen Binden mit dem DBL hat dies u.a. folgende Auswirkungen:
Wenn in einer PLAM-Bibliothek sowohl vom Compiler generierte Einzelmodule als auch LLMs mit eingebundenem Laufzeitsystem stehen, werden beim dynamischen Binden der Einzelmodule die Externverweise auf das Laufzeitsystem aus irgendeinem vorgebundenen Modul befriedigt und nicht aus der Laufzeitbibliothek. In diesem Fall erhält man diverse „DUPLICATES“-Warnungen vom DBL. Auf Grund des Autolink-Mechanismus wird zuerst die Bibliothek durchsucht, in der das Einzelmodul steht und erst anschließend die mit den Linknamen BLSLIBnn zugewiesenen Laufzeitbibliotheken.

Um sicherzustellen, dass beim Binden die Externverweise immer aus der aktuellen Laufzeitbibliothek und nicht aus einem beliebigen Modul befriedigt werden, müssen

- entweder Einzelmodule und vorgebundene Module in unterschiedlichen Bibliotheken gehalten werden
- oder beim Binden mit dem BINDER die Symbole mit der Anweisung MODIFY-SYMBOL-VISIBILITY maskiert werden.

4.1.3 Gemeinsam benutzbare C/C++-Programme

Bei großen Programmen kann es von Vorteil sein, einzelne Programmteile, auf die mehrere Benutzer (Tasks) gleichzeitig zugreifen, gemeinsam benutzbar (shareable) zu machen.

Um gemeinsam benutzbare Programme zu erzeugen, muss bei der Übersetzung folgende Compileroption angegeben werden:

```
//MODIFY-MODULE-PROP SHAREABLE-CODE=*YES
```

Der Compiler erzeugt dann pro Übersetzungseinheit ein LLM mit einer nicht gemeinsam benutzbaren Daten-CSECT und einer gemeinsam benutzbaren Code-CSECT. Die Code-CSECT ist mit dem Attribut PUBLIC gekennzeichnet.

Bei einem anschließenden Binderlauf wird aus der Code-CSECT eine PUBLIC-Slice gebildet und aus der Daten-CSECT eine PRIVATE-Slice.

Die PUBLIC-Slice kann der Systemverwalter mit dem ADD-SHARED-Kommando als gemeinsam benutzbar erklären. Beim Aufruf mit dem START-EXECUTABLE-PROGRAM-Kommando wird dann nur die PRIVATE-Slice geladen.

Bespiel

```
/START-CPLUS-COMPILER
//MOD-SOURCE-PROP LANG=*C
//MOD-MODULE-PROP SHAREABLE-CODE=*YES
//COMPILE SOURCE=MODUL1.C,MODULE-OUTPUT=*LIB-ELEM(LIB=A.TEST)
//COMPILE SOURCE=MODUL2.C,MODULE-OUTPUT=*LIB-ELEM(LIB=A.TEST)
//END
/START-BINDER
//START-LLM-CREATION INT-NAME=TEST,
//              SLICE-DEFINITION=BY-ATTRIBUTE(PUBLIC=*YES)
//INCLUDE-MODULES LIB=A.TEST,ELEM=(MODUL1)
//INCLUDE-MODULES LIB=A.TEST,ELEM=(MODUL2)
//SAVE-LLM LIB=B.TEST,ELEM=TEST
//END
/ADD-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME=$.SYSLNK.CRTE
/START-EXECUTABLE-PROGRAM FROM-FILE=*LIBRARY-ELEMENT(-
/                           LIB=B.TEST,ELEM=TEST),-
/             DBL-PAR=*PAR(RESOLUTION=*PAR(ALTERNATE-LIBRARIES=*BLSLIB##))
```

4.1.4 Einschränkung beim Binden von C++-Programmen

Es besteht ausschließlich bei Verwendung der BIND-Anweisung des Compilers die Gewähr, dass ein C++ V3 bzw. C++ 2017 Programm korrekt gebunden wird. U.a. wird mit der BIND-Anweisung die automatische Template-Instanziierung durch den Prälinker durchgeführt (siehe auch "Template-Instanziierung").

4.2 Programmablauf

In diesem Abschnitt werden folgende Themen behandelt:

- Parametereingaben nach dem Programmstart
- Umlenken der Standard-Ein-/Ausgabedateien
- Eingabe der Parameter für die main-Funktion
- Definition der main-Funktion mit Parametern
- Dialogtesthilfe AID
- Voraussetzungen für das symbolische Testen

4.2.1 Parametereingaben nach dem Programmstart

Nach dem Aufruf mit dem START-EXECUTABLE-PROGRAM-Kommando wird ein C/C++-Programm standardmäßig sofort ausgeführt.
Mit der PARAMETER-PROMPTING-Option der MODIFY-RUNTIME-PROPERTIES-Anweisung (siehe "MODIFY-RUNTIME-PROPERTIES") lässt sich steuern, ob vor der Programmausführung die Standard-Ein-/Ausgabedateien umgelenkt und der `main`-Funktion Parameter übergeben werden können.

PARAMETER-PROMPTING = *NO

Standardmäßig wird das Programm nach dem Start mit dem START-EXECUTABLE-PROGRAM-Kommando ohne vorhergehenden Dialogschritt für Parametereingaben sofort ausgeführt. An die `main`-Funktion wird lediglich der Name des Programms übergeben.

PARAMETER-PROMPTING = *YES

Nach dem Start mit dem START-EXECUTABLE-PROGRAM-Kommando meldet sich das Programm mit

```
CCM0001 enter options :
*
```

Mit dem START-EXECUTABLE-PROGRAM- oder SRX-Kommando lassen sich die Parameter auch gleich mit übergeben:
`/START-EXECUTABLE-PROGRAM..,PROGRAM-PARAMETERS='...'`

Ähnlich wie in der Kommandozeile im UNIX-Betriebssystem lassen sich nun in einer oder mehreren Parameterzeilen

- die C-Standard-Ein-/Ausgabedateien `stdin`, `stdout` und `stderr` sowie die C++-Standard-Ein-/Ausgabdateien `cin`, `cout`, `cerr` und `clog` umlenken (siehe "Umlenken der Standard-Ein-/Ausgabedateien") und
- Parameter an die `main`-Funktion übergeben (siehe "Eingabe der Parameter für die main-Funktion").

  Damit diese Parameter im Programm angesprochen werden können, muss die Funktion `main` zwei formale Parameter enthalten, die üblicherweise *argc* (argument count) und *argv* (argument vector) heißen (siehe "Definition der main-Funktion mit Parametern")

Die einzelnen Eingaben (Umlenkangaben und main-Parameter) sind in der Parameterzeile durch Leerzeichen zu trennen.

Wenn eine Parameterzeile für die Eingabe nicht ausreicht, kann man sie mit einem Gegenschrägstrich (\) beenden und den nächsten Parameter am Beginn der nächsten Zeile eingeben. Der Gegenschrägstrich wird dabei (wie das Leerzeichen) als Trennzeichen zwischen zwei Parametern interpretiert (weitere Bedeutungen von \ siehe "Eingabe der Parameter für die main-Funktion").

Beispiel

```
*par1 par2 par3\
*par4
```

4.2.2 Umlenken der Standard-Ein-/Ausgabedateien

Bei Beginn der Programmausführung sind die Standard-Ein-/Ausgabedateien folgenden BS2000-Systemdateien zugeordnet:

stdin/cin	SYSDTA
stdout/cout	SYSOUT
stderr/cerr/clog	SYSOUT

In der Parameter-Zeile können diese Voreinstellungen geändert werden. Für die Platzhalter *eingabe* und *ausgabe* sind aktuelle Werte einzusetzen.

<*eingabe*

Die Standardeingabe (`stdin`, `cin`) soll von *eingabe* gelesen werden.

>*ausgabe*

Die Standardausgabe (`stdout`, `cout`) soll nach *ausgabe* geschrieben werden.
Ist die Datei bereits vorhanden, wird sie überschrieben. Ist sie nicht vorhanden, wird sie neu erstellt.

>>*ausgabe*

Die Standardausgabe (`stdout`, `cout`) soll an *ausgabe* angehängt werden.

Ist die Datei noch nicht vorhanden, wird sie neu erstellt.

2>*ausgabe*

Die Standard-Fehlerausgabe (`stderr`, `cerr`, `clog`) soll nach *ausgabe* geschrieben werden. Ist die Datei bereits vorhanden, wird sie überschrieben. Ist sie nicht vorhanden, wird sie neu erstellt.

2>>*ausgabe*

Die Standard-Fehlerausgabe (`stderr`, `cerr`, `clog`) soll an *ausgabe* angehängt werden. Ist die Datei noch nicht vorhanden, wird sie neu erstellt.

Für *eingabe* bzw. *ausgabe* lassen sich folgende aktuelle Werte einsetzen:

(SYSDTA)
bezeichnet die Systemdatei SYSDTA. Diese Angabe kann nur für *eingabe* stehen.

(SYSOUT)
(SYSLST)
bezeichnen die Systemdateien SYSOUT und SYSLST. Diese Angaben können nur für *ausgabe* stehen.

dateiname
bezeichnet den Namen einer katalogisierten BS2000-Datei. Diese Angabe kann sowohl für *eingabe* als auch für *ausgabe* stehen.

*POSIX(*dateiname*)
dateiname bezeichnet den Namen einer POSIX-Datei. Diese Angabe kann sowohl für *eingabe* als auch für *ausgabe* stehen. Diese Umlenkung ist nur möglich, wenn der POSIX-Bindeschalter eingebunden wird (siehe Punkt 3. im Abschnitt "Benutzen der POSIX-Bibliotheksfunktionen").

Eingabedateien müssen bereits existieren.

Ist eine Ausgabedatei noch nicht vorhanden, wird sie neu erstellt. Ist sie bereits vorhanden, wird sie entweder überschrieben (> bzw. 2>) oder um die neue Ausgabe erweitert (>> bzw. 2>>).

Hinweis

Die Umlenkung der Standard-Ein-/Ausgabe-Dateien betrifft alle Ein-/Ausgabe-Funktionen, die standardmäßig von der Standardeingabe lesen bzw. auf die Standardausgabe schreiben sowie alle Funktionen, die die Dateizeiger `stdin`/`stdout` bzw. die Dateikennzahlen 0/1 als Argument benutzen. Von der Umlenkung nicht betroffen sind Ein/Ausgaben auf Dateien, die explizit mit den Namen "(SYSDTA)", "(SYSOUT)" oder "(SYSTERM)" eröffnet wurden.

4.2.3 Eingabe der Parameter für die main-Funktion

Eingabedaten in der Parameter-Zeile, die nicht der Umlenkung von Standard-Ein-/Ausgabedateien dienen (siehe "Umlenken der Standard-Ein-/Ausgabedateien"), werden als Parameter (d.h. aktuelle Argumente) an die `main`-Funktion übergeben. Im Programm können diese Parameter als mit dem Nullbyte (\0) abgeschlossene Zeichenketten verarbeitet werden.

Für die im Folgenden kursiv geschriebenen Platzhalter sind jeweils aktuelle Werte einzusetzen.

%*muster*

Alle Dateinamen, die dem angegebenen Muster entsprechen, werden als Parameter übergeben.

muster ist ein voll- oder teilqualifizierter Dateiname mit Wildcard-Syntax.

Außerdem können aus Kompatibilitätsgründen weitere Parameter angegeben werden, die die Auswahl der Dateien beeinflussen, z.B:

- Datei- und Katalogeigenschaften (FCBTYPE, SHARE etc.)
- Erstellungs- und Zugriffsdatum (CRDATE, EXDATE etc.)

Diese Parameter müssen in der Syntax des ISP-Kommandos FSTAT angegeben werden.

Z.B. liefert folgende Angabe die Namen aller Dateien mit Suffix `.C` und heutigem Erstellungsdatum.

```
%*.c,cr=t
```

'*zeichenkette*' oder "*zeichenkette*"

zeichenkette kann beliebige Zeichen inkl. Leerzeichen enthalten; sie werden unverändert ohne die begrenzenden Hochkommata (' bzw. ") dem Programm übergeben. Wenn das als Begrenzer verwendete Hochkomma bzw. Anführungszeichen in der Zeichenkette auftritt, muss ihm ein Gegenschrägstrich (\) vorangestellt werden, z.B.
'\'Zitat\'' oder "\"Zitat\"".

Das Zeilenende-Zeichen (`\n`) kann mitübergeben werden, wenn man innerhalb der Zeichenkette die Zeile mit dem Gegenschrägstrich beendet und ggf. weitere Zeichen sowie das abschließende Hochkomma in einer Fortsetzungszeile eingibt, z.B.
'1. Teil\
2. Teil'

sonstige-parameter

Das oder die Zeichen werden direkt an das Programm übergeben. Leerzeichen und der Gegenschrägstrich am Ende der Zeile gelten als Trennzeichen zwischen zwei Parametern.

Sollen Zeichen an das Programm übergeben werden, die eine Sonderbedeutung bei der Parameterübergabe haben (z.B. % oder >), muss ihnen ein Gegenschrägstrich vorangestellt werden. In diesem Fall wird der Gegenschrägstrich entfernt und das Zeichen selbst übergeben. Auf diese Weise lässt sich auch die Sonderbedeutung des Gegenschrägstrichs aufheben: \\ steht für einen Gegenschrägstrich ohne Sonderbedeutung.

Um Kleinbuchstaben über Prozedurparameter übergeben zu können, wird ein mit dem Gegenschrägstrich versehener Großbuchstabe in den entsprechenden Kleinbuchstaben umgewandelt.

Wirkung des Gegenschrägstrichs (Zusammenfassung):

buchstabe

Der Buchstabe wird als Kleinbuchstabe übergeben. Damit ist es möglich, selbst da Kleinbuchstaben zu übergeben, wo sie vom BS2000-Kommandointerpreter automatisch in Großbuchstaben übersetzt werden (z.B. in Prozedurparametern).

zeilenende

Gegenschrägstrich unmittelbar gefolgt vom Zeilenende.

Außerhalb von quotierten Zeichenketten wird der Gegenschrägstrich als Trennzeichen zwischen zwei Parametern interpretiert. Auf diese Weise können weitere Parametereingaben in Fortsetzungszeilen erfolgen.

Innerhalb von quotierten Zeichenketten wird die Sonderbedeutung des Zeilenendes aufgehoben und als Zeilenende-Zeichen (\n) mit an das Programm übergeben.

sonstige-zeichen

Der Gegenschrägstrich wird entfernt und das *sonstige-zeichen* übergeben. Auf diese Weise lässt sich die Sonderbedeutung von bestimmten Zeichen aufheben.

Beispiele

```
Eingabe:              Übergebene Parameter:
\AB\C                 "aBc"
%bsp.                 Alle Dateinamen mit Präfix BSP.
'string mit \         "string mit \nnewline"
newline'
```

4.2.4 Definition der main-Funktion mit Parametern

Um die in der Parameter-Zeile eingegebenen Daten (siehe "Eingabe der Parameter für die main-Funktion") im Programm ansprechen zu können, sind für die Funktion main zwei formale Parameter vorzusehen:

```
int main(int argc, char *argv[])
```

Die Namen der Parameter (*argc*, *argv*) können beliebig gewählt werden, sind aber im UNIX-Betriebssystem so gebräuchlich.

Der erste Parameter *argc* zeigt die Anzahl der übergebenen Parameter (inkl. Programmname) an.

Der zweite Parameter *argv* ist ein Zeiger auf einen Vektor von char-Zeigern (Zeichenketten). In ihm werden der Programmname (in argv[0]) und alle eingegebenen Parameter als mit dem Nullbyte (\0) abgeschlossene Zeichenketten abgespeichert.

Beispiel

Folgendes Beispiel gibt den Programmnamen und alle eingegebenen Parameter aus.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
 int i;
 printf("Programmname: %s\n", argv[0]);
 for (i=1; i<argc; ++i)
   printf("%d.ter Parameter ist: %s\n", i, argv[i]);
 return 0;
}
```

4.2.5 Dialogtesthilfe AID

C- und C++-Programme können mit der Dialogtesthilfe AID getestet werden.

In diesem Benutzerhandbuch soll AID lediglich kurz vorgestellt werden. Die ausführliche Beschreibung dieser Testhilfe finden Sie im Handbuch „AID, Testen von C/C++-Programmen“ [9].

AID zeichnet sich durch folgende Leistungsmerkmale aus:

- Es bietet die Möglichkeit, „symbolisch“ zu testen, d.h. in den Kommandos an Stelle sedezimaler Adressen auch symbolische Namen aus dem Quellprogramm anzugeben, vorausgesetzt, beim Übersetzen werden LSD-Informationen erzeugt und an das geladene Programm weitergegeben.

  Dabei ist es nicht unbedingt erforderlich, diese Informationen stets für das Gesamtprogramm zusammen mit diesem Programm zu laden. AID erlaubt ein Nachladen der LSD-Informationen für jede Übersetzungseinheit, falls die zugehörigen Module mit den LSD-Informationen in einer PLAM-Bibliothek stehen. Dadurch lassen sich Betriebsmittel wirtschaftlicher einsetzen:

  - Der Programmspeicher wird entlastet, da LSD-Informationen nur dann geladen werden müssen, wenn sie zum Testen benötigt werden.
  - Ein Programm, das im Test fehlerfrei bleibt, muss für den Produktiveinsatz nicht unbedingt neu (ohne LSD-Informationen) übersetzt oder gebunden werden.
  - Falls sich für ein Programm während seines Produktiveinsatzes ein Test als nötig erweist, stehen dafür LSD-Informationen zur Verfügung, ohne dass das Programm erneut übersetzt und gebunden werden muss.
- Es stellt Funktionen zur Verfügung, die es insbesondere gestatten,
  - den Programmablauf auf symbolischer Ebene zu verfolgen und zu protokollieren (TRACE-Funktion),
  - den Programmablauf an festgelegten Stellen oder beim Eintreten definierter Ereignisse zu unterbrechen, um AID- oder BS2000-Kommandos (so genannte Subkommandos) ausführen zu lassen,
  - sich die Inhalte von Variablen in einer Form ausgeben zu lassen, welche die Datendefinitionen des Quellprogrammes berücksichtigt,
  - die Inhalte von Variablen zu verändern,
  - Aufrufhierarchien auch ohne LSD-Informationen rückzuverfolgen (%SDUMP %NEST).
- Es unterstützt neben der Diagnose geladener Programme auch die Analyse von Speicherabzügen in Plattendateien.
- Es kann außer im Dialog- auch im Stapelbetrieb eingesetzt werden. Für einen Programmtest empfiehlt sich allerdings der Dialog, da die Folge der Kommandos nicht im voraus festgelegt werden muss, sondern der jeweiligen Testsituation angepasst werden kann.

In C können folgende Arten von Symbolen angesprochen werden:

- einfache (skalare) Typen
- Arrays und Arrayelemente
- Strukturen/Unionen und Struktur-/Union-Komponenten
- Aufzählungs-Konstanten (enum)
- Bitfelder
- Zeiger
- Funktionen

- Label

Nicht referenzierbar sind jedoch Präprozessor-Konstanten und -Makros (#DEFINEs), typedef-Namen, Aufzählungs-, Struktur- und Union-Typen (Etiketten), inline-generierte Funktionen.

In C++ können zusätzlich folgende Arten von Symbolen angesprochen werden:

- Funktionen und Datenelemente innerhalb von Klassen
- Überladene Funktionen und Operatoren
- Referenzen
- Templates
- Namespaces

Auf einzelne Anweisungszeilen wird über Source-Referenzen, auf den Beginn von Blöcken über Block-Referenzen Bezug genommen:

```
S'[<UNIQUE-Nr.> -] <Zeilen-Nr.> [: <rel.Anweisungs-Nr.>]' bzw.
```

```
BLK = '[<UNIQUE-Nr.> -] <Zeilen-Nr.> [: <rel.Block-Nr.>]'
```

Die UNIQUE- und Zeilennummer wird auch in der Quellprogramm-/Fehlerliste abgebildet. Dabei entspricht die UNIQUE-Nr. der FILE-NO des Listings.

4.2.6 Voraussetzungen für das symbolische Testen

Beim Testen auf symbolischer Ebene erlaubt es AID, Variablen mit den im Quellprogramm definierten Namen anzusprechen und sich auf einzelne Anweisungszeilen anhand von Source-Referenzen zu beziehen. Dafür müssen AID sog. LSD-Informationen zur Verfügung gestellt werden.

Die Erzeugung bzw. Weitergabe dieser Informationen wird bei jedem der folgenden Schritte durch Angabe entsprechender Operanden in den Steueranweisungen bzw. Kommandos gesteuert:

- Übersetzen mit dem C/C++-Compiler
- Binden und Laden mit dem dynamischen Bindelader
- Binden mit dem BINDER
- Binden mit der Compiler-Anweisung BIND

Übersetzen mit dem C/C++-Compiler

Das Erzeugen von LSD-Informationen wird mit folgender Option gesteuert:

```
//MODIFY-TEST-PROPERTIES TEST-SUPPORT = *NO / *YES
```

*NO Bei der Voreinstellung *NO erzeugt der Compiler keine LSD-informationen.
Auch ohne LSD-Informationen ist eine Rückverfolgung der Aufrufhierarchien möglich. Das heißt, bei Programmabbruch ist stets %SDUMP %NEST möglich.

*YES Der Compiler erzeugt LSD-Informationen.
Dies ist jedoch nur für nicht optimierte Programme möglich. Sollte die Optimierung dennoch eingeschaltet sein (siehe MODIFY-OPTIMIZATION-PROPERTIES-Anweisung), bewertet der Compiler den Wunsch nach Testhilfe höher, schaltet die Optimierungsstufe auf *LOW und gibt eine entsprechende Meldung aus.

Binden, Laden und Starten

Nachdem die LSD-Informationen bei der Übersetzung erzeugt worden sind, ist es möglich,

- sie zusammen mit dem Gesamtprogramm zu laden oder
- sie erst bei Bedarf für jede Übersetzungseinheit nachzuladen, falls die zugehörigen Module in einer PLAM-Bibliothek stehen.

Die folgende Tabelle gibt für beide Fälle einen Überblick über die Operanden, die zur Weitergabe der LSD-Informationen angegeben werden müssen:

Binde-/Lade-Art:	**LSD-Information mit dem Gesamtprogramm laden:**	**LSD-Information durch AID nachladen (%SYMLIB):**
Binden, Laden und Starten mit dem DBL	LOAD-EXECUTABLE-PROGRAM ..., TEST-OPTIONS=*AID oder START-EXECUTABLE-PROGRAM ..., TEST-OPTIONS=*AID	LOAD-EXECUTABLE-PROGRAM ..., [TEST-OPTIONS=*NONE] oder START-EXECUTABLE-PROGRAM ..., [TEST-OPTIONS=*NONE]

Binden mit dem BINDER	START-LLM-CREATION ..., INC-DEF=*PAR(..., TEST-SUPPORT=*YES) oder INCLUDE-MODULES ..., TEST-SUPPORT=*YES	START-LLM-CREATION ..., [INC-DEF=*PAR(..., TEST-SUPPORT=*NO)] oder INCLUDE-MODULES ..., [TEST-SUPPORT=*NO]
Binden mit der BIND-Anweisung	MODIFY-BIND-PROP ..., TEST-SUPPORT=*YES	MODIFY-BIND-PROP ..., [TEST-SUPPORT=*NO]
Laden und Starten mit dem DBL	LOAD-EXECUTABLE-PROGRAM ..., TEST-OPTIONS=*AID oder START-EXECUTABLE-PROGRAM ..., TEST-OPTIONS=*AID	LOAD-EXECUTABLE-PROGRAM ..., [TEST-OPTIONS=*NONE] oder START-EXECUTABLE-PROGRAM ..., [TEST-OPTIONS=*NONE]

Ein Programm, das getestet werden soll, wird mit dem LOAD-EXECUTABLE-PROGRAM-Kommando geladen, damit anschließend AID-Kommandos eingegeben werden können.Ein Programm, das nur bei Auftreten eines Fehlers mit AID bearbeitet werden soll, kann mit dem START-EXECUTABLE-PROGRAM-Kommando geladen und gestartet werden.

Hinweis zum Binden des C-Laufzeitsystems

> Wenn die Module aus der CRTE-Bibliothek SYSLNK.CRTE.PARTIAL-BIND eingebunden werden und das C-Laufzeitsystem selbst erst zum Ablaufzeitpunkt dynamisch nachgeladen wird (Voreinstellung beim Binden mit der BIND-Anweisung des Compilers), gibt es beim Testen mit AID folgende Einschränkung: Bei bestimmten Programmfehlern, z.B. fehlerhaften Parameterübergaben an C-Bibliotheksfunktionen, kann AID die Aufrufhierarchie nicht vollständig ausgeben, da u.U. die letzte Funktion vor Auftritt des Fehlers fehlt. Diese Einschränkung kann durch statisches Einbinden des C-Laufzeitsystems aufgehoben werden. Beim Binden mit der BIND-Anweisung des Compilers ist hierzu folgende Angabe erforderlich:
> MODIFY-BIND-PROPERTIES STDLIB=*STATIC.

Name der Übersetzungseinheit in der S-Qualifikation

Bei LLMs wird der sog. „Sourcemodulname“ angegeben, der vom C/C++-Compiler wie folgt aus dem Namen des Quellprogramms abgeleitet wird:

1. Ggf. vorhandene Namensteile <cat-id> und <user-id> werden nicht übernommen.
2. Ist der Datei- bzw. Elementname des Quellprogramms länger als 32 Zeichen, wird er auf 32 Zeichen von rechts verkürzt.

Falls der Sourcemodulname Punkte enthält, muss er in der S-Qualifikation mit n’name’ angegeben werden.

Beispiel

Der Dateiname des Quellprogramms lautet HALLO.C.

S-Qualifikation z.B. im %TRACE-Kommando:

```
/%t 1 in s=n'hallo.c'
```

5 Funktions- und Sprachverknüpfung

Der C/C++-Compiler erfüllt die Konventionen der Programm-Kommunikationsschnittstelle ILCS (Inter-Language Communication Services). Er erzeugt bei der Übersetzung generell ILCS-Objekte.

Diese Objekte lassen sich mit Objekten in anderen Sprachen verknüpfen, die ebenfalls die ILCS-Konventionen erfüllen.

Die ausführliche Beschreibung der Programm-Kommunikationsschnittstelle ILCS sowie die allgemeinen Sprachverknüpfungsregeln finden Sie im CRTE-Benutzerhandbuch [4].

Das vorliegende Kapitel enthält ergänzende Informationen zu den Besonderheiten bei der Verknüpfung von C- und C++-Programmen.

5.1 C/C++-spezifische Verknüpfungskonventionen

Zusätzlich zu den im CRTE-Benutzerhandbuch beschriebenen ILCS-Konventionen sind in C/C++ die im Folgenden aufgeführten Besonderheiten zu beachten.

Parameterübergabe per Wert

In C/C++ werden (laut Sprachdefinition) die Parameter „by value“ übergeben. In der Parameterliste stehen deshalb die Werte der einzelnen Parameter. Diese Form der Parameterübergabe ist nur möglich, wenn ausschließlich C- und C++-Objekte miteinander verknüpft werden.

Informationen zum internen Aufbau der Parameterliste finden Sie im Abschnitt „Implementierungsabhängiges Verhalten gemäß dem ANSI-/ISO-C-Standard“.

Bei der Verknüpfung mit Objekten in anderen Sprachen müssen folgende Maßnahmen getroffen werden:

- Bei Aufruf der anderssprachigen Routine muss die Adresse des zu übergebenden Datenelements angegeben werden, z.B. &par. Technisch gesehen wird dann die Adresse des Datenelements „by value“ übergeben.
- Wird eine C/C++-Routine von einer anderssprachigen Routine aufgerufen, müssen die Formalparameter in C und C++ als Zeiger auf die zu übergebenden Datenelemente deklariert werden, z.B. f(T *par).

Übergabe von Gleitkommazahlen

Bei der Verknüpfung von K&R-C mit C89-, C11- oder C++-Objekten muss bezüglich der Übergabe von Gleitkommazahlen Folgendes beachtet werden:
Im K&R-Modus werden `float`-Werte immer als `double`-Werte übergeben. In den übrigen C-Modi werden `float`-Werte nur dann als `double`-Werte übergeben, wenn keine Prototyp-Deklaration vorhanden ist. Im anderen Fall werden `float`-Werte tatsächlich als `float`-Werte übergeben.
In den C++-Modi werden `float`-Werte immer als `float`-Werte übergeben.

Eine problemlose Übergabe von Gleitkommazahlen zwischen K&R-C und C89-, C11- oder C++-Objekten ist nur dann sichergestellt, wenn im C89- bzw. C11- bzw. C++-Programm die Parameter als `double` deklariert werden.

Übergabe von Funktions-Returnwerten

Bei einer Strukturfunktion wird die Ergebnisstruktur in die Parameterliste kopiert und zusätzlich in R1 ein Zeiger auf die Parameterliste geliefert.

5.2 Sprachverknüpfung C und C++

Die Programmiersprache C ist die Basis für C++. Einer der großen Vorteile von C++ ist die leichte Benutzbarkeit von in C geschriebenen Bibliotheken.

Allerdings weichen die Implementierungen von C und C++ bezüglich der Behandlung von externen Namen voneinander ab: Im Gegensatz zu C werden in C++ externe Namen für den Binder codiert.

Sollen Funktionen und Daten in C++ und in C (oder anderen Sprachen) gemeinsam benutzt werden, muss die Codierung der externen Namen unterdrückt werden. Das C++-Sprachmittel hierfür lautet:

```
extern "C" Deklaration;
```

bzw.

```
extern "C" {
    Deklarationen
}
```

Im C++-Programm sind `extern "C"`-Deklarationen erforderlich sowohl für die in C++ definierten als auch für die in C (oder anderen Sprachen) definierten Daten und Funktionen.

Für den Aufruf der Standard-C-Bibliotheksfunktionen aus C++-Quellen sind lediglich die entsprechenden Standard-Include-Dateien (z.B. stdio.h) einzubinden. Diese enthalten die `extern "C"`-Deklarationen für sämtliche C-Bibliotheksfunktionen.

Bei Sprachverknüpfungen zwischen C und C++ sollten nur C89- bzw. C11-Objekte Verwendung finden. Verknüpfungen von C++-Objekten mit K&R-C-Objekten sind zwar technisch möglich, jedoch u.U. problematisch (es gelten unterschiedliche Restriktionen bzgl. Funktionsdeklarationen, `float`-Parameter werden unterschiedlich übergeben etc.).

5.2.1 Gemeinsame Typen

Um Daten und Funktionen in C und C++ benutzen zu können, müssen sie gleiche oder vergleichbare Typen haben. C++ enthält die gleiche Menge an Basisdatentypen wie C. C++ ist jedoch eine echte Obermenge von C und enthält somit mehr Typen als C.

Folgende Typen können bei der Sprachverknüpfung benutzt werden:

- Elementare Typen: `char`, `short`, `int`, `long`, `long long`, `float`, `double`, `long double`, `void`
- Qualifizierer: `unsigned`, `const`, `volatile`
- Zeiger auf gemeinsame Typen
- Arrays von gemeinsamen Typen (werden als Zeiger übergeben)
- Strukturen und Unions
  Diese dürfen in C++ folgende Komponenten nicht enthalten: Elementfunktionen, Konstruktoren, Destruktoren, Basisklassen, Zugriffsspezifizierer. Die Struktur-/Unionkomponenten müssen gemeinsame Typen haben, d.h. sie müssen syntaktisch gleich aussehen.
- Einfache Aufzählungen
- Funktionen mit gemeinsamen Argument- und Ergebnistypen
  Die Ellipse (...) ist erlaubt.

Folgende Typen dürfen bei der Sprachverknüpfung nicht benutzt werden:

- Referenztypen
- Zeiger auf die Elemente von Klassen, Unions oder Strukturen
- Klassen
- Strukturen und Unions, die Elementfunktionen, Konstruktoren, Destruktoren, Basisklassen oder Zugriffsspezifizierer enthalten
- Aufzählungen mit expliziter Typangabe und/oder Scope: `enum E1 : int; enum struct E2;`
- Typen, die lokal in einer Klasse definiert sind
- Elementfunktionen
- Templates
- `std::nullptr_t`
- `bool`
- `_Complex`, `_Imaginary` (ein C11 Feature)
- Arrays mit variabler Länge (ein C11 Feature)

5.2.2 Aufruf von C-Funktionen in C++

C-Funktionen können in C++ ohne Probleme benutzt werden. Die C-Funktion muss dazu in C++ mit `extern "C"` und ihrem vollständigen Prototyp deklariert werden.

Beispiel

C-Quelle:

```
/* file = C_file.c */
int error_level;
void error(int number, char *text)
{
  printf("Fehler %d, Grund: %s\n", number, text);
}
```

C++-Quelle:

```
// file = C++_file.C
extern "C" int error_level;
extern "C" void error(int, char *);
int main(void)
{
  error_level = 100;
  error(error_level, "TEST");
  return 0;
}
```

Die C++-Quelle enthält `extern "C"`-Deklarationen für alle dort verwendeten C-Bezeichner.

5.2.3 Aufruf von C++-Funktionen in C

C++-Funktionen können in C nur dann benutzt werden, wenn sie einen Typ haben, der auch in C darstellbar ist.

Je nachdem, ob die C++-Quelle verfügbar ist oder nicht, gibt es verschiedene Lösungen für die Funktionsverknüpfung.

Wenn die C++-Quelle zur Verfügung steht, kann die C++-Funktion dort mit `extern "C"` deklariert werden. Die Funktion muss dann allerdings in allen C++-Quellen, die diese Funktion aufrufen, mit `extern "C"` deklariert werden!
In der aufrufenden C-Quelle ist eine „normale“ `extern`-Deklaration für die C++-Funktion erforderlich.

Wenn die C++-Quelle nicht zur Verfügung steht oder eine allgemeine Neu-Deklaration und Neu-Übersetzung nicht tragbar ist, kann eine zusätzliche Funktionsebene eingeschoben werden. In C++ wird eine sog. „Umhüllungsfunktion “ geschrieben und als `extern "C"` definiert. Diese Umhüllungsfunktion kann dann in C ohne Einschränkungen aufgerufen werden.

Beispiel

Aufzurufende C++-Funktion:

```
int hidden(int i)
{
  // ...
}
```

Umhüllungsfunktion:

```
extern int hidden(int);
extern "C" int wrapper(int i)
{
  return hidden(i);
}
```

C-Quelle, die die Umhüllungsfunktion aufruft:

```
extern int wrapper(int);
int main(void)
{
  printf("%d\n", wrapper(100));
  return 0;
}
```

Dieses Verfahren kann auch benutzt werden, wenn eine C++-Funktion einen Parameter enthält, dessen Typ nicht direkt einem C-Datentyp entspricht. Z.B. könnte für eine C++-Funktion mit einem Referenztyp-Parameter eine Umhüllungsfunktion mit einem Parameter vom Typ Zeiger geschrieben werden.

5.2.4 Probleme und Einschränkungen

Die gemeinsame Nutzung von C und C++ in einem Programmsystem unterliegt den nachfolgend aufgeführten Einschränkungen.

- Freispeicherverwaltung

  C++ bietet mit den Sprachmitteln `new` und `delete` eine eigene Freispeicherverwaltung. In C werden für die Speicherverwaltung die Funktionen `malloc` und `free` verwendet. Beide Verfahren dürfen nicht gemischt werden!

  Wenn ein mit `malloc` angeforderter Speicherbereich mit `delete` freigegeben wird (bzw. ein mit `new` angeforderter Speicherbereich mit `free`), ist das Verhalten undefiniert.

- Standard-Ein-/Ausgabedateien

  C++ bietet eine neue Schnittstelle zur Dateiverarbeitung. Die C++-Standarddateien `cin`, `cout`, `cerr` und `clog` entsprechen den C-Standarddateien `stdin`, `stdout` und `stderr`.

  Werden korrespondierende Standarddateien sowohl in C als auch in C++ verwendet, ist kein fehlerfreies Verhalten garantiert.

  Sollen in einem C++-Programm die C-Standarddateien und die C++-Standarddateien gemischt eingesetzt werden, muss die C++-Funktion `ios::sync_with_stdio()` aufgerufen werden (siehe Handbuch „C++-Bibliotheksfunktionen“ [5]).

5.3 Sprachverknüpfung unterschiedlicher C++-Sprachmodi

Die unterstützen C++-Sprachmodi des Compilers fallen in 3 Gruppen:

- C++ V2 (Cfront-C++)
- C++ V3
- C++ 2017

Module innerhalb einer Gruppe können ohne Probleme miteinander gebunden werden.

Module aus zwei verschiedenen Gruppen können nicht miteinander kombiniert werden. Wird eine Funktion in einem Modul einer Gruppe definiert und in einem Modul einer anderen Gruppe aufgerufen, so werden diese Funktionen beim Binden nicht miteinander verknüpft. Es entsteht eine offene Referenz.

5.4 Hinweise zur Verknüpfung mit ILCS-Programmen in anderen Sprachen

Sprachverknüpfung mit C++

In C++ werden externe Namen für den Binder codiert.

Sollen Funktionen und Daten in C++ und in anderen ILCS-Sprachen gemeinsam benutzt werden, muss die Codierung der externen Namen unterdrückt werden. Das C++-Sprachmittel hierfür lautet:

```
extern "C" Deklaration;
```

bzw.

```
extern "C" {
    Deklarationen
}
```

Im C++-Programm sind `extern "C"`-Deklarationen erforderlich sowohl für die in C++ definierten als auch für die in der anderen ILCS-Sprache definierten Daten und Funktionen.

Gemeinsame Dateiverarbeitung

Gemeinsam bearbeitete Dateien müssen sowohl im C/C++-Teil als auch im anderssprachigen Teil eröffnet werden. Ihre Verarbeitung ist intern über verschiedene FCBs realisiert.

Da die Verarbeitung einer gemeinsamen Datei über getrennte FCBs erfolgt, ist ein verzahntes Einlesen nicht möglich. Sämtliche Zeichen der Datei werden dem C-Teil und dem anderssprachigen Teil geliefert.

Aufruf der main-Funktion

Der Aufruf einer `main`-Funktion ist möglich. Als Einsprungadresse ist dann MAIN zu verwenden.
Wenn mehrere Module mit `main`-Funktionen in einer Bibliothek stehen, muss die Auswahl der gewünschten `main`-Funktion durch explizites Einbinden des entsprechenden Moduls sichergestellt werden (INCLUDE- statt RESOLVE-Anweisung).

Die Umlenkung der Standard-Ein-/Ausgabe-Dateien sowie Parameterübergaben an die `main`-Funktion sind nicht möglich.

Gemeinsame STXIT-Ereignisbehandlung

ILCS unterscheidet zwischen einer impliziten sprachspezifischen Ereignisbehandlung und einer explizit an- und abmeldbaren Ereignisbehandlung.
Die implizite sprachspezifische Ereignisbehandlung ist nur für die STXIT-Ereignisse ERROR und PROCHK möglich, die explizite An- und Abmeldung für alle STXIT-Ereignisse.

Die durch verschiedene Sprachen explizit angemeldeten STXIT-Routinen werden parallel verwaltet, d.h. bei Auftritt eines Ereignisses werden die angemeldeten Routinen aller beteiligten Sprachen aufgerufen (in der Reihenfolge der Anmeldung).
Die implizite Ereignisbehandlung wird durch eine explizite Anmeldung außer Kraft gesetzt.

C/C++ hat keine implizite sprachspezifische Ereignisbehandlung. Eine explizite An- und Abmeldung ist mit den C-Bibliotheksfunktionen `signal` und `cstxit` möglich.

Die Anmeldung von Ereignisbehandlungsroutinen für die STXIT-Ereignisklassen ERROR und PROCHK setzt die implizite Ereignisbehandlung anderer Sprachen außer Kraft. Bei Verlassen der C/C++-Sprachumgebung sollten daher die Ereignisbehandlungsroutinen für ERROR und PROCHK explizit abgemeldet werden, um die implizite Ereignisbehandlung anderer Sprachen wieder wirksam werden zu lassen.

Die Abmeldung erfolgt bei der C-Bibliotheksfunktion `signal` durch Zuordnung von SIG_DFL.

6 C-Sprachunterstützung des Compilers

Der Compiler unterstützt optional sowohl den von Kernighan/Ritchie als auch den vom ANSI-/ISO-Standard definierten C-Sprachumfang.

Die Definition von Kernighan/Ritchie ist dokumentiert in:

> „Programmieren in C“, von B.W. Kernighan und D.M. Ritchie, erste Ausgabe, 1983,Hanser-Verlag, ISBN 3-446-13878-1

Die ANSI-/ISO-Definition ist dokumentiert in:

> „International Standard ISO/IEC 9899 : 1990, Programming languages - C“
>
> „International Standard ISO/IEC 9899 : 1990, Programming languages - C /Amendment 1 : 1994“
>
> „International Standard ISO/IEC 9899 : 2011, Programming languages - C“

Ergänzend zur o.g. nicht herstellerspezifischen Literatur werden in den nächsten Abschnitten die implementierungs- und maschinenabhängigen Eigenschaften dieses Compilers sowie die diversen Erweiterungen gegenüber den Sprachdefinitionen beschrieben.

Der Abschnitt Die C-Sprachmodi im Überblick vergleicht die C-Sprachmodi des Compilers und weist auf die wichtigsten Unterschiede hin.
Der Abschnitt Implementierungsabhängiges Verhalten gemäß dem ANSI-/ISO-C-Standard führt Implementierungsabhängigkeiten („Implementation-defined behavior“) gemäß dem ANSI-/ISO-Standard auf.
Der Abschnitt Erweiterungen gegenüber ANSI-/ISO-C beschreibt die C-Spracherweiterungen gegenüber der Definition im ANSI-/ISO-Standard.
Der Abschnitt Pragmas beschreibt die von diesem Compiler unterstützten `#pragma`-Anweisungen.

6.1 Die C-Sprachmodi im Überblick

Der Compiler hat fünf C-Übersetzungsmodi, die sich an unterschiedlichen Sprachstandards orientieren:

K&R-C-Modus (POSIX-Option -X kr, SDF-Option MODE=KERNIGHAN-RITCHIE)

Der Compiler akzeptiert C-Code gemäß der Definition von Kernighan/Ritchie einschließlich einiger ANSI-spezifischer Erweiterungen.

erweiterter C89-Modus (POSIX-Option -X 1990 -X nostrict, SDF-Option MODE=1990, STRICT=NO)

Der Compiler akzeptiert C-Code gemäß der ANSI-C89 Definition bzw. dem ISO C Standard von 1990 (inklusive des ISO-C Amendments 1) im erweiterten Modus.

strikter C89-Modus (POSIX-Option -X 1990 -X strict, SDF-Option MODE=1990, STRICT=YES)

Der Compiler akzeptiert C-Code gemäß der ANSI-C89 Definition bzw. dem ISO C Standard von 1990 (inklusive des ISO-C Amendments 1) im strikten Modus.

erweiterter C11-Modus (POSIX-Option -X 2011 -X nostrict, SDF-Option MODE=2011, STRICT=NO)

Der Compiler akzeptiert C-Code gemäß dem ISO C Standard von 2011 im erweiterten Modus.

strikter C11-Modus (POSIX-Option -X 2011 -X strict, SDF-Option MODE=2011, STRICT=YES)

Der Compiler akzeptiert C-Code gemäß dem ISO C Standard von 2011 im strikten Modus.

Folgende Tabelle gibt einen Überblick über die im jeweiligen ANSI-/ISO-C-Standard definierten Sprachelemente sowie über die in den Übersetzungsmodi unterstützten Sprachelemente.

Die Tabelleneinträge bedeuten:

X	Volle Unterstützung im strikten und im erweiterten Sprachmodus
o	nur syntaktische, keine semantische Unterstützung
-	keine Unterstützung

Lexikalische Elemente

C Sprachelemente	Sprachdefinition			Übersetzungs-Modus		
	C11	C89	K&R	C11	C89	K&R
Multibytezeichen	X	X	-	X	X	X

Trigraph-Sequenzen
`??=` `#`
`??(` `[`
`??/` `\`
`??)` `]`
`??'` `^`
`??<` `{`
`??!` `|`
`??>` `}`
`??-` `~`
	X	X	-	X	X	X

Digraph-Sequenzen [1)]
`<:` `[`
`:>` `]`
`<%` `{`
`%>` `}`
`%:` `#`
`%:%:` `##`
	X	X	-	X	X	X

Escape-Sequenzen						
`\a`	X	X	-	X	X	-
`\b`	X	X	X	X	X	X
`\e`, `\E`	-	-	-	X	X	X
`\f`	X	X	X	X	X	X
`\n`	X	X	X	X	X	X
`\r`	X	X	X	X	X	X
`\t`	X	X	X	X	X	X
`\v`	X	X	-	X	X	X
`\'`	X	X	X	X	X	X
`\"`	X	X	-	X	X	X
`\?`	X	X	-	X	X	X

\ \	X	X	X	X	X	X
\ *octdigits*	X	X	X	X	X	X
\u *decdigits* \U *decdigits*	X	-	-	X	-	-
\x *hexdigits*	X	X	-	X	X	X
Namenslängen						
intern	63	31	8	alle Zeichen signifikant		
extern	31	6	<8	30/32/32K [2]		
Schlüsselwörter [3]						
Konstanten						
integer	X	X	X	X	X	X
float	X	X	X	X	X	X
character	X	X	X	X	X	X
L'character'	X	X	-	X	X	X
u'character' U'character'	X	-	-	X	-	-
string	X	X	X	X	X	X
L"string"	X	X	-	X	X	X
u"string" U"string"	X	-	-	X	-	-
enum	X	X	X	X	X	X
Suffixe						
integer L,l	X	X	X	X	X	X
integer U, u	X	X	-	X	X	X
integer LL, ll	X	-	-	X	X	X
float F, f	X	X	-	X	X	X
float L,l	X	X	-	X	X	X

Lexikalische Elemente

Anmerkungen

1) Digraph-Sequenzen

Digraph-Sequenzen sind erstmals im ISO-C-Amendment 1 definiert. Sie können mit der POSIX-Option `-K alternative_tokens` bzw. der SDF-Option ALTERNATIVE-TOKENS=*YES aktiviert werden. Im Modus C11 sind sie per Default aktiviert.

2) Externe Namenslängen

Genauere Informationen dazu unter "Implementierungsabhängiges Verhalten gemäß dem ANSI-/ISO-C-Standard", Unterpunkt Bezeichner.

3) Reservierte Schlüsselwörter

asm	continue	extern	int	signed	union
auto	default	float	long	sizeof	unsigned
break	do	for	register	static	void
case	double	goto	restrict	struct	volatile
char	else	if	return	switch	while
const	enum	inline	short	typedef	

Das Schlüsselwort `asm` ist im K&R-Modus und im erweiterten C-Modus reserviert. Da die Inline-Generierung von Assembler-Code jedoch nicht unterstützt wird, führt eine Anwendung dieses Schlüsselworts zu einem Fehler. Im strikten C-Modus ist `asm` nicht reserviert.

Die Schlüsselwörter `inline` und `restrict` sind nur in C11 definiert und verfügbar.

Datentyp-Vereinbarung

C Sprachelemente	Sprachdefinition			Übersetzungs-Modus		
	C11	C89	K&R	C11	C89	K&R
Typ-Spezifizierer						
void [1)]	X	X	-	X	X	X
void * [1)]	X	X	-	X	X	X
char	X	X	X	X	X	X
short	X	X	X	X	X	X
int	X	X	X	X	X	X
long	X	X	X	X	X	X
long long	X	-	-	X	X	X
float	X	X	X	X	X	X

double	X	X	X	X	X	X
long double	X	X	-	X	X	X
signed	X	X	-	X	X	X
unsigned	X	X	X	X	X	X
array [ ]	X	X	X	X	X	X
structure [2)]	X	X	X	X	X	X
union [2)]	X	X	X	X	X	X
(*)	X	X	X	X	X	X
enum	X	X	X	X	X	X
()	X	X	X	X	X	X
Typ-Attribute						
const	X	X	-	X	X	o
volatile	X	X	-	X	X	o
Initialisierung						
auto aggregate	X	X	-	X	X	X
Speicherklasse						
typedef	X	X	X	X	X	X
extern	X	X	X	X	X	X
static	X	X	X	X	X	X
auto	X	X	X	X	X	X
register	X	X	X	X	X	X
Bitfeld-Typ						
int	X	X	X	X	X	X
signed int	X	X	X	X	X	X
_Bool	X	-	-	X	X	X
all integral	-	-	-	X	X	X

Datentyp-Vereinbarung

Anmerkungen

1) void

Der Typ `void` bezeichnet eine leere Menge von Werten. Er kann auf folgende drei Arten verwendet werden:

1. Resultattyp von Funktionen, die keinen Wert zurückliefern.
2. Ein Zeiger auf `void` zeigt auf ein Objekt beliebigen Datentyps.
3. In einer Funktionsdeklaration können Anzahl und Datentypen der Parameter angegeben werden (vgl. Prototyping). Mit `void` an Stelle der Parameterliste wird vereinbart, dass die Funktion keine Parameter hat.

2) structure, union

Gemäß ANSI-C lassen sich Strukturen und Unions einander zuweisen (falls gleichen Typs), als Parameter an Funktionen übergeben und als Funktions-Returnwerte zurückliefern.
Diese Möglichkeiten werden auch im K&R-Modus unterstützt.

Konversionsregeln

C Sprachelemente	Sprachdefinition			Übersetzungs-Modus		
	C11	C89	K&R	C11	C89	K&R
value preserving	X	X	-	X	X	-
sign preserving	-	-	X	-	-	X

Konvertierungsregeln

Eine wesentliche Änderung von ANSI gegenüber K&R ergibt sich bei impliziten arithmetischen Konversionen.

Bei K&R werden die Operanden eines Ausdrucks entsprechend der „unsigned-preserving“-Regel konvertiert: Wird ein Operand vom Typ `unsigned char` oder `unsigned short` erweitert, ist das Ergebnis vom Typ `unsigned int`. Kommen in einem Ausdruck `unsigned`-Typen zusammen mit anderen Typen vor, hat das Ergebnis immer einen `unsigned`-Typ.

Bei ANSI gilt jedoch die Regel, den Wert zu bewahren („value-preserving“): Der Ergebnistyp hängt von der Größe des Operandentypen ab. Wird ein Operand vom Typ `unsigned char` oder `unsigned short` erweitert, ist das Ergebnis vom Typ `int`, wenn `int` groß genug ist, um alle Werte des kleineren Typs darzustellen. Ansonsten ist das Ergebnis vom Typ `unsigned int`.

Diese Änderung kann u.U. zu unterschiedlichen Ergebnissen von arithmetischen Ausdrücken und damit zu unterschiedlichem Programmverhalten führen. Dies muss beim Übergang von K&R-C auf ANSI-C berücksichtigt werden.

Funktionen

C Sprachelemente	Sprachdefinition			Übersetzungs-Modus		
	C11	C89	K&R	C11	C89	K&R
Definition „alt“ [1)]	X	X	X	X	X	X
Definition „neu“	X	X	-	X	X	X
Prototyping [2)]	X	X	-	X	X	o
Parametertypangleichung [3)]	X	X	-	X	X	-

Funktionen

Anmerkungen

[1] Definition von Funktionen

ANSI hat gegenüber K&R eine neue Syntax für die Definition der formalen Funktionsparameter eingeführt, lässt jedoch auch noch die „alte“ (K&R) Syntax zu.
Beide Definitionsarten werden auch im K&R-Modus unterstützt.

[2] Prototyping

ANSI definiert gegenüber K&R Funktions-Prototypen. Dies sind Funktions-Deklarationen, in denen zusätzlich Anzahl und Datentypen der einzelnen Parameter angegeben werden. Damit können vom Compiler die Datentypen der Aktualparameter gegen die Formalparameter der Deklaration geprüft und an die Formalparameter angeglichen werden.

Im K&R-Modus sind Prototyp-Deklarationen syntaktisch zugelassen, haben jedoch keine semantische Auswirkung.

[3] Parametertyp-Angleichung

Prototyping bietet den weiteren Vorteil, dass die in der Funktionsdeklaration angegebenen Parameter nicht den Standard-Konversionsregeln unterworfen werden. Ein Parameter, der dort als `float` deklariert ist, wird auch als `float` übergeben. Im anderen Falle würde er vor der Übergabe nach `double` konvertiert werden. Bei Verknüpfung von K&R- und ANSI-Objekten sollten die Gleitkomma-Parameter immer als `double` deklariert werden.

Die automatische Parametertyp-Angleichung wird im K&R-Modus nicht unterstützt.

Präprozessoranweisungen

C Sprachelemente	Sprachdefinition			Übersetzungs-Modus		
	C11	C89	K&R	C11	C89	K&R
# (stringizing)	X	X	-	X	X	X
## (token pasting)	X	X	-	X	X	X
#assert / #unassert	-	-	-	X	X	X
#define	X	X	X	X	X	X
defined	X	X	-	X	X	X
#elif	X	X	-	X	X	X
#else	X	X	X	X	X	X
#endif	X	X	X	X	X	X
#error	X	X	-	X	X	X
#include	X	X	X	X	X	X
#if	X	X	X	X	X	X

#ifdef	X	X	X	X	X	X
#ifndef	X	X	X	X	X	X
#ident	-	-	-	o	o	o
#line	X	X	X	X	X	X
#line (old style)	-	-	X	X	X	X
#pragma	X	X	-	X	X	X
#pragma STDC	X	-	-	o	o	o
#undef	X	X	X	X	X	X
# (null directive)	X	X	-	X	X	X

Präprozessoranweisungen

Vordefinierte Makronamen

C Sprachelemente	Sprachdefinition			Übersetzungs-Modus		
	C11	C89	K&R	C11	C89	K&R
`__LINE__`	X	X	-	X	X	X
`__FILE__`	X	X	-	X	X	X
`__DATE__`	X	X	-	X	X	X
`__TIME___`	X	X	-	X	X	X
`__STDC__`	X	X	-	X	X	X
`__STDC_VERSION__` [1)]	X	X	-	X	X	-

Vordefinierte Makronamen

Anmerkungen

1) `__STDC_VERSION__`

Genauere Informationen dazu unter "Vordefinierte Präprozessornamen".

6.2 Implementierungsabhängiges Verhalten gemäß dem ANSI-/ISO-C-Standard

Bezeichner

Grundsätzlich können Namen in beliebiger Länge gebildet werden. Bei internen Namen sind alle Zeichen signifikant. Bei externen Namen wertet der Compiler standardmäßig maximal 32 Zeichen aus (siehe Regeln unten). Folgende Zeichen sind zur Bildung von Namen zulässig: Gemäß ANSI-/ISO-C die Ziffern 0 bis 9, die Großbuchstaben A bis Z, die Kleinbuchstaben a bis z und der Unterstrich _. Als Erweiterung gegenüber ANSI-/ISO-C sind standardmäßig auch das Dollarzeichen $ und das at-Zeichen @ in Namen zugelassen. Dies lässt sich mit entsprechenden Optionen ausschalten (`-K no_dollar`, `-K no_at` bzw. DOLLAR-ALLOWED=*NO, AT-ALLOWED=*NO).
Multibyte-Zeichen in Bezeichnern werden nicht unterstützt. Im Modus C11 können aber universal-character-names genutzt werden. Diese haben die Form \u0123 oder \U01234567. Nicht alle Ziffernfolgen sind erlaubt.

Für externe Namen gilt Folgendes:

- Standardmäßig, d.h. die Optionen `-K c_names_std` bzw. C-NAMES=*STD sind gesetzt, können externe Namen maximal 32 Zeichen lang sein. Längere Namen werden vom Compiler auf 32 Stellen verkürzt.
  Bei der Generierung von gemeinsam benutzbarem Code (Optionen `-K share` bzw. SHAREABLE-CODE=*YES) können nur maximal 30 Zeichen genutzt werden.

  Bei Angabe der Optionen `-K c_names_unlimited` bzw. C-NAMES=*UNLIMITED findet keine Namensverkürzung statt. Der Compiler generiert Entry-Namen im EEN-Format. EEN-Namen können eine Länge von maximal 32000 Zeichen erreichen.
- Standardmäßig werden Kleinbuchstaben in Großbuchstaben und Unterstriche (_) in Dollarzeichen ($) übersetzt. Durch die Angabe entsprechender Optionen (`-K llm_case_lower`, `-K llm_keep` bzw. LOWER-CASE-NAMES=*YES, SPECIAL-CHARACTERS=*KEEP) können Kleinbuchstaben und Unterstriche in den externen Namen beibehalten werden.
- Externe Namen dürfen nicht mit „`I`“ beginnen.
- Enthält der Identifier universal-character-names, so werden diese als Zeichensequenz im externen Namen abgebildet. Dabei wird der Gegenschrägstrich durch ein Minus-Zeichen ersetzt. Der Buchstabe u bzw. U und die Ziffern bleiben erhalten.

Die obigen Regeln gelten auch für externe Namen, die in C++ als `extern "C"` deklariert sind und außerdem für Static-Funktionen.

main-Funktion

Der Compiler lässt für die Funktion `main` die Return-Typen `int` und `void` zu. Der Return-Typ `void` führt immer zu einer Compiler-Meldung (meist eine Warnung).

Um einem Programm beim Aufruf Argumente übergeben zu können, sind für die Funktion `main` zwei formale Parameter vorzusehen:

```
int main(int argc, char *argv[])
```

Der erste Parameter *argc* zeigt die Anzahl der übergebenen Argumente an. Da das erste Argument argv[0] gemäß Konvention der Programmname ist, ist die Argumentanzahl mindestens 1.

Der zweite Parameter *argv* ist ein Zeiger auf einen Vektor von Zeichenketten. In ihm werden der Programmname (in argv[0]) und alle beim Programmaufruf eingegebenen Argumente als mit dem Nullbyte (\0) abgeschlossene Zeichenketten abgespeichert.

Als Erweiterung gegenüber ANSI-/ISO-C kann für die Funktion `main` ein dritter Parameter `char *envp[]` vereinbart werden (siehe "Erweiterungen gegenüber ANSI-/ISO-C").

Weitere Einzelheiten zur Übergabe von Parametern an die `main`-Funktionen finden Sie im Abschnitt „Eingabe der Parameter für die main-Funktion“.

Zeichen (character)

Der Datentyp `char` wird von diesem Compiler standardmäßig als `unsigned` behandelt (siehe auch Optionen `-K uchar`, `-K schar` bzw. SIGNED-CHARACTER=*NO/*YES).

Der Wert eines EBCDIC-Zeichens ist immer positiv.
Der Wert von '\377' (oktal) oder '\xFF' (sedezimal) ist also 255.
Das Verhalten ist undefiniert, wenn eine character-Konstante einen numerischen Wert enthält, der nicht im EBCDIC-Zeichensatz enthalten ist.

Der Wert einer character-Konstante, die mehr als ein Zeichen enthält (z.B. 'ab'), berechnet sich aus dem EBCDIC-Wert der Zeichen als Zahl zur Basis 256. Das erste (rechte) Zeichen wird mit 1 multipliziert, das zweite Zeichen mit 256, das dritte Zeichen mit 256 * 256, das vierte Zeichen mit 256 * 256 * 256.
Z.B. ergibt 'abcd' den Wert 'a' * 256^3 + 'b' * 256^2 + 'c' * 256 + 'd' (= 2172814212).

Der Wert einer Multibyte-Zeichenkonstante in der Form L'ab' ist in dieser Implementierung identisch mit dem Wert einer Zeichenkonstante in der Form 'ab'.

Wenn eine character-Konstante fünf oder mehr Zeichen enthält, wird ein Error ausgegeben und kein Code generiert.
Die Zuweisung eines `int` an `char` geschieht modulo 256.

Multibytezeichen

In dieser Implementierung haben Multibytezeichen immer die Länge 1 Byte und `wchar_t`-Werte sind immer Integer-Werte der Größe 32 bit.

Zeiger

Ein Zeiger wird in 4 Bytes dargestellt, mit Ausrichtung auf Wortgrenze. Die Differenz zwischen zwei Zeigern ist vom Typ `int` (`ptrdiff_t`).

Arrays

Arrays in C haben üblicherweise feste Grenzen, die Größe eines Arrays ist in solchen Fällen bereits zur Übersetzungszeit bekannt. Eine Ausnahme bilden die VLA (variable lenght array) aus C11. Diese können nur auf Block-Scope vorkommen.

Ein Arrayname wird in C immer wie ein Zeiger behandelt, der auf das erste Element des Arrays zeigt.

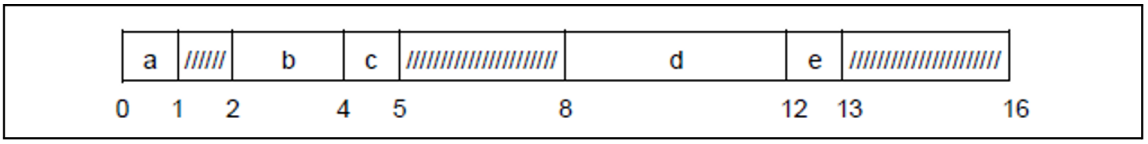
Die Elemente werden sequenziell im Speicher abgelegt, das erste Element hat den Index 0. Bei mehrdimensionalen Arrays werden die Elemente so im Speicher abgelegt, dass der letzte Index am schnellsten variiert. Jedes Element wird, wie das Array selbst, entsprechend dem Elementtyp ausgerichtet.

Strukturen

In Strukturen belegen die Komponenten Platz in der Reihenfolge ihrer Deklaration. Jede Komponente wird dabei entsprechend ihrem Typ ausgerichtet. Die Struktur selbst wird auf die maximal erforderliche Ausrichtungsgröße einer Komponente ausgerichtet. Die Strukturgröße ist ein Vielfaches dieser Ausrichtung, damit Arrays von diesen Strukturen gebildet werden können. Siehe auch „Interne Darstellung der Datentypen (Ausrichtung und Darstellung in Registern)“ und Präprozessor-Anweisung `#pragma aligned`, "aligned-Pragma".

Beispiel

```
                          Größe:       Ausrichtung:      Offset:
struct {  char  a;        1 Byte       Bytegrenze         0 (Wortgrenze)
          short b;        2 Byte       Halbwortgrenze     2
          char  c;        1 Byte       Bytegrenze         4
          long  d;        4 Byte       Wortgrenze         8
          char  e;        1 Byte       Bytegrenze        12
       };                                                 16 (Strukturende)
```



Bitfelder

Bitfelder werden von links nach rechts in maximal 32 Bit (einem Wort) abgespeichert.

Bitfelder können folgendermaßen definiert sein:

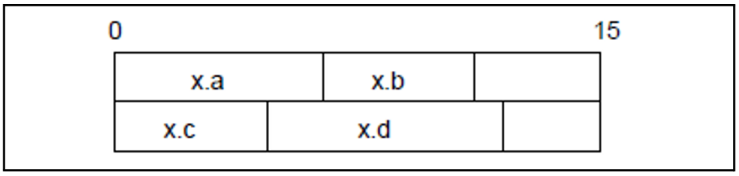
```
int         unsigned int         signed int
long        unsigned long        signed long
short       unsigned short       signed short
char        unsigned char        signed char
```

Bitfelder ohne den Zusatz `unsigned` oder `signed` werden entsprechend dem Basis-Datentyp dargestellt, d.h. `char` als `unsigned` und `int`, `long` und `short` als `signed`. Bei expliziter Angabe von `signed` oder `unsigned` werden die Bitfelder entsprechend dieser Angabe dargestellt. Dieses voreingestellte Verhalten kann durch folgende Optionen verändert werden: `-K schar` , `-K signed_fields_unsigned` und `-K plain_fields_unsigned` bzw. SIGNED-FIELDS=*UNSIGNED, PLAIN-FIELDS=*UNSIGNED, SIGNED-CHARACTER=*YES.

Die angegebene Anzahl von Bits wird ohne Ausrichtung angelegt, falls das Bitfeld noch ganz im aktuellen Byte, Halbwort, Wort oder Doppelwort Platz findet; andernfalls wird das Bitfeld entsprechend dem Grundtyp auf Byte-, Halbwort-, Wort- oder Doppelwortgrenze ausgerichtet (siehe Beispiel unten).

Beispiel

```
struct
{
    unsigned short  a : 7;
    unsigned short  b : 5;
    unsigned short  c : 5;
    unsigned short  d : 8;
} x;
```



Aufzählung (enum)

Ohne explizite Wertzuweisung werden den Konstanten bei der Definition eines Aufzählungstyps nacheinander die Zahlen 0, 1, etc. zugeordnet. Wird einer Konstanten ein Wert explizit zugewiesen, erhalten die nachfolgenden Konstanten automatisch einen entsprechend höheren Wert.

Standardmäßig wird ein Aufzählungstyp je nach den äußeren Grenzen (höchster und niedrigster Wert) dargestellt wie `char`, `short` oder `long`. Mit den Compileroptionen `-K enum_long` bzw. ENUM-TYPE=*LONG kann erreicht werden, dass enum-Daten unabhängig von ihrem tatsächlichen Platzbedarf stets als `long` dargestellt werden.

Typqualifizierer volatile

`volatile` verhindert die Optimierung beim Zugriff auf eine Variable. Anstelle der Verwendung des alten Inhalts wird stets neu aus dem Speicher eingelesen. Bei allen Zuweisungen, auch bei redundanten, wird der entsprechende Wert unmittelbar in den Speicher geschrieben. Es wird von der Implementierung garantiert, dass Referenzen auf `volatile`-Objekte auf Werte zeigen, die im Speicher stehen, im Gegensatz zu nicht-`volatile`-Objekten, die weitreichenden Optimierungen unterzogen und beispielsweise in Registern gehalten werden.

Im K&R-Modus wird `volatile` nur syntaktisch akzeptiert.

size_t

`size_t` entspricht in dieser Implementierung `unsigned int`.

ptrdiff_t

`ptrdiff_t` entspricht in dieser Implementierung `int`.

Konvertierung von Datentypen

- Integer --> Integer

  Bei der Konvertierung eines vorzeichenlosen Integer-Werts in einen vorzeichenbehafteten Integer-Typ gleicher Größe wird das Bitmuster beibehalten. Wenn der Wert nicht aufgenommen werden kann, entspricht das Ergebnis der Subtraktion der größtmöglichen Zahl + 1 von der gegebenen Größe.

  Wenn bei der Konvertierung eines Integer-Werts in einen kleineren Integer-Typ der Wert nicht aufgenommen werden kann, wird das Bitmuster beibehalten. Die höherwertigen Bits werden abgeschnitten.

- Gleitkommazahl --> Integer

  Bei der Konvertierung einer Gleitkommazahl nach Integer wird in Richtung 0 abgeschnitten.

  Beispiel

  (int)(-1.5) ist -1

  (int)(1.5) ist 1

  Das Ergebnis ist nicht definiert, wenn die zu konvertierende Gleitkommazahl zu groß ist, um als Integer-Wert dargestellt werden zu können.
- Integer --> Gleitkommazahl

  Bei der Konvertierung eines Integer- in einen Gleitkomma-Typ, der den korrekten Wert nicht aufnehmen kann, wird gerundet.
- Gleitkommazahl --> Gleitkommazahl

  Bei der Konvertierung einer Gleitkommazahl in eine kleinere Gleitkommazahl (z.B. `double` nach `float`) wird gerundet.
- Integer <--> Zeiger

  Bei der Konvertierung Integer nach Zeiger und umgekehrt wird das Bitmuster nicht verändert (einfache Uminterpretierung).

Vorzeichen des Divisionsrestes

Der Rest einer ganzzahligen Division hat immer dasselbe Vorzeichen wie der Dividend.

Beispiel

(-5) / 2 ist -2, (-5) % 2 ist -1
5 / (-2) ist -2, 5 % (-2) ist 1

Rechts-Shift logisch und arithmetisch

Rechts-Shift ist logisch (Auffüllen von 0-Bits), wenn der linke Operand `unsigned` ist, sonst arithmetisch (Auffüllen von Vorzeichen-Bits).

Beispiel

(-8) >> 1 ist -4

Bitweise Operationen auf vorzeichenbehaftete Integerwerte

Bitweise Operationen (Operatoren ~, <<, &, ^, und |) werden bei Interpretation als vorzeichenlose Integer ausgeführt, das Ergebnis ist wieder vorzeichenbehaftet.

Deklaratoren

Für die Vereinbarung eines Typs sind beliebig viele Deklaratoren zugelassen.

switch-Anweisung

Pro `switch`-Anweisung sind beliebig viele `case`-Zweige zugelassen.

Präprozessor-Anweisungen

- #include

  Eine Folge von Include-Dateien <name> bzw. "name" ist nicht zulässig. Es wird nur der erste Name akzeptiert.

  #include-Anweisungen, in denen die Namen der Include-Dateien Schrägstriche (/) für Verzeichnisse enthalten, werden vom Compiler auch im Falle von PLAM-Bibliothekselementen akzeptiert. Jeder Schrägstrich in den Namen von benutzereigenen und Standard-Include-Dateien wird intern zur Suche in PLAM-Bibliotheken in einen Punkt umgewandelt.
  In Quellprogrammen, die z.B. aus dem POSIX- oder UNIX-System portiert werden, müssen deshalb die Schrägstriche nicht in Punkte umgewandelt werden.

  Beispiel

  ```
  #include <sys/types.h>
  ```

  Der Compiler sucht die Standard-Include-Datei SYS.TYPES.H in der CRTE-Bibliothek $.SYSLIB.CRTE.

  Bezüglich der Schachtelung von Include-Dateien gibt es keine Einschränkungen.

- #pragma

  Siehe Abschnitt „Pragmas“.

- __DATE__, __TIME__

  Falls Datum und Uhrzeit der Übersetzung nicht verfügbar sind, sind diese Makros folgendermaßen definiert:

__DATE__	"Jan 1 1970"
__TIME__	"01:00:00"

Größe und Wertebereiche der elementaren Datentypen

Typ	Bit	Wertebereiche
char	8	0 .. 255
signed char	8	-128 .. 127
short	16	-32768 .. 32767
unsigned short	16	0 .. 65535
int	32	-2147483648 .. 2147483647 (-2^{31} .. 2^{31}-1)
unsigned int	32	0 .. 4294967295 (0 .. 2^{32}-1)
long	32	wie int
unsigned long	32	wie unsigned int
long long	64	-9223372036854775808 .. 9223372036854775807 (-2^{63}.. 2^{63}-1)
unsigned long long	64	0 .. 18446744073709551615 (0 .. 2^{64}-1)
float	32	10^{-75} .. $0.79*10^{76}$

double	64	wie float
long double	124	wie float

Interne Darstellung der Datentypen (Ausrichtung und Darstellung in Registern)

Im Folgenden wird zusammenfassend gezeigt, wie die einzelnen C-Datentypen intern im Speicher abgebildet werden.

Bei skalaren Typen wird zusätzlich auf ihre Abbildung in Registern eingegangen. Dadurch wird zum einen festgelegt, wie die Variablen mit der Speicherklasse `register` dargestellt werden, zum anderen, wie der Wert einer solchen Variablen in Ausdrücken interpretiert wird.

Datentyp	**Größe**	**Ausrichtung**	**Darstellung in Registern**
char, unsigned char, signed char	1 Byte	Bytegrenze	rechtsbündig
short, unsigned short	2 Byte	Halbwortgrenze	rechtsbündig
int, unsigned int	4 Byte	Wortgrenze	wie im Speicher
long, unsigned long	4 Byte	Wortgrenze	wie im Speicher
long long, unsigned long long	8 Byte	Doppelwortgrenze	keine Darstellung in Registern
Zeiger	4 Byte	Wortgrenze	wie im Speicher
float	4 Byte	Wortgrenze	linksbündig
double	8 Byte	Doppelwortgrenze	Für Darstellung wird keine Konvertierung benötigt
long double	16 Byte	Doppelwortgrenze	Für Darstellung wird ein Gleitpunkt-Registerpaar verwendet

Datentyp	**Größe und Ausrichtung**
Aufzählung	Je nach den Grenzwerten dargestellt wie `char`, `short` oder `long` mit entsprechender Ausrichtung.
Arrays	Größe und Ausrichtung entsprechend dem Elementtyp.
Strukturen	Größe und Ausrichtung für jede einzelne Komponente nach obigen Regeln. Gesamtausrichtung nach maximaler Ausrichtung der Komponenten.
Bitfelder	Die angegebene Anzahl von Bits wird ohne Ausrichtung angelegt, wenn dadurch die Ausrichtungsgrenze des Grundtyps nicht überschritten wird, ansonsten werden die Bits gemäß der Ausrichtung des Grundtyps angelegt.

Implementierungsspezifische Grenzwerte

Die meisten Grenzwerte werden von den Systemressourcen vorgegeben (z.B. vom virtuellen Speicher). Nur die folgenden Grenzwerte sind von der Implementierung vorgegeben:

Merkmal	Maximalwert
Anzahl der Parameter in einer Makrodefinition	2^{24}-1
Anzahl der Argumente in einem Makroaufruf	2^{24}-1
sizeof-Grenzwert	2^{31}

Speicherklassen

In diesem Abschnitt wird zusammenfassend gezeigt, wie den Variablen in Abhängigkeit von ihrer Speicherklasse Speicher zugeordnet wird.

Speicherklasse register

Variablen können mit `register` als Registervariablen deklariert werden. Dies ist ein Hinweis an den Compiler, dass die Variablen relativ oft benutzt werden und deshalb möglichst in Registern gehalten werden sollen. Beim Lesen und Schreiben dieser Variablen werden teure Speicherzugriffe eingespart. Die Optimierung des Compilers kann sich allerdings über diese Hinweise hinwegsetzen und nach einem eigenen Algorithmus bestimmte Variablen als Registervariablen realisieren.

Speicherklasse auto (default)

Für lokale Variablen mit der (voreingestellten) Speicherklasse `auto` wird Speicher in einer Automatic Data Area reserviert.

Parameter in der Parameterliste

Funktionsparameter werden in der Reihenfolge ihres Auftretens in einer Parameterliste übergeben.

Alle `unsigned`... Parameter werden wie `unsigned` dargestellt, alle anderen ganzzahligen Parameter (`char`, `short`) wie `int`: rechtsbündig in je einem Wort, ausgerichtet auf Wortgrenze und evtl. nach links aufgefüllt mit Vorzeichenbits (`int`) oder Nullen (`unsigned`...). Zeiger belegen ein Wort.

Abhängig vom Sprachmodus, werden Gleitkommazahlen unterschiedlich übergeben.
Im K&R-Modus werden Gleitkommazahlen (`float`, `double`) immer in doppelter Genauigkeit übergeben, also als Doppelwort ausgerichtet auf Doppelwortgrenze.
Im ANSI-Modus werden `float`-Werte nur dann in doppelter Genauigkeit übergeben, wenn keine Prototyp-Deklaration vorhanden ist. Im anderen Fall werden `float`-Werte in einfacher Genauigkeit übergeben, also als Wort ausgerichtet auf Wortgrenze.
In den C++-Sprachmodi werden `float`-Werte immer in einfacher Genauigkeit übergeben, da Prototyp-Deklarationen vorhanden sein müssen.

`long double` wird in zwei Doppelworten übergeben, ausgerichtet auf Doppelwortgrenze.

Strukturparameter werden je nach Bedarf auf Wort- oder Doppelwortgrenze ausgerichtet. Die Größe einer Struktur wird nach der maximal erforderlichen Ausrichtungsgröße einer Komponente aufgefüllt. Wenn z.B. eine Struktur nur `short`- und `char`-Komponenten enthält, ergibt sich als Größe ein Vielfaches von 2 Bytes.

Arrays können nicht als Wert übergeben werden. Es wird ein Zeiger auf das erste Array-Element übergeben.

Statische Variablen

Für die folgenden Arten von statischen Variablen reserviert der Compiler bereits bei der Übersetzung Speicher:

- Lokale static-Variablen
- Globale static-Variablen
- Globale extern-Variablen

Der Unterschied zwischen diesen Speicherklassen liegt im Gültigkeitsbereich:

- Lokale static-Variablen sind innerhalb einer Funktion mit dem Speicherklassenattribut `static` definierte Variablen. Sie sind nur der Funktion bekannt, in der sie definiert wurden.
- Globale static-Variablen sind außerhalb einer Funktion mit dem Speicherklassenattribut `static` definierte Variablen. Diese sind nur innerhalb einer Übersetzungseinheit bekannt.
- Globale extern-Variablen sind Variablen, die außerhalb einer Funktion ohne das Speicherklassenattribut `static` definiert wurden. Auf diese Variablen kann auch in anderen Übersetzungseinheiten zugegriffen werden, wenn diese dort mit dem Attribut extern deklariert werden.

Funktionen ohne Prototyp

Wenn eine Funktion ohne Prototyp aufgerufen wird und es sind Parameter-Informationen vorhanden, wird in einigen Fällen ein Fehler ausgegeben. Dies geschieht, wenn eine Definition „alten Stils“ oder ein Prototyp im K&R-Modus vorgefunden werden.

Sind Argument und Parameter – nach der üblichen Typ-Erweiterung – verschiedenen Typs und einer der folgenden Punkte trifft zu, wird ein Fehler ausgegeben:

- Parameter und Argument sind von unterschiedlicher Größe
- Parameter und Argument sind von unterschiedlicher Ausrichtung
- Der Parameter ist vom Typ float, double oder long double
- Das Argument ist vom Typ float, double oder long double

Der Fehler kann zu einer Warnung herabgestuft werden. Wird dies getan, wird der Aufruf zur Laufzeit in aller Regel scheitern.

6.3 Abweichungen zu ANSI-/ISO-C

In diesem Abschnitt werden folgende Themen behandelt:

- Erweiterungen gegenüber ANSI-/ISO-C
- Einschränkungen gegenüber ANSI-/ISO-C

6.3.1 Erweiterungen gegenüber ANSI-/ISO-C

Die Verwendung nicht ANSI-/ISO-konformer Sprachfeatures, wozu die im Folgenden beschriebenen Erweiterungen zählen, führt zu potenziell nicht portablen Quellprogrammen.

Sonderzeichen $ und @ in Bezeichnern

Standardmäßig sind das Dollarzeichen $ bzw. das at-Zeichen @ in internen und externen Namen zugelassen. Dies kann durch Optionen unterdrückt werden (siehe „Bezeichner“ (Implementierungsabhängiges Verhalten gemäß dem ANSI-/ISO-C-Standard)).

main-Funktion mit drei Parametern

Zusätzlich zu den Parametern *argc* und *argv* (siehe „main-Funktion" (Implementierungsabhängiges Verhalten gemäß dem ANSI-/ISO-C-Standard)) kann ein dritter Parameter `char *`*envp*`[]` vereinbart werden. *envp* ist ein Zeiger auf einen Vektor von Zeichenketten, der mit Informationen zur Systemumgebung versorgt wird. Weitere Einzelheiten hierzu finden Sie im Handbuch „C-Bibliotheksfunktionen für POSIX-Anwendungen“ [3].

Gültigkeitsbereich von Funktionen

`extern`-Deklarationen von Funktionen innerhalb von Blöcken gelten für die gesamte Übersetzungseinheit. Wenn mehrere `extern`-Deklarationen für die gleiche Funktion vorliegen, werden sie auf Übereinstimmung überprüft.

Schreibzugriff auf Zeichenketten-Literale

Zeichenketten-Literale sind in dieser Implementierung standardmäßig überschreibbar. Dabei ist sichergestellt, dass sich die Literale nicht überlappen. Identische Literale werden in separaten Bereichen abgespeichert.
Bei Angabe der Optionen `-K rostr` bzw. STRING-LITERALS=*READ-ONLY kann auf Zeichenketten-Literale nur lesend zugegriffen werden.

Konvertierung von Funktions-Zeigern

Es ist erlaubt, mit dem cast-Operator Zeiger auf Objekte in Zeiger auf Funktionen sowie Zeiger auf Funktionen in Zeiger auf Objekte umzuwandeln. Bei impliziten Umwandlungen gibt der Compiler Warnungsmeldungen aus.

Nicht-Integer Bitfelder

Es können alle Integral-Typen (außer `long long`) als Bitfelder verwendet werden (siehe auch „Bitfelder“ (Implementierungsabhängiges Verhalten gemäß dem ANSI-/ISO-C-Standard)). Der Standard definiert lediglich die Typen `int`, `unsigned int` und `signed int`.

Schlüsselwort asm

Das Schlüsselwort `asm` ist im K&R-Modus und im erweiterten C-Modus reserviert. Da die Inline-Generierung von Assembler-Code jedoch nicht unterstützt wird, führt eine Anwendung dieses Schlüsselworts zu einem Fehler. Im strikten C-Modus ist `asm` nicht reserviert.

Mehrfachdefinitionen von externen Variablen

Wenn in mehreren Übersetzungseinheiten sog. „tentative“ Definitionen für dasselbe Objekt vorliegen (dies sind externe Deklarationen von Variablen ohne das Attribut `extern` oder `static`), müssen diese immer vom selben Typ sein. Unterschiedliche Typ-Deklarationen für dasselbe externe Objekt werden vom Compiler nicht erkannt. Mehrere Initialisierungen von externen Variablen führen zu Fehlern beim Binden. Dieses Verhalten lässt sich mit Optionen steuern (`-K external_multiple`, `-K external_unique` bzw. EXTERNAL-DEFINITION=*UNIQUE /*MULTIPLY-ALLOWED).

Leere Makro-Argumente

Bei dem Aufruf von Makros können auch leere Argumente übergeben werden.

Beispiel

```
#define F(a,b)   f(a)+f(b);
F(1)    /* ergibt f(1)+f(); */
F(,1)   /* ergibt f()+f(1); */
```

Vordefinierte Makros

Es sind einige Präprozessor-Makros aus Kompatibilitätsgründen vordefiniert, die nicht mit dem Unterstrich (_) beginnen (siehe „Vordefinierte Präprozessornamen“).

Zusätzliche Präprozessor-Anweisungen

Folgende Präprozessor-Anweisungen werden vom Compiler zusätzlich akzeptiert: `#line` (altes Format), `#ident`, `#assert` und `#unassert`.

`#line`-Anweisung (altes Format):

```
# Ziffern-Folge [Include-Datei]
```

Diese Anweisung ist gleichbedeutend mit der `#line`-Anweisung; es fehlt lediglich das Schlüsselwort `line`.

`#ident`-Anweisung:

```
#ident "zeichenkette"
```

Die `#ident`-Anweisung wird, wie auch die Anweisung `#pragma ident`, syntaktisch akzeptiert, führt jedoch zu keiner Veränderung im erzeugten Objekt. Der Compiler gibt keine Notes oder Warnings aus, da diese Anweisungen intern in System-Headern benutzt werden.

`#assert`-Anweisung:

```
#assert name[(token-Folge)]
```

Mit der `#assert`-Anweisung kann ein Prädikat (Assertion) definiert werden. Prädikate sind unabhängig von Makrodefinitionen.

name ist der Name des Prädikats. *token-Folge* ist der Wert, für den das Prädikat gilt.
Ein einzelnes *token* kann eines der folgenden lexikalischen Einheiten sein: Name, Schlüsselwort, Konstante, Zeichenkette, Operator, Trenn-/Interpunktionszeichen. Ist keine *token-Folge* angegeben, gilt wie bei einer symbolischen Konstante das Prädikat als definiert, allerdings ist kein Wert zugeordnet.

Mit der `#if`-Anweisung kann geprüft werden, ob ein Prädikat für einen Wert gilt:

`#if #`*name*`(`*nicht-leere-token-Folge*`)`

name ist der Name des Prädikats. *nicht-leere-token-Folge* ist der Wert, der überprüft werden soll. Beispielsweise würde die folgende Abfrage des vordefinierten Prädikats `compiler` die Bedingung „wahr“ liefern:

```
#if #system(bs2000)
```

Die vordefinierten Prädikate finden Sie im Anhang unter „Vordefinierte Präprozessor-Prädikate (#assert)“ (Vordefinierte Präprozessornamen).

`#unassert`-Anweisung:

```
#unassert name[(token-Folge)]
```

Ein Prädikat kann mit der `#unassert`-Anweisung wieder gelöscht werden. Die `#unassert`-Anweisung hat dieselbe Syntax wie die `#assert`-Anweisung.
Ist eine *token-Folge* angegeben, wird nur für diesen Wert das Prädikat gelöscht. Ist keine *token-Folge* angegeben, wird das Prädikat insgesamt gelöscht.

6.3.2 Einschränkungen gegenüber ANSI-/ISO-C

Folgende Sprachmittel werden vom hier beschriebenen Compiler nicht oder nur eingeschränkt unterstützt:

Datentyp long long

Bitfelder und Ausdrücke in `switch`-Anweisungen vom Typ `long long` werden nicht unterstützt.

Explizite Ausrichtung

Es gibt 3 Möglichkeiten, die Ausrichtung zu verändern: Das `#pragma aligned`, das C11-Schlüsselwort `_Alignas` und das C++-Schlüsselwort `alignas`. Diese Möglichkeiten wirken nur für Strukturelemente. Sie wirken nicht auf einfache Variablen.

Deklaration ohne Typ

Im Sprachmodus C11 muss jede Deklaration einen expliziten Typ haben. In C89 konnte der Typ fehlen, es wurde dann `int` angenommen. Fehlt der Typ, so kommt im Modus C11 ein Fehler, der zu einer Warnung herabgestuft werden kann.

Beispiel

```
%  CFE1260[*ERROR]: x.c /      3: explicit type is missing ("int" assumed)
  const var1 = 7;
        ^
%  CFE1077[*ERROR]: x.c /      5: this declaration has no storage class or type specifier
  var2;
  ^
%  CFE1260[*ERROR]: x.c /      7: explicit type is missing ("int" assumed)
  extern function (int a);
         ^
%  CFE1260[*ERROR]: x.c /      9: explicit type is missing ("int" assumed)
  function2 (int a);
  ^
```

Unicode

Unter dem Begriff Unicode ist ein Standard gemeint, der für viele Zeichen der Welt eine binäre Darstellung angibt. Der Standard spezifiziert mehr als 100.000 verschiedene Zeichen. Der Standard definiert mehrere Darstellungen, die bekanntesten sind UTF-8 und UTF-16. In diesen Darstellungen sind die meisten Zeichen durch mehr als 1 Byte repräsentiert.

Der C11 Standard hat Sprachmittel geschaffen, um diese Zeichen im Sourcecode zu beschreiben: Zeichen mit \u, Typen für Unicode-Zeichen und Literale in Unicode. Diese Sprachmittel werden vom C/C++ 4.0 Compiler unterstützt.

Zu beachten ist jedoch, dass die Funktionen des CRTE nur Zeichen und Strings des normalen EBCDIC oder ASCII unterstützen. Dabei wird angenommen, dass jedes Zeichen in einem Byte dargestellt wird. Dies bedeutet, dass die CRTE-Funktionen zur Bearbeitung von Unicode-Strings nicht geeignet sind.

Float-Format IEEE

Der Standard IEC 60559 wird vom Compiler nicht unterstützt. Der Compiler erlaubt es zwar, Float-Konstanten in dem IEEE-Format abzulegen. Es fehlen jedoch einige Funktionen zur Konfiguration des Verhaltens. Auch werden die Werte infinity und NaN nicht sinnvoll unterstützt. Im Zusammenhang mit diesen Werten kann es zu seltsamem Verhalten kommen.

Unterstützung von Threads

Der C/C++ Compiler unterstützt kein bestimmtes Thread-Paket. Im Detail heisst dies, dass die Header <threads.h> und <stdatomic.h> nicht vorhanden sind. Die Schlüsselworte `_Thread_local` und `_Atomic` führen im Modus C11 zu einer Fehlermeldung. Die Prädefines `__STDC_NO_THREADS__` und `__STDC_NO_ATOMICS__` sind gesetzt.

Komplexe Zahlen

Der C/C++ Compiler unterstützt nicht die im C11 Standard erwähnten komplexen Zahlen. Im Detail heisst dies, dass der Header <complex.h> nicht unterstützt wird. Der im CRTE vorhandene Header <complex.h> wird für komplexe Zahlen in C++ verwendet, seine Verwendung in C führt zu einer Fehlermeldung. Die Schlüsselworte `_Complex` und `_Imaginary` führen im Modus C11 zu einer Fehlermeldung. Das Prädefine `__STDC_NO_COMPLEX__` ist gesetzt.

C11 Anhang K

Im C11-Standard, Anhang K werden Funktionen beschrieben, die unter den Begriff "Bounds-checking interfaces" fallen. Diese Funktionen werden nicht angeboten.

C11 Anhang L

Im C11-Standard, Anhang L werden Funktionen beschrieben, die unter den Begriff "Analyzability" fallen. Diese Funktionen werden nicht angeboten.

6.4 Pragmas

Dieser Abschnitt beschreibt die vom Compiler akzeptierten, lt. ANSI-/ISO-Standard implementierungsabhängigen `#pragma`-Anweisungen:

- aligned-Pragma
- pack-Pragma
- ETPND-Pragma
- Pragmas zum Steuern des Listenbildes
  - LISTING-Pragma
  - PAGE-Pragma
  - SPACE-Pragma
  - TITLE-Pragma
- inline-Pragma
- int_to_unsigned-Pragma
- weak-Pragma
- ident-Pragma
- VIRTUAL_FUNCTION_TAB-Pragma
- Pragmas zur Steuerung der Template-Instanziierung
- deprecated-Pragma
- STDC-Pragma
- __printf_args-Pragma
- __scanf_args-Pragma

6.4.1 aligned-Pragma

Mit dem `aligned`-Pragma können die Datenelemente innerhalb von Klassen, Strukturen und Unions auf eine größere Anzahl Bytes ausgerichtet werden, als in der standardmäßigen Minimal-Ausrichtung durch den Compiler vorgesehen ist.

Das Pragma kann in allen Sprachmodi des Compilers verwendet werden.

```
#pragma aligned n
```

n gibt eine Anzahl Bytes in Zweierpotenzen bis maximal 8 an. *n* kann als Dezimal-, Oktal- oder Sedezimalzahl angegeben werden. Zulässige Angaben (dezimal) sind 1, 2, 4, 8 und 16 (syntaktisch noch erlaubt) Byte.

Hinweise

> Zur Vereinfachung wird im Folgenden nur noch von Strukturen gesprochen. Die Hinweise gelten sinngemäß auch für Klassen und Unions.

- Pragmas, die weniger Bytes angeben als für die Minimal-Ausrichtung des entsprechenden Datentyps vorgesehen (siehe Tabelle unten), sind nicht zulässig und werden ignoriert.

Datentyp	Minimal-Ausrichtung (Anzahl Bytes)
char	1
short	2
int	4
long	4
long long	8
float	4
Zeiger	4
double	8
long double	8

- Das auszurichtende Datenelement kann auch vom Typ Bitfeld sein. In diesem Fall wird das Bitfeld in einem neuen Basisfeld mit der angeforderten Ausrichtung angelegt.
- Bei `static`-Elementen wird das Pragma ignoriert.
- Das Pragma muss innerhalb der Strukturdefinition unmittelbar vor dem auszurichtenden Datenelement stehen. An anderer Stelle wird es ignoriert.
- Stehen mehrere Pragmas vor einem Datenelement, wird dasjenige mit der größten Ausrichtungsangabe berücksichtigt.
- Steht ein Pragma vor der Deklaration mehrerer Strukturelemente, bezieht es sich nur auf das erste deklarierte Element.
- Die Ausrichtung einer Struktur richtet sich nach der maximalen Ausrichtung ihrer Elemente.

- Tritt eine Struktur als Element einer anderen Struktur auf, bezieht sich das davorstehende Pragma auf die Ausrichtung der Gesamtstruktur, nicht auf die Ausrichtung ihrer Elemente.
- Ein Pragma vor einem Strukturelement vom Typ Array bezieht sich auf die Ausrichtung des gesamten Arrays (also auf das erste Arrayelement), nicht auf die übrigen Arrayelemente.
- aligned-Pragma und pack-Pragma (siehe "pack-Pragma") können gleichzeitig für ein bestimmtes Strukturelement aktiv sein. In diesem Fall hat das aligned-Pragma Vorrang.

Beispiel

```
...
class bsp1
  {
    int a;                  // Ausrichtung auf 4 Bytes.
#pragma aligned 8
    int b,c;                // b wird auf 8 Bytes ausgerichtet.
                            // c wird auf 4 Bytes ausgerichtet.
                            // Die maximale Ausrichtung eines Elements
                            // der Klasse bsp1 beträgt daher 8 Bytes.
                            // Die Klasse bsp1 wird daher auf 8 Bytes
                            // ausgerichtet.
    int d;                  // Ausrichtung auf 4 Bytes.
  };
class bsp2
  {
public:
    double dens;            // Ausrichtung auf 8 Bytes.
#pragma aligned 4           // Wird ignoriert. Da die maximale Ausrichtung
                            // eines Elements der Struktur stru1 8 Bytes
                            // beträgt, wird auch stru1 auf 8 Bytes
                            // ausgerichtet. Eine entsprechende Warnung
                            // wird ausgegeben.
    struct
      {
        int istru11;        // Ausrichtung auf 4 Bytes.
        double dstru12;     // Ausrichtung auf 8 Bytes.
      } stru1;              // Ausrichtung auf 8 Bytes (s.o.).
    struct
      {
        short s1;           // Ausrichtung auf 2 Bytes.
        short s2;           // Ausrichtung auf 2 Bytes.
      } stru2;              // Wird auf 2 Bytes ausgerichtet, da die maximale
                            // Ausrichtung eines Elements 2 Bytes beträgt.
#pragma aligned 4
    char c;                 // Ausrichtung auf 4 Bytes.
    char c1;                // Ausrichtung auf 1 Byte.
#pragma aligned 8
    short ar1[16];          // Das Array wird auf 8 Bytes ausgerichtet,
                            // nicht jedoch die einzelnen Arrayelemente.
    ...
  }
...
```

6.4.2 pack-Pragma

Dieses Pragma steuert das Layout von Strukturen. Die Ausrichtung der Strukturelemente aller Strukturen wird auf die im Pragma spezifizierte Zahl reduziert. Auf diese Weise wird die Größe der Strukturen reduziert. Der Gültigkeitsbereich eines pack-Pragmas erstreckt sich bis zum nächsten pack-Pragma.

```
#pragma pack ([n])
```

n kann als Dezimal-, Oktal- oder Sedezimalzahl angegeben werden. Zulässige Angaben (dezimal) sind 1, 2, 4 und 8.

Defaultwert: 8

Achtung!

> Die Verwendung des pack-Pragmas erhöht die Laufzeit, da der Zugriff auf nicht ausgerichtete Strukturelemente aufwändiger ist als der Zugriff auf ausgerichtete Strukturelemente.
>
> Die Verwendung der Adresse eines Strukturelements ist riskant. Der Versuch, mithilfe dieser Adresse auf das Strukturelement zuzugreifen, kann zu einem Dump führen.

6.4.3 ETPND-Pragma

Mit diesem Pragma kann in jedem Modul (Code- und Daten-CSECT) des erzeugten LLM ein Dokumentationsbereich angelegt werden. Dieser Bereich enthält allgemeine Informationen über die Übersetzungseinheit, wie z.B. die Versionsnummer oder das Erstellungsdatum. Angaben über die Funktion des Moduls sind darin jedoch nicht enthalten. Für später eventuell erforderliche Korrekturen kann ein Patchbereich reserviert werden.

Format:

```
#pragma ETPND { CODE | DATA }
              [,VER= version ]
              [,DATE= datum ]
              [,COMPNR= compnr ]
              [,PATCH= anzahl ]
              [,MODULLENGTH= länge ]
```

CODE
DATA — Der ETPND-Bereich wird im Code- oder Datenmodul angelegt.

version — Modulversion als Dezimalzahl im Wertebereich [0..999]. Fehlt diese Angabe, wird 0 angenommen.

datum — Erstellungsdatum im Format jjjjmmtt, jjjj-mm-tt oder jjmmtt.
Bei Angabe des sechsstelligen Formats liegt die Jahreszahl zwischen 1960 und 2059. Die beiden fehlenden Stellen der Jahreszahl (19 oder 20) werden passend ergänzt.

Das Datum muss zwischen dem 1.1.1905 und dem 1.1.2035 liegen.

Wenn Überläufe bei den Monaten oder Tagen auftreten, werden diese kanonisch fortgeschrieben. Die Angabe 19961335 (35.13.1996) entspricht also 19970204 (4.2.1997).

Wird kein Erstellungsdatum angegeben, verwendet der Compiler das Übersetzungsdatum.

compnr — Komponentennummer als Dezimalzahl im Wertebereich [0..99999999]. Fehlt diese Angabe, wird 0 angenommen.

anzahl — Größe des Patchbereichs in byte. Der Wert darf nicht größer als 4294967295 (0xFFFFFFFF) sein. Fehlt diese Angabe, wird im Codemodul ein Bereich von 200 Bytes reserviert und im Datenmodul kein Bereich (0). Wird PATCH=0 angegeben, wird auch im Codemodul kein Patchbereich reserviert. Der Wert kann als Dezimal-, Oktal- oder Sedezimalzahl angegeben werden.

länge Länge des Moduls einschließlich des ETPND-Bereichs (24+7 Bytes wegen Ausrichtung auf Doppelwortgrenze) in byte; dieser Operand kann dazu verwendet werden, ein Modul auf Seitengrenze abschließen zu lassen. Der Wert darf nicht größer als 4294967295 (0xFFFFFFFF) sein. Wenn der Wert für *länge* kleiner als die tatsächliche Modullänge ist, wird die Angabe ignoriert. Der Wert kann als Dezimal-, Oktal- oder Sedezimalzahl angegeben werden.

Hinweise

- In einem ETPND-Pragma dürfen nicht gleichzeitig MODULLENGTH und PATCH angegeben werden. Wenn das ETPND-Pragma beide Angaben enthält, wird MODULLENGTH ignoriert.
- Pro Modul (d.h. pro Code- und Daten-CSECT) darf nur ein ETPND-Pragma angegeben werden. Werden mehrere ETPND-Pragmas zu einem Modul angegeben, wird das zuletzt angegebene verwendet.

6.4.4 Pragmas zum Steuern des Listenbildes

Es besteht die Möglichkeit, vom Quelltext aus mit #`pragma`-Anweisungen die Gestaltung der Quellprogramm-/Fehlerliste und der Präprozessorliste zu beeinflussen.
Die anderen Compilerlisten sind durch Pragmas nicht beeinflussbar.

Mit den Optionen `-K pragmas_interpreted`, `-K pragmas_ignored` bzw. LISTING-PRAGMAS=... kann global angegeben werden, welche der im Quelltext auftretenden Pragmas berücksichtigt bzw. ignoriert werden sollen.

6.4.4.1 LISTING-Pragma

Mit dem LISTING-Pragma kann die Ausgabe von Quelltextzeilen unterdrückt werden.

```
#pragma LIST[ING] { OFF | ON }
```

`LIST OFF` bewirkt, dass alle auf diese Anweisung folgenden Quelltextzeilen in der Liste nicht abgebildet werden.

Mit `LIST ON` wird die Wirkung von `LIST OFF` aufgehoben, d.h. die darauf folgenden Quelltextzeilen werden wieder abgebildet.

Die Quelltextzeilen einer Include-Datei werden nicht abgebildet, wenn die entsprechende `#include`-Anweisung durch ein `#pragma LIST OFF` und ein `#pragma LIST ON` eingeschlossen wird.

Beziehen sich Fehlermeldungen des Compilers auf Quelltextzeilen, deren Ausgabe unterdrückt wird, so werden die Meldungen an der Stelle eingefügt, wo normalerweise der betroffene Quelltext stehen würde. Als Zusatzinformation erhält man den entsprechenden Datei-/Elementnamen und die Zeilennummer.

Beispiel

Quelltext:

```
98     ...
99     ...                    <-- Meldung zu Zeile 99
100    ...
101    #pragma LIST OFF
102    #include "msg.h"
  1    ...
...                            <-- Meldungen zu nicht abgebildeten Zeilen
 45    ...
103    #pragma LIST ON
104    ...
```

Quellprogramm-/Fehlerliste:

```
98     ...
99     ...
****>  CFE2234 [ERROR]    : ...
100    ...
101    #pragma LIST OFF
****>  CFE1004 [NOTE]     : msg.h / 13: ...
****>  CFE1386 [WARNING]  : msg.h / 37: ...
103    #pragma LIST ON
104    ...
```

6.4.4.2 PAGE-Pragma

Das PAGE-Pragma bewirkt einen Seitenvorschub. Wahlweise kann analog zum TITLE-Pragma ein zusätzlicher Text für den Listenkopf definiert werden.

```
#pragma PAGE [text]
```

text in Anführungszeichen eingeschlossene Zeichenkette aus druckbaren Zeichen; Steuerzeichen wie `\n`, `\t` etc. sind nicht zulässig.

Die Anweisung `#pragma PAGE` ohne *text* bewirkt nur einen Seitenvorschub.

Die Anweisung `#pragma PAGE` *text* bewirkt einen Seitenvorschub, wobei der Listenkopf zusätzlich den angegebenen Text enthält. Bzgl. der Ausgabe von Textzeilen gelten die gleichen Bedingungen wie beim TITLE-Pragma.

6.4.4.3 SPACE-Pragma

Mit dem SPACE-Pragma können in der Liste Leerzeilen erzeugt werden.

```
#pragma SPACE [n]
```

n ist eine nicht-negative ganze Zahl. In der Liste werden *n* Leerzeilen eingefügt.
n kann als Dezimal-, Oktal- oder Sedezimalzahl angegeben werden.

Fehlt die Angabe *n*, wird genau eine Leerzeile eingefügt.

6.4.4.4 TITLE-Pragma

Das TITLE-Pragma definiert einen Text, der zusätzlich zu den standardmäßig generierten Zeilen in den Listenkopf geschrieben wird.

```
#pragma TITLE text
```

text in Anführungszeichen eingeschlossene Zeichenkette aus druckbaren Zeichen; Steuerzeichen wie `\n`, `\t` etc. sind nicht zulässig.

Ist der Text länger als die mit den Optionen `-N output` bzw. LINE-SIZE=... definierte Zeilenlänge, wird er in mehrere Zeilen entsprechender Länge aufgeteilt.

Der Text wird erst ab der zweiten Seite in den Listenkopf geschrieben. Ein mit der INITIAL-TITLE-TEXT-Option definierter Text wird durch das TITLE-Pragma überschrieben. Auf der ersten Seite der Liste steht entweder eine Leerzeile oder der mit der INITIAL-TITLE-TEXT-Option definierte Text.

Die Definition des Textes gilt bis zum nächsten TITLE- oder PAGE-Pragma bzw. bis zum Dateiende.

Unzulässige Zeichen (z.B. Steuerzeichen) führen zu einer Warnung und werden durch Leerzeichen ersetzt.

Wenn durch die Generierung mehrerer zusätzlicher Textzeilen im Listenkopf die minimal vorgesehene Anzahl von 11 Zeilen pro Seite (siehe Optionen `-N output` bzw.LINES-PER-PAGE) nicht ausreicht, um zusätzlich zum Listenkopf und -fuß mindestens eine Quelltextzeile auszugeben, gibt der Listengenerator eine Warnung aus und wählt eine entsprechend höhere Zeilenanzahl.

6.4.5 inline-Pragma

Mit dem inline-Pragma können die Namen von benutzereigenen C-Funktionen angegeben werden, die der Compiler inline generieren soll.

```
#pragma inline name
```

name ist der Name einer C-Funktion, die inline generiert werden soll.

Die angegebenen C-Funktionen werden nur dann inline generiert, wenn die folgenden Optionen angegeben werden: In POSIX die Optionen `-F inline_by_source` oder `-F i`, in SDF die Option INLINING=*YES. In den C++-Sprachmodi wird das inline-Pragma nicht unterstützt, da es in C++ eigene Sprachmittel für die Inline-Generierung von Funktionen gibt.

6.4.6 int_to_unsigned-Pragma

`#pragma int_to_unsigned` *name*

Dieses Pragma wird unterstützt, weil es in älteren C-Quellen, die beispielsweise aus einem UNIX-System portiert werden, enthalten sein kann. Es wirkt nur im K&R-Modus und veranlasst den Compiler, eine Funktion *name* vom Ergebnistyp `unsigned` so zu behandeln, als hätte sie weiterhin den Ergebnistyp `int`.

Die Deklaration der Funktion *name* mit dem Ergebnistyp `unsigned` muss vor der `#pragma` Anweisung stehen, z.B.

```
unsigned int function(const char*);

#pragma int_to_unsigned function
```

6.4.7 weak-Pragma

```
#pragma weak name
```

name wird als globales Symbol mit dem Attribut „bedingter Externverweis“ (WXTRN) deklariert (zu WXTRNs siehe auch Handbuch „BINDER“ [14]). Ein Symbol, das als `weak` deklariert ist, kann beim Binden offen bleiben. Wenn eine Referenz auf *name* nicht aufgelöst werden kann, gibt der BINDER lediglich eine Informationsmeldung aus. Bedingte Externverweise werden nur beim expliziten Einbinden (INCLUDE-MODULES) aufgelöst, nicht beim Einbinden von Modulen per Autolink (RESOLVE-BY-AUTOLINK).
Externe C++-Namen dürfen nicht als `weak` deklariert werden.

6.4.8 ident-Pragma

```
#pragma ident "zeichenkette"
```

Dieses Pragma wird vom Compiler nur syntaktisch akzeptiert. Es kann in C-Quellen, die beispielsweise aus einem UNIX-System portiert werden, enthalten sein.

6.4.9 VIRTUAL_FUNCTION_TAB-Pragma

```
#pragma VIRTUAL_FUNCTION_TAB = {GLOBALLY_DEFINED | EXTERNALLY_DECLARED}
```

Dieses Pragma kann genutzt werden, um im Modus C++ die Generierung der Tabelle für virtuelle Funktionen (virtual function table) zu kontrollieren. Diese Tabelle wird implizit generiert, wenn eine Klasse mindestens eine virtuelle Funktion hat.

Normalerweise bestimmt der Compiler ein Modul, welches die Definition der Tabelle enthält, in anderen Modulen wird dann nur ein Externverweis generiert. Dafür wird geprüft, ob die Klasse eine virtuelle Funktion hat, die weder inline noch eine reine virtuelle Funktion ist. Es wird dann das Modul genommen, in dem die erste dieser Funktionen definiert ist.

Es gibt jedoch Klassen, für die diese Heuristik fehlschlägt. Für diese Klassen wirkt dieses Pragma:

- Wird das Pragma nicht angegeben, wird die Tabelle als static-Variable angelegt.
- Wird GLOBALLY-DEFINED angegeben, so wird die Tabelle in diesem Modul als externe Variable definiert.
- Wird EXTERNALLY-DECLARED angegeben, so wird nur ein Extern-Verweis auf die Tabelle generiert. Sie muss dann in einem anderen Modul definiert sein.

6.4.10 Pragmas zur Steuerung der Template-Instanziierung

```
#pragma { instantiate | do_not_instantiate | can_instantiate } argument
```

Bei C++-Übersetzungen kann die Instanziierung einzelner Templates oder auch einer Gruppe von Templates mit folgenden Pragmas gesteuert werden:

- Das Pragma `instantiate` bewirkt, dass die als Argument angegebene Template-Instanz erzeugt wird. Dieses Pragma kann in allen Instanziierungsmodi benutzt werden.
- 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. Dieses Pragma kann in allen Instanziierungsmodi benutzt 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.

Folgende Argumente können mit den Pragmas angegeben werden:

Argument	*Beispiele*
Name eines Klassen-Templates	`A<int>`
Name einer Elementfunktion	`A<int>::f`
Name eines statischen Datenelements	`A<int>::i`
Deklaration einer Elementfunktion	`void A<int>::f(int, char)`
Deklaration eines Funktions-Templates	`char * f(int, float)`

Wenn als Argument der Name eines Klassen-Templates angegeben wird (z.B. `A<int>`), so hat dies die gleiche Wirkung, als wenn das Pragma für jede Elementfunktion und für jedes statische Datenelement dieses Klassen-Templates angegeben worden wäre. Bei der Instanziierung einer gesamten Klasse kann die Instanziierung von einzelnen Elementfunktionen oder statischen Datenelementen mit dem Pragma `do_not_instantiate` unterdrückt werden.

Beispiel

```
#pragma instantiate A<int>
#pragma do_not_instantiate A<int>::f
```

Für die Instanziierung von Templates müssen die Template-Definitionen in der aktuellen Übersetzungseinheit verfügbar sein. Es führt zu einer Fehlermeldung (ERROR), wenn eine Instanziierung explizit mit dem Pragma `instantiate` angefordert wird und keine Template-Definition oder nur eine spezifische Definition (Spezialisierung) vorliegt.

Beispiel

```
template <class T> void f1(T); // no body provided
template <class T> void g1(T); // no body provided
template <> void f1(int) { } // specific definition
int main()
{
   int i;
   double d;
   f1(i);
   f1(d);
   g1(i);
   g1(d);
}
#pragma instantiate void f1(int) // error - specific definition
#pragma instantiate void g1(int) // error - no body provided
```

`f1(double)` und `g1(double)` werden wegen der fehlenden Template-Definitionen ebenfalls nicht instanziiert, jedoch in diesem Fall ohne die Ausgabe einer Fehlermeldung während der Übersetzung. Zum Bindezeitpunkt führen fehlende Template-Definitionen zu Binder-Fehlern.

Wenn als Pragma-Argument der Name einer Elementfunktion (z.B. `A<int>::f`) angegeben wird, darf es sich um keine überladene Funktion handeln. Bei überladenen Elementfunktionen muss die vollständige Funktions-Deklaration angegeben werden, z.B.`#pragma instantiate char * A<int>::f(int, char *)`

Als Argumente der Instanziierungs-Pragmas sind Compiler-generierte Funktionen, Inline-Funktionen und reine virtuelle Funktionen generell unzulässig.

6.4.11 deprecated-Pragma

```
#pragma deprecated
```

Das Pragma muss vor einer Deklaration stehen. Der dort deklarierte Name wird als *deprecated* markiert. Dies hat die gleiche Wirkung wie das C++-Attribut `[[deprecated]]`. Eine Nutzung eines solchen Symbols führt zu einer Warnung.

6.4.12 STDC-Pragma

Der C11 Standard definiert 3 spezielle Pragmas. Sie dienen der Kontrolle von Randbereichen der Gleitpunkt-Arithmetik. Der C/C++ Compiler akzeptiert diese Pragmas in den Sprachmodi C11 und C++ 2017, sie haben aber keinen Effekt. Die Übersetzung erfolgt immer, als ob die Pragmas wie folgt gesetzt sind:

```
#pragma STDC CX_LIMITED_RANGE ON
#pragma STDC FENV_ACCESS OFF
#pragma STDC FP_CONTRACT ON
```

6.4.13 __printf_args-Pragma

Dieses Pragma wird genutzt, um eine zusätzliche Parameter-Überprüfung für eine Funktion zu aktivieren. Ein Beispiel wäre:

```
#pragma __printf_args

void my_printf(int arg1, const char *format, ...);

void foo()

{

  my_printf(5, "falsches Argument %s\n", 42);

}
```

Das Pragma muss vor der Deklaration einer Funktion stehen. Für diese Funktion wird angenommen, dass sie der Funktion `printf` ähnelt. Bedingung ist, dass die Funktion eine variable Anzahl von Argumenten hat. Es wird dann angenommen, dass der letzte feste Parameter einen Formatstring bezeichnet, der die Typen der folgenden Parameter beschreibt. Die Formatanweisungen in diesem Formatstring haben die gleiche Semantik wie bei der Funktion `printf`.

Wird beim Aufruf der Funktion dieser Parameter mit einem String-Literal versorgt, so prüft der Compiler den Typ aller folgenden Argumente. Der Typ wird mit der dazu passenden Formatanweisung verglichen. Passen diese nicht zusammen, so wird eine Warnung ausgegeben.

In dem Beispiel oben kommt so eine Warnung. Die Formatanweisung "%s" erwartet ein Argument vom Typ `char *` , es wurde jedoch ein `int` übergeben.

6.4.14 __scanf_args-Pragma

Dieses Pragma wird genutzt, um eine zusätzliche Parameter-Überprüfung für eine Funktion zu aktivieren. Ein Beispiel wäre:

```
#pragma __scanf_args
void my_scanf(int arg1, const char *format, ...);
void foo()
{
  int i;
  my_scanf(5, "falsches Argument %d\n", i);
}
```

Das Pragma muss vor der Deklaration einer Funktion stehen. Für diese Funktion wird angenommen, dass sie der Funktion `scanf` ähnelt. Bedingung ist, dass die Funktion eine variable Anzahl von Argumenten hat. Es wird dann angenommen, dass der letzte feste Parameter einen Formatstring bezeichnet, der die Typen der folgenden Parameter beschreibt. Die Formatanweisungen in diesem Formatstring haben die gleiche Semantik wie bei der Funktion `scanf`.

Wird beim Aufruf der Funktion dieser Parameter mit einem String-Literal versorgt, so prüft der Compiler den Typ aller folgenden Argumente. Der Typ wird mit der dazu passenden Formatanweisung verglichen. Passen diese nicht zusammen, so wird eine Warnung ausgegeben.

In dem Beispiel oben kommt so eine Warnung. Die Formatanweisung "%d" erwartet ein Argument vom Typ `int *`, es wurde jedoch ein `int` übergeben. Es fehlt ein Adress-Operator "&".

7 C++-Sprachunterstützung des Compilers

Der Compiler unterstützt optional sowohl die zur Vorgängerversion C/C++ V3.2 kompatiblen Sprachmodi als auch den aktuellen C++ Standard ISO/IEC 14882:2017.

Die alten Sprachmodi sind in der 2. und 3. Ausgabe von "Die C++ Programmiersprache" von Bjarne Stroustrup beschrieben. Der aktuelle Standard kann bei DIN (Deutsche Institut für Normung) bestellt werden.

Die folgenden Abschnitte ergänzen die Sprachbeschreibungen hinsichtlich der herstellerspezifischen, implementierungsabhängigen C++-Spracheigenschaften.

7.1 Die C++-Sprachmodi im Überblick

Der Compiler hat fünf C++-Übersetzungsmodi, die sich an unterschiedlichen Sprachdefinitionen orientieren:

*Cfront-C++-Modus (POSIX-Option -X V2-COMPATIBLE, SDF-Option MODE=*V2-COMPATIBLE)*

Der Compiler akzeptiert Cfront V3.0.3-kompatiblen C++-Code.

*erweiterter C++ V3-Modus (POSIX-Option -X V3-COMPATIBLE -X nostrict, SDF-Option MODE=*V3-COMPATIBLE, STRICT=*NO)*

Der Compiler akzeptiert C++-Code analog zum C/C++ V3-Compiler im erweiterten Modus.

*strikter C++ V3-Modus (POSIX-Option -X V3-COMPATIBLE -X strict, SDF-Option MODE=*V3-COMPATIBLE, STRICT=*YES)*

Der Compiler akzeptiert C++-Code analog zum C/C++ V3-Compiler im strikten Modus.

*erweiterter C++ 2017-Modus (POSIX-Option -X 2017 -X nostrict, SDF-Option MODE=*2017,STRICT=*NO)*

Der Compiler akzeptiert C++-Code gemäß dem C++-Standard von 2017 im erweiterten Modus.

*strikter C++ 2017-Modus (POSIX-Option -X 2017 -X strict, SDF-Option MODE=*2017,STRICT=*YES)*

Der Compiler akzeptiert C++-Code gemäß dem C++-Standard von 2017 im strikten Modus.

Die folgende Tabelle gibt einen Überblick über die wichtigsten Unterschiede zwischen den C++-Sprachmodi.

Merkmal / Spracheigenschaft	**Cfront C++**	**C++ V3**	**C++ 2017**
reservierte Schlüsselwörter [1)]			
`overload`	ja	nein	nein
Ausnahmebehandlung `catch`, `throw`, `try`	nein [2)]	ja	ja
Templates `template`	nein [2)]	ja	ja
Sprachkonstrukte von C++ 1998			
Laufzeit-Typinformation (RTTI) `typeid, dynamic_cast`	nein [3)]	ja	ja
Array-new/delete `new[], delete[]`	nein	ja	ja
Namensraum `namespace, using`	nein	ja	ja
Template-Parameter `typename`	nein	ja	ja
Konstruktor-Typ `explicit`	nein	ja	ja

Datentyp `wchar_t`	nein	ja [4]	ja
Datentyp `bool`	nein	ja [5]	ja
Speicherklasse `mutable`	ja	ja	ja
Casting-Schlüsselwörter `const_cast, reinterpret_cast, static_cast`	ja	ja	ja
`export` [6]	nein	nein	ja
externe `inline`-Funktionen	nein	nein	ja
Symbol-Suche in Namenräumen einer Argument-Klasse (Koenig-Lookup)	nein	nein	ja
Sprachkonstrukte von C++ 2017			
Alignment-Kontrolle `alignas`, `alignof`	nein	nein	ja
Semantik von `auto`	Speicherklasse	Speicherklasse	Typ
Unicode Unterstützung: Literale u"x", U"x", u8"x", u'x', U'x', u8'x', `char16_t`, `char32_t`	nein	nein	ja
Berechnung zur Übersetzungszeit `constexpr`	nein	nein	ja
`decltype`	nein	nein	ja
`long long`	ja	ja	ja
`noexcept` [7]	nein	nein	ja
`nullptr`	nein	nein	ja
`register` (wurde in C++17 verboten)	ja	ja	nein
`static_assert`	nein	nein	ja
`thread_local`	nein	nein	ja [8]
Trigraph (z.B. ??<)	ja	ja	nein
vordefinierte Präprozessornamen			
`__STDC_VERSION__`	==199409L	==199409L	==199409L
`__cplusplus`	==1	==2 (erweitert) ==199612L (strikt)	==201703L

[1] Reservierte Schlüsselwörter

Alle im Kapitel „C-Sprachunterstützung des Compilers“ (Die C-Sprachmodi im Überblick) aufgeführten Schlüsselwörter, außer `restrict`, sind auch in den C++-Sprachmodi reserviert. Hinzu kommen die folgenden C++-spezifischen Schlüsselwörter. Schlüsselwörter in **Fettdruck** sind nicht in allen C++-Sprachmodi reserviert (siehe obige Tabelle).

alignas	**constexpr**	mutable	public	try
alignof	**decltype**	**namespace**	reinterpret_cast	**typeid**
asm	delete	new	**static_assert**	**typename**
bool	dynamic_cast	**noexcept**	static_cast	**using**
catch	**explicit**	**nullptr**	template	virtual
char16_t	**export**	operator	this	**wchar_t**
char32_t	**false**	**overload**	**thread_local**	
class	friend	private	throw	
const_cast	inline	protected	**true**	

Reservierte Namen in C++

Die folgenden Schlüsselwörter können als Ersatzdarstellungen für C-Operatoren verwendet werden. Sie sind abhängig von den Optionen `-K alternative_token` ,

`-K no_alternative_token` bzw. ALTERNATIVE-TOKENS=*YES/*NO reserviert oder nicht. Im Cfront-C++-Modus ist nicht reserviert voreingestellt, in den C++ V3-Modi und den C++ 2017-Modi reserviert.

and	&&	not_eq	!=
and_eq	&=	or	||
bitand	&	or_eq	|=
bitor	|	xor	^
compl	~	xor_eq	^=
not	!		

Schlüsselwort-Operatoren in C++

[2] Ausnahmebehandlung, Templates

Im Cfront-C++-Modus werden Ausnahmebehandlung und Templates nicht unterstützt. Die Schlüsselwörter `catch`, `throw`, `try` und `template` sind jedoch nicht frei verfügbar und führen bei Gebrauch zu einer Fehlermeldung.

[3] Laufzeit-Typinformation (RTTI)

Im Cfront-C++-Modus ist das Schlüsselwort `dynamic_cast` aktiv. Es wird geprüft, ob die damit angesprochene Operation echte Laufzeit-Information benötigt oder ob sie mit Hilfe der statischen Typ-Information gemacht werden kann. Reicht die statische Information, wird der passende Code generiert. Braucht es dynamische Information, wird eine Fehlermeldung ausgegeben.

4) Datentyp `wchar_t`

In den C++ V3-Modi ist das Schlüsselwort `wchar_t` abhängig von den Optionen `-K wchar_t_keyword`, `-K no_wchar_t_keyword` bzw. KEYWORD-WCHAR=*YES/*NO reserviert. `-K wchar_t_keyword` bzw. KEYWORD-WCHAR=*YES ist voreingestellt.

5) Datentyp `bool`

In den C++ V3-Modi sind die Schlüsselwörter `bool`, `true` und `false` abhängig von den Optionen `-K bool`, `-K no_bool` bzw. KEYWORD-BOOL=*YES/*NO reserviert. `-K bool` bzw. KEYWORD-BOOL=*YES ist voreingestellt.

6) export

Die Nutzung des Wortes `export` führt in allen C++-Modi zu einer Meldung. In C++ 2017 ist es ein Fehler, in den anderen Modi eine Warnung. Die hinter dem Schlüsselwort stehende Semantik wird vom C/C++ Compiler nicht angeboten.

7) noexcept: Ausnahmebehandlung als Teil des Typs

Es gibt in C++ die Möglichkeit, für eine Funktion anzugeben, ob sie Ausnahmen (Exceptions) auslöst oder nicht. Ab C++17 ist nur noch die Angabe ja/nein erlaubt, und diese Angabe ist Teil des Typs der Funktion. Vorher konnte eine Liste von Typen angegeben werden, die geworfen werden können. Diese Liste wurde dynamisch geprüft und war kein Teil des Typs der Funktion.

8) `thread_local`

Der Standard C++ 2017 definiert `thread_local` als Schlüsselwort. Es kann im Rahmen von Threads genutzt werden. Da der C/C++ Compiler keine Thread-Implementierung unterstützt, wird das Schlüsselwort als solches erkannt und mit einer Fehlermeldung abgewiesen.

7.2 Implementierungsabhängiges Verhalten gemäß dem ANSI-/ISO-C++-Standard

Alle bereits im Kapitel „C-Sprachunterstützung des Compilers“ (Implementierungsabhängiges Verhalten gemäß dem ANSI-/ISO-C-Standard) beschriebenen Implementierungsabhängigkeiten gemäß dem ANSI-/ISO-C-Standard treffen auch in den C++-Sprachmodi zu und werden in diesem Abschnitt nicht mehr aufgeführt.

Im Folgenden werden nur die über die Sprache C hinausgehenden implementierungsabhängigen C++-Spracheigenschaften beschrieben. Welche C++-Spracheigenschaften in welchen C++ Modi unterstützt werden, können Sie der Übersichtstabelle im Abschnitt "Die C++-Sprachmodi im Überblick" entnehmen.

Bezeichner mit externer Bindung

- Übersetzung im Cfront-C++-Modus
  Externe C++-Namen werden vom Compiler auf 32 Zeichen verkürzt. Bei der Generierung von gemeinsam benutzbarem Code (Optionen `-K share` bzw. SHAREABLE-CODE=*YES) können nur maximal 30 Zeichen genutzt werden. Standardmäßig werden Kleinbuchstaben in Großbuchstaben umgewandelt. Durch die Angabe der Option `-K llm_case_lower` bzw. LOWER-CASE-NAMES=*YES können Kleinbuchstaben in den externen Namen beibehalten werden.
- Übersetzung in den Modi C++ V3 bzw. C++ 2017
  Für die Bildung von externen C++-Namen sind alle Zeichen signifikant. Es findet keine Namensverkürzung und keine Umwandlung von Klein- in Großbuchstaben statt. Der Compiler generiert Entry-Namen im EEN-Format, die eine Länge von maximal 32000 Zeichen erreichen können. Längere Namen führen zu einem Fehler.

Externe C++-Namen werden für den Binder codiert, so dass sie nur noch erlaubte Zeichen enthalten und eindeutig sind (name mangling). Diese interne Namenscodierung erfolgt im Cfront-C++-Modus anders als in den anderen C++-Modi. Mit der Option `-N project` bzw. PROJECT-INFORMATION=*YES erhält man eine Liste mit einer Gegenüberstellung aller im Quellprogramm original verwendeten externen C++-Namen und den Namen, die der Compiler für den Binder intern generiert.

Linkage der main-Funktion

Die Funktion `main` hat externe C-Linkage.

Datentyp bool

Der Typ `bool` hat die Größe `sizeof(bool) == 1`.

reinterpret_cast

Das Zielobjekt enthält dasselbe Bitmuster wie der Ausdruck, der durch `reinterpret_cast` konvertiert wurde. Aber das Zielobjekt ist möglicherweise kein gültiges Objekt (z.B. `reinterpret_cast<float>(int)`) des gewünschten Datentyps.

Folgende Konversionen sind möglich:

1. Ein Zeiger kann explizit in einen Integer-Typ konvertiert werden, der groß genug dafür ist. In Abhängigkeit davon, ob der Zieltyp `signed` oder `unsigned` ist, kann der Ergebniswert der Konversion vorzeichenbehaftet sein.
2. Der Wert eines Integer-Typs kann explizit in einen Zeiger konvertiert werden.

Array-new

Für jedes allokierte Array wird am Beginn des für das Array allokierten Speicherblocks eine Struktur reserviert. Diese Struktur enthält zwei `size_t`-Elemente, eine für die Größe des Arrays in Bytes und eine für die Anzahl der Array-Elemente (dieses Feld wird verschlüsselt, um erkennen zu können, ob die Struktur überschrieben worden ist).

Schlüsselwort asm

Während das Schlüsselwort `asm` gegenüber dem ANSI-/ISO-C-Standard eine Erweiterung darstellt, ist es im ANSI-C++-Standard als reserviertes Schlüsselwort definiert. Da jedoch die Generierung von Inline-Assemblercode nicht unterstützt wird, führt der Gebrauch zu einer Fehlermeldung.

Linkage-Spezifizierer

Es werden die Linkage-Spezifizierer "C++" und "C" unterstützt.

Der voreingestellte Linkage-Spezifizierer ist "C++". Namen mit externer C++-Linkage werden vom Compiler intern umgewandelt, damit sie vom Binder verarbeitet werden können (name mangling). Diese interne Namenscodierung erfolgt in den verschiedenen C++-Modi unterschiedlich.

Bei Namen mit externer C-Linkage wird keine interne Namenscodierung durchgeführt. C-Linkage kann benutzt werden, um Funktionen zu verknüpfen und aufzurufen, die in C oder in einer Sprache geschrieben sind, die sich an ihrer Namenscodierungs-Schnittstelle wie C verhält.

Ein Zeiger auf eine C-Funktion ist kompatibel mit einem Zeiger auf eine C++-Funktion desselben Typs, wenn die Argumenttypen C-kompatible PODs sind. In C++ 2017 sind dies zwei verschiedene Typen. Im erweiterten C++ 2017-Modus gibt es eine implizite Konversion zwischen zwei Funktionstypen, die sich nur in der Linkage unterscheiden.

Linkage von Templates

Es werden nur Templates mit C++-Linkage unterstützt.

Template-Instanziierung

siehe "Template-Instanziierung"

Referenztypen

Die Referenz für ein Rvalue ist an ein vom Compiler erzeugtes, temporäres Objekt gebunden. Intern wird der C++-Datentyp Referenz behandelt wie ein Zeiger (siehe "Implementierungsabhängiges Verhalten gemäß dem ANSI-/ISO-C-Standard").

Allokierung von nicht-statischen Datenelementen einer Klasse

Der Speicher für die Datenelemente wird in der strikten Reihenfolge ihrer Deklarationen allokiert, d.h. unabhängig von den Zugriffs-Spezifizierern `public`, `private` oder `protected`.

Bitfelder innerhalb von Klassen

Bitfelder innerhalb von Klassen werden behandelt wie Bitfelder innerhalb von Strukturen. Dies betrifft die Allokierung, die Ausrichtung sowie die Behandlung von „einfachen“ Bitfeldern ohne das Attribut `signed` bzw. `unsigned`.

Konstruktoren und Destruktoren für globale und lokale statische Objekte

C++ unterstützt die Initialisierung und Finalisierung von Objekten mit Konstruktoren und Destruktoren. Konstruktor- und Destruktoraufrufe können auf dynamische sowie auf globale und lokale statische Objekte angewendet werden.

Konstruktoren und Destruktoren für dynamische Objekte

Konstruktoren für dynamische Objekte werden aufgerufen, wenn das Objekt mit dem `new`-Operator erzeugt wird, beim Eintritt in Funktionen oder lokale Blöcke, beim Aufbau von Aktualparametern oder wenn temporäre Objekte erzeugt werden.

Destruktoren für dynamische Objekte werden in umgekehrter Reihenfolge ihrer Konstruktion aufgerufen, z.B. bei Funktionsende (`return`, `exit`), Blockende usw.

Werden für eine Expression temporäre Objekte angelegt, so werden die Destruktoren dafür am Ende der Expression aufgerufen. Dies gilt insbesondere auch für Parameter von Funktionen.

Konstruktoren und Destruktoren für globale und lokale statische Objekte

Die Art und Reihenfolge der Konstruktor- und Destruktoraufrufe für globale und lokale statische Objekte ist implementierungsabhängig. Technisch gesehen bedeutet dies: Die Konstruktoraufrufe werden pro Übersetzungseinheit in eine Funktion gebündelt. Die Adresse dieser Funktionen wird in einer speziell markierten Variablen im Datenteil des Programms abgelegt. Bei der Initialisierung des Laufzeitsystems wird eine Liste dieser Variablen erstellt. Vor Aufruf der `main`-Funktion werden die Konstruktoren aufgerufen. Der Destruktoraufruf erfolgt bei jeder normalen Programmbeendigung, d.h. bei Aufruf von `exit`, `_exit`, `bs2exit` oder `return` aus der `main`-Funktion, nicht jedoch bei Aufruf von `abort`.

Auf der Benutzerseite sind folgende Punkte zu beachten, wenn globale und lokale statische Objekte mit Konstruktoren und Destruktoren verwendet werden:

- Das Umlenken der Standard-Ein-/Ausgabe-Dateien (siehe "Umlenken der Standard-Ein-/Ausgabedateien") ist innerhalb der globalen und lokalen statischen Objekte nicht wirksam.
- Die Reihenfolge der Konstruktoraufrufe ist absolut undefiniert. Sie kann sogar bei Ablauf des gleichen Programms unterschiedlich sein. Es ist streng darauf zu achten, dass keine Abhängigkeiten zwischen den einzelnen Konstruktoraufrufen existieren.
- Die Namen der speziellen Variablen für die Konstruktor-Funktionsadressen (s.o.) beginnen mit ICP. Dieses Präfix sowie generell der Anfangs-Buchstabe `I` darf nicht für die Bildung von benutzereigenen externen Namen verwendet werden, da sonst die Konstruktorbehandlung fehlerhaft arbeitet.
- Beim Binden darf die ESD-Information nicht unterdrückt werden.
- Die Daten des Programms müssen im Klasse-6-Speicher stehen.
- Bei gemeinsam benutzbarem Code werden alle Code-Teile nachgeladen, um die Konstruktoren aufzurufen.
- Innerhalb eines Destruktors darf keine Sprachverknüpfung stattfinden (nur Fremdsprachenverknüpfung).
- Es darf kein Code „entladen“ werden, da sonst bei Programmende Destruktoren aufgerufen würden, für die kein Code mehr vorhanden ist.
- Bei echten Subsystemen, für die Daten und Code durch einen expliziten Aufruf nachgeladen werden, muss das Laufzeitsystem erneut initialisiert werden. Dabei wird die Tabelle der Konstruktoren ergänzt, und die neuen Konstruktoren werden aufgerufen.

Ausnahmebehandlung

Ausnahmen auslösen (throw)

Der Speicherplatz für Ausnahmeobjekte wird aus einem vorallokierten Speicher genommen, der ggf. durch `malloc`-Aufrufe erweitert wird.

Ausnahmen behandeln

In manchen Situationen wird implizit die Funktion `terminate()` aufgerufen. Der C++-Standard lässt in zwei Situationen die Frage offen, ob Destruktoren für automatische Objekte aufgerufen werden:

Wenn kein passender Ausnahme-Handler gefunden wird, werden die Destruktoren nicht aufgerufen.

Wenn auf dem Stack eine Funktion mit der Angabe `noexcept(true)` steht, wird nur ein Teil der Destruktoren aufgerufen.

Diese Destruktoren könnten mit der Funktion `unwind_exit()`, (siehe "Zusätzliche Laufzeitfunktionen") ausgeführt werden.

Funktion unexpected()

Für das ausgelöste Ausnahmeobjekt, das die Ursache für eine abgebrochene `unexpected()`-Funktion ist, wird ein Destruktor aufgerufen. Es wird durch ein neues `bad_exception`-Objekt ersetzt.

Weitere Einzelheiten zur Ausnahmebehandlung siehe "Ausnahmebehandlung".

Rückgabetyp von `operator->`

Der Rückgabetyp von `operator->()` wird nicht mehr an dem Punkt geprüft, an dem die Funktion deklariert wird, sondern ausschließlich an dem Punkt, an dem sie als „->“-Operator verwendet wird. (Diese Prüfung wurde bis Version V3.0C nur für Elemente von Klassen-Templates spät durchgeführt.)

Klassen und Ellipsis

Funktionen können mit variablen Argumentlisten deklariert werden (z.B. `int foo(int, ...);`). Ein Objekt einer Klasse kann ein Parameter sein, wenn das passende Argument die Ellipsis ist. In diesem Fall wird das Objekt Byte-weise kopiert. Ein evtl. vorhandener Konstruktor wird dabei ignoriert. Ein evtl. vorhandener Destruktur wird für den Parameter nicht aufgerufen.

7.3 Template-Instanziierung

In diesem Abschnitt werden folgende Themen behandelt:

- Grundlegende Aspekte
- Automatische Instanziierung
- Generieren von expliziten Template-Instanziierungsanweisungen (ETR-Dateien)
- Implizites Inkludieren
- extern-inline Funktionen

7.3.1 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.
- Gemäß des C++ Standard ist es 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. Da der Compiler beim Übersetzen einer Referenz auf eine Template-Einheit im C++V3 Modus 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 zur Verfügung:

- Die Optionen zur Auswahl der Instanziierungsmodi lauten MODIFY-SOURCE-PROP INSTANTIATION=*NONE / *AUTO / *LOCAL / *ALL (siehe "MODIFY-SOURCE-PROPERTIES").
- Die Instanziierung einzelner Templates oder auch einer Gruppe von Templates kann mit den Pragmas `instantiate`, `do_not_instantiate` und `can_instantiate` gesteuert werden (siehe "Pragmas zur Steuerung der Template-Instanziierung").

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 IMPLICIT-INCLUDE=*NO), wenn Templates aus der C++ V3-Bibliothek (SYSLNK.CRTE.STDCPP) benutzt werden, da in diesem Fall Definitionen nicht gefunden werden.

Instanziierungsmodi != INSTANTIATION=*AUTO: Hier besteht die Gefahr, dass unbefriedigte Externverweise (*NONE), Duplikate (*ALL) oder ggf. Ablauffehler (*LOCAL) auftreten können.

7.3.2 Automatische Instanziierung

Der Compiler unterstützt als Voreinstellung in den Sprachmodi C++ V3 und C++ 2017 die automatische Instanziierung (INSTANTIATION=*AUTO). Dadurch können Sie Quellcode übersetzen und die erzeugten Module 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 Standard-Suffix für C++-Quelldateien (`.C`, `.CPP`, `.CXX` oder `.CC`). 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 mit der COMPILE-Anweisung übersetzt werden, werden noch keine Template-Einheiten instanziiert. Für jede Quelldatei, die ein Template benutzt, wird, falls noch nicht vorhanden, eine Instanziierungs-Informationsdatei erzeugt. Diese Instanziierungs-Informationsdatei wird in die PLAM-Bibliothek des zu erzeugenden Moduls abgelegt. Der Name entspricht dem um das Suffix `.II` erweiterten Namen des Moduls (siehe Abschnitt „Bildung von Modulnamen“). Bei der Übersetzung eines Quellprogramms `ABC.CC` (ohne explizite Angabe eines Modulnamens) würde z.B. die Datei `ABC.II` erzeugt werden. Die Instanziierungs-Informationsdatei darf vom Benutzer nicht verändert werden.
2. Module erzeugen
   Die erzeugten Module enthalten Informationen darüber, welche Instanzen bei der Übersetzung einer Quelldatei erzeugt werden könnten und ggf. benötigt werden.
3. Template-Instanziierungen zuweisen
   Wenn die Module mit der BIND-Anweisung gebunden werden, wird vor dem eigentlichen Binden der Prälinker aufgerufen. Dieser durchsucht die Module 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 einem Modul, in dem angegeben ist, dass es die Template-Einheit instanziieren könnte. Wenn er ein solches Modul findet, weist er die Instanziierung diesem Modul zu.

4. Instanziierungs-Informationsdatei aktualisieren
   Für alle Instanziierungen, die einem Modul 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 Module erzeugen
   Wenn der Compiler eine Datei übersetzt, liest er die Instanziierungs-Informationsdatei für diese Übersetzungseinheit und erzeugt ein neues Modul 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 Module 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).
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 mit der COMPILE-Anweisung übersetzt werden, werden noch keine Template-Einheiten instanziiert. Für jede Quelldatei, die ein Template benutzt, wird, falls noch nicht vorhanden, eine Instanziierungs-Informationsdatei erzeugt. Diese Instanziierungs-Informationsdatei wird in die PLAM-Bibliothek des zu erzeugenden Moduls abgelegt. Der Name entspricht dem um das Suffix `.II` erweiterten Namen des Moduls (siehe Abschnitt „Bildung von Modulnamen“). Bei der Übersetzung eines Quellprogramms `ABC.CC` (ohne explizite Angabe eines Modulnamens) würde z.B. die Datei `ABC.II` erzeugt werden. Die Instanziierungs-Informationsdatei darf vom Benutzer nicht verändert werden.
2. Module erzeugen
   Die erzeugten Module 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 Module 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 Modulen 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 Module erzeugen
   Wenn der Compiler eine Datei nachübersetzt, liest er die Instanziierungs-Informationsdatei für diese Übersetzungseinheit und erzeugt ein neues Modul 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 und 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 Module 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 Modulen 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.

Zusammenspiel der Übersetzungs- und Prälinkerläufe

Da bei der automatischen Template-Instanziierung durch den Prälinker u.a. Nachübersetzungen stattfinden, spielen die Bedingungen, unter denen die Übersetzung (COMPILE) stattfindet, auch beim Prälinkerlauf (BIND ACTION=*PRELINK,...) eine wichtige Rolle. Technisch gesehen werden alle für die Übersetzung, Code-Generierung und Compilerausgaben relevanten Optionen in die entsprechenden Instanziierungs-Informationsdateien eingetragen und bei der automatischen Template-Instanziierung ausgewertet. Damit die Nachübersetzungen und andere Aktualisierungen beim Instanziieren funktionieren, sind insbesondere folgende Punkte zu beachten:

- Quellprogrammeingaben über SYSDTA sowie Listenausgaben auf SYSLST und SYSOUT werden nicht unterstützt und mit einer entsprechenden Fehlermeldung abgewiesen.

- Sämtliche Ein-/Ausgabe-Dateien und -Bibliotheken des Übersetzungslaufs müssen auch für den Prälinkerlauf auffindbar und zugreifbar sein und dürfen demzufolge weder namentlich geändert noch gelöscht werden. Wenn der Prälinkerlauf in einer anderen Umgebung (z.B. Benutzerkennung) stattfindet, müssen bei der Übersetzung vollqualifizierte Dateinamen verwendet werden, d.h. bei BS2000-Dateien und PLAM-Bibliotheken Namen inklusive <cat-id> und <user-id>, bei POSIX-Dateien absolute Pfadnamen (beginnend mit /).
- Die Option *INCREMENT ist nicht erlaubt für alle folgenden Ausgabedateien:
  - module element name
  - cif element name
  - listing element name

  Achtung: Die Kompilierung wird in diesem Fall abgebrochen.
- Wenn man einen expliziten Versionsstring für ’module name’ eingibt, wird eine ii-Element-Datei mit demselben Versionsstring gesucht, gelesen und geschrieben. (Sofern ein existierendes ii-Element mit einem anderen Versionsstring benutzt werden soll, muss dieses Element kopiert und unter neuem Namen abgelegt werden, bevor es gebunden wird bzw. die Template-Instanziierung beginnt.) Dieses Verhalten ist inkompatibel zum C++-Compiler V3.0.

Vorbinden und dynamisches Binden

Der Prälinker führt die automatische Instanziierung nur in den vom Compiler generierten Einzelmodulen durch, da er hierzu die eindeutige Zuordnung von Modul und Instanziierungs-Informationsdatei (`.II`-Datei) benötigt. Der Prälinker wird ausschließlich mit der BIND-Anweisung des Compilers aktiviert (ACTION=*PRELINK). Wenn in der BIND-Anweisung nur ACTION=*MODULE-GENERATION angegeben wird, wird keine Template-Instanziierung durchgeführt.

Mit der BIND-Anweisung vorgebundene „Großmodule“ oder dynamisch zu bindende Module, die Instanzen von Template-Einheiten benötigen, müssen beim Erstellen des fertigen Programms (durch statisches oder dynamisches Binden)

- entweder diese Instanzen bereits enthalten; dies kann durch explizite Instanziierung und/oder das Vorinstanziieren der Module mit der BIND-Anweisung (ACTION=*PRELINK) erreicht werden
- oder entsprechende Include-Dateien mit `can_instantiate`-Pragmas bereitstellen.

Beim Vorinstanziieren mit der BIND-Anweisung ist unbedingt darauf zu achten, dass die C++-Bibliotheken und C++-Laufzeitsysteme des CRTE (Standard-C++-Bibliothek, C++-Laufzeitsystems, Tools.h++-Bibliothek) vom Prälinker mitbetrachtet werden, da sonst Duplikat-Definitionen entstehen können. Die Bibliotheken werden automatisch berücksichtigt, wenn in der MODIFY-BIND-PROPERTIES-Anweisung die STDLIB-Option auf *DYNAMIC oder *STATIC gesetzt ist und die TOOLSLIB-Option auf *YES. Wenn die CRTE-Bibliotheken selbst dynamisch gebunden werden sollen, können Duplikat-Definitionen folgendermaßen vermieden werden:

1. In einem ersten Schritt die Module unter Berücksichtigung der verwendetet CRTE-Bibliotheken vorinstanziieren

   ```
   //MODIFY-BIND-PROPERTIES STDLIB=*DYNAMIC / *STATIC [,TOOLSLIB=*YES], ...
   //BIND ACTION=*PRELINK, ...
   ```

2. In einem zweiten Schritt die Module mit offenen Externverweisen auf die verwendeten CRTE-Bibliotheken binden

```
//MODIFY-BIND-PROPERTIES STDLIB=*NONE [,TOOLSLIB=*NO], ...
//BIND ACTION=*MODULE-GENERATION, ...
```

Wenn der Prälinker- und Bindelauf nicht getrennt erfolgen sollen, muss die ADD-PRELINK-FILES-Option verwendet werden. Diese Methode hat den Nachteil, dass die Namen der CRTE-Bibliotheken dann explizit angegeben werden müssen.
Bei einer Standardinstallation des CRTE und im C++ V3-Modus z.B. mit:

```
//MODIFY-BIND-PROPERTIES ADD-PRELINK-FILES=(*LIB(LIB=$.SYSLNK.CRTE.STDCPP),-
  *LIB(LIB=$.SYSLNK.CRTE.RTSCPP)   [,*LIB(LIB=$.SYSLNK.CRTE.TOOLS)] ),-
  STDLIB=*NONE, ...
//BIND ACTION=(*PRELINK,*MODULE-GENERATION), ...
```

7.3.3 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 werden mit dem Operanden GENERATE-ETR-FILE der SDF-Anweisung MODIFY-DIAGNOSTIC-PROPERTIES (siehe "MODIFY-DIAGNOSTIC-PROPERTIES") festgelegt. Wesentlich sind die Angaben ***ALL**..., bei der alle relevanten Informationen, und ***ASSIGNED**..., bei der nur ein Teil dieser Informationen ausgegeben wird.

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 vier logische Zeilen erzeugt (vgl. Beispiel):
  - eine Kommentarzeile mit dem Text ’The following template was’
  - eine Kommentarzeile, die die Art der Instanz gemäß der obigen Klassifizierung angibt (z.B ’explicitly instantiated’)
  - eine Kommentarzeile mit dem externen Namen der Instanz. Dieser Name entspricht dem Eintrag in der ii-Datei (siehe Abschnitt „Das Tool II-UPDATE“) und kann auch dem Binder-Listing bzw. der Binder-Fehlerliste entnommen werden
  - eine Zeile, die die explizite Instanziierung für diese Instanz beschreibt

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 vier 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 INSTANTIATION=*AUTO der SDF-Anweisung MODIFY-SOURCE-PROPERTIES und der Option GENERATE-ETR-FILE=*ASSIGNED der SDF-Anweisung MODIFY-DIAGNOSTIC-PROPERTIES aufgerufen.
   Die in den ETR-Dateien ausgegebenen Instanziierungs-Anweisungen werden in die zugehörigen Sourcen mit aufgenommen. Der Produktivbetrieb wird dann mit der Option INSTANTIATION=*NONE bzw. *AUTO in der SDF-Anweisung MODIFY-SOURCE-PROPERTIES 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 INSTANTIATION=*NONE und der Option GENERATE-ETR-FILE=*ALL-INSTANTIATIONS (SDF-Anweisungen wie oben) 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. 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 zweier Dateien **x.c** und **y.c** entsteht:

Beim Übersetzen wurden folgende Anweisungen verwendet:

//MOD-DIAG GEN-ETR=ALL
//COMPILE (*LIB(OLIB,X.C), *LIB(OLIB,Y.C)), MODULE-OUTPUT=*SOURCE-LOCATION

Source x.c:

```
template <class T> void f(T) {}
template <class T> void g(T);
template void f(long);
void foo()
{
         f(5);
         f('a');
         g(5);
}
```

Source y.c:

```
template <class T> void f(T) {}
void bar()
{
         f(5);
}
```

ETR-File x.etr (steht in Bibliothek OLIB als Element vom Typ S):

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

// The following template was
// explicitly instantiated
// __1f__tm__2_l__FZ1Z_v&_
template void f(long);

// The following template was
// used in this module and can be instantiated here
// __1f__tm__2_i__FZ1Z_v&_
template void f(int);

// The following template was
// used in this module and can be instantiated here
// __1f__tm__2_c__FZ1Z_v&_
template void f(char);

// The following template was
// used in this module
// __1g__tm__2_i__FZ1Z_v&_
template void g(int);
```

ETR-File y.etr (steht in Bibliothek OLIB als Element vom Typ S):

```
// 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
// __1f__tm__2_i__FZ1Z_v&_
template void f(int);
```

Der Benutzer kann nun entscheiden, in welcher Source er explizite Instanziierungen vornimmt (diese Entscheidung muss immer getroffen werden für Einträge mit „used in this module and can be instantiated here“) z.B. Eintrag von `template void f(int)` und `template void f(char)` in `x.c` (siehe Source in Beispiel 2).
Danach muss nicht mehr mit der automatischen Template-Instanziierung weitergearbeitet werden.

Beispiel 2

Gegeben seien zwei Dateien **x.c** und **y.c** in der Bibliothek **test**.

Source x.c

```
template <class T> void f(T){}
template <class T> void g(T);
template void f(long);
void foo()
{
        f(5);
        f('a');
        g(5);
}
```

Source y.c:

```
template <class T> void f(T) {}
void bar()
{
        f(5);
}
```

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

```
//mod-source-prop instantiation=*auto
//mod-diagnostic-prop generate-etr-file=*assigned-instantiations
//compile *lib-elem(test, X.C), module-output=*source-location
//compile *lib-elem(test, Y.C), module-output=*source-location
//mod-bind-prop start-llm-creation=*yes
//mod-bind-prop include=*lib-elem(test,X)
//mod-bind-prop include=*lib-elem(test,Y)
//bind action=*prelink
```

Danach existiert in der Bibliothek **test** folgende Datei **x.etr**:

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

// The following template was
// instantiated automatically by the compiler
// __1f__tm__2_i__FZ1Z_v&_
template void f(int);

// The following template was
// instantiated automatically by the compiler
// __1f__tm__2_c__FZ1Z_v&_
template void f(char);
```

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);
}
template void f(int);
template void f(char);
```

Im Anschluss kann die Produktion mit folgenden Kommandos durchgeführt werden:

```
//mod-source-prop instantiation=*none
//mod-diagnostic-prop generate-etr-file=*no
//compile *lib-elem(test,X1.C), module-output=*source-location
//compile *lib-elem(test,Y.C), module-output=*source-location
```

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:

```
//mod-source-prop instantiation=*auto
//mod-diagnostic-prop generate-etr-file=*no
//compile *lib-elem(test,Y.C), module-output=*source-location
//mod-bind-prop start-llm-creation=*yes
//mod-bind-prop include=*lib-elem(test,y)
//bind action=*prelink (dadurch wird f(int) y zugeordnet)
//mod-diagnostic-prop generate-etr-file=*all-instantiations
//compile *lib-elem(test,X.C), module-output=*source-location
//mod-bind-prop start-llm-creation=*yes
//mod-bind-prop include=*lib-elem(test,X)
//mod-bind-prop include=*lib-elem(test,Y)
//bind action=*prelink
```

Daraufhin entsteht in der Bibliothek **test** folgende ETR-Datei **x.etr**:

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

// The following template was
// explicitly instantiated
// __1f__tm__2_l__FZ1Z_v&_
template void f(long);

// The following template was
// used in this module and can be instantiated here
// __1f__tm__2_i__FZ1Z_v&_
template void f(int);

// The following template was
// instantiated automatically by the compiler
// __1f__tm__2_c__FZ1Z_v&_
template void f(char);

// The following template was
// used in this module
// __1g__tm__2_i__FZ1Z_v&_
template void g(int);
```

7.3.4 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 IMPLICIT-INCLUDE, "MODIFY-SOURCE-PROPERTIES") und kann mit IMPLICIT-INCLUDE=*NO 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 einem Include-Element *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 enthalten ist,

- die als Quellprogramm-Element in derselben PLAM-Bibliothek steht wie das Include-Element
- und deren Name aus dem Basisnamen des Include-Elements und einem Standard-Suffix für C++-Quelldateien `.C`, `.CPP`, `.CXX` oder `.CC` gebildet ist (z.B. *basisname*`.CC`).

Es sei beispielsweise in dem Include-Element `XYZ.H` in der Bibliothek `PLAM.T` eine Template-Einheit `ABC::f` 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 der Bibliothek `PLAM.T` nach einer Quelldatei mit dem Basisnamen `XYZ` und einem Standard-Suffix für C++-Quelldateien `.C`, `.CPP`, `.CXX` oder `.CC` (z.B. `XYZ.CC`). Wenn sie existiert, wird sie so behandelt, als ob sie am Ende der Quelldatei inkludiert wäre.

7.3.5 extern-inline Funktionen

Im Sprachmodus C++ 2017 unterstützt der Compiler *extern-inline* Funktionen (siehe auch „extern inline vs. static inline"). Diese haben eine ähnliche Problematik wie die Templates: Es muss eine eindeutige Adresse geben. Der Compiler muss die Funktion also einem Modul zuordnen.

Die extern-inline Funktionen werden ähnlich behandelt wie Template-Einheiten. Für sie werden Einträge in der Instanziierungs-Informationsdatei angelegt. Die Automatische Instanziierung weist die Funktion einem Modul zu.

Das hat zur Folge, dass die Template-Instanziierung auch ablaufen muss, wenn keine Templates verwendet werden.

Die Möglichkeit der Inline-Expansion als Optimierung bleibt bestehen (siehe „Inline-Generierung von benutzereigenen Funktionen“ (Verlauf der Optimierung)). Diese Optimierung ist in allen Modulen möglich.

Gegenüber normalen Template-Einheiten ergeben sich die folgenden Unterschiede:

- Die Pragmas `instantiate`, `do_not_instantiate` und `can_instantiate` können nicht für extern-inline Funktionen genutzt werden.
- Das implizite Inkludieren ist nicht relevant. Alle benutzten Typen müssen deklariert worden sein, wie bei nicht-inline Funktionen auch.
- Die syntaktische und semantische Prüfung erfolgt in jedem Modul, in dem die extern-inline Funktion definiert ist.
- Es gibt zur Zeit (Version 4.0A10) keine Möglichkeit, das Modul explizit auszuwählen, dem die Funktion zugeordnet wird.
- Zur Zeit (Version 4.0A10) werden extern-inline Funktionen nicht in den ETR-Dateien berücksichtigt.

7.4 Abweichungen zu ANSI-/ISO-C++

In diesem Abschnitt werden folgende Themen behandelt:

- Erweiterungen gegenüber ANSI-/ISO-C++
- Einschränkungen gegenüber ANSI-/ISO-C++
- extern inline vs. static inline

7.4.1 Erweiterungen gegenüber ANSI-/ISO-C++

Die im Folgenden beschriebenen Erweiterungen werden im C++ 2017-Sprachmodus akzeptiert, im strikten C++ 2017-Sprachmodus allerdings nur, wenn die Option `-R strict_errors` bzw. ANSI-VIOLATIONS=*ERROR nicht gesetzt ist. Darüber hinaus werden in den C++-Modi auch alle Erweiterungen gegenüber ANSI-/ISO-C unterstützt (siehe Abschnitt „Erweiterungen gegenüber ANSI-/ISO-C“).

- In der Deklaration eines Klassenelements kann ein qualifizierter Name benutzt werden:

```
struct A {
   int A::f(); // laut ANSI int f();
};
```

- Im erweiterten C++ 2017 Modus gibt es eine implizite Konversion zwischen Funktionen mit "C" und "C++" Linkage:

```
extern "C" void f();
void (*pf)() = &f;
```

- Im erweiterten C++ 2017 Modus gibt es eine implizite Konversion von einem String-Literal zu dem Typ `char*` (In C++ 2017 haben String Literale den Typ `const char *`):

```
char *p1 = "abc";
char *p2 = x ? "abc" : "def";
```

- Das Präprozessormakro `c_plusplus` wird zusätzlich zum standardkonformen Makro `__cplusplus` definiert. Es wird keine Warnung ausgegeben.

7.4.2 Einschränkungen gegenüber ANSI-/ISO-C++

Einige Sprachmittel werden vom hier beschriebenen Compiler nicht unterstützt. Neben den hier aufgeführten Einschränkungen gelten die Einschränkungen gegenüber ANSI-/ISO-C bezüglich `long long`, Ausrichtung, Unicode und dem Float-Format IEEE auch in C++.

Unterstützung von Threads

Der C/C++ Compiler unterstützt kein bestimmtes Thread-Paket. Im Detail heisst dies, dass die Header <thread>, <mutex>, <shared_mutex>, <condition_variable> und <future> nicht verfügbar sind. Das Schlüsselwort `thread_local` führt im Modus C++2017 zu einer Fehlermeldung. Das Prädefine `__STDCPP_THREADS__` ist nicht gesetzt.

Atomics

Die atomaren Typen des C++-Standard werden nicht unterstützt. Sie sind zum Teil mit dem Thread-Paket verbunden. Das bedeutet, dass die dafür vorgesehenen Schlüsselwörter mit einer Fehlermeldung abgewiesen werden. Der Header <atomic> ist nicht verfügbar.
Der Compiler enthält eine rudimentäre Unterstützung für Atomarität. Details sind auf besondere Anfrage verfügbar.

Headers <execution> and <memory_resource>

Diese beiden Header werden nicht unterstützt. Die vom Standard dort vorgesehenen Klassen und Funktionen werden nicht angeboten.

Besondere mathematische Funktionen

Der C++-Standard von 2017 beschreibt in Kapitel 29.9.5 ein paar besondere mathematische Funktionen, zum Beispiel Laguerre und Legendre. Diese Funktionen werden nicht angeboten.

7.4.3 extern inline vs. static inline

Eine inline-Funktion kann als *static-inline* oder *extern-inline* interpretiert werden. Diese Frage wird wichtig, wenn in der Funktion lokale static-Variablen verwendet werden oder wenn die Adresse einer solchen Funktion in einem Vergleich genutzt wird.
Die *Beispiele 1* und *2* am Ende dieses Abschnitts zeigen das Problem.

Eine static-inline-Funktion verhält sich aus Sicht des Programmierers genau wie eine static-Funktion. Die Funktion selbst und alle in der Funktion deklarierten Elemente sind lokal zu der aktuellen Übersetzungseinheit. Die Funktion selbst hat keine Interaktion mit Elementen anderer Übersetzungseinheiten; solche müssen von anderen (externen) Funktionen hergestellt werden.

Eine extern-inline-Funktion verhält sich wie eine normale, externe Funktion. Der Rumpf der Funktion muss in jeder benutzenden Übersetzungseinheit auftreten. Dieser muss aber jeweils identisch sein. Das Verhalten lässt sich mit einem Vergleich beschreiben: Es ist so, als ob der Rumpf als einzige Definition in einer zusätzlichen Übersetzungseinheit steht und in allen anderen Übersetzungseinheiten diese Funktion aufgerufen wird.

Zusätzlich zu obiger Definition gibt es den Hinweis an den Optimierer, dass eine Inline-Expansion wünschenswert ist. Diese Optimierung muss aber unter Beibehaltung der Bedeutung erfolgen.

Die Sicht des C++- Standard

Wird eine Funktion mit dem Schlüsselwort `inline` deklariert, so gilt sie implizit als extern-inline. Eine static-inline muss mit den Schlüsselwörtern `static` und `inline` deklariert werden.
Eine Member-Funktion, die innerhalb einer Klasse definiert ist, gilt implizit als extern-inline.

Die Implementierung im Sprachmodus C++ 2017

Der Sprachmodus C++ 2017 unterstützt extern-inline wie vom Standard verlangt. Siehe hierzu "extern-inline Funktionen".

Die Implementierung in den Sprachmodi Cfront-C++ und C++ V3

In den Sprachmodi Cfront-C++ und C++ V3 kennt der Compiler das Konzept einer extern-inline-Funktion nicht. Alle inline-Funktionen werden von ihm als static-inline interpretiert.
Dies trifft sowohl die mit dem Schlüsselwort `inline` deklarierten Funktionen als auch die innerhalb einer Klasse definierten Member-Funktionen.

Die folgenden Abschnitte gelten nur für die Sprachmodi Cfront-C++ und C++ V3.

Die problematischen Konstrukte

In vielen Fällen ist nicht relevant, ob eine Funktion als static-inline oder extern-inline interpretiert wird. Dazu muss eines von drei Konstrukten verwendet werden. Des Weiteren wird es nur problematisch, wenn die Funktion in mehreren Übersetzungseinheiten verwendet wird.

Betroffene Funktionen sind solche, die als extern-inline gedacht sind (siehe *Beispiel 3*). Diese Funktionen stehen fast immer in einem Header-File, komplett mit Rumpf. Es kann sich dabei um Templates oder um Member-Funktionen handeln.

Bei Templates ist zu beachten, dass der C/C++-Compiler weitere Sourcen inkludiert, wenn dies nicht abgeschaltet wurde. Siehe hierzu die Optionen //MODIFY-SOURCE-PROPERTIES IMPLICIT-INCLUDE = bzw. `-K implicit_include`, die im Default-Fall aktiv sind (siehe *Beispiel 4*).

- lokale static-Variablen

  Eine lokale `static`-Variable ist in einer solchen Situation in der Compiler-Implementierung mehrfach vorhanden, während der Standard eine einzige Kopie verlangt. Ob dies zu einem Problem führt, hängt davon ab, aus welchem Grund die Variable als lokale `static` definiert wurde (siehe *Beispiel 5*).
  Wenn tatsächlich eine einzige Kopie erwartet wird, gibt es ein Problem. Ob dies zu Ablauf-Fehlern führt, hängt stark von der realen Benutzung ab.
  Eine andere Situation liegt vor, wenn der Grund die Lebensdauer des Speichers ist. Es gibt viel Code, der intern einen String aufbaut, um ihn als Return-Wert zurückzugeben. Hier kann keine `auto`-Variable genutzt werden, da der Speicher beim Return freigegeben wird. Ein lokale `static` bleibt aber bis zum nächsten Aufruf der Funktion gültig. Dies wird häufig genutzt, wenn Texte für eine Ausgabe aufbereitet werden. In diesem Fall wird die Variable zwar verdoppelt, aber dies hat praktisch keine Auswirkung auf den Programmablauf.

- Vergleich von Adressen

  Eine extern-inline-Funktion hat eine eindeutige Adresse, eine static-inline jeweils eine Adresse pro Übersetzungseinheit. Dies ist nicht relevant, solange die Adressen nur zum Aufruf der Funktion genutzt werden. Kritisch wird es, wenn solche Adressen miteinander verglichen werden. Dies passiert aber eher selten. Denkbar wäre so etwas beim Anmelden von Callback-Funktionen, wo jede Funktion nur einmal eingetragen werden sollte.

- String-literal

  Dieses Konstrukt liefert wohl nur in Ausnahmefällen ein Problem. Der Standard legt nicht fest, ob String-Literale mit gleichem Inhalt auch die gleiche Adresse haben. Auch ist es sehr unüblich, die Adresse eines String-Literal in Vergleichen oder ähnlichem zu benutzen.

Beispiel 1: eine problematische Situation

```
// file bsp.h
       #include <stdio.h>
       class BSP {
       public:
          inline int mf1(void);
       };
       inline int BSP::mf1(void)
       {
          static int i = 1;
          return i++;
       }
       extern void f();
// file bsp1.c
       #include “bsp.h”
       void f()
       {
          BSP bsp;
          printf (“Wert 1: %d\n”, bsp.mf1() );
       }
// file bsp2.c
       #include “bsp.h”
       void g()
       {
          BSP bsp;
          printf (“Wert 2: %d\n”, bsp.mf1() );
       }
       int main()
       {
          f();
          g();
          return 0;
       }
```

Erläuterung

Kritisch ist die Funktion `mf1` und die darin enthaltene Variable `i`. Nach Standard sollte diese nur einmal existieren. Die beiden Aufrufe in `bsp1.c` und `bsp2.c` sollen also die gleiche Variable betreffen. Die gewünschte Ausgabe ist

```
Wert 1: 1

Wert 2: 2
```

In der Implementierung vom C/C++-Compiler wird die Funktion aber in `bsp1.c` und `bsp2.c` jeweils getrennt bearbeitet. Es wird beim Aufruf in `bsp2.c` eine andere Variable angesprochen als beim Aufruf in `bsp1.c`. Beide Ausprägungen sind mit 1 initialisiert.

Die Ausgabe ist dann

```
Wert 1: 1

Wert 2: 1
```

Beispiel 2: eine problematische Situation mit Templates

```
// file tmpl.h
       #include <stdio.h>
       template <class T>
       inline int tmpl(T t)
       {
          static int i = 1;
          return i++;
       }
       extern void f();
// file tmpl1.c
       #include “tmpl.h”
       void f()
       {
          printf (“Wert 1: %d\n”, tmpl(5) );
       }
// file tmpl2.c
       #include “tmpl.h”
       void g()
       {
          printf (“Wert 2: %d\n”, tmpl(7) );
       }
       int main()
       {
          f();
          g();
          return 0;
       }
```

Erläuterung

Es gelten die gleichen Hinweise wie für *Beispiel 1*. Nur handelt es sich hier um ein Template statt um eine Member-Funktion. Die vom Standard erwartete Ausgabe ist

```
Wert 1: 1

Wert 2: 2
```

Die vom C/C++-Compiler gelieferte Ausgabe ist

```
Wert 1: 1

Wert 2: 1
```

Beispiel 3: betroffene Funktionen

```
// file bf.h
       inline int f1(T) { }                              // betroffen
       int f2(T);                                        // nicht betroffen
       static inline int f3(T) { }                       // nicht betroffen,
                                                            da explizit static
       template <class T> inline int tf1(T) { }          // betroffen
       template <class T> int tf2(T);                    // nicht betroffen
       template <class T> static inline int tf3(T) { }   // nicht betroffen,
                                                            da explizit static
       class BSP {
       public:
          inline int mf1(void);                          // betroffen
          int mf2(void) { }                              // betroffen
          int mf3(void);                                 // nicht betroffen
       };
       inline int BSP::mf1(void) { }
```

Beispiel 4: Implizites Include

```
// file ii.h
       template <class T> inline int ii(T);
// file ii.c
       template <class T> inline int ii(T)    // diese Funktion ist betroffen,
                                                 obwohl sie in einer .c-Datei
                                                 steht
       {
          static int i;
          return i++;
       }
```

Beispiel 5: Problematische static Variablen

Hier wird nur die Nutzung der `static` als problematisch / nicht problematisch dargestellt. Um zu einem echten Problem zu werden, muss die Funktion eine problematische inline-Funktion sein.

```
inline void no_recursion (void)
{
       static int active = 0;          // problematisch
       active ++;
       if (active > 1)
       {
          illegal_recursion ();
       } else {
          do_something ();
       }
       active --;
}
```

Dies ist problematisch, da die Rekursion nicht zuverlässig erkannt wird.

```
inline char * debug_text ()
{
       static char buffer [200];       // nicht problematisch
       sprintf (buffer, "...", ...);
       return buffer;
}
```

Dies ist nicht problematisch, da der Inhalt des `buffer` immer vor dem nächsten Aufruf verwendet wurde. Es gibt normalerweise niemand, der die Adresse abspeichert und annimmt, dass sich ihr Inhalt zuverlässig ändert.

```
inline void f (void)
{
       static int actions = 0;         // (siehe unten)
       actions++;
       if (actions > 200)
       {
          actions = 0;
          optimize_datastructures ();  // Nur aufräumen, keine relevante
                                          Änderung
       }
       do_something ();
}
```

Ob dies problematisch wird, hängt stark von der Funktion `optimize_datastructures` ab. Wenn dort wirklich nur Optimierungen vorgenommen werden, läuft das Programm trotz der Verdoppelung von actions korrekt ab.
Die Performance ist jedoch anders, da die Aufrufe von `optimize_datastructures` nicht mehr regelmäßig sind.

7.5 Besonderheiten im Cfront-C++-Modus

Im Cfront-C++-Modus verhält sich der Compiler kompatibel zu Cfront-C++ V3.0, d.h. er unterstützt eine Reihe entsprechender Funktionsmerkmale und spezifischer Eigenschaften. Der Cfront-C++-Modus wird unterstützt, damit existierender Code, der Erweiterungen der Cfront-Version V3.0 enthält, ohne Eingriffe übersetzt werden kann. Eine vollständige Kompatibilität zum C++-Compiler V2.2 ist jedoch nicht gewährleistet.

Wenn ein Programm beim Übersetzen mit dem C++-Compiler V2.2 einen Fehler produziert hat, kann es sein, dass der C/C++-Compiler V4.0 im Cfront-C++-Modus einen anderen oder auch keinen Fehler produziert. Im Folgenden werden einige Sprachbesonderheiten beschrieben, die nur für den Cfront-C++-Modus gelten.

- `const`-Qualifikationen für den `this`-Parameter können in gewissen Kontexten ignoriert werden, z.B.

  ```
  struct A {
     void f() const;
  };
  void (A::*fp)() = &A::f;
  ```

  Dies ist tatsächlich eine sichere Operation. Einem Zeiger auf eine Funktion vom Typ `const` kann ein Zeiger auf einen Typ zugewiesen werden, der nicht `const` ist, da es in einem Aufruf, der den Zeiger benutzt, erlaubt ist, das Objekt zu verändern, und die Funktion, auf die gezeigt wird, das Objekt nicht verändert. Eine Zuweisung in die umgekehrte Richtung würde nicht sicher sein.
- Konversions-Operatoren, die nach `void` konvertieren sind erlaubt.
- Eine `friend`-Deklaration kann einen neuen Typ einführen. Eine `friend`-Deklaration, die einen ausführlichen Typspezifizierer weglässt, ist im C++-Modus erlaubt, aber im Cfront-Modus kann auch ein neuer Typname eingeführt werden.

  ```
  struct A {
     friend B;
  };
  ```

- Der dritte Operator des ?-Operators ist syntaktisch ein Bedingungsausdruck statt eines Zuweisungsausdrucks.
- Eine Referenz auf einen Zeigertyp kann von einem Zeigerwert initialisiert werden, ohne dass eine temporäre Variable benutzt werden muss. Dies ist sogar dann möglich, wenn der Referenz-Zeigertyp zusätzliche Typ-Qualifikationen zu denen im Zeigerwert hat.

  ```
  int *p;const int *&r = p; // No temporary used
  ```

- Eine Referenz-Variable kann mit 0 initialisiert werden.
- Da im Cfront-Modus die Zugreifbarkeit von Typen nicht überprüft wird, werden Typ-Zugriffsfehler als Warnung und nicht als Fehler ausgegeben.
- Beim Aufruf von überladenen Funktionen muss ein Nullzeiger als Zeichenkette in der Form "0" geschrieben werden. Andere Schreibweisen, z.B.`const`-Variablen mit dem Wert 0 oder Konstanten in der Form '\0' werden vom Compiler im Falle von überladenen Funktionen nicht als Nullzeiger interpretiert.
- Wenn eine `operator()()`-Funktion Default-Argument-Ausdrücke hat, wird keine Warnung ausgegeben.

- Für die Deklaration von Zeigervariablen auf Elementfunktionen wird eine alternative Form unterstützt. Diese wird im folgenden Beispiel verdeutlicht:

```
struct A {
   void f(int);
   static void sf(int);
   typedef void A::T3(int);    // nonstd  typedef decl
   typedef void T2(int);       // std typedef
};
typedef void A::T(int);        // nonstd typedef decl
T* pmf = &A::f;                // nonstd ptr-to-member decl
A::T2* pf = A::sf;             // std ptr to static mem decl
A::T3* pmf2 = &A::f;           // nonstd ptr-to-member decl
```

  In diesem Beispiel bezeichnet `T` einen Routinentyp für eine nicht statische Elementfunktion der Klasse `A`, der ein `int`-Argument übergeben wird und die `void` zurückliefert. Der Gebrauch solcher Typen ist beschränkt auf nicht standardkonforme Deklarationen von Zeigern auf Elementfunktionen. Die kombinierte Deklaration von `T` und `pmf` entspricht einer einzigen standardkonformen Deklaration der folgenden Form:

  `void (A::* pmf)(int) = &A::f;`

  Eine nicht standardkonforme Deklaration außerhalb einer Klassendeklaration, wie z.B. die Deklaration von `T`, ist normalerweise ungültig und würde einen Fehler erzeugen. Für Deklarationen wie `A::T3` innerhalb einer Klassendeklaration, ändert dieses Sprachmerkmal die Bedeutung einer gültigen Deklaration.

- `protected`-Klassenelemente werden nicht überprüft, wenn die Adresse eines `protected`-Elements angegeben wird.

```
class B { protected: int i; };
class D : public B {void mf(); };
void D::mf() {
   int B::* pmi1 = &B::i;      // Error except in Cfront mode
   int D::* pmi2 = &D::i;      // OK
}
```

  Beachten Sie, dass die Überprüfung von `protected`-Klassenelementen für andere Operationen - alle außer der Deklaration von Zeigeradressen auf Klassenelemente - im Cfront-Modus standardkonform behandelt wird.

- Der Destruktor einer abgeleiteten Klasse kann implizit den `private`-Konstruktor einer Basisklasse aufrufen.

```
class A {
   ~A();
};
class B : public A {
   ~B();
};
B::~B(){}          // Error except in Cfront mode
```

- Wenn es zur Vermeidung von Mehrdeutigkeiten notwendig ist, zu entscheiden, ob etwas eine Parameterdeklaration oder ein Argumentausdruck ist, wird die Zeichenfolge *Typname-oder-Schlüsselwort (Identifizierer...)* als Argument behandelt.

```
class A { A(); };
double d;
A x(int(d));
A(x2);
```

  Gemäß Standard wird `int(d)` als Parameterdeklaration (mit redundanten Klammern) behandelt; entsprechend ist `x` eine Funktion. Im Cfront-C++-Modus wird `int(d)` als Argument interpretiert und `x` als Variable.

  Die Deklaration `A(x2);` wird im Cfront-C++-Modus ebenfalls nicht standardkonform interpretiert. Gemäß Standard sollte die Anweisung als Deklaration des benannten Objektes `x2` interpretiert werden. Im Cfront-C++-Modus wird sie jedoch als `cast` von `x2` in den Typ `A` interpretiert.

  Auch die folgende Deklaration wird im Cfront-Modus abweichend vom Standard interpretiert:

  `int xyz(int());`

  Gemäß Standard wird hier die Funktion `xyz` deklariert, der als Parameter eine Funktion ohne Argument vom Typ `int` übergeben wird. Im Cfront-Modus wird diese Anweisung als Deklaration eines Objekts interpretiert, das mit dem Wert 0 initialisiert wird.

- Bitfelder

  Ein benanntes Bitfeld kann die Größe 0 haben. Die Deklaration wird behandelt, als ob kein Name deklariert worden wäre.

  Einfache Bitfelder, d.h. Bitfelder, die mit dem Typ `int` deklariert sind, sind immer `unsigned`.

- Der Name für einen Typspezifizierer kann ein `typedef`-Name sein, der synonym mit einem Klassennamen ist.

```
typedef class A T; class T *pa;
// No error in Cfront mode
```

- Wenn Datentyp-Spezifizierer (auch `signed` oder `unsigned`) verdoppelt werden, wird keine Warnung ausgegeben:

  `short short int i; // No warning in Cfront mode`

- Nach dem letzten Argument in einer Argumentliste ist ein zusätzliches Komma erlaubt:

  `f(1,2,);`

- Der cast eines konstanten Zeigers auf eine Elementfunktion in einen Zeiger auf eine Funktion ist möglich. Es wird nur eine Warnung ausgegeben:

```
struct A {int f();};
main() {
   int (*p)();
   p = (int (*)())A::f;    // OK, with warning
}
```

- Argumente von Klassentypen, die bitweises Kopieren erlauben und auch Destruktoren haben, werden als Wert übergeben (wie C-Strukturen) und der Destruktor wird für die „Kopie“ nicht aufgerufen. In den anderen C++-Modi wird das Klassenobjekt in ein temporäres Objekt kopiert, die Adresse des temporären Objekts wird als Argument übergeben und der Destruktor wird für das temporäre Objekt aufgerufen, nachdem der Aufruf zurückgekehrt ist. In der Praxis ist das kein großes Problem, da Klassen, die bitweises Kopieren erlauben, normalerweise keine Destruktoren haben.
- Wenn eine `typedef`-Deklaration eine unbenannte Klasse enthält, kann der `typedef`-Name als Klassenname verwendet werden.

```
typedef struct {int i, j; } S;
struct S x; // No error in Cfront mode
```

- Ein `typedef`-Name kann in einem expliziten Destruktoraufruf verwendet werden:

```
struct A { ~A(); };
typedef A B;
int main() {
   A *a;
   a->~B();            // Permitted in Cfront mode
}
```

- Zwei Elementfunktionen können mit denselben Parametertypen deklariert werden, wenn die eine statisch und die andere nicht statisch mit einem Funktionsqualifizierer deklariert wird.

```
class A {
   void f(int) const;
   static void f(int);    // no error in Cfront mode
};
```

- Zwei Funktionen, die den gleichen Namen und sehr ähnliche Parametertypen haben, können nicht gemeinsam benutzt werden. Dies ist der Fall, wenn sich die Parametertypen nur in den folgenden Aspekten unterscheiden:
  - char vs. signed char;

    ```
    f(char);
    f(signed char);
    ```

  - Array-Grenzen

    ```
    f(char(*x)[15]);
    f(char(*x)[18]);
    ```

  - Eine const-Qualifikation eines typedef

    ```
    typedef char c;
    f(const c*);
    f(c*);
    ```

  - Indirekte oder kombinierte Nutzung dieser Aspekte

    ```
    f(char*,int);
    f(signed char*,int);
    typedef char c;
    g(char,c*);
    g(signed char,const c*);
    ```

  Werden doch beide Funktionen definiert, so wird das als Duplikat gewertet. Es wird keine Meldung beim Übersetzen ausgegeben.

8 C++-Bibliotheken und C++-Laufzeitsystem

Mit CRTE werden folgende C++-Bibliotheken bereitgestellt:

- Eine C++-Bibliothek für den Sprachmodus C++ 2017
- Eine C++-Bibliothek für den Sprachmodus C++ V3
- Eine C++-Bibliothek für den Cfront-C++ Sprachmodus V2
- Eine C++-V3-Bibliothek Tools.h++ V7.0

8.1 Die C++-Bibliothek für den Sprachmodus C++ 2017

Diese C++-Bibliothek kann nur im C++ 2017-Modus des Compilers genutzt werden. Sie umfasst folgende Schnittstellen (basierend auf den Kapiteln des C++ Standards von 2017):

- Einige Hilfsklassen (Kapitel 23), u.a. Allokatoren, Functions-Objekte, Smart Pointer, Paare, Tupel und mehr,
- String-Klassen (Kapitel 24), insbesondere einen String von `char` Elementen, der mit EBCDIC gut umgehen kann,
- Klassen für die `locale`-Behandlung (Kapitel 25),
- Container Klassen (Kapitel 26), insbesondere bitset, deque, queue, vector, list, map, multimap, set,
- Iteratoren (Kapitel 27),
- generische Algorithem (Kapitel 28), u.a. suchen, sortieren, minimum, maximum, partitionieren, permutieren,
- numerische Klassen (Kapitel 29), u.a für komplexe Zahlen oder für Arrays von arithmetischen Werten,
- Ein- und Ausgabe (Kapitel 30), in dieser Library ist die Implementierung Template-basiert,
- reguläre Ausdrücke (Kapitel 31).

Die Klassen für atomare Operationen (Kapitel 32) und Threads (Kapitel 33) werden nicht angeboten.

Die Include-Dateien für die o.g. Schnittstellen sind in der Bibliothek SYSLIB.CRTE.CXX01 enthalten, die Module in der Bibliothek SYSLNK.CRTE.CXX01.

Alle Namen der C++-Bibliothek befinden sich im Namensraum `std`.

Ein weiterer Bestandteil der C++-Bibliothek sind die unten aufgelisteten Include-Dateien (ebenfalls in SYSLIB.CRTE.CXX01 enthalten), in denen alle ANSI-C-Bibliotheksfunktionen im Namensraum `std` definiert sind. Die Namen dieser Include-Dateien sind wie folgt aus den Namen der ANSI-C-Include-Dateien abgeleitet: Dem Namen wird der Buchstabe c vorangestellt, das Suffix `.h` entfällt.

`<cassert>`	`<cinttypes>`	`<csetjmp>`	`<cstddef>`	`<ctgmath>`
`<cctype>`	`<ciso646>`	`<csignal>`	`<cstdint>`	`<ctime>`
`<cerrno>`	`<climits>`	`<cstdalign>`	`<cstdio>`	`<cuchar>`
`<cfenv>`	`<clocale>`	`<cstdarg>`	`<cstdlib>`	`<cwchar>`
`<cfloat>`	`<cmath>`	`<cstdbool>`	`<cstring>`	`<cwctype>`

Benutzen der Bibliothek

Programmentwicklung in BS2000-Umgebung (SDF)

Beim Übersetzen muss die Bibliothek SYSLIB.CRTE.CXX01 nach Standard-Includes durchsucht werden. Dies ist sichergestellt, wenn in der MODIFY-INCLUDE-LIBRARY-Anweisung für die Suche nach den Standard-Includes die Option STD-INCLUDE-LIBRARY=*STANDARD-LIBRARY angegeben wird (Voreinstellung).

Das Programm muss mit der BIND-Anweisung des Compilers gebunden werden. In der MODIFY-BIND-PROPERTIES-Anweisung ist die (voreingestellte) Angabe RUNTIME-LANGUAGE =*CPLUSPLUS (MODE=*LATEST/*2017) erforderlich.

Für weitere Einzelheiten siehe "MODIFY-BIND-PROPERTIES".

Programmentwicklung in POSIX-Umgebung

Zum Einfügen der Include-Dateien und zum Binden der Module muss im CC-Kommando der C++ 2017-Modus angegeben werden (`-X 2017`). `-X 2017` ist beim CC-Kommando voreingestellt. Für weitere Einzelheiten siehe Handbuch „POSIX-Kommandos des C/C++-Compilers“ [1].

Dokumentation

Die Standard-C++-Bibliothek ist in der allgemeinen C++ Literatur beschrieben.

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

Diese C++-Bibliothek kann nur im C++ V3-Modus des Compilers genutzt werden. Sie umfasst folgende Schnittstellen:

- eine String-Klasse

  `<string>`
- Container-Klassen

  `<bitset>`
  `<deque>`
  `<list>`
  `<map>`
  `<queue>`
  `<set>`
  `<stack>`
  `<vector>`
- Iteratoren

  `<iterator>`
- Generische Algorithmen

  `<algorithm>`
- Numerische Klassen und Operationen

  `<complex>`

  `<numeric>`
- Ein-/Ausgabe-Klassen

  `<iostream.h>`
  `<fstream.h>`
  `<strstream.h>`
  `<stdiostream.h>`
  `<iomanip.h>`

  Die Ein-/Ausgabe-Klassen sind nicht Standard-konform und entsprechen der zu Cfront V3.0.3 kompatiblen Ein-/Ausgabe-Bibliothek iostream.

Die Include-Dateien für die o.g. Schnittstellen sind in der Bibliothek SYSLIB.CRTE enthalten, die Module in der Bibliothek SYSLNK.CRTE.STDCPP.

Mit Ausnahme der Ein-/Ausgabe-Klassen befinden sich sonst alle Namen der C++-Bibliothek im Namensraum `std`.

Ein weiterer Bestandteil der C++-Bibliothek sind die unten aufgelisteten Include-Dateien (ebenfalls in SYSLIB.CRTE enthalten), in denen alle ANSI-C-Bibliotheksfunktionen im Namensraum `std` definiert sind. Die Namen dieser Include-Dateien sind wie folgt aus den Namen der entsprechenden C-Include-Dateien abgeleitet: Dem Namen wird der Buchstabe c vorangestellt, das Suffix `.h` entfällt.

`<cassert>`	`<ciso646>`	`<csetjmp>`	`<cstdio>`	`<ctime>`
`<cctype>`	`<climits>`	`<csignal>`	`<cstdlib>`	`<cwchar>`
`<cerrno>`	`<clocale>`	`<cstdarg>`	`<cstring>`	`<cwctype>`

<cfloat>	<cmath>	<cstddef>		

Benutzen der Bibliothek

Programmentwicklung in BS2000-Umgebung (SDF)

Beim Übersetzen muss die Bibliothek SYSLIB.CRTE nach Standard-Includes durchsucht werden. Dies ist sichergestellt, wenn in der MODIFY-INCLUDE-LIBRARY-Anweisung für die Suche nach den Standard-Includes die Option STD-INCLUDE-LIBRARY=*STANDARD-LIBRARY angegeben wird (Voreinstellung).

Das Programm muss mit der BIND-Anweisung des Compilers gebunden werden. In der MODIFY-BIND-PROPERTIES-Anweisung ist die Angabe RUNTIME-LANGUAGE =*CPLUSPLUS(MODE=*V3-COMPATIBLE) erforderlich.

Für weitere Einzelheiten siehe "MODIFY-BIND-PROPERTIES".

Programmentwicklung in POSIX-Umgebung

Zum Einfügen der Include-Dateien und zum Binden der Module muss im CC-Kommando der C++ V3-Modus angegeben werden (`-X V3-COMPATIBLE`). Für weitere Einzelheiten siehe Handbuch „POSIX-Kommandos des C /C++-Compilers“ [1].

Dokumentation

Diese C++-Bibliothek ist ausführlich in folgendem Handbuch beschrieben:„Standard C++ Library V1.2, User‘s Guide and Reference“ [6] (ein Band).

Die zu Cfront V3.0.3 kompatiblen Ein-/Ausgabe-Klassen sind beschrieben in:„C++ V2.1 (BS2000) C++-Bibliotheksfunktionen“ [5].

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

Die zu Cfront V3.0.3 kompatible C++-Bibliothek kann nur im Cfront-C++-Modus des Compilers genutzt werden. Sie enthält folgende Schnittstellen:

- eine Klasse für komplexe Mathematik

  `<complex.h>`
- Klassen für stromorientierte Ein-/Ausgaben

  ```
  <iostream.h>
  <fstream.h>
  <strstream.h>
  <stdiostream.h>
  <iomanip.h>
  <generic.h>
  <new.h>
  ```

Die Include-Dateien für die o.g. Schnittstellen sind in der Bibliothek SYSLIB.CRTE.CPP enthalten, die Module in der Bibliothek SYSLNK.CRTE.CPP.

Benutzen der Bibliothek

Programmentwicklung in BS2000-Umgebung (SDF)

> ¡ Die Header der Cfront-C++-Bibliothek enthalten Deklarationen, die das „at“-Zeichen (@) verwenden. Deshalb kann die Cfront-C++-Bibliothek nicht werwendet werden, wenn gleichzeitig in der MODIFY-SOURCE-PROPERTIES-Anweisung die Option AT-ALLOWED=*NO spezifiziert wird.

Beim Übersetzen muss die Bibliothek SYSLIB.CRTE.CPP vor der Bibliothek SYSLIB.CRTE nach Standard-Includes durchsucht werden. Dies ist sichergestellt, wenn in der MODIFY-INCLUDE-LIBRARY-Anweisung für die Suche nach den Standard-Includes die Option STD-INCLUDE-LIBRARY=*STANDARD-LIBRARY angegeben wird (Voreinstellung).

Beim Binden mit der BIND-Anweisung des Compilers, muss in der MODIFY-BIND-PROPERTIES-Anweisung lediglich folgende Angabe gemacht werden:
RUNTIME-LANGUAGE=*CPLUSPLUS(MODE=*V2-COMPATIBLE).

Programmentwicklung in POSIX-Umgebung

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

Zum Einfügen der Include-Dateien und zum Binden der Module muss im CC-Kommando lediglich die Sprachmodus-Option `-X V2-COMPATIBLE` angegeben werden. Dabei ist zu beachten, dass diese Option sowohl beim Übersetzen als auch beim Binden angegeben werden muss:

```
CC -X V2-COMPATIBLE -c x.C y.C  # Übersetzen
CC -X V2-COMPATIBLE x.o y.o     # Binden
```

Dokumentation

Die Cfront-C++-Bibliothek ist ausführlich in folgendem Handbuch beschrieben:„C++ V2.1 (BS2000) C++-Bibliotheksfunktionen“ [5].

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

Die C++-V3-Bibliothek Tools.h++ V7.0 kann nur im C++ V3-Modus des Compilers genutzt werden.

Sie bietet ein breites Spektrum an „Foundation Classes“:

- String-Klassen mit pattern-matching Mechanismen
- Klassen zur Handhabung von Datum und Uhrzeit
- virtuelle Ströme
- Datei- und Dateimanager-Klassen
- Containerklassen (Collectable) mit der Möglichkeit zur Realisierung von Persistenz und zugehörigen Iterator-Klassen:
  - Smalltalk-ähnliche Containerklassen (ohne Template-Benutzung)
  - Template-Containerklassen zur Speicherung von Werten (`RWTVal`...)
  - Template-Containerklassen zur Speicherung von Zeigern (`RWTPtr`...)
- Klassen zur Internationalisierung

Die Include-Dateien für die o.g. Schnittstellen sind in der Bibliothek SYSLIB.CRTE enthalten, die Module in der Bibliothek SYSLNK.CRTE.TOOLS.

Benutzen der Bibliothek

Programmentwicklung in BS2000-Umgebung (SDF)

Beim Übersetzen muss die Bibliothek SYSLIB.CRTE nach Standard-Includes durchsucht werden. Dies ist sichergestellt, wenn in der MODIFY-INCLUDE-LIBRARY-Anweisung für die Suche nach den Standard-Includes die Option STD-INCLUDE-LIBRARY =*STANDARD-LIBRARY angegeben wird (Voreinstellung).

Das Programm muss mit der BIND-Anweisung des Compilers gebunden werden.
In der MODIFY-BIND-PROPERTIES-Anweisung sind folgende Angaben erforderlich:
RUNTIME-LANGUAGE=*CPLUSPLUS(MODE=*V3-COMPATIBLE) und TOOLSLIB=*YES.

Für weitere Einzelheiten siehe "MODIFY-BIND-PROPERTIES".

Programmentwicklung in POSIX-Umgebung

Zum Einfügen der Include-Dateien und zum Binden der Module muss im `CC`-Kommando einer der C++ V3 Modus angegeben werden (`-X V3-COMPATIBLE`).

Beim Binden des Programms muss zusätzlich die Option `-l RWtools` angegeben werden. Für weitere Einzelheiten siehe Handbuch „POSIX-Kommandos des C/C++-Compilers“ [1].

Dokumentation

Die C++-V3-Bibliothek Tools.h ist in folgenden Handbüchern ausführlich beschrieben:
„Tools.h++ V7.0, User‘s Guide“ und
„Tools.h++ V7.0, Class Reference“

Hinweise zur Dokumentation

Um die Bibliothek *tools.h++* unabhängig von inkompatiblen künftigen Versionen der Standard-Bibliothek zu machen, setzt sie nicht auf die Standardbibliothek auf (d.h. das define RW_NO_STL wurde im zentralen Konfigurationsheader rw/compiler.h gesetzt).
Bestimmte Klassen sind daher nicht oder nur eingeschränkt verfügbar. Diese sind im Handbuch „Tools.h++ Class Reference“ [8] entsprechend gekennzeichnet, z.B.
„RWTValMap requires the Standard C++ Library.“
Bei Benutzung der zugehörigen Header wird eine *#error*-Fehlermeldung ausgegeben, wie z.B.

```
„Cannot include header if RW_NO_STL macro is defined for your compiler“
```

oder

```
„You must have both Standard Library and Exceptions to use this class.“
```

Einige Klassen realisieren Funktionen, die an bestimmte Systeme gebunden sind:

header rw/xdrstrea.h; Klasse RWXDRistream und RWXDRostream

Diese Klassen sind im BS2000/POSIX nutzbar, im BS2000 native aber nicht.

header rw/winstrea.h; Klasse RWCLIPstreambuf

Im Handbuch „Tools.h++ Class Reference“ [8] steht dazu:
„Class RWCLIPstreambuf is a specialized streambuf that gets and puts sequences of characters to Microsoft Windows global memory. It can be used to exchange data through Windows clipboard facility.“
Da es im BS2000 weder das Konzept eines „Microsoft Windows global memory“ noch das eines „Windows clipboard “ gibt, steht diese Klasse hier nicht zur Verfügung.

8.5 Das C++-Laufzeitsystem

In diesem Abschnitt werden folgende Themen behandelt:

- Initialisierung
- Ausnahmebehandlung
  - Zusätzliche Laufzeitfunktionen
  - C-Signalbehandlung und C++-Ausnahmebehandlung
  - longjmp-Unterstützung
  - Verknüpfung von alten C-Modulen mit C++ V3- und C++ 2017-Modulen

8.5.1 Initialisierung

Ausnahmen, die während der Initialisierung von globalen Objekten „geworfen“ werden (`throw`), bedingen den Aufruf von `terminate` und den Abbruch des Programms ohne Diagnosemeldungen. Mit der Funktion `set_terminate` können Sie eine Funktion vorsehen, die im Fall eines unvorgesehenen Programmendes als Ausnahmebehandlungsroutine fungiert. Diese Funktion muss jedoch vor der Initialisierung der globalen Objekte aufgerufen werden.

Das C++-Laufzeitsystem bietet folgende Lösung an, um Funktionen anzugeben, die als „initial current handler“ benutzt werden:

- Sie können eine eigene `__initial_terminate_handler`-Funktion vom Typ `terminate_handler` zu Ihrem Programm binden. Diese Funktion wird im C++-Laufzeitsystem als `weak` deklariert. Wenn `__initial_terminate_handler` definiert ist, wird die Funktion als „initial handler“ für die Ausnahmebehandlung aufgerufen.
- Sie können auch die Funktionen `__initial_unexpected_handler` und `___initial_new_handler` mit demselben Mechanismus verwenden. Diese Routinen haben den Typ `unexpected_handler` bzw. `new_handler`.

Für die Schnittstellen `__initial_terminate_handler` und `__initial_unexpected_handler` muss die Header-Datei `<exception>` inkludiert werden, für die Schnittstelle `__initial_new_handler` die Header-Datei `<new>`.

8.5.2 Ausnahmebehandlung

In diesem Abschnitt werden folgende Themen behandelt:

- Zusätzliche Laufzeitfunktionen
- C-Signalbehandlung und C++-Ausnahmebehandlung
- longjmp-Unterstützung
- Verknüpfung von alten C-Modulen mit C++ V3- und C++ 2017-Modulen

8.5.2.1 Zusätzliche Laufzeitfunktionen

Zusätzlich zu den Laufzeitfunktionen, die durch den Sprachstandard definiert sind, bietet das C++-Laufzeitsystem weitere nützliche Funktionen an, mit denen Programme verlässlicher gestaltet werden können. Allerdings sind Programme, die diese Funktionen verwenden, nicht ANSI-C++-konform und deshalb nicht portabel.
Es handelt sich um die Funktionen `unwind_exit`, `get_caught_object_typeid`, `can_throw` und `can_rethrow`.
Sie sind alle Bestandteil des Namensraums `_SNI_extensions`.

Ebenfalls im Namensraum `_SNI_extensions` definert ist der für die Argumente verwendete Datentyp `EO_flag_set`.

Datentyp EO_flag_set

Der Datentyp `EO_flag_set` wird verwendet um Eigenschaften eines Ausnahmeobjekts anzuzeigen.
`EO_flag_set` ist in den Sprachmodi C++ V3 und C++ 2017 unterschiedlich definiert.

EO_flag_set im Sprachmodus C++ 2017

Im C++ 2017-Sprachmodus ist `EO_flag_set` folgendermaßen definiert:

```
namespace _SNI_extensions {
      typedef unsigned int EO_flag_set;
}
```

Folgende Flag-Werte sind für Datenfelder vom Typ `EO_flag_set` definiert:

```
EO_NO_FLAGS             : kein Zeigertyp / kein Flag gesetzt
EO_IS_POINTER           : Zeigertyp
EO_POINTER_TO_CONST     : Zeiger auf konstantes Objekt
EO_POINTER_TO_VOLATILE  : Zeiger auf flüchtiges Objekt
EO_LAST                 : letztes Element eines Flag-Arrays
```

EO_flag_set im Sprachmodus C++ V3

Im C++ V3-Sprachmodus ist `EO_flag_set` folgendermaßen definiert:

```
namespace _SNI_extensions {
      typedef unsigned char EO_flag_set;
}
```

Folgende Bit-Werte sind für Datenfelder vom Typ `EO_flag_set` definiert:

```
EO_NO_FLAGS             : kein Zeigertyp
EO_IS_POINTER           : Zeigertyp
EO_POINTER_TO_CONST     : Zeiger auf konstantes Objekt
EO_POINTER_TO_VOLATILE  : Zeiger auf flüchtiges Objekt
```

unwind_exit - Stack vor der Programmbeendigung abbauen

Die Funktion `unwind_exit` dient, ähnlich wie `exit`, zur Beendigung eines Programms. Im Unterschied zu `exit` werden beim Aufruf von `unwind_exit` vor der Programmbeendigung die folgenden zusätzlichen Aktionen durchgeführt:

- die Destruktion aller noch nicht gelöschten `automatic` Objekte auf dem Laufzeitstack
- die Destruktion aller noch nicht beendeten Ausnahmeobjekte

Anschließend erfolgt analog zu `exit` die Destruktion der globalen Objekte.

Eine Destruktion noch nicht freigegebener Objekte auf dem Heap erfolgt jedoch weder durch `exit` noch durch `unwind_exit`.

Wird ein von `unwind_exit` aufgerufener Destruktor mit einer Ausnahme beendet, so wird implizit `terminate` aufgerufen. Es ist daher ratsam, im `terminate_handler` ebenfalls `unwind_exit` aufzurufen. Dadurch ist garantiert, dass in jedem Fall nach und nach die Destruktoren für alle `automatic`- und Ausnahme-Objekte aufgerufen werden. Eine Endlosschleife kann dabei nicht entstehen.

Die Funktion `unwind_exit` kann, wie auch `exit`, von jeder Stelle des Programms aus aufgerufen werden, insbesondere in einer Beendigungsbehandlung (`terminate_handler`). Als Argument wird ihr, analog und mit gleicher Wirkung wie bei `exit`, ein Exit-Status mitgegeben.

Der Prototyp der Funktion `unwind_exit` ist in der Header-Datei `<exception>` deklariert:

```
#include <exception>
namespace _SNI_extensions {
      void unwind_exit(int status);
}
```

get_caught_object_typeid - Ermittlung des Typs abgefangener Ausnahmeobjekte

Mit der Funktion `get_caught_object_typeid` kann der Typ eines abgefangenen Ausnahmeobjekts ermittelt werden. Geliefert wird der Typ des jüngsten abgefangenen und noch nicht beendeten Ausnahmeobjekts. Falls kein abgefangenes und noch nicht beendetes Ausnahmeobjekt existiert, wird eine Ausnahme vom Typ `bad_typeid` geworfen (`throw`).

Die Funktion `get_caught_object_typeid` kann in einer beliebigen Behandlungsroutine (`terminate_handler`, `unexpected_handler`, `catch(...) {...}`) aufgerufen werden, um Informationen über den Typ des abgefangenen Ausnahmeobjekts zu erhalten. Dies kann die Diagnose von Programmablauf-Fehlern erleichtern.

Die Klasse `type_info` und der Prototyp der Funktion `get_caught_object_typeid` sind in der Header-Datei `<typeinfo>` deklariert.

Die Funktion `get_caught_object_typeid` ist in den Sprachmodi C++ V3 und C++ 2017 unterschiedlich definiert.

get_caught_object_typeid im Sprachmodus C++ 2017

Der Prototyp der Funktion `get_caught_object_typeid` ist in der Header-Datei `<typeinfo>` folgendermaßen deklariert:

```
#include <typeinfo>
namespace _SNI_extensions {
      const type_info &get_caught_object_typeid(EO_flag_set *pflags, EO_flag_set **ppflags);
}
```

Die beiden Argumente dienen der Rückgabe von Informationen, ob es sich um bei dem Typ um einen Zeiger handelt oder nicht. Bei Werten von Null wird keine Information zurückgegeben.

Das Verhalten hängt davon ab, ob der Typ kein Zeiger ist, ein einfacher Zeiger oder ein Zeiger auf einen Zeiger.

Ist der Typ kein Zeiger, so wird er als Referenz auf ein `type_info` Objekt geliefert. `*pflags` erhält den Wert EO_NO_FLAGS. `*ppflags` erhält den Wert Null.

Ist der Typ ein einfacher Zeiger, so wird der Typ des Objekts, auf das der Zeiger zeigt, als Referenz auf ein `type_info` Objekt geliefert . `*pflags` erhält einen der Werte `EO_IS_POINTER`, `(EO_IS_POINTER | EO_POINTER_TO_CONST)`, `(EO_IS_POINTER | EO_POINTER_TO_VOLATILE)` oder `(EO_IS_POINTER | EO_POINTER_TO_CONST | EO_POINTER_TO_VOLATILE)` je nachdem, ob der Zeiger konstant und/oder flüchtig ist. `*ppflags` erhält den Wert Null.

Ist der Typ ein Zeiger auf einen Zeiger-Typ, so wird die Kette der Zeiger verfolgt, bis ein Objekt erreicht wird, welches kein Zeiger ist. Der Typ dieses Objekts wird als Referenz auf ein `type_info` Objekt geliefert. `*pflags` erhält den Wert `EO_NO_FLAGS`. Es wird ein Array angelegt, der pro Zeiger ein Element hat. Die Elemente zeigen an, ob der Zeiger dieser Stufe konstant und/oder flüchtig ist. Dabei werden die Flag-Werte `EO_POINTER_TO_CONST` und `EO_POINTER_TO_VOLATILE` genutzt. Der Flag-Wert `EO_IS_POINTER` wird nicht verwendet. Das letzte Element des Arrays enthält zusätzlich den Flag-Wert `EO_LAST`. `*ppflags` wird auf die Adresse dieses Arrays gesetzt.

Ein Beispiel: für den Typ "`volatile char * *const *`" wird der Array `{ EO_POINTER_TO_CONST, EO_NO_FLAGS, EO_POINTER_TO_VOLATILE|EO_LAST }` gebildet.

get_caught_object_typeid im Sprachmodus C++ V3

Der Prototyp der Funktion `get_caught_object_typeid` ist in der Header-Datei `<typeinfo>` folgendermaßen deklariert:

```
#include <typeinfo>
namespace _SNI_extensions {
      const type_info &get_caught_object_typeid(EO_flag_set *pflags);
}
```

Wenn der Typ des abgefangenen Ausnahmeobjekts kein Zeigertyp ist, wird er als Referenz auf ein `type_info` Objekt geliefert. Falls das Ausnahmeobjekt von einem Zeigertyp ist, so wird der Typ des Objekts, auf das der Zeiger zeigt, geliefert. Ist das der Funktion `get_caught_object_typeid` übergebene Argument ungleich Null, so wird es als Adresse interpretiert, bei der hinterlegt wird, ob das abgefangene Objekt von einem Zeigertyp ist, und falls ja, welche Typ-Attribute für das Objekt gelten, auf das der Zeiger zeigt (Werte siehe oben). Wird zum Beispiel ein `const char *` geworfen, so wird die typeid von `char` zurückgegeben und `*pflags` erhält den Wert `(EO_IS_POINTER | EO_POINTER_TO_CONST)`.

can_throw - Prüfen auf terminate oder unexpected beim Werfen eines Objekts

Die Funktion `can_throw` (Prädikat) prüft, ob ein Ausnahmeobjekt des angegebenen Typs geworfen werden kann, ohne dass dies zum Aufruf von `terminate` oder `unexpected` führt.

```
#include <typeinfo>
namespace _SNI_extensions {
    bool can_throw(const std::type_info &typeid_to_check,
                   EO_flag_set flags = EO_NO_FLAGS);
}
```

`typeid_to_check` muss ein Nicht-Zeigertyp eines Ausnahmeobjekts sein.

Mit `flags` kann angegeben werden, ob der tatsächlich zu überprüfende Typ des Ausnahmeobjekts ein Zeigertyp ist und ob er mit den Typ-Attributen `const` oder `volatile` versehen ist (Werte siehe oben).

Wenn das Argument `flags` nicht angegeben wird, ist `EO_NO_FLAGS` (kein Zeigertyp) voreingestellt.

can_rethrow - Prüfen auf terminate oder unexpected beim Weiterwerfen eines Objekts

Die Funktion `can_rethrow` (Prädikat) prüft, ob ein Ausnahmeobjekt weitergeworfen werden kann (`throw;`), ohne dass dies zum Aufruf von `terminate` oder `unexpected` führt.

```
#include <typeinfo>
namespace _SNI_extensions {
    bool can_rethrow(void);
}
```

8.5.2.2 C-Signalbehandlung und C++-Ausnahmebehandlung

Gemäß der C++-Sprachdefinition ist es nicht erlaubt, C++-Ausnahmebehandlung in C-Signalroutinen zu benutzen. Wenn während einer durch eine Ausnahme ausgelösten Stack-Abwicklung durch das Laufzeitsystem dennoch eine Signalroutine erreicht wird, wird so getan, als ob das Ende des Stacks erreicht ist. Es ist daher nicht möglich, eine Ausnahme aus einer Signalroutine zu werfen oder eine Ausnahme weiterzuwerfen, die mit `catch` vor dem Aufruf der Signalroutine abgefangen wurde. In solchen Fällen wird `terminate` aufgerufen.

Die Funktion `unwind_exit` (siehe "Zusätzliche Laufzeitfunktionen") führt die beschriebenen Destruktionen nicht vollständig durch, wenn sie aus Signalroutinen aufgerufen wird. Es erfolgt keine Destruktion der vor dem Aufruf der Signalroutine noch nicht gelöschten `automatic` Objekte sowie der bis dahin noch nicht beendeten Ausnahmeobjekte. Die Beendigung des Programms mit `exit` findet jedoch statt.

8.5.2.3 longjmp-Unterstützung

Nach einem `longjmp`-Aufruf führt das C++-Laufzeitsystem für jede Funktion, die übersprungen wird, folgende Aktionen durch:

- die Destruktion aller in der Funktion konstruierten und noch nicht gelöschten `automatic` Objekte
- die Destruktion aller in der Funktion abgefangenen und noch nicht beendeten Ausnahmeobjekte

Diese Destruktionen werden allerdings nicht für die Zielfunktion des `longjmp`-Aufrufs selbst durchgeführt. Deshalb sollte in einer Funktion, die einen `setjmp`-Aufruf enthält, zwischen dem `setjmp`-Aufruf und dem Ansprung der Routine, die durch den `longjmp` abgebrochen wird, kein Objekt konstruiert, destruiert oder gefangen werden. Dies lässt sich dadurch erreichen, dass sämtlicher Code für Konstruktion, Destruktion und Auffangen von Ausnahmeobjekten in eine eigene Funktion ausgelagert wird. Damit diese Funktion nicht inline generiert wird, muss sie mittels eines externen Funktionszeigers aufgerufen werden.

Betrachten Sie hierzu das folgende Beispiel:

Beispiel

Statt der Codierung der Funktion `f()` in der Version 1 sollte die Codierung der Version 2 verwendet werden:

/* Version 1 */

```
#include <setjmp>
jmp_buf target;
void g();
class X { ~X(); };
void g()
{
    longjmp(target, 1);
}
void f()
{
    if (setjmp(target) == 0)
    {
        X x;             // Keine Destruktion bei longjmp-Aufruf durch g()
        g();
    }
}
```

/* Version 2 */

```
void f1()
{
    X x;                // Destruktion bei longjmp-Aufruf durch g()
    g();
}
extern void (*f1p)() = f1;
void f()
{
    if (setjmp(target) == 0)
    {
        (*f1p)();         // f1() wird hier nicht inline generiert
    }
}
```

longjmp und Signalroutinen

Wenn `longjmp` aus einer Signalroutine aufgerufen wird, werden in der Funktion, in der das Signal aufgetreten ist, keine Destruktionen für `automatic` Objekte und Ausnahmeobjekte durchgeführt.

8.5.2.4 Verknüpfung von alten C-Modulen mit C++ V3- und C++ 2017-Modulen

C-Module, die mit dem C- oder C++-Compiler V2.2 erzeugt wurden, können mit C++ V3 bzw. C++2017-Modulen verknüpft werden. Bezüglich der Stack-Abwicklung durch das C++-Laufzeitsystem gibt es folgende Einschränkungen:

- Es darf keine Ausnahme geworfen werden, wenn zwischen dem `throw`-Aufruf und dem Ausnahme-Handler (`catch`), der die Ausnahme auffängt, noch eine C-V2.2-Funktion aktiv ist.
- Es darf keine Ausnahme weitergeworfen werden, wenn zwischen dem Aufruf `throw;` und dem Ausnahme-Handler (`catch`), der die weitergeworfene Ausnahme aufgefangen hat, noch eine C-V2.2-Funktion aktiv ist.
- Es darf kein `longjmp` aus einer C++-Funktion über eine C-V2.2-Funktion hinweg erfolgen.

9 Anhang

In diesem Kapitel werden folgende Themen behandelt:

9.1 Beschreibung der Listenbilder

Die Listengeneratoren des Compilers erzeugen - je nach Anforderung mit der MODIFY-LISTING-PROPERTIES-Anweisung - folgende Listen:

Liste	Option der MODIFY-LISTING-PROPERTIES-Anweisung
Optionenliste	OPTIONS
Quellprogramm-/Fehlerliste	SOURCE
Präprozessorliste	PREPROCESSING-RESULT
Adressliste	DATA-ALLOCATION-MAP
Querverweisliste	CROSS-REFERENCE
Projektliste	PROJECT-INFORMATION
Objektcodeliste	ASSEMBLER-CODE
Übersichtsliste	SUMMARY

Auf den nächsten Seiten werden folgende Listen anhand von Beispielen erläutert: Quellprogramm-/Fehlerliste, Adressliste, Querverweisliste und Objektcodeliste.

Zur Vermeidung von Zeilenumbrüchen wurden in einigen Listen Leerzeichen entfernt.

9.1.1 Quellprogramm-/Fehlerliste

Die Quellprogramm-/Fehlerliste wird mit der SOURCE-Option der MODIFY-LISTING-PROPERTIES-Anweisung angefordert.

```
*** SOURCE - ERROR - LISTING **    BS2000 C/C++ COMPILER 04.0A10    DATE:2020-04-02 PAGE: 1
                         SOURCENAME:*BS2000(MAINPROG)              TIME=17:37:07
_____________________________________________________________________________________

  EXP  INC  FILE    SRC BLOCK
  LIN  LEV   NO     LIN  LEV
    1    0     0      1    0  #include <stdio.h>
 1746    0     0      2    0  #include "incl1.h"                                  __
 1747    1    10      1    0  class A                                             |
 1748    1    10      2    0  {                                                   |
 1749    1    10      3    0       int i;                                         |
 1750    1    10      4    0     public:                                          | (1)
 1751    1    10      5    0       A(int x = 1) : i(x) {};                        |
 1752    1    10      6    0       void foo() { printf( "A::foo called\n" ); };   |
 1753    1    10      7    0  } a;                                               __|
 1754    0     0      3    0  #include "incl2.h"
 1755    1    11      1    0  class B : public A
 1756    1    11      2    0  {
 1757    1    11      3    0       int i;
 1758    1    11      4    0     public:
 1759    1    11      5    0       B(int x = 2) : i(x) {};
 1760    1    11      6    0       void foo() { printf( "B::foo called\n" ); };
 1761    1    11      7    0  } b;
 1762    0     0      4    0  extern "C" int jj;
 1763    0     0      5    0  extern int ii;
 1764    0     0      6    0
 1765    0     0      7    0  int main(void)
 1766    0     0      8    0  {
 1767    0     0      9    1    char *string = "AbCdEfG";
 1768    0     0     10    1    float xx = 1.0;
 1769    0     0     11    1    int ii = 1;
 1770    0     0     12    1    int jj = 2;
 1771    0     0     13    1    A* aptr = &a;
 1772    0     0     14    1    A* bptr = &b;
 1773    0     0     15    1
 1774    0     0     16    1    printf("%d\n", ii);
 1775    0     0     17    1    printf("%d\n", jj);
 1776    0     0     18    1    printf("%s\n", string);
 1777    0     0     19    1    printf("%f\n", xx);
 1778    0     0     20    1    a.foo();
 1779    0     0     21    1    aptr->foo();
 1780    0     0     22    1    b.foo();
 1781    0     0     23    1    bptr->foo();
 1782    0     0     24    1
 1783    0     0     25    1    return 0;
 1784    0     0     26    1  }

_____________________________________________________________________________________
```

(1) Der Inhalt von Include-Elementen, abhängig von der INCLUDE-INFORMATION-Option der MODIFY-LISTING-PROPERTIES-Anweisung (hier nur die benutzereigenen Include-Elemente).

Erklärung

EXP, LIN

Quellprogrammzeilennummer einschließlich aller Zeilen der im Quellprogramm verwendeten Include-Elemente. Die Zeilen aus den Include-Elementen werden generell, d.h. unabhängig davon mitgezählt, ob sie in der Quellprogrammliste abgebildet werden oder nicht (siehe INCLUDE-INFORMATION-Option).

INC, LEV

Schachtelungsebene der Include-Elemente

FILE, NO

Nummer der Datei (Quelldatei bzw. Include-Element), deren Inhalt jeweils in der Quellprogrammliste abgebildet ist. Die Nummer (beginnend mit 0 für die Quelldatei) wird pro #include- bzw. #line-Anweisung um 1 erhöht. Am Ende jedes Include-Elements wird die Nummer wieder auf den Wert der Datei zurückgesetzt, die die zugehörige #include-Anweisung enthält. Diese Nummer ist für die Source-Referenz beim Testen mit AID relevant, wenn Include-Elemente ausführbare Anweisungen enthalten oder #line-Anweisungen in Quellprogrammen mit ausführbaren Anweisungen eingestreut sind.
Im Beispiel werden nur die benutzereigenen Include-Elemente expandiert: incl1.h (10) und incl2.h (11). Dem Standard-Include-Element stdio.h sowie den weiteren darin enthaltenen #include-Anweisungen sind die Nummern 1 bis 9 zugeordnet (siehe auch FILETABLE-Teil der Querverweisliste).

SCR, LIN

Original-Zeilennummer in der Quelldatei bzw. im Include-Element unter Berücksichtigung von #line-Anweisungen

BLOCK, LEV

Schachtelungstiefe der Anweisungsblöcke

9.1.2 Adressliste

Die Adressliste wird mit der DATA-ALLOCATION-MAP-Option der MODIFY-LISTING-PROPERTIES-Anweisung angefordert. Sie gibt Informationen über alle im Programm verwendeten symbolischen Adressen (Variablennamen, Funktionsnamen).

```
******** MAP - LISTING ********    BS2000 C/C++ COMPILER 04.0A10    DATE:2020-04-02 PAGE: 3
                          SOURCENAME:*BS2000(MAINPROG)              TIME=17:37:33

_____________________________________________________________________________________________
name                    stcl/type  size  slice  offs  xoffs    enuval   stroffs   xstroffs
a                       nospec        4      1   104 0x0068         -         -          -
                        class
aptr                    auto          4      1    24 0x0018         -         -          -
                        pointer to class
b                       nospec        8      1   108 0x006C         -         -          -
                        class
bptr                    auto          4      1    28 0x001C         -         -          -
                        pointer to class
ii                      extern        4      -     -      -         -         -          -
                        signed int
ii                      auto          4      1    16 0x0010         -         -          -
                        signed int
jj                      extern        4      -     -      -         -         -          -
                        signed int
jj                      auto          4      1    20 0x0014         -         -          -
                        signed int
main                    nospec        0      1    48 0x0030         -         -          -
                        entry_var
string                  auto          4      1     8 0x0008         -         -          -
                        pointer to char
x                       param         4      -     -      -         -         -          -
                        signed int
x                       param         4      -     -      -         -         -          -
                        signed int
xx                      auto          4      1    12 0x000C         -         -          -
                        float
A                       nospec        4      -     -      -         -         -          -
                        class
B                       nospec        8      -     -      -         -         -          -
                        class
_____________________________________________________________________________________________
```

Erklärung

name Name der symbolischen Adresse

stcl storage class: Speicherklasse der symbolischen Adresse. Folgende Bezeichnungen werden verwendet:

auto	Variablen auf Blockebene, außer static-Variablen
extern	externe Variablen und Funktionen, die in einem anderen Modul definiert sind
member	Elemente von Klassen, Strukturen oder Unions Werte von Aufzählungstypen
param	Funktionsparameter
reg	Variablen, die mit dem Schlüsselwort `register` deklariert wurden
static	static-Variablen auf Blockebene, d.h. mit Gültigkeit auf Blockebene
statmem	statisches Element einer Klasse
typedef	typedef-Name
nospec	Es wurde keine Speicherklasse angegeben

type Datentyp der symbolischen Adresse (in eigener Zeile unter der Speicherklasse).
Die meisten Einträge sind selbsterklärend. Besondere Werte sind:

ass_proc	Zuweisungs-Operator
bit	Bitfeld
entry_var	Funktion
enum	Aufzählungstyp
enum_val	Element eines Aufzählungstyps
lab_const	Label
mem_pointer	Zeiger auf Element
opr_func	Operator
tmpl	template
tmpl_inst	template instance
tmpl_par_cl	template parameter

size Größe der Variablen im Speicher (in Byte)

slice Eine Slice ist ein durch ein Basisregister adressierbarer Bereich (Code- oder Datenstück) von 4096 Byte. Die Ziffer gibt an, in welcher Slice des Datenmoduls die Variable angelegt ist.

offs Relative Adresse innerhalb einer Slice (dezimal)

xoffs Relative Adresse innerhalb einer Slice (sedezimal)

enuval Bei Elementen eines Aufzählungstyps (enum) gibt enuval den Wert dieses Elements an.

stroffs Byteposition der symbolischen Adresse innerhalb einer Struktur (dezimal)

xstroffs Byteposition der symbolischen Adresse innerhalb einer Struktur (sedezimal)

9.1.3 Querverweisliste

Die Querverweisliste wird mit der CROSS-REFERENCE-Option der MODIFY-LISTING-PROPERTIES-Anweisung angefordert.

Sie besteht aus folgenden Teilen:

- Der FILETABLE-Teil enthält die Namen aller Dateien bzw. Bibliotheken/Elemente, die der Compiler als Quelle (Quellprogramm oder Include-Element) verwendet hat und ordnet diesen jeweils eine Nummer zu. Auf diese Nummern nehmen die anderen Teile der Querverweisliste Bezug.
- Der PREPROCESSING-INFO-Teil enthält eine Liste der vom Präprozessor bearbeiteten Namen in `#include`- und `#define`-Anweisungen (Makros, Include-Elementnamen etc.)
- Der TYPES-Teil enthält eine Liste der benutzerdefinierten Typen (typedefs, Klassen-, Struktur-, Union- und Aufzählungstypen)
- Der VARIABLES-Teil enthält eine Liste der Variablen
- Der FUNCTIONS-Teil enthält eine Liste der Funktionen
- Der LABELS-Teil enthält eine Liste der Labels
- Der TEMPLATES-Teil enthält eine Liste der Templates (nur in den Modi C++ V3 und C++ 2017)

Die Namen sind in den Listen jeweils alphabetisch sortiert.

Der PREPROCESSING-INFO-, TYPES- und TEMPLATES-Teil sind in der Querverweisliste standardmäßig nicht enthalten und müssen mit PREPROCESSING-INFO=*YES, TYPES=*YES bzw. TEMPLATES=*YES explizit angefordert werden.

FILETABLE-Teil der Querverweisliste

```
******* XREF - LISTING ********    BS2000 C/C++ COMPILER 04.0A10    DATE:2020-04-02 PAGE: 1
         FILETABLE      SOURCENAME:*BS2000(MAINPROG)                TIME=17:37:52
_____________________________________________________________________________________
 SOURCE FILE      0 =  *BS2000(:2OSC:$TST30B.MAINPROG)
 INCLUDE FILE     1 =  *LIBRARY-ELEMENT(:2OSL:$TSOS.SYSLIB.CRTE,STDIO.H(V04.3A10),S)
 INCLUDE FILE     2 =  *LIBRARY-ELEMENT(:2OSL:$TSOS.SYSLIB.CRTE,IOBUF.H(V04.3A10),S)
 INCLUDE FILE     3 =  *LIBRARY-ELEMENT(:2OSL:$TSOS.SYSLIB.CRTE,POSFILE.H(V04.3A10),S)
 INCLUDE FILE     4 =  *LIBRARY-ELEMENT(:2OSL:$TSOS.SYSLIB.CRTE,STDIO.BS21.H(V04.3A10),S)
 INCLUDE FILE     5 =  *LIBRARY-ELEMENT(:2OSL:$TSOS.SYSLIB.CRTE,STDIO.COMMON.H(V04.3A10),S)
 INCLUDE FILE     6 =  *LIBRARY-ELEMENT(:2OSL:$TSOS.SYSLIB.CRTE,STDIO.BS22.H(V04.3A10),S)
 INCLUDE FILE     7 =  *LIBRARY-ELEMENT(:2OSL:$TSOS.SYSLIB.CRTE,CGLOBALS.H(V04.3A10),S)
 INCLUDE FILE     8 =  *LIBRARY-ELEMENT(:2OSL:$TSOS.SYSLIB.CRTE,IOBUF.H(V04.3A10),S)
 INCLUDE FILE     9 =  *LIBRARY-ELEMENT(:2OSL:$TSOS.SYSLIB.CRTE,ERRNO.H(V04.3A10),S)
 INCLUDE FILE    10 =  *LIBRARY-ELEMENT(:2OSC:$TST30B.PLAM.INCL,INCL1.H(*UPPER-LIMIT),S)
 INCLUDE FILE    11 =  *LIBRARY-ELEMENT(:2OSC:$TST30B.PLAM.INCL.INCL2.H(*UPPER-LIMIT),S)
_____________________________________________________________________________________
```

Dateinummer in globalen Querverweislisten

In einer Querverweisliste, die mit dem globalen Listengenerator aus mehreren CIF-Dateien erstellt wurde (siehe Abschnitt „Steuerung des globalen Listengenerators“) wird die Dateinummer in der Form *n*(*m*) angegeben. *n* ist die fortlaufende Nummer der pro Übersetzungseinheit (= CIF-Datei) verwendeten Quell- und Include-Dateien (analog zur lokalen Querverweisliste, s.o.), *m* ist die Nummer der jeweiligen Übersetzungseinheit. Die Nummerierung der Übersetzungseinheiten beginnt mit 0.

PREPROCESSING-INFO-Teil der Querverweisliste

```
******* XREF - LISTING ********    BS2000 C/C++ COMPILER 04.0A10    DATE:2020-04-02 PAGE: 2
         PREPRO       SOURCENAME:*BS2000(MAINPROG)                  TIME=17:37:52
_____________________________________________________________________________________
     56/12:5
 *LIBRARY-ELEMENT(:2OSC:$TST30B.PLAM.INCL,INCL1.H(*UPPER-LIMIT),S) / include file
     2%0
 *LIBRARY-ELEMENT(:2OSC:$TST30B.PLAM.INCL,INCL2.H(*UPPER-LIMIT),S) / include file
     3%0
_____________________________________________________________________________________
'.'applied ':'def ':^'undef '%'included
```

TYPES-Teil der Querverweisliste

```
******* XREF - LISTING ********    BS2000 C/C++ COMPILER 04.0A10    DATE:2020-04-02 PAGE: 3
         TYPES          SOURCENAME:*BS2000(MAINPROG)                  TIME=17:37:52
_____________________________________________________________________________________
a0000270              / struct of size 16 (0x10)
std                   / namespace
A                     / class of size 4 (0x4)                ----------------- (1)
    1/7:10           1/18.11         13/3.0           14/3.0  ----------------- (2)
                        public baseclass of class 'B'
  i                     / member, signed int, private          ----------------- (3)
  A                     / inline constructor( signed int ), public
  foo                   / member, inline function( void ) ret void, public
B                     / class of size 8 (0x8)
    1/7:11
  A                     / baseclass, public
  i                     / member, signed int, private
  B                     / inline constructor( signed int ), public
  foo                   / member, inline function( void ) ret void, public
_____________________________________________________________________________________
'.'used ':'def '&'decl
```

Erklärung

(1) Name und Beschreibung des benutzerdefinierten Typs, ggf. mit Angabe der Größe (dezimal und sedezimal)

(2) Jeweils von links nach rechts:
Quellprogrammzeile und Spalte, in der der Typ erscheint,
Abkürzungssymbol zur Verwendung des Typs,
Nummer der Quelldatei bzw. des (Include-)Elements aus der FILETABLE

14/3.0 bedeutet z.B.: In Zeile 14, Spalte 3 des Quellprogramms mainprog (0) wird eine Variable vom Typ der Klasse A angelegt. Der Typ wird benutzt (.)

(3) Bei strukturierten Typen werden auch die Komponenten dieser Typen (jeweils eingerückt) beschrieben.

Die Datenelemente von Strukturen, Klassen und Unions werden ausschließlich im TYPES-Teil aufgelistet (sind keine Variablen). Die Funktionselemente erscheinen noch einmal im FUNCTIONS-Teil.

VARIABLES-Teil der Querverweisliste

```
******* XREF - LISTING ********    BS2000 C/C++ COMPILER 04.0A10    DATE:2020-04-02 PAGE: 4
         VARIABLES      SOURCENAME:*BS2000(MAINPROG)                TIME=17:37:52
_____________________________________________________________________________________
a                      / class 'A'                            ------------------ (1)
    7/3=10            7/3:10          13/14&0          20/3&0 ------------------ (2)
aptr                   / automatic, pointer to class 'A',
                         local in main( void ) ret signed int, init value = &a
   13/6=0            13/6:0           21/3.0
b                      / class 'B'
    7/3=11            7/3:11          14/14&0          22/3&0
bptr                   / automatic, pointer to class 'A',
                         local in main( void ) ret signed int, init value = &b
   14/6=0            14/6:0           23/3.0
i                      / member, signed int,
                         member of class 'A'
    3/10:10           5/21=10
i                      / member, signed int,
                         member of class 'B'
    3/10:11           5/21=11
ii                     / automatic, signed int,
                         local in main( void ) ret signed int, init value = 1
   11/7=0            11/7:0           16/18.0
ii                     / extern, signed int
    5/12:%0
jj                     / extern, signed int
    4/16:%0
jj                     / automatic, signed int,
                         local in main( void ) ret signed int, init value = 2
   12/7=0            12/7:0           17/18.0
string                 / automatic, pointer to char,
                         local in main( void ) ret signed int, init value = "AbCdEfG"
    9/9=0             9/9:0           18/18.0
x                      / param of constructor A::A, signed int
    5/12:10           5/23.10
x                      / param of constructor B::B, signed int
    5/12:11           5/23.11
xx                     / automatic, float,
                         local in main( void ) ret signed int, init value = 1
   10/9=0            10/9:0           19/18.0
_____________________________________________________________________________________
'='write '.'read '*='indir-write '*.'indir-read '&'read-addr ':'def '%'decl ':%'extdecl
'%%'use
```

Erklärung

(1) Name, Speicherklasse und Datentyp der Variablen

(2) Jeweils von links nach rechts:
Quellprogrammzeile und Spalte, in der die Variable erscheint,
Abkürzungssymbol zur Verwendung der Variablen und
Nummer der Quelldatei bzw. des (Include-)Elements aus der FILETABLE

7/3:10 bedeutet z.B.: In Zeile 7, Spalte 3 des Include-Elements `incl1.h` (10) wird die Variable a definiert (:).

FUNCTIONS-Teil der Querverweisliste

```
******* XREF - LISTING ********    BS2000 C/C++ COMPILER 04.0A10    DATE:2020-04-02 PAGE: 5
         FUNCTIONS      SOURCENAME:*BS2000(MAINPROG)                TIME=17:37:52
_____________________________________________________________________________________
foo                    / public member of class 'B', inline function( void ) ret void  -- (1)
    6/11:11          22/5.0                                                            -- (2)
foo                    / public member of class 'A', inline function( void ) ret void
    6/11:10          20/5.0           21/9.0           23/9.0
main                   / function( void ) ret signed int
    7/5:0
A                      / public member of class 'A', inline constructor( signed int )
    5/6:10            7/3.10           5/26.11
B                      / public member of class 'B', inline constructor( signed int )
    5/6:11            7/3.11
_____________________________________________________________________________________
'.'call ':'def '&'decl '%'extdecl ':^'forward '&.'read-addr
```

Erklärung

(1) Name der Funktion, Geltungsbereich, Signatur (Typ der Parameter und Returntyp) und Speicherklasse; bei Elementfunktionen zusätzliche Informationen über das Zugriffsrecht (z.B. public) und die sie enthaltende Klasse oder Union

(2) Jeweils von links nach rechts:
Quellprogrammzeile und Spalte, in der die Funktion erscheint,
Art der Verwendung an dieser Stelle und
Nummer der Quelldatei bzw. des (Include-)Elements aus der FILETABLE

6/11:11 bedeutet z.B.: In Zeile 6, Spalte 11 des Include-Elements `incl2.h` (11) wird die Funktion `foo` definiert (:).

9.1.4 Objektcodeliste

Die Objektcodeliste wird mit der ASSEMBLER-CODE-Option der MODIFY-LISTING-PROPERTIES-Anweisung angefordert.

Sie enthält

- die sedezimale Abbildung des vom Code-Generator erzeugten Objektcodes
- den Objektcode in Assembler-Notation
- Kommentare zum Objektcode in Assembler-Notation
- die Quellprogrammzeilen in C/C++-Notation als Kommentare

Die Gesamtliste ist gegliedert in Code- und Datenmodulliste. Jede Modulliste ist unterteilt in Bereiche (Areas). Der Beginn der jeweiligen Modul- bzw. Bereichsliste ist durch Kommentarzeilen im Objektcode gekennzeichnet.

```
***** ASSEMBLER - LISTING *****    BS2000 C/C++ COMPILER 04.0A10    DATE:2020-04-02 PAGE: 1
                                SOURCENAME:*BS2000(MAINPROG)             TIME=17:37:22
_____________________________________________________________________________________

                                 1   ******************************
                                 2   *        CODE MODULE
                                 3   *
                                 4   *        OPTIMIZER:     ON
                                 5   *        SHARED CODE:   OFF
                                 6   *        OBJECT FORMAT: LLM
                                 7   ******************************
 000000                          8   MAINPROG\&@ CSECT       READ
 000000                          9   MAINPROG\&@ AMODE       ANY
 000000                         10   MAINPROG\&@ RMODE       ANY
                                11   ******************************
                                12   *        CODE AREA (main)
                                13   ******************************
 000000                         14            ENTRY  MAIN
 000000                         15   MAIN     DS     0A
 000000                         16            USING  *,15
 000000  90 EF D00C             17            STM    14,15,12(13)
 000004  90 2C D01C             18            STM    2,12,28(13)
 000008  58 90 D04C             19            L      9,76(0,13)
 00000C  98 AB F068             20            LM     10,11,#DC#CONT#10       104(15)
 000010  58 E0 9018             21            L      14,24(0,9)
 000014  41 00 E0D8             22            LA     0,216(0,14)
 000018  55 00 9010             23            CL     0,16(0,9)
 00001C  47 20 F044             24            BC     2,#OFLOW#10     68(0,15)
 000020  50 00 9018             25            ST     0,24(0,9)
 000000  00 00                  26   #OFLOWOK#10 EQU *
 000024  50 E0 900C             27            ST     14,12(0,9)
 000028  58 00 F088             28            L      0,#DC#SAVID#10  136(0,15)
 00002C  50 00 E000             29            ST     0,0(0,14)
 000030  58 00 F08C             30            L      0,#DC#EHL#10    140(0,15)
 000034  50 00 E050             31            ST     0,80(0,14)
 000038  50 D0 E004             32            ST     13,4(0,14)
 00003C  50 90 E04C             33            ST     9,76(0,14)
 000040  18 DE                  34            LR     13,14
 000042  07 FA                  35            BCR    15,10
                                 .
                                 .
                                 .
 000094  58 80 F084             69            L      8,132(0,15)
                                70   *****    LINE   7: (*BS2000(:2OSC:$TST30B.MAINPROG))
                                71   *****      int main(void)
 000098                         72   @0000001 DS     0H
                                73   *****    LINE   8
                                74   *****    {
 000098                         75   @0000002 DS     0H
 000098  58 F0 B024             76            L      15,36(0,11)     <17>    <17>
 00009C  41 00 0000             77            LA     0,0(0,0)
 0000A0  0D EF                  78            BASR   14,15
                                79   *****    LINE   16
                                80   *****      printf("%d\n", ii);
   |     |        |             |   |                              | |            |
   |     |________|             |   |______________________________| |____________|
   |         |                  |                  |                       |
  (1)       (2)                (3)                (4)                     (5)
```

Erklärung

(1) Adresspegel, sedezimal

(2) Objektcode, sedezimal

(3) Zeilennummer des Assembler-Codes

(4) Assembler-Code (symbolische Adresse, Assembler-Mnemonic, Operanden) und Quellprogrammzeile als Kommentar

(5) Erklärungen zum Assembler-Code

9.2 Vordefinierte Präprozessornamen

Bei der Übersetzung mit dem C/C++-Compiler in SDF-Umgebung (Anweisungen COMPILE, CHECK-SYNTAX, PREPROCESS) werden abhängig vom gewählten Sprachmodus und von der Eingabe weiterer 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)

Die im Folgenden angegeben Optionen sind, sofern nicht explizit anders erwähnt, Bestandteil der Anweisung MODIFY-SOURCE-PROPERTIES.

`_BOOL`	in den Sprachmodi C++ V3 und C++ 2017 bei der Option KEYWORD-BOOL=*YES
`__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
`c_plusplus`	in allen C++-Sprachmodi
`__CFRONT_V3`	im Sprachmodus Cfront-C++
`__EDG_NO_IMPLICIT_INCLUSION`	in den Sprachmodi C++ V3 und C++ 2017, wenn im Rahmen der Template-Instanziierung implizites Inkludieren mit der Option IMPLICIT-INCLUDE=*NO ausgeschaltet wurde
`__EXISTCGLOB`	immer gesetzt
`__HALF_TAG_LOOKUP`	immer gesetzt
`_IEEE`	Option FP-ARITHMETICS=*IEEE der Anweisung MODIFY-MODULE-PROPERTIES
`LANGUAGE_C`	immer gesetzt
`_LANGUAGE_C`	immer gesetzt
`__LITERAL_ENCODING_ASCII`	Option LITERAL-ENCODING=*ASCII[-FULL]
`__LITERAL_ENCODING_EBCDIC`	Option LITERAL-ENCODING=*EBCDIC[-FULL]/*NATIVE

_LONGLONG	Option LONGLONG=*YES
__OLD_SPECIALIZATION_SYNTAX	== 1 bei der Option SPECIALIZATION=*OLD
__SHORT_NAMES	Ist definiert, wenn C-NAMES=*SHORT angegeben wurde
__SIGNED_CHARS__	Option SIGNED-CHARACTER=*YES
__SMALL_VA_DCL	immer gesetzt
__SNI	in allen C-Modi und im Cfront-C++-Modus
__SNI_HOST_BS2000	immer gesetzt
__SNI_HOST_BS2000_POSIX	nie gesetzt (reserviert für Übersetzung in POSIX-Umgebung)
__SNI__STDCplusplus	in allen C++-Sprachmodi: == 0 in den erweiterten Sprachmodi (STRICT=*NO) == 1 in den strikten Sprachmodi (STRICT=*YES)
__SNI_TARG_BS2000	immer gesetzt
__SNI_TARG_BS2000_POSIX	nie gesetzt (reserviert für Übersetzung in POSIX-Umgebung)
__STDC__	immer gesetzt: == 0 in den erweiterten Sprachmodi (STRICT=*NO) == 1 in den strikten Sprachmodi (STRICT=*YES)
__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 und C++ 2017

`__STDC_VERSION__`	im K&R-C-Modus undefiniert == 199409L im Sprachmodus C89 und in allen C++ Sprachmodi == 201112L im Sprachmodus C11
`_STRICT_STDC`	in den strikten Sprachmodi (STRICT=*YES)
`_WCHAR_T`	Option KEYWORD-WCHAR=*YES (Voreinstellung in den Modi C++ V3 und C++ 2017) 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 KEYWORD-WCHAR=*YES (Voreinstellung in den Modi C++ V3 und C++ 2017)

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

9.3 Konzept der Namens-Adaptermodule im C-Laufzeitsystem

Bezüglich der C-Bibliotheksfunktionen besteht das Problem, dass einerseits die Namen, mit denen die Funktionen auf Quellprogrammebene angesprochen werden, durch Standards vorgeschrieben sind (z.B. `printf`, `fopen`), dass andererseits die BS2000-Namenskonventionen Entry-Namen verlangen, die mit dem Präfix „IC“ beginnen. Außerdem ist gefordert, dass die C-Bibliotheksfunktionen durch benutzereigene Funktionen ersetzt werden können, in denen die Entry-Namen aus dem Funktionsnamen ohne das Präfix „IC“ gebildet werden.

Dieses Problem wurde bis zur CRTE-Version 1.0B dadurch gelöst, dass der Compiler mithilfe einer Tabelle die Funktionsnamen erkannt und entsprechend umgesetzt hat. Dies hatte zur Folge, dass Änderungen im C-Laufzeitsystem immer mit Änderungen im Compiler verbunden waren. Diese Technik wird aus Kompatibilitätsgründen für die bis dahin im C-Laufzeitsystem vorhandenen C-Bibliotheksfunktionen (Funktionen des C90-Standards sowie ca. 50 BS2000-spezifische Erweiterungen) beibehalten. Für alle mit CRTE ab V2.0A namentlich neu hinzugekommenen POSIX-Funktionen und in Zukunft neu hinzukommenden Funktionen wird das Problem compilerunabhängig mit Namens-Adaptermodulen gelöst. Diese Adaptermodule enthalten den aus dem Funktionsnamen abgeleiteten Entry-Namen ohne das Präfix „IC“ und rufen die eigentliche „Standard“-Funktion mit dem Entry-Namen IC... auf.

Pro C-Bibliotheksfunktion stehen folgende Adaptermodule zur Verfügung:

- Ein Objektmodul (OM) mit dem ggf. auf 8 Zeichen verkürzten Funktionsnamen als Entry-Name, in dem die Kleinbuchstaben in Großbuchstaben und der Unterstrich in das Dollarzeichen umgewandelt sind. Z.B. `FPATHKON` für die Funktion `fpathkonv`.

  Diese sind nur noch relevant, wenn Objektmodule gebunden werden, die mit den C/C++-Vorgänger-Compilern (bis V2.2) erzeugt wurden. C/C++ ab V3 erzeugt ausschließlich Module im LLM-Format.
- Bis zu vier LLMs, in denen jeweils der unverkürzte Funktionsname als Entry-Name enthalten ist, aber einmal in Kleinbuchstaben oder in Großbuchstaben, einmal mit beibehaltenem oder mit umgewandeltem Unterstrich (vgl. die Optionen LOWER-CASE-NAMES und SPECIAL-CHARACTERS in der MODIFY-MODULE-PROPERTIES-Anweisung).

  Z.B. liegen für die Funktion `fpathkonv` zwei LLMs vor, die die Entry-Namen `FPATHKONV` bzw. `fpathkonv` enthalten.

Die Adaptermodule gehören zu den nicht vorladbaren Bestandteilen des C-Laufzeitsystems und müssen deshalb in das Anwendungsprogramm eingebunden werden. Sie sind sowohl in der Bibliothek SYSLNK.CRTE als auch in der Bibliothek SYSLNK.CRTE.PARTIAL-BIND enthalten.

Wenn an Stelle der Bibliotheksfunktion die benutzereigene Funktion aufgerufen werden soll, muss das entsprechende Benutzermodul vorrangig vor der Bibliothek SYSLNK.CRTE bzw. SYSLNK.CRTE.PARTIAL-BIND eingebunden werden. Das Benutzermodul kann ein Objektmodul oder ein LLM (mit oder ohne Umwandlung der Kleinbuchstaben und des Unterstrichs) sein.

9.4 Das Tool II-UPDATE

Beim Binden oder Neu-Übersetzen von C++ V3- bzw. C++ 2017-Programmen wird auf vorhandene ii-Informationen (ii=Instanziierungs Informationen) zugegriffen. Diese Informationen enthalten u.a. die Namen von Source-Dateien /Bibliotheken und Source-Elementen, von Include-Bibliotheken und von Listen - bzw CIF-Dateien/Bibliotheken /Elementen. Werden diese Dateien, Bibliotheken und/oder Elemente umbenannt oder unter einer anderen Kennung (oder auch einer anderen cat-id) gehalten, so **müssen** diese Änderungen in den ii-Dateien nachgezogen werden.

ii-Dateien, auch ii-Elemente genannt, werden vom Compiler beim Übersetzen von Programmen erzeugt, die Templates enthalten.
II-UPDATE ist ein compilerunabhängiges Tool mit SDF-Oberfläche, das nach Umbenennung einer Source - bzw. Include- Bibliothek, diese Umbenennung(en) in der ii-Datei nachzieht, ohne dass neu übersetzt werden muss. Pro Aufruf werden die ii-Elemente nur **einer** Bibliothek angepasst.

Einträge in ii-Elementen, die Namen von BS2000- oder POSIX-Dateien oder POSIX-Directories enthalten, können nicht geändert werden, aber II-UPDATE kann diese fehlerfrei anzeigen. Dies gilt sowohl für Source-, Include-, CIF-, als auch für Listing-Einträge.

Die SDF-Syntaxdatei des II-UPDATE wird nicht automatisch aktiviert. Vor der Nutzung des Kommandos muss sie pro Sitzung einmal aktiviert werden. Das Kommando dazu lautet /MODIFY-SDF-OPTIONS *ADD($.SYSSDF.CPP.040.IU.USER).

START-II-UPDATE

```
CONTAINER= *LIBRARY-ELEMENT(...)
    *LIBRARY-ELEMENT(...)
        |   LIBRARY= <filename 1..54>
        |   ,ELEMENT= *ALL / <composed-name 1..64 with-under>(...)
        |       <composed-name 1..64 with-under>(...)
        |           VERSION= *HIGHEST-EXISTING / <composed-name 1..24 with-under>
,OLD-NAME= *LIBRARY-ELEMENT(...) / *LIBRARY(...)
    * LIBRARY-ELEMENT(...)
        |   LIBRARY= <filename 1..54>
        |   ,ELEMENT= <composed-name 1..64 with-under>(...)
        |       <composed-name 1..64 with-under>(...)
        |           |   VERSION= *DEFAULT / <composed-name 1..24 with-under>
    *LIBRARY(...)
        |   LIBRARY= <filename 1..54>
,NEW-NAME= *LIBRARY-ELEMENT(...) / *LIBRARY(...)
    * LIBRARY-ELEMENT(...)
        |   LIBRARY= *SAME / <filename 1..54>
        |   ,ELEMENT= *SAME / <composed-name 1..64 with-under>(...)
        |       <composed-name 1..64 with-under>(...)
        |           |   VERSION=*UNCHANGED / <composed-name 1..24 with-under>
    *LIBRARY(...)
        |   LIBRARY= <filename 1..54>
,CONTEXT= *INCLUDE / *CIF / *SOURCE / *LISTING / *ALL
,ACTION= list-poss(2): *REPLACE / *SHOW
```

CONTAINER = *LIBRARY-ELEMENT(...)
Mit dieser Option wird angegeben, in welcher Objekt-Bibliothek welche ii-Elemente angepasst werden sollen.

LIBRARY = <filename 1..54>
<filename> ist der Name einer PLAM-Bibliothek.

ELEMENT = *ALL
Es sollen alle ii-Elemente in der mit LIBRARY= angegebenen PLAM-Bibliothek angepasst werden. Das sind alle Elemente vom Typ S und einem Namen mit dem Suffix ".II".

ELEMENT = <composed-name 1..64 with-under>(...)
<composed-name> bezeichnet den vollqualifizierten Namen eines ii-Elementes aus der mit LIBRARY= angegebenen PLAM-Bibliothek.

VERSION = *HIGHEST-EXISTING
Enthält die ii-Elementangabe keine Versionsbezeichnung, nimmt II-UPDATE das ii-Element mit der höchsten Version.

VERSION = <composed-name 1..24 with-under>
II-UPDATE nimmt das ii-Element mit der angegebenen Version.

OLD-NAME = *LIBRARY-ELEMENT(...) / *LIBRARY
Mit dieser Option werden der ursprüngliche Bibliotheksname bzw. die ursprünglichen Bibliotheks-Elementnamen (z. B. von Source-Bibliotheks-Elementen) angegeben, die in einer ii-Datei geändert werden sollen. Der Name muss exakt so, wie in der ii-Datei vorgefunden, also ggf. mit `cat-id` und `user-id` angegeben werden (siehe auch Beispiel 6).

LIBRARY-ELEMENT (...)

LIBRARY = <filename 1..54>

<filename> ist der Name der PLAM-Bibliothek, der sich geändert hat bzw. in der geändert wurde.

ELEMENT = <composed-name 1..64 with-under>(...)
<composed-name> bezeichnet den vollqualifizierten Namen eines Elementes, der sich geändert hat.

VERSION = *DEFAULT
Unabhängig vom Kontext wird der im ii-Element vorgefundene Versions-Wert übernommen.

VERSION = <composed-name 1..24 with-under>
Bezeichnet die Version des Elementes, das sich geändert hat.

LIBRARY = <filename 1..54>
<filename> ist der Name einer PLAM-Bibliothek, der sich geändert hat.

NEW-NAME = *LIBRARY-ELEMENT(...) / *LIBRARY
Mit dieser Option wird der neue Bibliotheks- bzw. Elementname für die in OLD-NAME genannten Namen angegeben.

LIBRARY-ELEMENT(...)

LIBRARY = *SAME / <filename 1..54>

Bei *SAME wird der im OLD-NAME Parameter angegebene Name der PLAM-Bibliothek übernommen.
<filename> ist der neue Name der PLAM-Bibliothek, die in der Option OLD-NAME angegeben wurde.

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

Bei *SAME wird der im OLD-NAME Parameter angegebene Element-Name übernommen.
<composed-name> bezeichnet den geänderten, vollqualifizierten Namen eines Elementes, das in der Option OLD-NAME angegeben wurde.

VERSION = *UNCHANGED
Die vorgefundene Version des Bibliothekselementes (im ii-Element) soll nicht geändert werden.

VERSION = <composed-name 1..24 with-under>
Der neue Versionsname des Elementes.

LIBRARY = <filename 1..54>
<filename> ist der neue Name der PLAM-Bibliothek, die in der Option OLD-NAME angegeben wurde.

Hinweis

Es werden nur gleichartige OLD-NAME / NEW-NAME-Kombinationen unterstützt, dh.
OLD-NAME = *LIBRARY / NEW-NAME = *LIBRARY bzw.
OLD-NAME = *LIBRARY-ELEMENT / NEW-NAME = *LIBRARY-ELEMENT

CONTEXT =
Es wird angegeben, welche Teile der ii-Information bearbeitet werden sollen. Diese entsprechen bestimmten Anweisungen an den C++-Compiler.

CONTEXT = *INCLUDE
Bearbeite die Einträge, die der Anweisung //MODIFY-INCLUDE-LIBRARIES entsprechen.

Für *INCLUDE wird die Anpassung des Bibliotheksnamen nur durchgeführt, wenn OLD-NAME und NEW-NAME beides Bibliotheksnamen sind. Sind OLD-NAME und NEW-NAME Bibliotheks-Elementnamen , so wird die Anpassung für *INCLUDE nicht vorgenommen.

CONTEXT = *CIF
Bearbeite die Einträge, die der Anweisung //MODIFY-CIF-PROPERTIES OUTPUT= entsprechen.

CONTEXT = *SOURCE
Bearbeite die Einträge, die der Anweisung //COMPILE SOURCE= entsprechen.

CONTEXT = *LISTING
Bearbeite die Einträge, die der Anweisung //MODIFY-LISTING-PROPERTIES OUTPUT= entsprechen.

CONTEXT = *ALL
Bearbeite alle Einträge.

Hinweis

Es wird eine Meldung ausgegeben, in der die Gesamtzahl(en) der Ersetzungen in den einzelnen Kontexten aufgeführt ist.

ACTION =
Mit dieser Option kann II-UPDATE zur Ersetzung bzw. Ausgabe aller potentiell ersetzbaren Bibliotheksnamen und Bibliotheks-Elementnamen eines ii-Elementes angewiesen werden.

ACTION = *REPLACE
Ersetzungen werden durchgeführt. Es werden nur die durchgeführten Ersetzungen ausgegeben.

ACTION = *SHOW
Es werden keine Ersetzungen durchgeführt. Alle möglichen Ersetzungen werden aber ausgegeben. Ebenso werden Namen von Bibliotheken und Elementen ausgegeben, die nicht ersetzt werden würden.

ACTION = (*SHOW, *REPLACE)
Ersetzungen werden durchgeführt. Zusätzlich zu den durchgeführten Ersetzungen werden auch die nicht durchgeführten Ersetzungen ausgegeben.

Hinweis

> In allen Fällen wird der angegebene Kontext mit berücksichtigt. II-UPDATE bleibt bezüglich Aufruf-, Ersetzungs- und Ausgabe-Verhalten kompatibel zu älteren Versionen.

Einige nachfolgende Beispiele sollen den beschriebenen Sachverhalt weiter erläutern.

Beispiel 1

Die Include-Bibliothek *INC-LIB-V1* wurde in *INC-LIB-V2* umbenannt. Die folgende Anweisung führt die entsprechenden Anpassungen aller in der Bibliothek *MY-OBJ-LIB* enthaltenen ii-Elemente durch.

```
/START-II-UPDATE CONTAINER=*LIB-ELEM(MY-OBJ-LIB, *ALL),-
/OLD-NAME=*LIBRARY(INC-LIB-V1),-
/NEW-NAME=*LIBRARY(INC-LIB-V2),-
/CONTEXT=*INCLUDE
```

Beispiel 2

Die Source-Bibliothek *MY-SOURCE-LIB.V1* wurde in *MY-SOURCE-LIB.V2* umbenannt. Folgende Anweisung führt die entsprechenden Anpassungen des Bibliotheks-Elementes *EL.O.II* in der Bibliothek *MY-OBJ-LIB* durch:

```
/START-II-UPDATE CONTAINER=*LIB-ELEM(MY-OBJ-LIB, EL.O.II),-
/OLD-NAME=*LIBRARY(MY-SOURCE-LIB.V1),-
/NEW-NAME=*LIBRARY(MY-SOURCE-LIB.V2),-
/CONTEXT=*SOURCE
```

Beispiel 3

Die Source-Bibliotheken MY-SOURCE-LIB1 und MY-SOURCE-LIB2 enthalten ein Programmsystem. Die zugehörigen Objekte liegen in den Bibliotheken MY-OBJ-LIB1 und MY-OBJ-LIB2. Die Source-Bibliotheken sollen nach SOURCE-LIB1-V1 bzw. SOURCE-LIB2-V1 umbenannt werden. Weiter wird angenommen, dass die Source-Bibliotheken sich wechselseitig benutzen (inkludieren):

```
/MODIFY-FILE-ATTRIBUTES MY-SOURCE-LIB1, SOURCE-LIB1-V1
/MODIFY-FILE-ATTRIBUTES MY-SOURCE-LIB2, SOURCE-LIB2-V1
```

Die ii-Dateien/Elemente werden durch folgende Anweisungen angepasst:

```
/START-II-UPDATE CONTAINER=*LIB-ELEM(MY-OBJ-LIB1, *ALL),-
/OLD-NAME=*LIBRARY(MY-SOURCE-LIB1),-
/NEW-NAME=*LIBRARY(SOURCE-LIB1-V1),-
/CONTEXT=*ALL

/START-II-UPDATE CONTAINER=*LIB-ELEM(MY-OBJ-LIB1, *ALL),-
/OLD-NAME=*LIBRARY(MY-SOURCE-LIB2),-
/NEW-NAME=*LIBRARY(SOURCE-LIB2-V1),-
/CONTEXT=*INCLUDE

/START-II-UPDATE CONTAINER=*LIB-ELEM(MY-OBJ-LIB2, *ALL),-
/OLD-NAME=*LIBRARY(MY-SOURCE-LIB1),-
/NEW-NAME=*LIBRARY(SOURCE-LIB1-V1),-
/CONTEXT=*INCLUDE

/START-II-UPDATE CONTAINER=*LIB-ELEM(MY-OBJ-LIB2, *ALL),-
/OLD-NAME=*LIBRARY(MY-SOURCE-LIB2),-
/NEW-NAME=*LIBRARY(SOURCE-LIB2-V1),-
/CONTEXT=*ALL
```

Beispiel 4

In dem in Beispiel 3 beschriebenen Programmsystem soll statt der Umbenennung der Source-Bibliotheken das Source-Bibliothekselement *MY-ELEM.C* aus der Bibliothek *MY-SOURCE-LIB1* in die Bibliothek *MY-SOURCE-LIB2* und das zugehörige Objekt *MY-ELEM.O*, sowie das zugehörige ii-Element *MY-ELEM.O.II* nach *MY-OBJ-LIB2* verschoben werden:

```
/LMS
//O-L MY-SOURCE-LIB1,*U
//COP-EL (,MY-ELEM.C,S),(MY-SOURCE-LIB2,MY-ELEM.C)
//DEL-EL (,MY-ELEM.C,S)
//O-L MY-OBJ-LIB1,*U
//COP-EL (,MY-ELEM.O,L),(MY-OBJ-LIB2,MY-ELEM.O)
//COP-EL (,MY-ELEM.O.II,S),(MY-OBJ-LIB2,MY-ELEM.O.II)
//DEL-EL (,MY-ELEM.O.L)
//DEL-EL (,MY-ELEM.O.II,S)
//END
```

Die ii-Dateien/Elemente werden durch folgende Anweisung angepasst:

```
/START-II-UPDATE CONTAINER=*LIB-ELEM(MY-OBJ-LIB2, MY-ELEM.O.II),-
/OLD-NAME=*LIB-ELEM(MY-SOURCE-LIB1, MY-ELEM.C),-
/NEW-NAME=*LIB-ELEM(MY-SOURCE-LIB2, MY-ELEM.C),-
/CONTEXT=*SOURCE
```

Beispiel 5

Die Source-Bibliotheken *MY-SOURCE-LIB1* und *MY-SOURCE-LIB2* enthalten ein Programmsystem (vgl. Beispiel 3). Die zugehörigen Listing-Bibliotheken *MY-LISTING-LIB1* und *MY-LISTING-LIB2* sollen nach *LISTING-LIB1-V1* bzw. *LISTING-LIB2-V1* umbenannt werden:

```
/MODIFY-FILE-ATTRIBUTES MY-LISTING-LIB1, LISTING-LIB1-V1
/MODIFY-FILE-ATTRIBUTES MY-LISTING-LIB2, LISTING-LIB2-V1
```

Die ii-Dateien/Elemente werden durch folgende Anweisungen angepasst:

```
/START-II-UPDATE CONTAINER=*LIB-ELEM(MY-OBJ-LIB1,*ALL),-
/OLD-NAME=*LIBRARY(MY-LISTING-LIB1),-
/NEW-NAME=*LIBRARY(LISTING-LIB1-V1),-
/CONTEXT=*LISTING

/START-II-UPDATE CONTAINER=*LIB-ELEM(MY-OBJ-LIB2,*ALL),-
/OLD-NAME=*LIBRARY(MY-LISTING-LIB2),-
/NEW-NAME=*LIBRARY(LISTING-LIB2-V1),-
/CONTEXT=*LISTING
```

Beispiel 6

Die Include-Bibliothek *HELLO-MAP.INCLIB* wurde in *HELLO-MAP.INCLIB.NEW* umbenannt. Durch die folgende Anweisungsfolge wird zunächst die exakte Schreibweise der Bibliothek ermittelt, die umbenannt werden soll (ACTION=*SHOW; die Werte der Parameter *old-name* und *new-name* sind in diesem Fall bedeutungslos) und danach werden die Ersetzungen mit ACTION=(*SHOW, *REPLACE) durchgeführt:

```
/START-II-UPDATE CONTAINER=*LIB-ELEM(HELLO-MAP.OLIB1,*ALL),-
/OLD-NAME=*LIBRARY(OLIB),-
/NEW-NAME=*LIBRARY(NLIB),-
/CONTEXT=*ALL,-
/ACTION=*SHOW
```

Ausgabe:

```
%  BLS0523 ELEMENT 'II-UPDATE', VERSION '04.0A10', TYPE 'L' FROM LIBRARY ':2OSG:
$TSOS.SYSLNK.CPP.040' IN PROCESS
%  BLS0524 LLM 'II-UPDATE', VERSION '04.0A10' OF '2020-04-02 14:33:19' LOADED
%  BLS0551 COPYRIGHT (C) 2020 Fujitsu Technology Solutions GmbH. ALL RIGHTS RESERVED
%  CDR9992 BEGIN II-UPDATE VERSION 04.0A10
%  CDR9810 Processing ii-element *LIBRARY-ELEMENT(:2OSC:$MTZ.HELLO-MAP.OLIB1,
HELLO-MAP-HP.II(*UPPER-LIMIT),S)
%  CDR9819 include lib no replacement: *LIBRARY(:2OSC:$MTZ.HELLO-MAP.INCLIB)
%  CDR9819 include lib no replacement: *LIBRARY(:2OSC:$MTZ.HELLO-MAP.INCLIB)
%  CDR9819 include lib no replacement: *LIBRARY(:2OSC:$TSOS.SYSLIB.CRTE)
%  CDR9819 source lib no replacement: *LIBRARY(:2OSC:$MTZ.HELLO-MAP.SLIB1)
%  CDR9810 Processing ii-element *LIBRARY-ELEMENT(:2OSC:$MTZ.HELLO-MAP.OLIB1,
HELLO-MAP-UP.II(*UPPER-LIMIT),S)
%  CDR9819 include lib no replacement: *LIBRARY(:2OSC:$MTZ.HELLO-MAP.INCLIB)
%  CDR9819 include lib no replacement: *LIBRARY(:2OSC:$MTZ.HELLO-MAP.INCLIB)
%  CDR9819 include lib no replacement: *LIBRARY(:2OSC:$TSOS.SYSLIB.CRTE)
%  CDR9819 source lib no replacement: *LIBRARY(:2OSC:$MTZ.HELLO-MAP.SLIB1)
%  CDR9814 Summary: 0 include / 0 source / 0 cif / 0 listing replaced
%  CCM0998 CPU TIME USED: 0.1485 SECONDS
```

```
/START-II-UPDATE CONTAINER=*LIB-ELEM(HELLO-MAP.OLIB1,*ALL),-
/OLD-NAME=*LIBRARY(:2OSC:$MTZ.HELLO-MAP.INCLIB),-
/NEW-NAME=*LIBRARY(:2OSC:$MTZ.HELLO-MAP.INCLIB.NEW),-
/CONTEXT=*ALL,-
/ACTION=(*REPLACE,*SHOW)
```

Ausgabe

```
%  BLS0523 ELEMENT 'II-UPDATE', VERSION '04.0A10', TYPE 'L' FROM LIBRARY ':2OSG:
$TSOS.SYSLNK.CPP.040' IN PROCESS
%  BLS0524 LLM 'II-UPDATE', VERSION '04.0A10' OF '2020-04-02 14:33:19' LOADED
%  BLS0551 COPYRIGHT (C) 2020 Fujitsu Technology Solutions GmbH. ALL RIGHTS RESERVED
%  CDR9992 BEGIN II-UPDATE VERSION 04.0A10
%  CDR9810 Processing ii-element *LIBRARY-ELEMENT(:2OSC:$MTZ.HELLO-MAP.OLIB1,
HELLO-MAP-HP.II(*UPPER-LIMIT),S)
%  CDR9819 include lib no replacement: *LIBRARY(:2OSC:$TSOS.SYSLIB.CRTE)
%  CDR9811 include lib replaced: *LIBRARY(:2OSC:$MTZ.HELLO-MAP.INCLIB) -->
*LIBRARY(:2OSC:$MTZ.HELLO-MAP.INCLIB.NEW)
%  CDR9819 source lib no replacement: *LIBRARY(:2OSC:$MTZ.HELLO-MAP.SLIB1)
%  CDR9810 Processing ii-element *LIBRARY-ELEMENT(:2OSC:$MTZ.HELLO-MAP.OLIB1,
HELLO-MAP-UP.II(*UPPER-LIMIT),S)
%  CDR9819 include lib no replacement: *LIBRARY(:2OSC:$TSOS.SYSLIB.CRTE)
%  CDR9811 include lib replaced: *LIBRARY(:2OSC:$MTZ.HELLO-MAP.INCLIB) -->
*LIBRARY(:2OSC:$MTZ.HELLO-MAP.INCLIB.NEW)
%  CDR9819 source lib no replacement: *LIBRARY(:2OSC:$MTZ.HELLO-MAP.SLIB1)
%  CDR9814 Summary: 2 include / 0 source / 0 cif / 0 listing replaced
%  CCM0998 CPU TIME USED: 0.2243 SECONDS
```

10 Literatur

Die Handbücher sind online unter https://bs2manuals.ts.fujitsu.com zu finden.

[1] **C/C++ V4.0A** (BS2000/OSD)
POSIX-Kommandos des C/C++-Compilers
Benutzerhandbuch

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

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

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

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

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

[7] **Tools.h++ V7.0**
User‘s Guide

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

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

[10] **AID** (BS2000)
Advanced Interactive Debugger
Benutzerhandbuch

[11] **BS2000/OSD-BC**
Kommandos
Benutzerhandbuch

[12] **SDF** (BS2000/OSD)
SDF Dialogschnittstelle
Benutzerhandbuch

[13] **BLSSERV**
Bindelader-Starter
Benutzerhandbuch

[14] **BINDER** (BS2000/OSD)
Benutzerhandbuch

[15] **EDT V17.0** (BS2000/OSD)
Anweisungen
Benutzerhandbuch

[16] **LMS** (BS2000/OSD)
SDF-Format
Benutzerhandbuch

[17] **BS2000/OSD**
Makroaufrufe an den Ablaufteil
Benutzerhandbuch

[18] **BS2000/OSD-BC**
Einführung in das DVS
Benutzerhandbuch

[19] **JV (BS2000/OSD)**
Jobvariablen
Beschreibung

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

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

Sonstige Literatur und Standards

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

[23] Die C++-Programmiersprache
(2. und 3. Ausgabe)
von Bjarne Stroustrup

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

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

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

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

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

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