Deutsch

FUJITSU Software BS2000

AID V3.4B

Testen von C/C++-Programmen

Benutzerhandbuch

Ausgabe Juni 2018

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 manuals@ts.fujitsu.com senden.

Nach DIN EN ISO 9001:2015 zertifizierte Dokumentationserstellung

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.

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

Copyright und Handelsmarken

Inhalt

1 Einleitung

Mit AID (Advanced Interactive Debugger) steht im Betriebssystem BS2000 eine leistungsstarke Dialog-Testhilfe zur Verfügung. Fehlerdiagnose, Test und vorläufige Fehlerbehebung aller im BS2000 erstellten Programme können Sie mit AID wesentlich schneller und mit weniger Aufwand durchführen als mit anderen Mitteln, wie z.B. dem Einfügen von Testhilfe-Anweisungen im Programm. AID ist permanent verfügbar und besitzt eine hohe Anpassungsfähigkeit an die jeweilige Programmiersprache. Ein Programm, das Sie mit AID getestet haben, muss nicht immer erneut übersetzt werden, sondern kann sofort in den produktiven Einsatz gehen. Der Funktionsumfang von AID und seine Testsprache, die AID-Kommandos, sind primär auf die Dialoganwendung zugeschnitten. AID kann aber ebenso gut im Batch-Betrieb eingesetzt werden. AID bietet Ihnen vielfältige Möglichkeiten zur Ablaufüberwachung und Ablaufsteuerung sowie zur Ausgabe und Änderung von Speicherinhalten. Außerdem können Sie Informationen über den Programmablauf und zur Handhabung von AID abfragen.

Mit AID können Sie sowohl auf der symbolischen Ebene der jeweiligen Programmiersprache als auch auf Maschinencode-Ebene testen. Beim „symbolischen Testen“ eines C/C++-Programms können Sie Anweisungen, Funktionen und Daten mit den im Quellcode vereinbarten Namen ansprechen und die Anweisungen ohne Namen mit der vom Compiler erzeugten Source-Referenz.

Die BS2000-Kommandos, die in der AID-Dokumentation vorkommen, sind im SDF-Format (System Dialog Facility), EXPERT-Form beschrieben. SDF ist die Dialogschnittstelle zum BS2000. Die SDF-Kommandosprache löst die bisherige Kommandosprache im ISP-Format ab.

Mit AID können Sie neben reinen BS2000-Programmen auch reine POSIX-Programme und Mixed-Mode-Programme testen. Reine POSIX-Programme laufen vollständig in der POSIX-Shell ab. Mit Mixed-Mode-Programmen werden BS2000-Programme bezeichnet, die POSIX-Schnittstellen benutzen.

Die in diesem Handbuch beschriebenen Möglichkeiten, Daten und Anweisungen eines C++-Programms anzusprechen, setzen den C/C++-Compiler ab V3.0 voraus.

Eine Gegenüberstellung der wesentlichsten Unterschiede beim Testen von Programmen, die mit dem C/C++-Compiler ab V3.0 übersetzt wurden, und älteren Objekten finden Sie im Anhang.

Standardmäßig können die mit dem neuen C/C++-Compiler übersetzten Programme source-basiert auf einer grafischen Oberfläche am PC getestet werden. Das grafische Testen ist wesentlich komfortabler, da Ihnen stets der aktuell ausgeführte Programmausschnitt am Bildschirm angezeigt wird und Sie AID-Kommandos per Mausklick eingeben können.

1.1 Zielsetzung und Zielgruppen der AID-Dokumentation

AID wendet sich an alle Software-Entwickler, die im BS2000 mit den Programmiersprachen COBOL, FORTRAN, C, C++, PL/I oder ASSEMBH arbeiten oder Programme auf Maschinencode-Ebene testen oder korrigieren wollen. Dieses Handbuch wendet sich an die Tester von C- und C++-Programmen.

1.2 Konzept der AID-Dokumentation

Die Dokumentation von AID besteht aus einem Basishandbuch und den sprachspezifischen Handbüchern für das symbolische Testen sowie dem Handbuch für das Testen auf Maschinencode-Ebene. Zusätzlich liegt für den geübten AID-Anwender als Nachschlagewerk ein Tabellenheft vor, in dem die Syntax aller Kommandos und die Operanden mit kurzen Erläuterungen aufgeführt sind. Außerdem gibt es darin die %SET-Tabellen und eine Gegenüberstellung AID - IDA. Zusammen mit dem Basishandbuch enthält das Handbuch für die von Ihnen gewählte Sprache alle Informationen, die Sie zum Testen brauchen. Das Handbuch für das Testen auf Maschinencode-Ebene kann statt oder zusätzlich zu einem der sprachspezifischen Handbücher eingesetzt werden.

AID - Basishandbuch [1]

Im Basishandbuch finden Sie einen Überblick über AID und die Beschreibung der Sachverhalte und Operanden, die für alle Programmiersprachen gleich sind. Im Überblick wird die BS2000-Umgebung beschrieben, es werden die grundlegenden Begriffe erläutert und der AID-Kommandovorrat vorgestellt. Die anderen Kapitel beschreiben die Testvorbereitung, die Kommandoeingabe, die Operanden Subkommando, komplexe Speicherreferenz und Medium-und-Menge, die AID-Literale und die Schlüsselwörter. Das Handbuch enthält außerdem die in Kommandofolgen unzulässigen BS2000-Kommandos.

AID-Benutzerhandbücher

In den Benutzerhandbüchern finden Sie die Kommandos in alphabetischer Reihenfolge. Alle einfachen Speicherreferenzen sind in diesen Handbüchern beschrieben. Neben dem vorliegenden Handbuch
AID - Testen von C- und C++-Programmen

gibt es noch die Benutzerhandbücher
AID - Testen von COBOL-Programmen [3]
AID - Testen von FORTRAN-Programmen [4]
AID - Testen unter Posix [5]
AID - Testen von ASSEMBH-Programmen [6]

In diesen sprachspezifischen Handbüchern ist die Beschreibung der Operanden auf die jeweilige Programmiersprache zugeschnitten. Es wird vorausgesetzt, dass Ihnen der Sprachumfang und die Handhabung des entsprechenden Compilers geläufig sind.

Die zusätzlichen Möglichkeiten des „maschinennahen Testens“ sind beschrieben in
AID - Testen auf Maschinencode-Ebene [2]

Das Handbuch für das Testen auf Maschinencode-Ebene können Sie für Programme einsetzen, zu denen keine LSD-Sätze vorhanden sind oder für die die Informationen aus dem symbolischen Testen zur Diagnose nicht ausreichen. Beim Testen auf Maschinencode-Ebene sind Sie bei der Anwendung der AID-Kommandos unabhängig von der Programmiersprache, in der das Programm erstellt wurde.

Readme-Datei

Funktionelle Änderungen der aktuellen Produktversion und Nachträge zu diesem Handbuch entnehmen Sie bitte ggf. der produktspezifischen Readme-Datei.

Readme-Dateien stehen Ihnen online bei dem jeweiligen Produkt zusätzlich zu den Produkthandbüchern unter *http://manuals.ts.fujitsu.com* zur Verfügung. Alternativ finden Sie Readme-Dateien auch auf der Softbook-DVD.

Informationen unter BS2000

Wenn für eine Produktversion eine Readme-Datei existiert, finden Sie im BS2000-System die folgende Datei:

```
SYSRME.<product>.<version>.<lang>
```

Diese Datei enthält eine kurze Information zur Readme-Datei in deutscher oder englischer Sprache (<lang>=D/E). Die Information können Sie am Bildschirm mit dem Kommando `SHOW-FILE` oder mit einem Editor ansehen.
Das Kommando `/SHOW-INSTALLATION-PATH INSTALLATION-UNIT=<product>` zeigt, unter welcher Benutzerkennung die Dateien des Produkts abgelegt sind.

Ergänzende Produkt-Informationen

Aktuelle Informationen, Versions-, Hardware-Abhängigkeiten und Hinweise für Installation und Einsatz einer Produktversion enthält die zugehörige Freigabemitteilung. Solche Freigabemitteilungen finden Sie online unter *http://manuals.ts.fujitsu.com*.

1.3 Änderungen gegenüber dem Vorgänger-Handbuch

AID V3.4B30 bietet gegenüber der Version V3.4B10 folgende neue Funktionalität:

- Erweiterung des %AID-Kommandos: neuer Operand *LEV*. Dieser Operand kann die Ausgabe des AID-Kommandos %SDUMP %NEST um die Ebenen innerhalb der Aufrufhierarchie erweitern.
- Neue Qualifikation *NESTLEV* in den Kommandos %DISPLAY, %MOVE, %SDUMP und %SET zur Qualifikation aller Instanzen rekursiver Daten.
- Erweiterung des %FIND-Kommandos, mit der es möglich wird, den *find-bereich* nach Zeichen aus einem von XHCS unterstützten Coded Character Set (CCS) zu durchsuchen.

1.4 Darstellungsmittel

kursiv	Im Fließtext werden Operanden in *kursiven kleinbuchstaben* geschrieben.
halbfett	Hervorhebungen werden ***halbfett*** gekennzeichnet. Außerdem werden in den Syntax-Notationen Sonderzeichen und Kleinbuchstaben, die Sie unverändert übernehmen müssen, halbfett gedruckt, um sie von den Zeichen der Metasyntax bzw. den Operandennamen zu unterscheiden. Beispiele sind die eckigen Klammern `[]`, die in C/C++ den Index eines Vektors einschließen oder auch der `sizeof()`-Operator, der stets in Kleinbuchstaben eingegeben werden muss.
	Mit diesem Symbol werden Stellen gekennzeichnet, die Sie besonders beachten sollten, z.B wenn bei gleicher Syntax von AID und C/C++ unterschiedliche Adressen errechnet werden, weil sich AID und C/C++ bei der Behandlung einzelner Adressoperanden unterscheiden, etc.
Farbe	Im Fließtext und in der Syntax werden Informationen, die nur C++ betreffen, farbig dargestellt, wenn es sich nicht um ganze Kapitel oder Abschnitte handelt, die sich ausdrücklich nur mit C++ befassen.

2 Metasyntax

Für die Darstellung der Kommandos wird folgende Metasyntax verwendet. Die Aufstellung zeigt die verwendeten Symbole und beschreibt ihre Bedeutung.

GROSSBUCHSTABEN
: Zeichenfolge, die Sie unverändert übernehmen müssen, wenn Sie eine Funktion auswählen.

kleinbuchstaben
: Zeichenfolge, die eine Variable bezeichnet. An ihre Stelle müssen Sie einen der zugelassenen Operandenwerte setzen.

```
{ alternativ }
{    ...     }
{ alternativ }
```

```
{alternativ | ... | alternativ}
```

: Alternativen, unter denen Sie eine auswählen müssen. Die beiden Formate sind gleichbedeutend.

```
[wahlweise]
```

: Die in eckige Klammern eingeschlossenen Angaben können entfallen.
Bei AID-Kommandonamen kann der in eckigen Klammern stehende Teil nur komplett entfallen, andere Abkürzungen ergeben einen Syntaxfehler.

```
[...]
```

: Wiederholbarkeit einer wahlfreien syntaktischen Einheit. Muss vor jede Wiederholung ein Trennzeichen, z.B. Komma gesetzt werden, steht es vor den Wiederholungspunkten.

```
{...}
```

: Wiederholbarkeit einer syntaktischen Einheit, die einmal angegeben werden muss. Muss vor jede Wiederholung ein Trennzeichen, z.B. Komma gesetzt werden, steht es vor den Wiederholungspunkten.

Unterstreichung

Die Unterstreichung kennzeichnet den Standardwert, den AID einsetzt, wenn Sie für einen Operanden keinen Wert angeben.

- Der dickere Punkt trennt Qualifikationen oder er steht für eine *vorqualifikation* (siehe %QUALIFY) oder er ist der Operator für einen Adressversatz oder er ist Teil des Durchlaufzählers bzw. Subkommandonamens. Eingegeben wird der dickere Punkt mit dem normalen Punkt, der auf der Tastatur ist. Er wurde nur der besseren Lesbarkeit wegen dicker dargestellt.

3 Voraussetzungen zum Testen

Für das symbolische Testen benötigt AID eine „List for Symbolic Debugging“ (LSD), in der die in einem Programm definierten symbolischen Namen verzeichnet sind. Diese LSD-Informationen werden vom Compiler erzeugt und können beim Binden übernommen und auch mitgeladen werden. In diesem Kapitel sind die Steueranweisungen für die Erzeugung der LSD durch den C/C++-Compiler auf BS2000- und auf POSIX-Ebene kurz beschrieben. Zusätzlich werden in den folgenden Abschnitten auch diejenigen Operanden aufgeführt, die Sie beim Übersetzen, Binden und Laden angeben müssen, um ein POSIX-fähiges Programm zu erzeugen und zum Ablauf zu bringen. Im AID-Basishandbuch, Kapitel „Voraussetzungen zum Testen mit AID“, finden Sie allgemeine Informationen zu LSD-Sätzen, zum Binden, Laden und Starten.

Außerdem gibt es bei AID die Möglichkeit, die LSD bei Bedarf nachzuladen, wenn das Programm zunächst ohne LSD geladen wurde. Dazu muss die LSD mit dem zugehörigen Programm in einer PLAM-Bibliothek stehen. Dort kann sie entweder direkt vom Compiler beim Übersetzen abgelegt worden sein, oder, falls das Programm unter POSIX übersetzt wurde, können Sie es mit den LSD-Sätzen aus dem POSIX-Dateisystem in eine PLAM-Bibliothek kopieren. Zum POSIX-Kommando `bs2cp`, das Sie zum Übertragen des Programms aus POSIX ins BS2000 benötigen, finden Sie in diesem Kapitel eine Kurzbeschreibung.
Der letzte Abschnitt dieses Kapitels enthält eine Zusammenstellung von Kommandos, mit denen Sie eineTestsitzung stets starten sollten.

3.1 Übersetzen in BS2000

Beim Übersetzen mit dem C/C++-Compiler V3.0 steuern Sie das Erzeugen der LSD-Informationen mit folgender Option:

```
//MODIFY-TEST-PROPERTIES TEST-SUPPORT = {*UNCHANGED|*YES|*NO}
```

*UNCHANGED
: Der zuletzt mit einer MODIFY-TEST-PROPERTIES-Anweisung vereinbarte Wert wird übernommen. Wurde in diesem Übersetzungslauf noch kein Wert festgelegt, so gilt *NO.

*YES
: Der Compiler erzeugt LSD-Informationen.

NO
: Bei der Voreinstellung NO erzeugt der Compiler keine LSD-Informationen. Auch ohne diese LSD-Informationen ist eine Rückverfolgung der Aufrufhierarchie (%SDUMP %NEST) möglich.

Die Erzeugung der LSD ist nur für nicht optimierte Programme möglich. Sollte die Optimierung dennoch eingeschaltet sein (vgl. Anweisung MODIFY-OPTIMIZATION-PROPERTIES), schaltet der Compiler die Optimierungsstufe auf *LOW und gibt eine entsprechende Meldung aus.
Außerdem wirkt sich die Erzeugung der LSD für C++-Programme auf die Generierung von Funktionen aus. Inline-Funktionen werden als Outline-Funktionen generiert. Eine ggf. angegebene Option INLINING=*YES wird vom Compiler auf INLINING=*NO zurückgesetzt.

In der Compiler-Anweisung, die das Binden des Moduls steuert, müssen Sie ebenfalls dafür sorgen, dass die LSD-Informationen mitgeführt werden, falls Sie nicht planen, die LSD-Informationen erst bei Bedarf mit %SYMLIB nachzuladen:

```
//MODIFY-BIND-PROPERTIES ...,TEST-SUPPORT = {*UNCHANGED|*YES|*NO}
```

*UNCHANGED
: Der zuletzt mit einer MODIFY-TEST-PROPERTIES-Anweisung vereinbarte Wert wird übernommen. Wurde in diesem Übersetzungslauf noch keine MODIFY-TEST-PROPERTIES-Anweisung angegeben, so gilt *NO.

*YES
: Die LSD-Informationen werden zum Modul dazugebunden.

*NO
: Bei der Voreinstellung *NO werden die LSD-Informationen nicht miteingebunden.

Eine weitere Option der Anweisung MODIFY-BIND-PROPERTIES, die sich auf das Testen mit AID auswirkt, ist STDLIB. Diese hat standardmäßig den Wert *DYNAMIC, was bedeutet, dass das C-Laufzeitsystem dynamisch nachgeladen wird. Bei bestimmten Programmfehlern, wenn zum Beispiel Teile des Codes von Bibliotheksfunktionen überschrieben werden, kann AID die Aufrufhierarchie nicht vollständig ausgeben, es fehlt u.U. die letzte Funktion vor dem Auftreten des Fehlers. In diesem Fall können Sie sich behelfen, indem Sie beim Binden STDLIB=*STATIC angeben. Das Laufzeitsystem wird dadurch statisch zum Programm dazugebunden (siehe auch Benutzerhandbuch „C/C++-Compiler“).

Die folgenden beiden Optionen müssen Sie angeben, falls das Programm POSIX-Schnittstellen benutzt:

– Vor Auftreten der ersten `#include`-Anweisung im Programm muss das Define `_OSD_POSIX` gesetzt sein. Dies erreichen Sie am einfachsten, indem Sie bei der Übersetzung die folgende Option angeben:
```
//MODIFY-SOURCE-PROPERTIES DEFINE = _OSD_POSIX
```

– Für die Suche der Standard-Include-Header müssen Sie beim Übersetzen zusätzlich zur CRTE-Bibliothek `SYSLIB.CRTE` die Bibliothek `SYSLIB.POSIX-HEADER` angeben, die die Standard-Include-Elemente für die POSIX-Funktionen enthält.
Dazu verwenden Sie die Option:
```
//MODIFY-INCLUDE-LIBRARIES STD-INCLUDE-LIBRARY=
  (*STD-LIBRARY,$.SYSLIB.POSIX-HEADER)
```

Wenn das Programm, wie in Unix-Systemen üblich, Parameter für die `main`-Funktion einlesen soll, muss beim Übersetzen die folgende Option gesetzt sein:

```
//MODIFY-RUNTIME-PROPERTIES PARAMETER-PROMPTING = *YES
```

Damit veranlassen Sie, dass das Programm unmittelbar nach dem Starten unterbrochen wird und auf die Eingabe von Parametern für die `main`-Funktion oder auf Umweisungen von `stdin/stdout` oder `stderr` wartet. Wird das Programm in der POSIX-Shell gestartet, dann hat die Angabe dieses Operanden keine Bedeutung, da Parameter und Umweisungen wie in Unix-Systemen direkt in der Kommandozeile angegeben werden.

Eine vollständige Darstellung der Operanden, die die Übersetzung steuern, finden Sie im C/C++-Benutzerhandbuch [8].

3.2 Binden, Laden und Starten in BS2000

Auch beim Binden, Laden und Starten müssen Sie darauf achten, dass stets die LSD-Informationen mitgeführt werden, damit Sie später symbolisch testen können.

Die für alle Sprachen gültigen SDF-Kommandos, mit denen Sie das Programm binden, laden und starten können, sind im AID-Basishandbuch, Kapitel „Voraussetzungen zum Testen mit AID“, beschrieben. Dort ist auch der jeweilige Parameter beschrieben, mit dem Sie veranlassen, dass die vom Compiler erzeugten LSD-Informationen an den Binder (BINDER) bzw. Dynamischen Bindelader DBL weitergereicht werden. Zusätzlich gibt es die Möglichkeit des Nachladens von LSD aus einer PLAM-Bibliothek mit Hilfe des Kommandos %SYMLIB (siehe Abschnitt „Nachladen der LSD“ auf Seite 18).

Wenn Sie die POSIX-Funktionen des C-Laufzeitsystems verwenden wollen, müssen Sie beim Binden die „Bindeschalter“-Bibliothek `SYSLNK.CRTE.POSIX` angeben. Das darin enthaltene Modul muss vorrangig vor Modulen anderer CRTE-Bibliotheken eingebunden werden. Daher müssen Sie beim dynamischen Binden mit dem DBL der Bibliothek `SYSLNK.CRTE.POSIX` einen niedrigeren Linknamen BLSLIB*nn* zuweisen als nachfolgenden weiteren CRTE-Bibliotheken.

Beispiel

```
ADD-FILE-LINK FILE-NAME=$.SYSLNK.CRTE.POSIX,LINK-NAME=BLSLIB00
ADD-FILE-LINK FILE-NAME=$.SYSLNK.CRTE.PARTIAL-BIND,LINK-NAME=BLSLIB01
LOAD-PROGRAM ...
```

Wenn Sie statisch mit dem BINDER binden und die Bibliothek `SYSLNK.CRTE.POSIX` mit einer `INCLUDE-MODULES`-Anweisung einbinden, ist sichergestellt, dass das Modul aus der „Bindeschalter“-Bibliothek vor den Modulen des Laufzeitsystems gebunden wird:
`INCLUDE-MODULES *LIB(LIB = $.SYSLNK.CRTE.POSIX, ELEM = *ALL)`

Weitergehende Informationen zur gemeinsamen Laufzeitumgebung CRTE finden Sie im Handbuch „CRTE - Common RunTime Environment“ [12].

3.3 Übersetzen und Binden unter POSIX

In der POSIX-Shell stehen Ihnen zum Übersetzen und Binden von C- oder C++-Programmen die folgenden POSIX-Kommandos zur Verfügung:

cc, c89 Aufruf des Compilers als C-Compiler

CC Aufruf des Compilers als C++-Compiler

Der C/C++-Compiler erzeugt LSD-Informationen, wenn Sie die Option `-g` angeben. Außerdem beinhaltet diese Option, dass Inline-Funktionen im C/C++-Quellprogramm nicht expandiert und dass keine Standardoptimierungen (`-O`) durchgeführt werden.

Ohne Angabe von `-g` können Sie das Programm nicht symbolisch testen. Das Programm kann jedoch auf Maschinencode-Ebene getestet werden.

Im Handbuch „POSIX-Kommandos des C/C++-Compilers“ [9] sind die Kommandos `cc`, `c89` und `CC` ausführlich beschrieben.

3.4 Laden und Starten unter POSIX

Um das Programm mit LSD zu laden, verwenden Sie das POSIX-Kommando `debug`. Es ist im Kapitel „POSIX-Kommando debug“ auf Seite 303 ausführlich beschrieben. Nach dem Laden gibt AID die Meldung AID0348 aus, der Sie die Prozessnummer (pid) des erzeugten Prozesses entnehmen können. Anschließend werden Sie mit dem Prompt des Testmodus zur Eingabe von AID-Kommandos aufgefordert. Mit %RESUME können Sie das Programm starten.

Wenn Sie das Programm in der POSIX-Shell direkt laden und starten, also ohne das `debug`-Kommando zu verwenden, wird das Programm bei Fehlerabbruch entladen. Sie haben dann im Gegensatz zur BS2000-Ebene keine Möglichkeit, die Fehlerumgebung und Fehlerursache sofort zu untersuchen und ggf. den Fehler zu beheben und das Programm weiterlaufen zu lassen.

3.5 Nachladen der LSD

Programme im Produktiveinsatz sind in der Regel ohne LSD geladen. Auch bei sehr großen Programmen, bei denen nur einzelne Module symbolisch getestet werden sollen, ist es sinnvoll, das Programm ohne LSD zu laden. In diesen Fällen kann AID nachträglich auf die zugehörige LSD zugreifen, falls das Modul zusammen mit der LSD in einer PLAM-Bibliothek abgespeichert wurde. Dazu müssen Sie im %SYMLIB-Kommando (siehe Seite 290) die PLAM-Bibliothek angeben, die das Programm mit den LSD-Informationen enthält. Wenn Sie daraufhin in einem AID-Kommando eine symbolische Speicherreferenz ansprechen, öffnet AID die PLAM-Bibliothek und sucht darin die benötigten Informationen. Diese Vorgehensweise können Sie auch anwenden, wenn das Programm in der POSIX-Shell abläuft. Da %SYMLIB den Zugriff auf POSIX-Dateien nicht unterstützt, muss auch in diesem Fall das Programm mit der LSD im BS2000 in einer PLAM-Bibliothek abgespeichert sein. Wurde das Programm in der POSIX-Shell übersetzt, müssen Sie das erzeugte Objekt mit dem POSIX-Kommando `bs2cp` ins BS2000 kopieren und dort als Element vom Typ L in einer PLAM-Bibliothek ablegen.

Bei Programmen, die über einen `exec()`-Aufruf aus einem anderen Programm heraus geladen und gestartet werden, kann die LSD generell nicht mitgeladen werden. Hier müssen Sie also stets das hier beschriebene Verfahren anwenden, wenn Sie symbolisch testen wollen.

3.6 Beschreibung zu bs2cp

`bs2cp` kopiert Dateien vom POSIX-Dateisystem in das BS2000 und umgekehrt. BS2000-Dateien können DVS-Dateien oder Elemente von BS2000-PLAM-Bibliotheken sein. Die vollständige Beschreibung von `bs2cp` steht im Handbuch „POSIX-Kommandos“ [11].

3.7 Kommandos zu Beginn einer Testsitzung

Beim Testen von C/C++-Programmen empfiehlt es sich, zu Beginn jeder Testsitzung das folgende Kommando einzugeben:

```
%AID C=YES
```

Damit schalten Sie die Behandlung von `char`-Vektoren als C-Strings ein. Sie können dann in AID mit C-Strings arbeiten, wie Sie es von C/C++ her gewohnt sind. Gleichzeitig beinhaltet die Einstellung `C=YES` das Einschalten von `LOW=ALL` und `SYMCHARS=NOSTD`.

– `LOW=ALL` bedeutet, dass AID die Groß- und Kleinschreibung in Namen aus dem Quellprogramm unterscheidet und auch kleingeschriebene Namen von Übersetzungseinheiten und andere BLS-Namen nicht in Großbuchstaben umsetzt. Alle anderen Angaben in BS2000- und AID-Kommandos werden wie gewohnt in Großbuchstaben konvertiert. Sie können daher Kommando- und Operandennamen und alle anderen Eingaben weiterhin mit Kleinbuchstaben eingeben. Falls Sie jedoch nicht unter POSIX testen, ist es vorteilhafter, `%AID LOW=ON` einzustellen. Sie müssen dann nicht darauf achten, Namen von Übersetzungseinheiten mit Großbuchstaben einzugeben.

– `SYMCHARS=NOSTD` bedeutet, dass der Bindestrich stets als Minuszeichen interpretiert wird. Da in C und C++ Bindestriche in Namen nicht zulässig sind, kann ein Bindestrich in einer Eingabe also nur ein Minuszeichen darstellen.

Die Angabe `%AID C=NO` beeinflusst jedoch die Einstellungen von LOW und SYMCHARS nicht, d.h. implizit durch `C=YES` gesetzte Werte `LOW=ALL` und `SYMCHARS=NOSTD` werden durch `C=NO` nicht zurückgenommen.
Während einer Testsitzung können Sie sich mit `%SHOW %AID` die aktuellen Einstellungen globaler Parameter ausgeben lassen (siehe Beschreibung des Kommandos %SHOW, Seite 284).

Unmittelbar nach dem Laden steht der Befehlszähler (PC) im Superblock. Deshalb können Sie nur globale und static vereinbarte Daten ansprechen. Zum Zugriff benötigt AID die entsprechende Qualifikation. Erst wenn der Befehlszähler auf dem ersten Befehl Ihres Programms steht, gibt es eine aktuelle Aufrufhierarchie und AID kann lokale Daten ansprechen und das Kommando %SDUMP ausführen. Diese Stelle erreichen Sie mit:

```
%insert main;%r
```

oder

```
%trace 1 in s=srcname
```

srcname ist der Name der Übersetzungseinheit, in der die `main`-Funktion enthalten ist.

Für C-Programme und C++-Programme ohne virtuelle Funktionen oder Konstruktoren sind beide Möglichkeiten gleichbedeutend: das Programm wird vor der ersten ausführbaren Anweisung der `main`-Funktion unterbrochen.

In der Regel beginnen C++-Programme jedoch mit einer vom Compiler erzeugten Funktion, in der z.B. Konstruktoren abgehandelt und Tabellen für virtuelle Funktionen aufgebaut werden. Nach dem %TRACE ist das Programm dann am Anfang dieser Funktion unterbrochen. Sie hat den vom Compiler erzeugten Namen _ _STI_ _, der auch in der STOP-Meldung des %TRACE ausgegeben wird. Um auch nach dem Laden eines C++-Programms stets unmittelbar vor der ersten ausführbaren Anweisung der main-Funktion anzuhalten, können Sie das folgende Kommando verwenden:

```
%trace 1 in main
```

Um eine längere AID-Ausgabe mit der K2-Taste unterbrechen zu können, muss mit dem Kommando /MODIFY-TERMINAL-OPTIONS
folgende Option eingestellt sein:
OVERFLOW-CONTROL=*USER-ACKNOWLEDGE

4 Adressierung in C- und C++-Programmen

In diesem Kapitel werden nur die Speicherreferenzen beschrieben, die für das symbolische Testen von C- und auch von C++-Programmen verwendet werden. Im Kapitel 5 sind zusätzlich Adressoperanden beschrieben, die nur für C++-Programme verwendbar sind. Eine allgemeine Beschreibung der Adressierung finden Sie im AID-Basishandbuch, Kapitel „Adressierung in AID“. Als symbolische Speicherreferenzen können Sie alle in den LSD-Sätzen verzeichneten Namen von Daten und Anweisungen aus dem Programm und die vom Compiler erzeugten Source-Referenzen verwenden. Eventuell sind davor Qualifikationen erforderlich, wie sie im Anschluss beschrieben sind.

In allen Operanden, in denen *kompl-speicherref* möglich ist, können Sie beliebig wechseln zwischen den in diesem Handbuch beschriebenen Speicherreferenzen und denen für das Testen auf Maschinencode-Ebene [2].

4.1 Qualifikationen

Qualifikationen verwenden Sie in den folgenden Fällen:

– wenn Sie ein Speicherobjekt ansprechen wollen, das nicht im aktuellen AID-Arbeitsbereich liegt,

– wenn die Unterbrechungsstelle nicht im Gültigkeitsbereich des adressierten Speicherobjekts liegt,

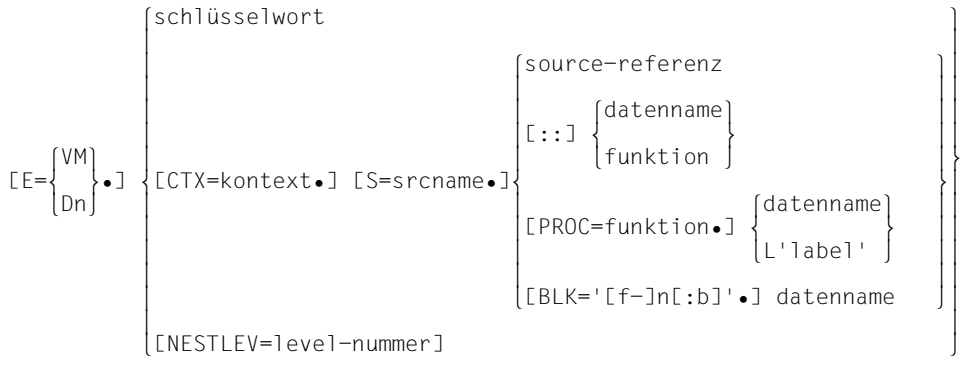
– wenn das gewünschte Speicherobjekt durch eine gleichnamige Definition verdeckt ist

– oder um einen zusammenhängenden Teilbereich des Pogrammspeichers zu bezeichnen.

Es gibt die Basisqualifikation, mit der Sie den AID-Arbeitsbereich vereinbaren und die Bereichsqualifikationen, mit denen Sie Teile des Arbeitsbereichs adressieren. Außerdem beschreiben Sie durch die Verbindung von Qualifikationen den Pfad zu einem Bereich bzw. einem Speicherobjekt.

Qualifikationen werden durch Punkte getrennt. Zwischen der letzten Qualifikation und einem anschließenden Operanden muss ebenfalls ein Punkt stehen. Eine Ausnahme davon bildet die Qualifikation für den Superblock, bei der Sie dem Adressoperanden zwei Doppelpunkte voranstellen; zwischen `::` und Adressoperand wird dann kein Punkt eingefügt.

Beim Testen von C- und C++-Programmen können Sie die Basisqualifikation und als Bereichsqualifikationen die S-, PROC- und BLK-Qualifikation verwenden. Globale Daten und Funktionen werden durch Voranstellen von `::` angesprochen.
Qualifikationen werden in der Syntax der Kommandos mit dem Operanden *qua* dargestellt. In der folgenden Übersicht wird gezeigt, wie Sie Qualifikationen einsetzen können:



Basisqualifikation

E={VM | Dn}
: Die Basisqualifikation legt fest, ob der AID-Arbeitsbereich im geladenen Programm (`E=VM`) oder in einer Dump-Datei (`E=Dn`) liegen soll. Die Basisqualifikation wird beim symbolischen Testen und beim maschinennahen Testen gleich verwendet und ist im AID-Basishandbuch, Kapitel „Adressierung in AID“ [1] und im Kommando %BASE, Seite 132 beschrieben.

Bereichsqualifikationen

Mit diesen Qualifikationen bezeichnen Sie einen Teil des Arbeitsbereiches. Endet ein Adressoperand mit einer dieser Qualifikationen, so wirkt das Kommando nur in dem Teil, der mit der letzten Qualifikation bezeichnet wurde. Mit einer Bereichsqualifikation begrenzen Sie den Wirkungsbereich eines Kommandos oder machen damit einen Daten- oder Anweisungsnamen im Arbeitsbereich eindeutig, oder Sie erreichen damit einen Namen, der an der aktuellen Unterbrechungsstelle sonst nicht ansprechbar ist.

CTX=kontext

Die CTX-Qualifikation bezeichnet einen Kontext (siehe AID Basishandbuch, Abschnitt „Bereichsqualifikationen“). Nur in den Kommandos %SDUMP und %QUALIFY kann ein Adressoperand mit der CTX-Qualifkation enden. Diese Qualifikation ist nur erforderlich, wenn gleichnamige Übersetzungseinheiten in verschiedenen Kontexten geladen sind und die gewünschte Übersetzungseinheit nur über die CTX-Qualifikation eindeutig ansprechbar ist. *kontext* ist der im `BIND`-Makro explizit vergebene Name des Kontextes oder der implizit vergebene Name `LOCAL#DEFAULT`. Auch mit dem DBL geladene Programme erhalten den standardmäßig vergebenen Kontextnamen `LOCAL#DEFAULT`. Weitere Kontexte eines Programms können durch Anschließen an ein Shared-Code-Programm entstehen. *kontext* kann bis zu 32 Stellen lang sein.

Um die Syntax für die Adressoperanden der einzelnen Kommandos nicht weiter aufzublähen, wurde die CTX-Qualifikation dort nicht aufgenommen.

Beispiele

```
%control in ctx=local#default.s=n'list.c'.proc=main
```

Der *control-bereich* liegt hier nicht im aktuellen Kontext, in dem das Programm unterbrochen wurde, sondern im Kontext `LOCAL#DEFAULT`.

```
%sdump ctx=ctxphase
```

Die aktuelle Unterbrechungsstelle liegt in einem anderen Kontext der Aufrufhierarchie. In diesem %SDUMP begrenzen Sie das Kommando auf den angegebenen Kontext.

```
%insert ctx=local#default.s=n'list.c'.s'30'
```

Die Übersetzungseinheit `LIST.C` ist sowohl im aktuellen Kontext als auch im Kontext `LOCAL#DEFAULT` vorhanden. Um den Testpunkt vereinbaren zu können, brauchen Sie die Kontext-Qualifikation.

S=srcname

Die S-Qualifikation bezeichnet eine Übersetzungseinheit.

Bei LLMs wird der Name der Quelldatei angegeben. Er kann bis zu 32 Stellen lang sein. Der C/C++-Compiler V3.0 erzeugt ausschließlich LLMs.

Bei OMs ist der *srcname* der Name der Code-CSECT und somit maximal 8 Stellen lang.

Enthält der Name Sonderzeichen, die nicht zum AID-Zeichenvorrat gehören, wie z.B. einen Punkt oder ein ‚&‘, so muss die S-Qualifikation mit `n'`*srcname*`'` angegeben werden. Weitere Informationen zur Bildung der Modulnamen finden Sie im C/C++-Benutzerhandbuch im Abschnitt „Standardnamensgenerierung“ [8].

srcname wird von AID auch bei der Voreinstellung `%AID LOW[=ON]` immer in Großbuchstaben umgesetzt.

Wenn das Programm in der POSIX-Shell übersetzt wurde und der Name der zugehörigen Quelldatei Kleinbuchstaben enthält, müssen Sie dagegen die Einstellung `%AID LOW=ALL` einschalten. Nur dann wird auch in der S-Qualifikation die Groß-/Kleinschreibung berücksichtigt. Implizit wird `LOW=ALL` gesetzt, wenn Sie `%AID C=YES` eingegeben.

Mit der S-Qualifikation können Sie den Bereich angeben, in dem die Kommandos %CONTROL*n*, %TRACE oder %SDUMP wirken.
Ansonsten geben Sie eine S-Qualifikation an, wenn Sie einen Namen (Funktion, Block, Datennamen, Marke oder Source-Referenz) aus den LSD-Sätzen ansprechen, der nicht innerhalb der aktuellen Übersetzungseinheit liegt.

Hinweis für Tester auf Maschinencode-Ebene:

Die CSECT-Namen von LLMs, die mit dem C/C++-Compiler ab V2.1C erzeugt wurden, enthalten ein '&' und müssen in AID-Kommandos in `n'...'` geschrieben werden. Ausführliche Informationen über das Arbeiten mit CSECTs bei AID finden Sie im Handbuch „Testen auf Maschinencode-Ebene“ [2].

NESTLEV=level-nummer

Die NESTLEV-Qualfikation bezeichnet eine Ebene in der aktuellen Aufrufhierarchie.

Ebenso wie die Qualifikation `S=`*srcname*`.PROC=`*funktion* dient die Qualifikation `NESTLEV=`*level-nummer* dazu, Datennamen zu manipulieren, die vom Anwender in den Source Units deklariert wurden. Die Qualifikation `NESTLEV=`*level-nummer* kann nur mit der Basisqualifikation `E={VM|Dn}` kombiniert werden.

Die Qualifikation NESTLEV akzeptiert als Eingabe die Nummer einer Ebene in der aktuellen Aufrufhierarchie, d.h. eine Referenz auf die aktuelle Aufrufhierarchie. Basierend auf dieser Referenz identifiziert AID eine komplette Liste von verfügbaren Datennamen, die auf der angegebenen Ebene definiert wurden.

Normalerweise müssen Sie sich die Aufrufhierarchie ausgeben lassen und diese analysieren, bevor Sie die Qualifikation NESTLEV verwenden können. Folgende AID-Kommandos geben die um die Aufrufebenen erweiterte aktuelle Aufrufherarchie aus:

```
%AID LEV=ON
%SDUMP %NEST
```

Die Qualifikation NESTLEV können Sie in den Kommandos %DISPLAY, %MOVE, %SDUMP and %SET verwenden. In diesen Kommandos liefert die Qualifikation `NESTLEV=`*level-nummer* das gleiche Resultat wie die Qualifikation `S=`*srcname*`.PROC=`*funktion*, sofern *level-nummer* korrekt ist.

Ein Beispiel zur Verwendung der NESTLEV-Qualifikation finden Sie im AID-Basishandbuch, Abschnitt „Bereichsqualifikationen“[1].

:: Mit den beiden Doppelpunkten adressieren Sie den Superblock. Die `::`-Qualifikation verwenden Sie, um globale Daten anzusprechen, die von einer gleichnamigen Definition an der Unterbrechungsstelle verdeckt sind, oder um globale Daten oder Funktionen zu bezeichnen, die nicht der aktuellen Übersetzungseinheit zugeordnet sind. Anders als bei den übrigen Qualifikationen wird zwischen den beiden Doppelpunkten und dem nachfolgenden Daten- oder Funktionsnamen kein Punkt geschrieben.

Beispiel

```
%display s=n'list.c'.::name
```

Die globale Variable `name` aus der Übersetzungseinheit `LIST.C` wird ausgegeben.

PROC=funktion

Die PROC-Qualifikation bezeichnet eine Funktion aus dem Quellprogramm. *funktion* ist der Name, der im Quellprogramm für eine Funktion vergeben wurde und kann bis zu 1000 Stellen lang sein.

Mit der PROC-Qualifikation können Sie den Bereich angeben, in dem die Kommandos %CONTROL*n*, %TRACE oder %SDUMP wirken.
Ansonsten geben Sie eine PROC-Qualifikation an, wenn Sie einen static vereinbarten Datennamen oder einen Anweisungsnamen (Marke) ansprechen wollen, der nicht der aktuellen Funktion zugeordnet ist. Außerdem geben Sie eine PROC-Qualifikation an, wenn Sie einen Datennamen ansprechen wollen, der zwar der aktuellen Funktion zugeordnet ist, aber von einer lokalen Definition mit demselben Namen an der Unterbrechungsstelle verdeckt wird, wenn z.B. in einem inneren Block eine gleichnamige Variable definiert ist.

BLK='[f-]n[:b]'

Die BLK-Qualifikation bezeichnet einen Block. Der Name für einen Block wird wie bei den Source-Referenzen aus Zeilennummer und gegebenenfalls FILE-Nummer und relativer Blocknummer gebildet.

Das äußerste Klammernpaar in einer Funktion umfasst die gesamte Funktion und ist für AID kein Block. Alle Definitionen, die sich dort befinden, werden der Funktion zugeordnet und mit der entsprechenden PROC-Qualifikation angesprochen. Erst mit dem zweiten geschweiften Klammernpaar in einer Funktion beginnt ein Block, den Sie mit einer BLK-Qualifikation ansprechen können.

f FILE-Nummer. Sie wird nur für Zeilen angegeben, die auf Grund einer `#include`- oder `#line`-Anweisung eingefügt wurden (siehe Abschnitt „Funktionen, Marken und Source-Referenzen“ auf Seite 53).

n Zeilennummer, in der der Block beginnt. Diese können Sie dem "Source-Error-Listing", Spalte `SRC-LIN` entnehmen. Sie ist identisch mit der Zeilennummer der Quelldatei.

b relative Blocknummer innerhalb einer Zeile. Sie ergibt sich aus der Anzahl der sich öffnenden geschweiften Klammern innerhalb einer Zeile, wobei jedoch nur die Klammern für Anweisungsblöcke als Blocknummer gelten.
Die Klammern von `struct`-, `union`- und `enum`-Deklarationen werden nicht mitgezählt. Die erste Klammer in einer Funktion zählt als relative Blocknummer, auch wenn sie nicht mit einer BLK-Qualifikation angesprochen werden kann.
b ist eine Nummer > 1, die Sie nur angeben, wenn Sie nicht den ersten Block in einer Zeile ansprechen wollen. Sie bezeichnen damit den *b*-ten Block in einer Zeile.

Mit der BLK-Qualifikation können Sie den Bereich angeben, in dem die Kommandos %CONTROL*n*, %TRACE oder %SDUMP wirken.
Ansonsten geben Sie eine BLK-Qualifikation an, wenn Sie einen static vereinbarten Datennamen ansprechen, der einem Block außerhalb der aktuellen Aufrufhierarchie zugeordnet ist. Außerdem geben Sie eine BLK-Qualifikation an, wenn Sie einen Datennamen ansprechen, der einem Block innerhalb der aktuellen Aufrufhierarchie zugeordnet ist und von einer gleichnamigen Definition an der Unterbrechungsstelle verdeckt wird.

schlüsselwort

Die Schlüsselwörter sind im AID-Basishandbuch, Kapitel „Schlüsselwörter“ [1], beschrieben. Außerdem finden Sie sie bei den Kommandos, in denen sie verwendet werden.

datenname

datenname ist im Abschnitt „Datennamen“ auf Seite 30 beschrieben.

{`L'label'` | `source-referenz` | `funktion`}

label, source-referenz und *funktion* sind im Abschnitt „Funktionen, Marken und Source-Referenzen“ auf Seite 53 beschrieben.

4.1.1 Zuordnung der Daten zu Übersetzungseinheiten, Funktionen und Blöcken

Bei der Adressierung berücksichtigt AID die Gültigkeitsbereiche der Programmiersprache C bzw. C++. Extern vereinbarte Namen gelten im gesamten Programm, Parameter-Namen und Marken innerhalb einer Funktion. Namen, die in einem Block vereinbart wurden, gelten nur innerhalb dieses Blocks.

Globale Daten werden außerhalb aller Funktionen definiert und sind dem Superblock zugeordnet. Lokale Daten sind der Funktion oder dem Block zugeordnet, in dem sie definiert sind.

Der AID-Arbeitsbereich umfasst entweder den gesamten nicht-priviligierten Adressbereich, der von Ihrem geladenen Programm belegt wird oder den entsprechenden Bereich in einem Speicherabzug und wird über die Basisqualifikation bestimmt. Alle Namen, die im mit %BASE festgelegten AID-Arbeitsbereich liegen, können Sie ohne explizite Basisqualifikation ansprechen.
Alle Namen, die in einer anderen Übersetzungseinheit liegen, benötigen immer eine S-Qualifikation und die dem Gültigkeitsbereich entsprechende PROC- oder BLK-Qualifikation bzw. die beiden Doppelpunkte (`::`), wenn es sich um Namen von globalen Daten oder von Funktionen handelt.
Namen, die in der aktuellen Übersetzungseinheit liegen, benötigen eine PROC- und evtl. eine BLK-Qualifikation, wenn sie einer anderen Funktion derselben Übersetzungseinheit zugeordnet sind. Namen in der aktuellen Funktion benötigen nur eine BLK-Qualifikation, wenn sie durch eine gleichnamige Definition lokal verdeckt sind oder wenn sie einem Block dieser Funktion zugeordnet sind, der nicht in der aktuellen Aufrufhierarchie liegt.

Ohne Qualifikation können Sie alle Namen ansprechen, die an der Unterbrechungsstelle gültig sind, d.h. die Sie auch im Programm an der Unterbrechungsstelle verwenden könnten; bei Namensgleichheit ist dies nur die erste Definition, die AID innerhalb der aktuellen Aufrufhierarchie (von innen nach außen) findet.
Übergeordnete Definitionen mit demselben Namen können Sie mit AID jedoch über Qualifikationen ansprechen. Sie verwenden jeweils die Qualifikation, die dem Gültigkeitsbereich des angegebenen Namens in C bzw. C++ entspricht.

Beispiele

1. Funktionsparameter

```
C-Programm
==========================================
 1   #include <ctype.h>
 2   ...
10   int main(int argc, char *argv[])
11   { ... }
==========================================
```

Der Parameter `argc` ist folgendermaßen anzusprechen, wobei `proc=main` in der zweiten Möglichkeit eine Überqualifizierung ist, die AID ignoriert:

```
argc
proc=main.argc
```

2. Geschachtelte Blöcke

```
C-Programm
==========================================
 1   #include <stdio.h>
 2   int main(void)
 3
 4   { int a; struct s {int i;}; {
     1                           2
 5              static int b;
 6              ...
 7              b++; }
 8       printf("%d\n", a); }
==========================================
```

Die aktuelle Unterbrechungsstelle liegt bei der Source-Referenz 8. Die Variable `b` ist mit folgender Block-Qualifikation anzusprechen:

```
blk='4:2'.b
```

3. Globale externe Deklarationen

```
C-Programm - Übersetzungseinheit TEST2.C
==========================================
 9   extern double d;
10   int main(void) {
11   int f1(void);
12   d = PI;
13   ...
14   {
15   int d = 15;
16   ...
==========================================
```

Es wird angenommen, dass das Programm aus drei Übersetzungseinheiten besteht: in `TEST1.C` soll die Variable `d` definiert sein, in `TEST2.C` ist `d` nur deklariert, und in `TEST3.C` wird `d` nicht verwendet. Die Variable `d` kann folgendermaßen ausgegeben werden:

– wenn die Unterbrechungsstelle S’13’ ist:
  `%display d`

– wenn die Unterbrechungsstelle S’16’ ist, und weil es ein lokales `d` gibt:
  `%display ::d`
– wenn die Unterbrechungsstelle in `TEST1.C` liegt:
  `%display d` bzw. `%display ::d`
– wenn die Unterbrechungsstelle in `TEST3.C` liegt:
  `%display s=n'test1.c'.::d`

  oder
  `%display s=n'test2.c'.::d`

4.2 Datennamen

Mit AID können Sie folgende Daten ansprechen:
– einfache (skalare) Typen
– Vektoren und Vektorelemente
– C-Strings
– Strukturen/Unionen und Struktur-/Union-Komponenten
– Aufzählungs (`enum`)-Konstanten
– Bitfelder
– Zeiger

Nicht ansprechen können Sie mit AID:
– Präprozessor-Konstanten und -Makros (`#define`)
– typedef-Namen
– Aufzählungs-, Struktur- und Union-Typen (Etiketten)

Daten können Sie im allgemeinen wie in C/C++ ansprechen. Dazu gibt es folgende Ausnahmen:

– Vektorelemente können Sie nur über Index ansprechen, nicht über Zeiger.

– Bei Variablen vom Typ `long double` wertet AID nur die ersten 8 Bytes aus.

– Variablen vom Typ `char` können Sie bei AID nicht in einem Ausdruck als arithmetischen Typ einsetzen. Sie können mit `char`-Variablen erst rechnen, wenn Sie an den Datennamen eine Typmodifikation anschließen. Dabei wandelt %A den Datentyp in `unsigned char` und %F in `signed char` um.
  Eine Variable vom Typ `signed char` wird auch von AID wie eine vorzeichenbehaftete Integer-Variable behandelt. Sie können den Inhalt einer solchen Variablen ohne Typmodifikation in einem Ausdruck verwenden. Entsprechendes gilt für Variablen vom Typ `unsigned char`.

datenname
: steht für alle im Quellprogramm definierten Namen von Daten.

  datenname wird allgemein wie im Quellprogramm angegeben. AID berücksichtigt bis zu 1000 Zeichen. Nach Eingabe von `%AID LOW={ON|ALL}` unterscheidet AID Groß- und Kleinschreibung. Wie in C/C++ sind C-Schlüsselwörter wie `int`, `char`, etc. auch bei AID nicht zugelassen und werden als Syntaxfehler abgewiesen.

  datenname kann in allen Kommandos zur Ausgabe und Änderung von Daten geschrieben werden; das sind die Kommandos %DISPLAY, %MOVE, %SDUMP und %SET. Außerdem kann *datenname* im %FIND-Kommando (Suchen einer Zeichenfolge) und im %ON-Kommando (Schreibüberwachung) angegeben werden.

AID stellt folgende Formate zur Verfügung:
– Indexschreibweise
– Zeigerschreibweise
– Strukturqualifikation
– Dereferenzierung

Die Formate können auch kombiniert werden, d.h. *datenname* kann in jedem der Formate auch durch jedes der anderen Formate ersetzt werden.

4.2.1 Indexschreibweise

```
datenname [ [index] {...} ]
```

Vektoren und typbezogene Zeiger können Sie indiziert ansprechen. Der Index wird wie in einer C/C++-Anweisung in eckigen Klammern angegeben. Die eckigen Klammern, in die ein Index gesetzt wird, sind in diesem Handbuch fettgedruckt, um sie von den eckigen Klammern der Metasyntax zu unterscheiden.

Zusätzlich besteht bei AID die Möglichkeit, den Vektornamen ohne Indizierung zu verwenden. Damit bezeichnen Sie den gesamten Vektor.

index Der Index kann einen Wert zwischen -2^{31} und $+2^{31}-1$ annehmen und besteht aus:

– einer Ganzzahl,
– einer Variablen vom Typ `int` oder
– einem arithmetischen Ausdruck

Der arithmetische Ausdruck kann aus den arithmetischen Operatoren (+, -, /, *), Ganzzahlen und numerischen Variablen gebildet werden. Die numerischen Variablen, die in einem Index verwendet werden, können nicht qualifiziert werden. Sie müssen daher an der Unterbrechungsstelle sichtbar sein, bzw., falls *datenname* qualifiziert ist, so müssen die Variablen aus *index* in dem Bereich sichtbar sein, der durch die Qualifikation bezeichnet wird.
Die Variablen, aus denen der Index gebildet wird, können wie *datenname* angegeben werden, d.h. sie können wiederum indiziert, zeiger- oder strukturqualifiziert oder dereferenziert werden.
Es ist zu beachten, dass AID zu einer indizierten Angabe in einem `%ON %WRITE(...)` sofort bei der Eingabe Anfangsadresse und Länge des zu überwachenden Bereichs berechnet. Wenn sich also während des Programmablaufs der Inhalt von *index* ändert und sich daraus eine Änderung der Anfangsadresse des

mit *datenname*`[`*index*`]{...}` bezeichneten Bereichs ergibt, so wird mit `%ON %WRITE(...)` weiterhin der bei der Eingabe des Kommandos gültige Bereich überwacht. Im Gegensatz dazu werden Angaben in Subkommandos der Kommandos %CONTROL*n*, %INSERT und %ON erst bei Eintritt des zu überwachenden Ereignisses ausgewertet, d.h. *datenname*`[`*index*`]{...}` muss an der Stelle des Programmablaufs, an der das vereinbarte Ereignis eintritt, also z.B. bei Erreichen des Testpunkts, sichtbar sein, nicht jedoch bei der Kommandoeingabe.

Um ein einzelnes Element eines Vektors anzusprechen, sind so viele Indizes erforderlich, wie in einer C/C++-Anweisung zum Zugriff angegeben werden müssen.

Wenn Sie *index* in der Form *index1*`:`*index2* angeben, so bezeichnen Sie damit den Bereich zwischen *index1* und *index2*.
Für *index1* und *index2* gilt:
Beide müssen innerhalb der Indexgrenzen liegen, und *index1* muss kleiner oder gleich *index2* sein.

Wenn Sie für *index* einen Stern (*) einsetzen, so bezeichnen Sie damit den gesamten Indexbereich der Dimension. Bei eindimensionalen Vektoren ist diese Angabe gleichbedeutend mit der Verwendung des Vektornamens ohne Indizierung.

Die Bereichsangabe können Sie nur im %DISPLAY-Kommando verwenden. Der Vektorname mit Bereichsangabe darf nicht in einer Adressrechnung eingesetzt werden. Es darf keine Typ- oder Längenmodifikation folgen.

Beispiele

1. `%DISPLAY array [*][3]`

   Von einem zweidimensionalen Vektor werden alle die Elemente ausgegeben, die zur ersten Dimension gehören und deren Index in der zweiten Dimension gleich 3 ist.

2. `%DISPLAY array [1:3][*][5:15]`

   Von einem dreidimensionalen Vektor werden folgende Elemente ausgegeben:
   – der Index der ersten Dimension ist 1, 2 oder 3,
   – der Index der zweiten Dimension läuft von der Indexuntergrenze bis zur Indexobergrenze,
   – der Index der dritten Dimension läuft von 5 bis 15.

Verwendung von Vektoren bei AID

Das Arbeiten mit Vektoren unterscheidet sich bei AID von dem, wie Sie es bei C/C++ gewohnt sind, da es die zusätzliche Möglichkeit gibt, den Namen eines Vektors ohne Index zu verwenden:

1. Ansprechen des Vektors im Gültigkeitsbereich seiner Definition

   In der Funktion oder dem Block, der die Definition des Vektors enthält, sprechen Sie mit *datenname* ohne Index den gesamten Vektor an. So wird z.B. bei
   `%FIND X'...' IN` *datenname* der gesamte Vektor durchsucht, bei
   `%ON %WRITE(`*datenname*`)` der gesamte Vektor überwacht.
   %DISPLAY und %SDUMP bereiten alle Vektorelemente mit Index und zugehörigem Inhalt als Tabelle auf. Dies gilt auch für Character-Vektoren, wenn `%AID C=NO` eingestellt ist. Wenn Sie jedoch mit `%AID C=YES` die Interpretation von Character-Vektoren als C-Strings eingeschaltet haben, gibt AID den Vektorinhalt als zusammenhängenden Character-String aus. Mit der Behandlung von C-Strings bei AID befasst sich ein eigener Abschnitt auf Seite 37.

2. Ansprechen eines Vektors als Übergabeparameter

   Wird ein Vektor beim Funktionsaufruf als Parameter an eine Funktion übergeben, so verbirgt sich hinter dem Vektornamen in dieser Funktion ein Zeiger auf den übergebenen Vektor. In der Funktion ist also nur die Anfangsadresse des Vektors bekannt. Einzelne Vektorelemente können Sie wie gewohnt über Index ansprechen. Wenn Sie hier jedoch den Parameternamen ohne Index verwenden, bezeichnen Sie damit die Anfangsadresse des Vektors. Den Vektor als Gesamtes können Sie nur mit anschließendem Pointer-Operator und entsprechender Längenmodifikation ansprechen:
   `%DISPLAY` *parametername*`->%`*typ*`L`*länge*
   Bei C-Strings können Sie für *typ* C angeben, der String wird dann im Character-Format ausgegeben. Bei Vektoren anderer Datentypen, wie z.B. `int`, ist für *typ* nur X sinnvoll; der Vektor wird dann in sedezimaler Darstellung ausgegeben.

   Um genau den aktuell belegten Inhalt eines Character-Vektors auszugeben, können Sie die folgenden beiden Kommandos verwenden:
   `%FIND X'00' IN` *parametername*`->%L`*n*
   `%DISPLAY` *parametername*`->%CL=(%0G -` *parametername*`)`
   Das erste Kommando (%FIND) ermittelt die Adresse des Nullbytes. AID speichert diese Adresse in das AID-Register %0G ab. Der anschließende %DISPLAY gibt den String bis zum Nullbyte in Character-Darstellung aus.

Beispiele

```
C-Programm                              SOURCE: PARR.C
================================================================================
SRC
LIN
  1   void foo (char*,int*);
  2   char a[25] = "abcdefgh";
  3   char *p = &a[5];
  4   int  iv[] = {0,10,20,30,40};
  .
  .   main()
  .   {
  .
 25     foo(a, iv);
  .
  .   }
  .
  .   void foo(char* str, int* nr)
  .   {
  .
 80     printf("String str:\n%-25s\n",str);
 81     printf("Array nr:\n");
 82     for (i=0; i<5; i++) printf("%6i",nr[i]);
  .
  .   }
================================================================================
```

1. Unterbrechungsstelle in `main`:

   Ausgabe eines einzelnen Elements eines Vektors:

```
SRC_REF: 25 SOURCE: PARR.C    PROC: main     ****************************
/%display a[6],p[1]
a( 6)           = |g|
*               = |g|
```

   Jeweils das siebte Element des Vektors `a` wird ausgegeben.
   Die Kopfzeile enthält die Source-Referenz der Unterbrechungsstelle und die Namen der aktuellen Übersetzungseinheit und der aktuellen Funktion.

   Ausgabe des gesamten Vektors im Dump-Format:

```
/%d a%x
CURRENT PC: 01000098    CSECT: PARR$O&@ ********************************
V'0100111A' = a      + #'00000000'
0100111A (00000000) 81828384 85868788 89000000 00000000    abcdefgh.....
0100117A (00000010) 00000000 00000000 00                    .........
```

Wegen der abschließenden Typmodifikation %X gibt AID den gesamten Vektor `a` in sedezimaler und in Character-Darstellung aus. Der sedezimalen Ausgabe können Sie die Position des Nullbytes entnehmen.
Da AID auf die Maschinencode-Ebene wechselt, wird eine zusätzliche Kopfzeile ausgegeben, die den aktuellen Stand des Befehlszählers und den Namen der zugehörigen CSECT enthält.

Ausgabe des `char`-Vektors als C-String:

```
/%aid c=yes
/%d a
SRC_REF: 25 SOURCE: PARR.C  PROC: main  ******************************
a               = "abcdefgh"
```

Mit `%AID C=YES` wird die Interpretation von `char`-Vektoren als Strings eingeschaltet. Der nachfolgende %DISPLAY gibt den belegten Teil des Strings als String-Literal in "..." aus.

Ausgabe eines numerischen Vektors:

```
/%d iv
iv( 0: 4)
( 0)          0  ( 1)         10  ( 2)         20  ( 3)         30
( 4)         40
```

Da der Name des Vektors ohne Index angegeben wurde, bereitet AID alle Vektorelemente in einer Tabelle auf und gibt diese aus.

2. Unterbrechungsstelle in der Funktion `foo`:

   Ausgabe der Anfangsadresse und eines einzelnen Elements des Vektors:

```
SRC_REF: 80 SOURCE: PARR.C    PROC: foo      ****************************
/%d str,nr,str[6],nr[3]
str              = 0100111A
nr               = 01001180
*               = |g|
*               =         30
```

   Anders als in `main` wird bei Angabe des Namens des Vektors ohne Index die Adresse des Vektors ausgegeben, da der Vektor als Zeiger übergeben wird.
   Indizierte Angaben sind genauso wie in `main` möglich. Da das Element jedoch über einen Zeiger angesprochen wird, gibt AID statt des Elementnamens einen Stern aus.

   Ausgabe des gesamten Character-Vektors:

```
/%d str->%xl(::a)
CURRENT PC: 0100020A    CSECT: PARR$0&@  ********************************
V'0100111A' = PARR$0&# + #'0000011A'
0100111A (00000000) 81828384 85868788 89000000 00000000    abcdefgh.....
0100112A (00000010) 00000000 00000000 0000                 .........
/%f x'00' in str->%xl(::a)
PARR$0&#+00000123=01001123 : 00000000 00000000 00000000  ............
/%d str->%cl=(%0g-str)
V'0100020A' = PARR$0&# + #'0000011A'
0100111A (0000011A) abcdefgh
```

   Wie in `main` ist es auch in der Funktion `foo` möglich, den gesamten Character-Vektor im Dump-Format auszugeben (erster %DISPLAY). Wenn Sie jedoch nur den belegten String im Character-Format sehen wollen, müssen Sie zunächst mit %FIND das Nullbyte suchen. Diese Information wird im zweiten %DISPLAY verwendet, um die Länge des Strings zu berechnen.

   Ausgabe eines numerischen Vektors:

```
/%d ::iv
SRC_REF: 80 SOURCE: PARR.C    PROC: foo      ****************************
iv( 0: 4)
( 0)          0  ( 1)         10  ( 2)         20  ( 3)         30
( 4)         40
/%d nr->%l(::iv)
CURRENT PC: 0100020A    CSECT: PARR$0&@  ********************************
V'01001180' = PARR$0&# + #'00000180'
01001180 (00000180) 00000000 0000000A 00000014 0000001E    ............
01001190 (00000190) 00000028                               ....
/%d nr->.8%fl4
01001188 (00000188)         +20
```

Bei Vektoren mit numerischen Elementen ist es nicht möglich, den gesamten Vektor über den Namen des Übergabeparameters aufbereiten zu lassen. Mit der entsprechenden Qualifikation (hier also mit den beiden Doppelpunkten, da es sich um ein globales Datum handelt) und dem Namen, mit dem der Vektor definiert wurde, können Sie jedoch stets den gesamten Vektor ansprechen.
Der zweite %DISPLAY zeigt, wie Sie den gesamten Vektor auf Maschinencode-Ebene über den Parameternamen adressieren können. Der Inhalt des Vektors wird im Dump-Format ausgegeben (sedezimal und Character-Darstellung).
Der letzte %DISPLAY gibt das dritte Element des Vektors als Integer-Wert aufbereitet aus.

4.2.2 C-Strings

AID unterstützt ab Version 2.3B die Stringnotation von C/C++, wenn Sie die Option `%AID C=YES` (siehe Seite 119) einschalten, d.h. Sie können C-String-Literale wie in C/C++ in Hochkommas ("...") schreiben, und `char`-Vektoren werden nicht länger als Vektor einzelner `char`-Elemente betrachtet, sondern wie in C/C++ als Strings behandelt.
Diese Funktionalität können Sie auch in alten C/C++-Objekten nutzen, die mit Compiler-Versionen < V3.0 übersetzt wurden.

4.2.2.1 C-String-Literale

C-String-Literale geben Sie in der folgenden Form an:

"x...x" maximale Länge: 1000 Zeichen bei der Eingabe; unbeschränkt bei der Ausgabe.

x kann jedes abdruckbare oder nicht abdruckbare Zeichen sein. Die nicht abdruckbaren Zeichen müssen mit einer Ersatzdarstellung angegeben werden. Die Ersatzdarstellung beginnt mit einem Fluchtsymbol, dem Gegenschrägstrich (\). Anschließend an das Fluchtsymbol können Sie den Wert des Zeichens auf verschiedene Weise schreiben:

– sedezimale Darstellung `\x`*ff*:

  Zeichenvorrat für *f*: 0-9, a-f, A-F

  Wertebereich von `00` bis `FF`

  Der sedezimale Wert muss stets zweistellig angegeben werden.

– oktale Darstellung `\`*ooo*:

  Zeichenvorrat für *o*: 0-7

  Wertebereich von `000` bis `377`

  Der oktale Wert muss stets dreistellig angegeben werden.

– symbolische Ersatzdarstellung:

  Für einige Zeichen, die nicht abdruckbar sind, wie z.B. das Klingelzeichen, sind bestimmte Ersatzdarstellungen vereinbart, so dass Sie den sedezimalen bzw. oktalen Wert des Zeichens nicht wissen müssen. Andere Zeichen wie z.B der Gegenschrägstrich selbst (\) oder die Anführungszeichen (") können, obwohl sie abdruckbar sind, nur in Verbindung mit einem Gegenschrägstrich eingegeben werden.

Alle Ersatzdarstellungen sind in der folgenden Übersicht zusammengestellt:

Ersatz-darstellung	**sedezimaler Wert**	**Bedeutung**
`\a`	X'2F'	Klingelzeichen
`\b`	X'16'	backspace
`\f`	X'0C'	Seitenvorschub
`\n`	X'15'	Zeilentrenner
`\r`	X'0D'	carriage return
`\t`	X'05'	Horizontal-Tabulator
`\v`	X'0B'	Vertikal-Tabulator
`\\`	X'BC'	Gegenschrägstrich
`\?`	X'6F'	Fragezeichen
`\'`	X'7D'	Anführungszeichen (einfach)
`\"`	X'7F'	Anführungszeichen (doppelt)
`\x`*ff*	X'*ff*'	Sedezimalzahl
ooo	-	Oktalzahl

Tabelle 1: Ersatzdarstellungen und ihre Bedeutung

In der Ausgabe wählt AID da, wo es möglich ist, das abdruckbare Äquivalent des Zeichens aus, unabhängig davon, in welcher Form das Zeichen eingegeben wurde. Nicht abdruckbare Zeichen werden in der Ersatzdarstellung abgebildet und zwar, wenn möglich, in der symbolischen Form. Die Bitkombinationen, die kein abdruckbares Zeichen darstellen und für die auch keine symbolische Darstellung vereinbart ist, werden in sedezimaler Form angezeigt.

C-String-Literale können Sie in den AID-Kommandos %DISPLAY und %SET sowie in Vergleichen in einem Subkommando verwenden. Ein C-String-Literal kann mit %SET nur in einen `char`-Vektor übertragen werden. Ist das Literal länger als das Empfangsfeld, wird rechts abgeschnitten; AID gibt eine Warnung aus. Ist das Literal dagegen kürzer als das Empfangsfeld, wird das Empfangsfeld mit binären Nullen aufgefüllt. Daraus ergibt sich, dass Sie mit dem Übertragen eines leeren Literals ("") einen `char`-Vektor auf binär Null setzen können.

Ein Vergleich mit einem C-String-Literal in einem Subkommando ist nur zulässig, wenn der zweite Vergleichsoperand ein `char`-Vektor ist.

Bei der Behandlung von `char`-Vektoren als C-Strings handelt es sich um eine reine High-Level-Funktionalität. Mit einem %MOVE können daher keine C-String-Literale übertragen werden. Auch muss in einem %SET das Empfangsfeld mit einer symbolischen Speicherreferenz bezeichnet werden, z.B. ist `%SET "abc" INTO V'...'` nicht möglich. Eine solche Eingabe wird als Syntaxfehler abgewiesen. Ebenso meldet AID bei der Verwendung eines C-String-Literals in allen AID-Kommandos außer %DISPLAY und %SET einen Syntaxfehler.

Wenn mit `%AID C=YES` die Unterstützung der C-String-Literale eingeschaltet ist, müssen Sie Kommentare in /*...*/ einschließen. Kommentare in "..." werden von AID dann nicht mehr als solche erkannt, sondern stets als C-String-Literale interpretiert, was i.d.R. zu einem Syntaxfehler führt.

Laufen AID-Kommandos in einer Prozedur ab, dann werden in C-String-Literalen keine Parameter ersetzt, da der BS2000-Kommandointerpreter Angaben in Hochkommas stets als Kommentare ansieht, unabhängig davon, ob `%AID C=YES` eingestellt ist oder nicht.

Beispiel

```
/%set "\xC5\x25\x15" into cstr; %d cstr
cstr            = "E\x25\n"
```

In den `char`-Vektor `cstr` werden mit %SET drei Zeichen übertragen und anschließend ausgegeben. Für das erste Zeichen, das mit "\xC5" angegeben wurde, gibt AID ein "E" aus, da "E" sedezimal mit C5 dargestellt wird; das zweite Zeichen wird als Sedezimal-Zahl ausgegeben, da es für" \x25" keine abdruckbare Entsprechung gibt. Das dritte Zeichen "\x15" erscheint in der Ausgabe als "\n", das ist die symbolische Ersatzdarstellung für Zeilentrenner.

4.2.2.2 C-String-Vektoren

Wenn `%AID C=YES` eingeschaltet ist, werden `char`-Vektoren von AID als C-Strings interpretiert. Ist der `char`-Vektor eindimensional, so beginnt der C-String mit dem ersten Vektorelement und endet mit dem Vektorelement, das den Wert X'00' hat. Mehrdimensionale `char`-Vektoren stellen Vektoren von C-Strings dar, und zwar werden, da der hinterste Index zuerst läuft, die Vektorelemente zu C-Strings zusammengefasst, die über den hintersten Index adressiert werden.

Mit %DISPLAY wird der Inhalt eines eindimensionalen `char`-Vektors als C-String-Literal ausgegeben. Den Namen des Vektors geben Sie ohne Index an. Für die Aufbereitung der einzelnen Zeichen gelten die im vorigen Abschnitt aufgeführten Regeln.
Bei der Ausgabe mehrdimensionaler `char`-Vektoren werden die Vektorelemente, die zum am weitesten rechts stehenden Index gehören, als C-String zusammengefasst, es wird also ein Vektor von C-Strings ausgegeben. Im Anschluss an den Vektornamen geben Sie einen Index weniger an, als Indexstufen in der Definition des Vektors enthalten sind. Das Endekriterium ist jeweils X'00'. Sind nach X'00' noch weitere Vektorelemente gesetzt, so werden diese in der Ausgabe nicht berücksichtigt.
Wenn Sie im %DISPLAY-Kommando einen `char`-Vektor mit einem Indexbereich angeben, so wird der Vektor bei der Ausgabe auch bei eingeschaltetem `%AID C=YES` in einzelne Vektorelemente aufbereitet.

Mit %SET können Sie C-String-Vektoren überschreiben. Sender kann dabei ein C-String-Literal oder ein anderer C-String-Vektor sein. Der Sender wird einschließlich des Endekriteriums X'00' in den Empfänger eingetragen, und es gilt:

- Ist der Sender länger als der Empfänger, wird rechts abgeschnitten; AID gibt eine Warnung aus.
- Ist der Sender kürzer als der Empfänger, wird rechts mit X'00' aufgefüllt.

Bei einem mehrdimensionalen Vektor können die Vektorelemente als C-String übertragen bzw. überschrieben werden, die der letzten Indexstufe zugeordnet sind.

Ein einzelnes `char`-Vektorelement oder ein `char`-Literal in der Form '*x*' können Sie nicht in einen C-String-Vektor übertragen, aber es kann selbstverständlich ein C-String-Literal, das nur aus einem Zeichen besteht, übertragen werden, also "*x*".

Was für die Übertragung von C-Strings gilt, müssen Sie auch bei Vergleichen von C-String-Vektoren in einem Subkommando beachten. Ein C-String-Vektor kann nur mit einem anderen C-String-Vektor oder mit einem C-String-Literal verglichen werden. Der Vergleich eines C-Strings mit einem einzelnen `char`-Zeichen wird ebenso wie die Übertragung eines einzelnen Characters in einen C-String mit der folgenden Meldung abgelehnt:

```
AID0388    Types are not convertible.
```

Beispiel

In einem C-Programm ist der `char`-Vektor `carray` folgendermaßen definiert und initialisiert:

```
char carray[3][10]={"1","22","333"};
```

```
/%aid c=yes
/%d carray
carray( 0: 2)
( 0) "1"  ( 1) "22"  ( 2) "333"
/%aid check=all
/%s "ab\n" into carray[1]
OLD CONTENT:
"22"
NEW CONTENT:
"ab\n"
%  AID0274 Change desired? Reply (Y=Yes; N=No)?y
```

Kommando `%AID C=YES` veranlasst AID, `char`-Vektoren als C-Strings zu interpretieren. Daher werden in der folgenden %DISPLAY-Ausgabe die Vektorelemente der zweiten Indexstufe zu C-Strings zusammengefasst. Anschließend wird mit `%AID CHECK=ALL` der Änderungsdialog eingeschaltet. Der nachfolgende %SET überschreibt den String in `carray[1]` mit der Zeichenfolge `"ab\n"`.

4.2.3 Zeigerschreibweise

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
datenname1 -> datenname2
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Mit dieser Schreibweise können Sie in AID nur Strukturkomponenten über Zeiger ansprechen. *datenname1* muss vom Typ Zeiger sein. Sie sprechen damit *datenname2* wie in einer C/C++-Anweisung an. AID bereitet *datenname2* entsprechend seinen Typ- und Längenattributen auf.

Beispiel

```
p1 -> var
```

Wie in C/C++ sprechen Sie ausgehend von der Adresse, die in `p1` hinterlegt ist, die Strukturkomponente `var` an.

4.2.4 Strukturqualifikation

```
übergeordneter datenname• {...•} datenname
```

Mit der Strukturqualifikation können Sie wie in C/C++ Komponenten von Strukturen ansprechen. Der erste *übergeordnete datenname* ist der Name der Struktur. Eventuell nachfolgende *übergeordnete datennamen* sind Namen von darin verschachtelten Strukturen. Der letzte *datenname* ist der Name der Strukturkomponenten, die Sie ansprechen wollen. Diese bereitet AID entsprechend ihren Typ- und Längenattributen auf. Ab C/C++ V2.1C müssen Sie in einem AID-Kommando, genau wie in einer C/C++-Anweisung, alle Stufen der Struktur angeben, beginnend beim ersten *übergeordneten datennamen* bis zum Namen der Komponenten, die Sie ansprechen wollen.
In der LSD, die der Compiler C/C++ V3.0 erzeugt, ist erstmalig die Beziehung von Basisklassen und abgeleiteten Klassen eingetragen, so dass AID die in C++ gültigen Scoperegeln für das Ansprechen von Komponenten aus Klassensystemen nachbilden kann. Daten- und Funktions-Member aus Basis- und abgeleiteten Klassen können Sie nun ohne Qualifikation oder teilqualifiziert ansprechen, solange die gewünschte Komponente dadurch eindeutig bezeichnet wird.

4.2.5 Dereferenzierung

```
[(]* {...} datenname[)]
```

Die Dereferenzierung können Sie in AID nur auf Zeiger, nicht jedoch auf Vektoren anwenden. Der Inhaltsoperator (*) wird wie in einer C/C++-Anweisung verwendet, er kann also auch mehrfach wiederholt werden. Die ganze Angabe kann in runde Klammern gesetzt werden. *datenname* ist der Name eines Zeigers, der (eventuell über weitere Zeiger) auf ein Speicherobjekt verweist.

Zeiger-Arithmetik für die Dereferenzierung von Zeigern können Sie in AID nur mit der Indexschreibweise angeben (siehe dazu das folgende Beispiel 4).

Beispiele

```
C-Programm
==========================================================================
     ...
     struct
     {   int x;
         char y;
         float *z3;
     } z, *p, p1[5];
==========================================================================
```

1. `z.x`
   bezeichnet die Strukturkomponente `x` in der Struktur `z`.
2. `p->y`
   bezeichnet die Strukturkomponente `y` in der von `p` referenzierten Struktur.
3. `*p`
   bezeichnet die gesamte von `p` referenzierte Struktur.
4. `*(p1+4)->z3`
   Diese in C/C++ übliche Adressangabe führt in AID zu einem Syntaxfehler. Um in einem AID-Kommando die gewünschte Speicherstelle zu adressieren, müssen Sie Folgendes schreiben:
   `*p1[4].z3`

4.2.6 Prioritäten der Operatoren

Die Operatoren für die symbolische Adressierung in AID haben dieselben Prioritäten wie in C/C++. Die Operatoren -> , • und [ ] haben gleiche Priorität, und alle haben höhere Priorität als der *. Auch in AID können Sie die Prioritäten durch Klammerung ändern.

Vorsicht beim zweiten Operanden des Punkt-Operators: ist dieser in Klammern eingeschlossen, führt AID einen Adressversatz durch (siehe unten Beispiel 2).

Beispiele

Die Beispiele beziehen sich auf die Definition der Struktur `z` auf der vorhergehenden Seite.

1.
```
*p -> z3
*(*p).z3
*p[0].z3
p[0].z3[0]
p -> z3[0]
(*p).z3[0]
```

   Alle Schreibweisen haben dieselbe Bedeutung: `p` ist ein Zeiger und verweist auf eine Struktur mit der Komponente `z3`. `z3` ist ebenfalls ein Zeiger und wird über Zeiger `p` angesprochen und dann referenziert. Die Bedeutung ist wie in C/C++.

2.
```
*(*p).(z3)
```

   Diese Angabe hat in AID eine andere Bedeutung als in C/C++:
   Da `z3` in Klammern steht, nimmt AID den Inhalt von `z3` als Wert für den Adressversatz. Dieser Inhalt muss vom Typ %F oder %A sein, sonst lehnt AID einen Adressversatz ab. Das Ergebnis sind 4 Bytes vom Typ %X.

4.2.7 Adressoperator & und Adressselektor %@(...)

Um beim Testen von C/C++-Programmen auf die Adressen von Daten zugreifen zu können, stehen Ihnen in AID zwei Möglichkeiten zur Verfügung: der Adressoperator &, den Sie von der Sprache C/C++ her kennen, und der AID-Adressselektor %@(...), den Sie unabhängig von der jeweiligen Programmiersprache verwenden können. Mit einem %DISPLAY-Kommando können Sie sich die Adresse eines Datums ausgeben lassen. Mit einem %MOVE- oder %SET-Kommando können Sie die Adresse eines Datums übertragen. Außer in den AID-Kommandos %DISPLAY, %MOVE und %SET können Sie Adressen von Daten im Vergleich eines Subkommandos oder in Ausdrücken verwenden.

Das Speicherobjekt, das Sie ansprechen, wenn Sie den Pointer-Operator anfügen, also &...-> bzw. %@(...)->, ist für AID vom Typ %XL4. Sie können aber mit einer Typ- und Längenmodifikation einen anderen Typ und eine andere Länge vereinbaren oder mit dem Typselektor %T(...) einen im Quellprogramm festgelegten Datentyp mit der zugehörigen Länge auf die Adresse anwenden

Der Adressoperator `&` kann auf alle symbolischen Adressen aus C/C++-Programmen angewandt werden und liefert die absolute Adresse des angesprochenen Datums im Speicher. Bitfeld- und Registervariablen sind als Argument nicht zugelassen. Auf Datennamen anderer Programmiersprachen oder auf komplexe Speicherreferenzen können Sie den AID-Adressselektor %@(...) anwenden. Dieser ist ausführlich im AID-Basishandbuch, Abschnitt „Adress-, Typ- und Längenselektor“, beschrieben.

Wird eine Adresse, die mit dem Adressoperator `&` ermittelt wurde, mit %SET übertragen, so muss das Empfangsfeld vom Typ Zeiger sein. Weitergehende Prüfungen führt AID nicht durch. Z.B. muss der Datentyp des Senders nicht mit dem Datentyp übereinstimmen, auf den der Zeiger verweist, der als Empfänger angegeben ist. Eine solche Überprüfung führt AID nur durch, wenn die Adresse eines Klassenobjekts in einen Zeiger auf eine Klasse übertragen wird (siehe „objekt“ auf Seite 46).

Syntax des Adressoperators `&`:

qua Basis- oder Bereichsqualifikation

Falls eine Basis- oder Bereichsqualifikation erforderlich ist, so muss diese vor den Adressoperator geschrieben werden.

{:: | namespace:: | klasse::}
Qualifikation für den globalen Bereich, Namespace- oder Klassenqualifikation

Die ::-Qualifikation für den globalen Bereich oder eine Namespace- oder Klassenqualifikation werden, falls erforderlich, anschließend an den Adressoperator geschrieben. Der Operand des Adressoperators darf nicht auf *namespace* oder *klasse* enden, sonst gibt AID eine Fehlermeldung aus (`AID0480`).

this this-Zeiger

In `this` ist die Adresse des zu einer Member-Funktion zugehörigen aktuellen Objekts hinterlegt.

Mit anschließendem Pointer-Operator können Sie den `this`-Zeiger im Pfad zu einer Komponente einer Klasse verwenden.

objekt Name eines Objekts einer Klasse

Mit anschließendem Punkt verwenden Sie *objekt*, um den Pfad zu einer Komponente einer Klasse zu beschreiben.
Endet der Operand des Adressoperators mit dem Namen eines Objekts, so bezeichnen Sie damit die Anfangsadresse des Objekts.

Wenn Sie mit %SET die Adresse eines Objekts in einen Zeiger auf eine Klasse übertragen wollen, so muss Folgendes gelten:

– Die Klasse, auf die der Empfänger zeigt, muss mit der Klasse, deren Adresse übertragen werden soll, übereinstimmen,

  oder

– sie muss Basisklasse der dem Sender zugeordneten Klasse sein. Diese Basisklasse muss in der Klasse des Senders ein eindeutiges Subobjekt besitzen.

datenname
Name eines Datums

datenname wird wie im Quellprogramm angegeben. Sie bezeichnen damit die Anfangsadresse des Datums.

Daten können Sie bis auf folgende Ausnahmen wie in C/C++ ansprechen:

– Mit einem Vektornamen ohne Index erhalten Sie die Adresse des ersten Elements des Vektors.

– Einzelne Vektorelemente können Sie nur über Index ansprechen, nicht über Zeiger.

– Wenn `%AID C=YES` gesetzt ist (siehe Seite 119), fasst AID die Vektorelemente eines `char`-Vektors, die der letzten Indexstufe zugeordnet sind, zu C-Strings zusammen. Mit der entsprechenden Indexangabe erhalten Sie die Anfangsadresse des C-Strings.

– Die Anfangsadresse von Vektorelementen, die über einen Indexbereich angesprochen werden, können Sie nicht ermitteln.

– Vektoren, die als Parameter an eine Funktion übergeben wurden, sind dort Zeiger auf die Vektoren im aufrufenden Programm; mit dem Adressoperator & erhalten Sie die Adresse des Zeigers und nicht die Adresse des Vektors.

Zum Arbeiten mit Vektoren siehe auch die Abschnitte „Indexschreibweise“ auf Seite 31 und „C-Strings“ auf Seite 37.

datenname kann wie folgt angegeben werden. Die Formate können auch kombiniert werden (siehe Abschnitt „Datennamen“ auf Seite 30). Es gelten die Vorrangregeln von C/C++.

Indexschreibweise:	*datenname* `[`*index*`]` `{...}`
Zeigerschreibweise:	*datenname1* `->` *datenname2*
Strukturqualifizierung:	*übergeordneter datenname*`.` `{...}` *datenname*
Dereferenzierung:	`[]*{...}` *datenname*`[]`
Pointer-to-Member-Dereferenzierung:	*datenname1*`.*`*datennname2* oder *datenname1*`->*`*datenname2*

Ist *datenname* ein dynamisches Daten-Member einer Klasse, dann ermittelt AID abhängig von der Unterbrechungsstelle und einer evtl. vorhergehenden Qualifikation entweder die absolute Adresse des Datums oder die relative Adresse des Daten-Members zum Anfang der Klasse:

– Die absolute Adresse des Datums wird ausgewählt, wenn das Programm außerhalb der Klasse unterbrochen ist und das Daten-Member über ein Objekt der Klasse angesprochen wird oder wenn das Programm in einer dynamischen Member-Funktion der Klasse unterbrochen ist und das Daten-Member entweder direkt angesprochen werden kann oder wenn Sie, um Eindeutigkeit herzustellen, eine entsprechende Klassenqualifikation davorschreiben müssen.

– Die Distanz zur Anfangsadresse der Klasse wird ausgewählt, wenn das Programm außerhalb der Klasse unterbrochen ist und das Daten-Member über eine Klassenqualifikation angesprochen wird. Ist das Programm in einer dynamischen Member-Funktion der Klasse unterbrochen, können Sie auf die relative Adresse nur zugreifen, wenn Sie die zugehörige Klasse über eine Bereichsqualifikation (S-, PROC- oder ::-Qualifikation) gewissermaßen von außen ansprechen.

Auch die Distanz zum Klassenanfang können Sie mit %SET nur in einen Zeiger übertragen, nicht jedoch in eine numerische Variable.

Bei statischen Daten-Membern liefert der Adressoperator & stets die absolute Adresse.

funktion

Name einer Funktion

Eine C-Funktion aus einer Übersetzungseinheit, die mit der Option
`//MODIFY-SOURCE-PROPERTIES LANGUAGE=C(...)`
übersetzt wurde, sprechen Sie bei AID mit ihrem Namen an. Die nachfolgenden beiden Klammern mit den Übergabeparametern (Signatur) werden weggelassen. Wie Sie die Namen von C++-Funktionen in AID angeben, ist auf Seite 60 ausführlich beschrieben.

Mit *&funktion* können Sie die Funktionsadresse in einem AID-Kommando angeben. Dieselbe Adresse sprechen Sie auch mit *funktion* ohne Adressoperator an, jedoch benötigen Sie die Angabe *&funktion*, wenn Sie die Funktionsadresse mit einem %SET-Kommando in einen Zeiger übertragen wollen.

Beispiel

```
C++-Programm                              SOURCE: EXADR.C
================================================================================
SRC
LIN
...
 20  class X {
 21    int a;
 22    static int c;
 23    public:
 24    void  gx(int) {...; return;}
 25  }x;
...
 68  int  main()
 69  {
 70    x.gx(3);
...
```

```
/%in s'70';%r
STOPPED AT SRC_REF: 70, SOURCE: EXADR.C , PROC: main
/%d &x.a,&x.c
010010F0
010010EC
/%d &X::a,&X::c
00000000
010010EC
```

Im ersten Schritt wird das Programm in `main` vor dem Funktionsaufruf von `gx(int)` unterbrochen. Die absoluten Adressen des dynamischen Daten-Members `a` und des statischen Daten-Members `c` werden ausgegeben. Der zweite %DISPLAY, in dem die Daten-Member über die zugehörige Klassenqualifikation angesprochen werden, liefert bei `a` die Distanz zum Klassenanfang; bei `c` wird wiederum, weil `c` statisch ist, die absolute Adresse ausgegeben.

```
/%in x.n'gx(int)';%r
STOPPED AT SRC_REF: 24, SOURCE: EXADR.C , PROC: X::gx(int)
/%d &a,&c,&X::a,&X::c
010010F0
010010EC
010010F0
010010EC
/%d &::X::a
00000000
```

Dann wird ein Testpunkt auf den Anfang der Member-Funktion `gx(int)` gesetzt und das Programm bis dorthin ausgeführt. Die Daten-Member `a` und `c` können jetzt direkt angesprochen werden. Aber auch mit der davorstehenden Klassenqualifikation `X::` werden jetzt die absoluten Adressen angezeigt, da die Unterbrechungsstelle in einer Member-Funktion der Klasse `X` liegt (erster %DISPLAY).
Wenn Sie auch von dieser Stelle aus die relative Adresse von `a` erfahren wollen, müssen Sie vor die Klassenqualifikation noch die beiden Doppelpunkte für den globalen Block schreiben. Damit erreichen Sie, dass AID von außen auf `X` zugreift (zweiter %DISPLAY).

4.2.8 Längenoperator sizeof() und Längenselektor %L(...)

Mit dem Längenoperator `sizeof()`, den Sie von C/C++ her kennen, und mit dem AID-Längenselektor %L(...) können Sie auf die Länge von Daten zugreifen. `sizeof()` kann nur beim Testen von C/C++-Programmen verwendet werden, während Sie den Längenselektor %L(...) unabhängig von der Programmiersprache des zu testenden Programms einsetzen können. Der AID-Längenselektor ist ausführlich im Basishandbuch [1], Abschnitt „Adress-, Typ- und Längenselektor“, beschrieben.
Mit %DISPLAY können Sie sich die Länge eines Datums ausgeben lassen, mit %MOVE oder %SET können Sie sie übertragen. Außerdem kann das Ergebnis einer Längenselektion in einem Vergleich eines Subkommandos oder in einem Ausdruck verwendet werden. Funktionsnamen sind als Operanden nicht erlaubt; die Länge einer Funktion kann also weder mit dem Längenoperator noch mit dem Längenselektor ermittelt werden.

Der Längenoperator `sizeof()` kann auf alle symbolischen Adressen aus C/C++-Programmen angewandt werden und liefert die Länge des angesprochenen Datums. Bitfeld- und Registervariablen sind als Argument nicht zugelassen.

`sizeof()` müssen Sie wie in C/C++ in Kleinbuchstaben angeben. `%AID LOW={ON|ALL}` muss also eingeschaltet sein.

Syntax des Längenoperators `sizeof()`:

qua Basis- oder Bereichsqualifikation

Falls eine Basis- oder Bereichsqualifikation erforderlich ist, so muss diese vor den Längenoperator geschrieben werden.

{:: | namespace::}
Qualifikation für den globalen Block und Namespace-Qualifikation

Die ::-Qualifikation für den globalen Block oder eine Namespace-Qualifikation werden, falls erforderlich, anschließend an den Adressoperator geschrieben. Der Operand des Adressoperators darf nicht auf *namespace* enden, sonst gibt AID eine Fehlermeldung aus (`AID0480`).

klasse Name einer Klasse

Den Namen einer Klasse können Sie mit den beiden anschließenden Doppelpunkten im Adressierungspfad zu einem Daten-Member der Klasse als Klassenqualifikation verwenden. Der Operand des Längenoperators kann auch auf *klasse* enden. Das Ergebnis ist dasselbe, als wenn Sie `sizeof()` auf ein Objekt von *klasse* anwenden. Sie erhalten die Anzahl Bytes, die ein Objekt dieser Klasse einnimmt, und zwar inklusive eventueller Füll-Bytes, die benötigt werden, um ein Objekt der Klasse in einen Vektor zu plazieren.

this this-Zeiger

`this` zeigt auf das zu einer Member-Funktion zugehörige aktuelle Objekt. `sizeof(*this)` liefert daher die Länge dieses Objekts.

Mit anschließendem Pointer-Operator können Sie den `this`-Zeiger im Pfad zu einer Komponente einer Klasse verwenden.

objekt Name eines Objekts einer Klasse

Mit anschließenden Punkt verwenden Sie *objekt*, um den Pfad zu einer Komponente einer Klasse zu beschreiben.
Endet der Operand des Längenoperators mit dem Namen eines Objekts, so erhalten Sie die Länge des Objekts. Die Länge entspricht der von `sizeof(`*klasse*`)`, wenn *klasse* der Name der dem Objekt zugeordneten Klasse ist (siehe oben).

datenname

Name eines Datums

datenname wird wie im Quellprogramm angegeben. Sie bezeichnen damit die Länge des Datums.

Daten können Sie bis auf folgende Ausnahmen wie in C/C++ ansprechen:
Mit einem Vektornamen ohne Index bezeichnen Sie die Gesamtlänge aller Vektorelemente.
Wenn `%AID C=YES` gesetzt ist (siehe Seite 119), fasst AID die Vektorelemente eines `char`-Vektors, die über den am weitesten rechts stehenden Index adressiert werden, zu C-Strings zusammen; jedoch liefert `sizeof()` in diesem Fall nicht die Länge des C-Strings, sondern die Gesamtlänge des zugrundeliegenden C-String-Vektors.
Im Gegensatz zu C/C++ ist es mit AID nicht möglich, zu einer ausgewählten Indexstufe den Speicherbedarf zu ermitteln.
Einzelne Vektorelemente können Sie nur über Index ansprechen, nicht über Zeiger.
Wird ein Vektor beim Funktionsaufruf als Parameter an eine Funktion übergeben, so ist in dieser Funktion nur die Anfangsadresse des Vektors bekannt. Wenn Sie in der Funktion den Namen des Parameters ohne Index verwenden, bezeichnen Sie damit diese Adresse. `sizeof()` auf den Übergabeparameter angewandt liefert als Ergebnis stets 4.

datenname kann wie folgt angegeben werden. Die Formate können auch kombiniert werden. Es gelten die Vorrangregeln von C/C++ (siehe Abschnitt „Datennamen“ auf Seite 30).

Indexschreibweise:	*datenname* `[`*index*`]` `{...}`
Zeigerschreibweise:	*datenname1* `->` *datenname2*
Strukturqualifizierung:	*übergeordneter datenname*`.` `{...}` *datenname*
Dereferenzierung:	`[()]*{...}` *datenname*`[()]`
Pointer-to-Member-Dereferenzierung:	*datenname1*`.*`*datennname2* `oder` *datenname1*`->*`*datenname2*

Beispiel

```
/%d sizeof(carray)
          30
/%aid c=yes
/%d sizeof(carray[0])
          10
```

`carray` sei ein `char`-Vektor mit folgender Definition:

```
char carray[3][10];
```

Der erste %DISPLAY gibt die Gesamtlänge des Vektors aus.

Nachdem `%AID C=YES` gesetzt wurde, können Sie sich mit `sizeof()` die Länge der über den zweiten Index adressierten C-Strings in `carray` anzeigen lassen. Es wird die Gesamtanzahl aller zum zweiten Index gehörenden Vektorelemente ausgegeben, unabhängig von der Position des Endekriteriums X’00’ innerhalb des dem C-String zugrunde liegenden Vektors.

4.3 Funktionen, Marken und Source-Referenzen

Funktionen und Marken sind Namen aus dem Quellprogramm, mit denen in AID Anweisungen angesprochen werden können. Marken und Funktionen aus C- und C++-Programmen sind als Adresskonstanten abgelegt. Sie enthalten die Adresse des ersten Befehls in einer Funktion bzw. nach einer Marke.
Namensgleichheit bei Funktionen und Marken sollten Sie vermeiden, da AID nicht unterscheiden kann, ob die Funktion oder die Marke gemeint ist.
Bei allen Namen berücksichtigt AID maximal 1000 Zeichen.

Source-Referenzen werden vom Compiler für jede ausführbare Anweisung erzeugt. Sie sind Adresskonstanten und enthalten die Adresse des ersten Befehls, der zu einer Anweisung erzeugt wurde.

Für Marken und Source-Referenzen ist die in der Adresskonstanten hinterlegte Adresse identisch mit der Adresse der zugehörigen ausführbaren Anweisung. Bei Funktionen steht jedoch in der Adresskonstanten die Anfangsadresse des Prologs der Funktion, die vor der Adresse der ersten ausführbaren Anweisung in der Funktion liegt. Aus diesem Grund kann AID bei der Lokalisierung von Adressen, z.B. in `%D %HLLOC(...)`, die Prologadressen nicht der entsprechenden Funktion zuordnen.

In den Kommandos %FIND und %ON *write-ereignis* können Sie Funktionen, Marken und Source-Referenzen nur mit anschließendem Pointer-Operator verwenden. Sie bezeichnen damit vier Bytes des Maschinencodes ab der Adresse, die in der Adresskonstanten hinterlegt ist, bei Funktionen also die Anfangsadresse des Prologs. In %DISPLAY, %MOVE und %SET bezeichnen Sie damit die Adresse. In %DISASSEMBLE und %INSERT sprechen Sie mit Funktionen, Marken und Source-Referenzen stets die erste ausführbare Anweisung an, die auf die Adresse folgt, die in der Adresskonstanten eingetragen ist. In %CONTROLn und %TRACE können Sie mit zwei Source-Referenzen einen Bereich festlegen.

funktion
: ist der Name einer Funktion, der im Quellprogramm vergeben wurde bzw. der Name einer Bibliotheksfunktion. Eine C-Funktion aus einer Übersetzungseinheit, die mit der Option `//MODIFY-SOURCE-PROPERTIES LANGUAGE=C(...)` übersetzt wurde, oder eine Bibliotheksfunktion sprechen Sie bei AID mit ihrem Namen an. Die nachfolgenden beiden Klammern mit den Übergabeparametern (Signatur) werden weggelassen.
Wie Sie die Namen von C++-Funktionen in AID angeben, ist auf Seite 60 ausführlich beschrieben.

L'label'

label ist eine Marke, die im Quellprogramm vergeben wurde. Sie können nur die Marken ansprechen, die im C/C++-Programm von einer `goto`-Anweisung angesprungen werden können. `case`- und `default`-Marken können Sie nicht ansprechen.
In den Kommandos %DISASSEMBLE und %INSERT können Sie *label* auch ohne `L'...'` angeben, da in diesen Kommandos eine Verwechslung mit einem Datennamen nicht möglich ist.

Source-Referenz

ist die vom Compiler erzeugte Bezeichnung einer Anweisung. Sie wird mit folgendem Format angegeben:

S'[f-]n[:a]'

Innerhalb der Hochkommas dürfen keine Leerzeichen stehen.

f `FILE`-Nummer; sie wird nur für Zeilen angegeben, die auf Grund einer `#include`-Anweisung eingefügt wurden, falls die eingefügten Zeilen ausführbare Anweisungen enthalten oder wenn durch eine `#line`-Anweisung die Zeilennumerierung für die Folgezeile explizit festgesetzt wurde. Die `FILE`-Nummer entnehmen Sie dem Source-Error-Listing, Spalte `FILE-NO`. *f* ist eine Ziffer > 0.

n Zeilennummer, die Sie dem Source-Error-Listing, Spalte `SRC-LIN` entnehmen; sie ist identisch mit der Zeilennummer der Quelldatei, falls das Quellprogramm keine `#include`- oder `#line`-Anweisungen enthält.
Geht eine Anweisung über mehrere Zeilen, so ist *n* die Zeilennummer der ersten Zeile der Anweisung.
Wenn ein Quellprogramm ohne `#include`- oder `#line`-Anweisungen mit der „Beautify"-Funktion des C-Strukturierers behandelt wird, steht jeweils nur eine Anweisung in einer Zeile. Die Source-Referenzen bestehen dann nur aus der Zeilennummer: S'`n`'.

a relative Anweisungsnummer innerhalb einer Zeile; sie ergibt sich aus der Anzahl der Anweisungen in einer Zeile. *a* ist eine Ziffer > 1, die Sie nur angeben, wenn Sie nicht die erste Anweisung in einer Zeile ansprechen wollen.
Sie bezeichnen damit die *a*-te Anweisung in einer Zeile.

Wenn Sie in %CONTROL*n* oder %TRACE einen Bereich mit zwei Source-Referenzen angeben wollen, müssen Sie beachten, dass aufsteigenden Source-Referenzen nur innerhalb eines Funktionsblocks aufsteigende Adressen entsprechen.
In C++-Programmen gilt außerdem, dass zu impliziten Konstruktor- und Destruktor-Aufrufen sowie bei Konversionsoperationen zusätzliche Source-Referenzen generiert werden, die zwar nicht im Source-Error-Listing erscheinen, die jedoch vom %TRACE protokolliert werden.

Beispiel

```
FILE    SRC
 NO     LIN
================================================================================
0       100          i++;
                     #line 17 "incl.h"
1        17          j++; k++;
================================================================================
```

Mit S'100' sprechen Sie die Anweisung `i++;` an.

Wegen der Zeile `#line 17 "incl.h"` nummeriert der Compiler ab der nächsten Zeile neu, beginnend mit 1-17:
Mit S'1-17' sprechen Sie die Anweisung `j++;` an.
Mit S'1-17:2' sprechen Sie die Anweisung `k++;` an.

4.3.1 Besonderheiten beim Ansprechen von Anweisungen

Eine Anweisung ist ein Statement im Sinne von ANSI-C und ist in der Grammatik definiert. Im folgenden sind die für AID geltenden Besonderheiten aufgeführt.

1. **Definitionen und Deklarationen von Daten und Funktionen**

   Sie sind keine Anweisungen, außer wenn bei einer Definition das Datum initialisiert wird.
   Daten werden angesprochen wie im Abschnitt „Datennamen“ auf Seite 30, Funktionen wie im Abschnitt „Funktionen, Marken und Source-Referenzen“ auf Seite 53 beschrieben. Das Ansprechen der zusätzlichen Sprachkonstrukte, die C++ anbietet, wird im Kapitel „C++-spezifische Adressierung“ auf Seite 59 behandelt.

2. **labeled-Statement / Marken an Anweisungen**

   Marken sind weder selbständige Anweisungen noch Bestandteil einer Anweisung, sie können mit einer Source-Referenz nicht angesprochen werden. Marken, die mit einer `goto`-Anweisung angesprungen werden können, können in AID mit dem Anweisungsnamen `L'`*label*`'` angesprochen werden.

Beispiel

```
SRC-LIN    Anweisung
-------------------------------------------------------------------------------
 99        lab5:
100                 a = b; c = d;
                    1      2
101        ...
102        case 'a': foo();
                    1
103        ...
104        default:  i = 0; break;
                    1      2
```

Mit `L'lab5'` sprechen Sie die Adresse der Anweisung `a = b;` an, also die Adresse der ersten Anweisung nach der Marke `lab5`.

Mit S’100’ sprechen Sie dieselbe Anweisung `a = b;` an.
Mit S’100:2’ sprechen Sie die Anweisung `c = d;` an.

`case`- und `default`-Marken können Sie mit `L'...'` nicht ansprechen.

Mit S’102’ sprechen Sie den Funktionsaufruf `foo();` an.
Mit S’104’ sprechen Sie die Anweisung `i = 0;` an.
Mit S’104:2’ sprechen Sie die Anweisung `break;` an.

3. **compound-Statement / Block**

Mit dem compound-Statement können mehrere Anweisungen zu einem Block zusammengefasst und an Stellen eingesetzt werden, wo die C/C++-Grammatik nur eine einzelne Anweisung zulässt. Weder das compound-Statement als solches, noch die öffnende oder die schließende geschweifte Klammer sind für AID Anweisungen. Für ein compound-Statement wird keine Source-Referenz erzeugt. Es kann nur als Block mit der BLK-Qualifikation angesprochen werden. Die einzelnen Anweisungen innerhalb eines compound-Statements können mit Source-Referenzen angesprochen werden.

Beispiel

```
SRC-LIN    Anweisung
-------------------------------------------------------------------------------
 30        { a = b; f(a); }
             1      2
```

BLK=’30’ bezeichnet das compound-Statement, damit können Sie aber nur den Bereich bei %CONTROLn oder %TRACE festlegen.

S’30’ bezeichnet die Zuweisung `a = b;`.
S’30:2’ bezeichnet den Funktionsaufruf `f(a);`.

4. **expression-Statement / Ausdruck als Anweisung**

   expression-Statements sind jeweils nur eine Anweisung. Zuweisungen, Funktionsaufrufe etc. innerhalb eines expression-Statements können Sie mit AID nicht ansprechen, da hierfür keine Source-Referenzen erzeugt werden.

   Beispiel

```
SRC-LIN    Anweisung
-------------------------------------------------------------------------------
 40         a = f(b);
 41         c = a > x ? b : a;
 42         a = (b = c);
 43         ...
```

   Für jede dieser Anweisungen wird nur je eine Source-Referenz erzeugt:

   Mit S'40' sprechen Sie die Anweisung `a = f(b);` in Zeile 40 an.
   Mit S'41' sprechen Sie die Anweisung `c = a > x ? b : a;` in Zeile 41 an.
   Mit S'42' sprechen Sie die Anweisung `a = (b = c);` in Zeile 42 an.

5. **selection-Statement / Auswahlanweisung**

   Das sind die if-, if else- und switch-Anweisungen. Sie bestehen für AID wie in C/C++ aus mehreren Anweisungen, die Sie alle mit Source-Referenzen ansprechen können.

   Beispiel

```
SRC-LIN    Anweisung
-------------------------------------------------------------------------------
 50        if (a < 0) a++; else a--;
           1          2        3
 51        if (a < 0) {a++;} else {a--;}
           1           2           3
 52        ...
 53        switch ( c = getchar()) {
           1
 54           case 'X': look('y');
                      1

 55           case 'Y': look('z'); return;
                      1          2
 56           ...
```

   Mit S'51' sprechen Sie den Bedingungsteil der if-Anweisung in Zeile 51 an.
   Mit S'51:2' sprechen Sie den then-Zweig der if-Anweisung an.
   Mit S'51:3' sprechen Sie den else-Zweig der if-Anweisung an.

   Mit S'55' sprechen Sie den case-Fall `'Y'` (Zeile 55) an.
   Mit S'55:2' sprechen Sie die `return`-Anweisung an.

6. **iteration-Statement / Wiederholungsanweisung**

   Dies sind die `while`-, `do`- und `for`-Schleifenkonstrukte. Sie bestehen für AID wie in C/C++ aus mehreren Anweisungen, die Sie alle mit Source-Referenzen ansprechen können. Bei der `for`-Schleife wird für den Bedingungs- und den Inkrementierungsteil eine zusätzliche Source-Referenz erzeugt.

   Beispiel

```
SRC-LIN    Anweisung
-------------------------------------------------------------------------------
 60        while (p->next) mknode(p->next);
           1               2
 61        ...
 62        do show() while( tick );
           1  2      3
 63        do { x++; z(x); } while(x);
           1    2    3       4
 64        ...
 65        for (i=0; i < 10; i++) j[i] = i;
                1    2       3    4
 66        a = b;
           1
 67        ...
```

   Mit S'65' sprechen Sie die Initialisierung `i=0` der for-Schleife in Zeile 65 an,
   mit S'65:2' den Bedingungsteil `i < 10`,
   mit S'65:3' die Inkrementierung `i++` und
   mit S'65:4' sprechen Sie den Ausführungsteil der for-Schleife `j[i] = i;` an.
   Mit S'66' sprechen Sie die erste Anweisung nach der `for`-Schleife an.

7. **jump-Statement / Sprunganweisung**

   Dies sind die `goto`-, `continue`-, `break`- und die `return`-Anweisungen.

   Beispiel

```
SRC-LIN    Anweisung
-------------------------------------------------------------------------------
 70        goto label1;
           1
 71        ...
 72        if (i < 10) continue;
           1           2
 73        ...
 74        if (k >= 1) return k * 2;
           1           2
 75        ...
```

   Mit S'72:2' sprechen Sie die Anweisung `continue;` in Zeile 72 an.
   Mit S'74:2' sprechen Sie die Anweisung `return k*2;` in Zeile 74 an.

5 C++-spezifische Adressierung

Als zusätzliche Sprachelemente zum Sprachumfang von C bietet C++ unter anderem Namespaces, Klassen, Templates, virtuelle Funktionen, überladene Funktionen und Operatoren sowie Referenzvariablen an. In diesem Kapitel ist beschrieben, wie Sie in AID-Kommandos auch diese Sprachelemente ansprechen können.

5.1 Qualifikationen

Beim Testen von C- und C++-Programmen gibt es hinsichtlich der Verwendung von Basis- und Bereichsqualifikationen, die im Abschnitt „Qualifikationen“ auf Seite 21 beschrieben sind, keine Unterschiede.
Um Daten und Funktionen aus Klassen und Namespaces anzusprechen, bietet AID die zusätzlichen Qualifikationen *klasse*`::` und *namespace*`::` an.
Besonderheiten sind auch bei der Schreibweise von C++-Funktionsnamen und bei der Angabe von Instanzen von Funktionstemplates oder Klassentemplates zu beachten.

klasse/namespace::[...]

Mit dieser Qualifikation bezeichnen Sie eine Klasse oder einen Namespace oder Sie geben bei abgeleiteten oder geschachtelten Klassen oder bei geschachtelten Namespaces den Pfad zu einer Klasse oder zu einem Namespace an. Den Namen des Datums oder der Funktion, den Sie über diese Qualifikation erreichen wollen, schließen Sie unmittelbar an die letzten beiden Doppelpunkte der Qualifikation an.
Wenn Sie mit der Qualifikation *klasse*`::` die Instanz eines Klassentemplates bezeichnen wollen, müssen Sie die folgende Syntax verwenden:
`t'`*k_template*`<`*arg*`[,...]>'::`
Gibt es zu einem Klassentemplate nur eine einzige Instanz, so können Sie diese Instanz mit dem Namen des Klassentemplates ansprechen und die Template-Argumente weglassen : `t'`*k_template*`'::`

Ausführliche Informationen zur Bildung der Template-Instanznamen finden Sie im Abschnitt „Template-Instanziierung“ auf Seite 98. Klassen sind auf Seite 65 und Namespaces auf Seite 89 beschrieben.

Beispiel

```
%DISPLAY ::A::B::i
```

`B` sei in diesem Beispiel eine in Klasse A geschachtelte Klasse und `i` ein statisches Daten-Member in `B`. Klasse `A` sei global definiert.
In diesem Fall können Sie sich `i` von jeder Stelle der Übersetzungseinheit aus mit dem obigen %DISPLAY-Kommando ausgeben lassen.

PROC-Qualifikation

Bei C++ müssen Sie beachten, dass Funktionen nicht wie in C stets unmittelbar einer Übersetzungseinheit zugeordnet sind, sondern dass sie sowohl innerhalb von Namespaces als auch innerhalb von Klassen definiert sein können. Zum Ansprechen dieser Funktionen müssen Sie dem Funktionsnamen die entsprechende *klasse*/*namespace*`::`-Qualifikation voranstellen.
Andererseits gibt es bei C++ überladene Funktionen, die den gleichen Funktionsnamen haben, sich aber in den Argumenten (Signatur) unterscheiden, und es gibt Funktionen, die durch Instanziierung aus einem Funktionstemplate hervorgegangen sind. Alle diese Merkmale müssen Sie beim Schreiben einer PROC-Qualifikation für eine C++-Funktion berücksichtigen. Es ergibt sich die folgende Syntax:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

PROC=[namespace::[...]][klasse::[...]]funktion

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

namespace::
: Namespace-Qualifikation, evtl. mehrstufig.

klasse::
: Klassen-Qualifikation, evtl. mehrstufig. Handelt es sich bei der Klasse um eine Instanz eines Klassentemplates, müssen Sie die oben beschriebene Form verwenden: `t'`*k_template*`<`*arg*`[,...]>'::`

funktion
: In C++ wird unterschieden zwischen:
  – normalen Funktionen, die den Funktionen in C entsprechen,
  – überladenen Funktionen,
  – virtuellen Funktionen und
  – Funktionen, die aus einem Funktionstemplate durch Instanziierung gebildet werden.

  Je nachdem, um welchen Typ von Funktion es sich handelt, den Sie in der PROC-Qualifikation angeben wollen, müssen Sie eine besondere Schreibweise wählen:

– Bei normalen, überladenen oder virtuellen Funktionen müssen Sie zur eindeutigen Bezeichnung der Funktion die Signatur mit angeben. Die Signatur `void` darf nicht geschrieben werden. In diesem Fall bezeichnen Sie die Funktion mit dem Funktionsnamen und den anschließenden Klammern, wie dies auch in C++ möglich ist. Da die Funktionsbezeichnungen somit Sonderzeichen enthalten (Klammern und evtl. Kommas), müssen sie in `n'...'` gesetzt werden. Es ergibt sich die folgende Syntax:
`n'`*funktion*`([`*signatur*`])'`

  Davon abweichend werden die `main`-Funktion und die vom Compiler erzeugte Funktion `__STI__` (siehe Abschnitt „Konstruktor und Destruktor“ auf Seite 75) in C++ nur mit ihrem Namen angesprochen. Auch alle Funktionen mit C-Linkage, die in einem C++-Programm aufgerufen werden, sprechen Sie nur über den Funktionsnamen ohne Klammern und ohne Signatur an.
  Beispiel: Daten aus der Funktion `main` qualifizieren Sie wie in C mit `PROC=main`.

– Den Instanznamen eines Funktionstemplates müssen Sie in `t'...'` einschließen. Die Template-Argumente geben Sie in spitzen Klammern und durch Kommas getrennt an, sodass sich die folgende Syntax ergibt:
`t'`*f_template*`<`*arg*`[,...]>([`*signatur*`])'`

  Gibt es zu einem Funktionstemplate nur eine einzige Instanz, so können Sie diese Instanz mit dem Namen des Funktionstemplates ansprechen und die Template-Argumente weglassen : `t'`*f_template*`([`*signatur*`])'`.

  Ausführlich beschrieben finden Sie die Namensbildung von Template-Instanzen im Abschnitt „Template-Instanziierung“ auf Seite 98.

– Funktionen aus Klassen, die über Zeiger angesprochen werden, wie virtuelle Funktionen oder über Pointer-to-Member adressierte Funktionen können Sie nicht in einer PROC-Qualifikation verwenden. Sie können jedoch auf die Anfangsadresse einer derart referenzierten Funktion zugreifen. Wie das geht, ist im Abschnitt „Virtuelle Funktionen“ auf Seite 76 und im Abschnitt „Pointer-to-Function-Member“ auf Seite 83 beschrieben.

Die beiden Schreibweisen `n'...'` und `t'...'` werden von AID unterschiedlich verarbeitet:

– Ist der Name in `n'...'` eingeschlossen, so übernimmt ihn AID ungeprüft und unverändert unter Beibehaltung der Groß-/Kleinschreibung. Daraus ergibt sich, dass innerhalb von `n'...'` keine zusätzlichen Leerzeichen geschrieben werden dürfen. Der Name kann bis zu 1000 Zeichen lang sein.

– Bei einem in `t'...'` eingeschlossenen Namen geht AID davon aus, dass es sich um eine Template-Instanz handelt und führt eine syntaktische und semantische Überprüfung des Namens durch. Stellt AID dabei fest, dass es sich um keine zulässige Bezeichnung einer Template-Instanz handelt bzw. dass die angegebene Instanz nicht existiert, wird eine entsprechende Fehlermeldung ausgegeben. In `t'...'` eingeschlossene Namen können beliebig lang sein.

Wenn Sie eine Funktion ansprechen wollen, die in einer lokalen Klasse definiert ist., so müssen Sie für eine vollständige explizite Qualifikation vor der PROC-Qualifikation, mit der Sie die gewünschte Funktion bezeichnen, eine weitere PROC-Qualifikation für die übergeordnete Funktion angeben, in der die Definition der lokalen Klasse enthalten ist. Falls die lokale Klasse in einem inneren Block der übergeordneten Funktion steht, müssen Sie zwischen den beiden PROC-Qualifikationen noch eine oder ggf. mehrere BLK-Qualifikationen schreiben. Wie Sie eine BLK-Qualifikation angeben, ist auf Seite 25 beschrieben.
Für die Qualifikation einer Funktion aus einer lokalen Klasse ergibt sich die folgende Syntax:

```
PROC=übergeordnete_fkt•[BLK='[f–]n[:b]'•[...]]PROC=klasse::[...]funktion
```

Funktionen, die in lokalen Klassen aus inneren Blöcken definiert sind, können Sie mit AID nur ansprechen, wenn das Programm mit C/C++ V3.0B übersetzt wurde. In Programmen der Compiler-Version 3.0A können diese Funktionen weder in einer PROC-Qualifikation angegeben werden noch können Sie sich in einem AID-Kommando auf die Anfangsadresse einer solchen Funktion beziehen.

Da auch lokal definierte Member-Funktionen in C++ global implementiert werden, ergibt sich für das Arbeiten mit AID, dass die Daten des Blocks, der die Definition des Objekts enthält sowie der Objektname selbst nicht zum Scope der Funktion gehören und Sie daher aus einer solchen Funktion heraus die Daten des Blocks sowie den Objektnamen nur über eine entsprechende Bereichsqualifikation ansprechen können. Das zugehörige Objekt können Sie jedoch stets über den `this`-Zeiger ansprechen (siehe Seite 66).

Ein Beispiel zu Funktionen aus lokalen Klassen finden Sie im Anschluss an diesen Abschnitt (Beispiel 3 auf Seite 64).

Beispiel

Verwendung von Member-Funktionen und überladenen Funktionen in der PROC-Qualifikation

```
C++-Programm                              SOURCE: BSP.C
================================================================================
SRC
LIN
  1   extern "C" int printf (const char*,...);
  2
  3  int FOO(int X)  {return X++; }
  4  long FOO(long X){return X--; }
  5  class A_global
  6  {
  7    public:
  8    A_global()  { printf("Constructor called\n");... };
  9    ~A_global() { printf("Destructor called\n");... };
 10    void f()    { static int k; printf("f called\n");
 11                        k = 5;... return;};
 12  } a_global;
...

 68  int main()
 69  {
 70    FOO(1);
 71    FOO(1L);
...
121    class A_lokal
122    {
123      public:
124      A_lokal()  { printf("Constructor called\n");... };
125      ~A_lokal() { printf("Destructor called\n");... };
126      void f()   { static int k; printf("f called\n");
127                         k = 5;... return;};
128    } a_lokal;
...
```

1. `%trace 1 in proc=n'FOO(int)'`

   AID hält vor der einzigen Anweisung der Funktion `FOO(int)` an und protokolliert diese.

2. `%control1 %proc in proc=A_global::n'~A_global()'`

   Im Destruktor der globalen Klasse `A_global` soll vor der Ausführung der ersten und der letzten Anweisung angehalten werden.

3. `%display proc=main.proc=A_lokal::n'f()'.k`

   Die statische Variable `k` der Member-Funktion `f()`, die in der lokalen Klasse `A_lokal` im äußersten Block der Funktion `main` definiert ist, wird ausgegeben.

5.2 In Blockmitte definierte Daten

In C++ können anders als in C auch mitten im Block Daten definiert werden.
Diese Daten können Sie mit AID wie auch in C++ erst nach ihrer Definition ansprechen.
Wird eine Qualifikation benötigt, so geben Sie den gesamten umfassenden Block oder die zugehörige Funktion in einer BLK- bzw. PROC-Qualifikation an.

Beispiele

```
C++-Programm                              SOURCE: BSPI.C
================================================================================
SRC
LIN
...
120  int main()
121  {
122     int i = 1;
123     {
124        i++;
125        int i = 3;
126        i++;
127        ...
```

1. `%insert s'124'; %resume`
   `%display i, proc=main.i`

   Durch die beiden Kommandos %INSERT und %RESUME wird das Programm bis zur Source-Referenz S’124’ ausgeführt und dann unterbrochen. Mit `i` und `proc=main.i` aus dem %DISPLAY-Kommando bezeichnen Sie beide Male dieselbe Variable `i` aus der `main`-Funktion.

2. `%trace 3`
   `%display i, proc=main.i`

   Der %TRACE führt das Programm bis zur Source-Referenz S’126’ aus. Diesmal bezeichnet der erste Operand des %DISPLAY die Variable `i` aus Block ’123’, die in Zeile 125 definiert ist . Mit `proc=main.i` sprechen Sie wie oben die Variable `i` aus `main` an.

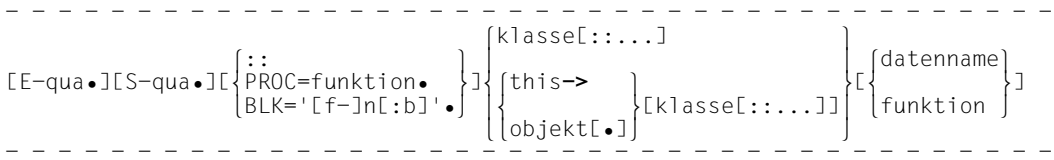
5.3 Klassen

Klassen und darin enthaltene Daten und Funktionen können Sie in AID analog der von C++ gewohnten Schreibweise ansprechen. Um Daten-Member von geschachtelten oder abgeleiteten Klassen anzusprechen, geben Sie genau wie in C++, abhängig von der Unterbrechungsstelle, von außen nach innen alle Klassen- bzw. Objektnamen an, die den Pfad zum Datum beschreiben. Bei geschachtelten Klassen müssen Sie stets auch alle Zwischenstufen angeben, bei abgeleiteten Klassen müssen nur die Zwischenstufen aufgeführt werden, die zur eindeutigen Ansprache des Datums erforderlich sind.
Bei Member-Funktionen, außer virtuellen Funktionen, beschreiben Sie den Pfad zur gewünschten Funktion in einer vorangestellten Klassen-Qualifikation (*klasse*`::`, siehe Abschnitt „Qualifikationen“ auf Seite 59). Wie Sie virtuelle Funktionen ansprechen können, finden Sie im Abschnitt „Virtuelle Funktionen“ auf Seite 76 beschrieben.

Objekte von Klassen werden analog den Strukturen in C den entsprechenden Gültigkeitsbereichen zugeordnet. Demnach werden Objekte von Klassen, die global vereinbart sind, genauso wie alle übrigen globalen Daten durch Voranstellen von `::` angesprochen. Lokal definierte Objekte von Klassen sind der Funktion bzw. dem Block zugeordnet, der die Definition enthält.

Geben Sie in einem %DISPLAY- oder %SDUMP-Kommando den Klassennamen an, so erhalten Sie eine Auflistung der statischen Daten-Member und der statischen und dynamischen Member-Funktionen (ohne virtuelle Funktionen) dieser und eventuell darin geschachtelter Klassen, bei abgeleiteten Klassen auch der Basisklassen. Zu den Daten wird der Inhalt ausgegeben. Zu den Funktionen wird der vollständige Funktionsname mit Signatur in C++-Standard-Schreibweise und die Anfangsadresse des Prologs der Funktion aufgeführt.
Die vollständige Klasse, also zusätzlich auch die dynamischen Daten-Member und die virtuellen Funktionen, gibt AID aus, wenn Sie in einem %DISPLAY- oder %SDUMP-Kommando den Namen eines Objekts der Klasse angeben. Außerdem erhalten Sie den vollständigen Inhalt der Klasse angezeigt, wenn das Programm in einer dynamischen Member-Funktion der Klasse unterbrochen ist und Sie die Klasse mit `*this` oder dem Klassennamen ansprechen.

Ob Daten und Funktionen innerhalb der Klassen als public, private oder protected erklärt sind, ist für AID unerheblich. Während des Testlaufs sind die im Programm vergebenen Zugriffsrechte aufgehoben.



klasse::
: Für *klasse* geben Sie den Namen einer Klasse an. Sie geben *klasse* auf der ersten Position an, wenn Sie die gesamte Klasse oder ein statisches Datum oder eine Funktion der Klasse ansprechen wollen. Zwischen Klassennamen und Daten- bzw. Funktionsnamen müssen Sie die beiden Doppelpunkte (`::`) schreiben.
Auf einer mittleren Position verwenden Sie den Klassennamen, um ausgehend von einer abgeleiteten Klasse eine Basisklasse anzusprechen, falls eine Komponente der Basisklasse adressiert werden soll, die durch eine gleichnamige Definition in der abgeleiteten Klasse verdeckt ist.

Handelt es sich bei der gewünschten Klasse um eine Instanz eines Klassentemplates, so müssen Sie die folgende Syntax verwenden:
`t'`*k_template*`<`*arg*`[,...]>'`
Gibt es zu dem Klassentemplate nur eine einzige Instanz, so reicht die Angabe `t'`*k_template*`'`.
Ausführliche Informationen zum Thema Templates finden Sie auf Seite 98.

this
: ist der vom C++-Compiler generierte Zeiger, der aus einer dynamischen Member-Funktion heraus auf das zugehörige Objekt verweist. An den `this`-Zeiger schließen Sie den Pointer-Operator an und eventuell im Pfad zum gewünschten Datennamen noch erforderliche Namen von Basisklassen oder bei geschachtelten Klassen die Namen äußerer Klassen, genauso wie Sie ausgehend vom Objektnamen den Pfad zum ausgewählten Datum der Klasse beschreiben. Bei abgeleiteten Klassen müssen Sie die Zwischenstufen jedoch nur dann angeben, wenn der Name des gewünschten Datums nicht eindeutig ist.
Da der `this`-Zeiger nur verwendet werden kann, wenn das Programm in einer dynamischen Member-Funktion unterbrochen wurde und `this` dann stets auf das zugehörige Objekt verweist, ist eine vorangehende Bereichsqualifikation nicht erforderlich.

In der %SDUMP-Ausgabe zu einer dynamischen Member-Funktion zeigt Ihnen AID den Zeiger und seinen Inhalt an.

`%DISPLAY this` gibt Ihnen die Adresse des aktuellen Objekts aus.
`%DISPLAY *this` listet das aktuelle Objekt auf.
Endet ein Adressoperand auf den `this`-Zeiger mit anschließendem Pointer-Operator (->), dann adressieren Sie damit die ersten 4 Bytes des aktuellen Objekts; es gilt der Speichertyp %X.

objekt ist der Name eines Objekts einer Klasse. *objekt* geben Sie bei allen dynamischen Daten-Membern an, wenn die Unterbrechungsstelle nicht in einer dynamischen Member-Funktion dieser Klasse liegt. Außerdem verwenden Sie *objekt*, um ein Datum der Klasse eindeutig zu bezeichnen, das durch eine gleichnamige Definition an der Unterbrechungsstelle lokal verdeckt ist.

5.3.1 Scoperegeln in Klassen

Für das Ansprechen von Daten und Funktionen, die in Klassen definiert sind, gelten die von C++ her bekannten Scoperegeln.

Statische Daten-Member können Sie unabhängig von einem Objekt der Klasse und auch unabhängig von der aktuellen Unterbrechungsstelle über die Klassen-Qualifikation ansprechen, also indem Sie den Klassennamen voranstellen und zwischen Klassennamen und Datennamen zwei Doppelpunkte schreiben. Bei geschachtelten Klassen beschreiben Sie den Pfad zum gewünschten Datum über mehrere Klassenstufen von außen nach innen; die Klassennamen werden dabei jeweils durch die beiden Doppelpunkte (`::`) getrennt.

Dynamische Daten-Member werden unterschiedlich angesprochen, je nachdem, an welcher Stelle Ihr Programm unterbrochen wurde:

– Liegt die Unterbrechungsstelle in einer dynamischen Member-Funktion der Klasse, so können Sie die zugehörigen dynamischen Daten-Member direkt ansprechen. AID verhält sich dabei gemäß den von C++ bekannten Gültigkeitsregeln. Das gesamte zugehörige Objekt wird über den `this`-Zeiger mit `*this` angesprochen (siehe Seite 66). Den Objektnamen können Sie nur mit der entsprechenden Qualifikation erreichen.

  Da AID die Beziehung zwischen Basisklassen und abgeleiteten Klassen nachbildet, gelten auch für das Ansprechen von Daten-Membern aus abgeleiteten oder Basisklassen die in C++ gültigen Scoperegeln.

  Für ältere Objekte, die mit einer C/C++-Compiler-Version bis 2.2C übersetzt wurden, fehlt die Information zu Basisklassen und abgeleiteten Klassen in der LSD. Daher kennt AID bei diesen Objekten den Bezug zwischen Basis- und abgeleiteten Klassen nicht. In diesem Fall gilt für das Ansprechen von Daten aus Klassensystemen das im Vorgängerhandbuch zu AID V2.1A beschriebene Verfahren.

– Liegt die aktuelle Unterbrechungsstelle **nicht** in einer dynamischen Member-Funktion derselben Klasse, in der auch die Daten-Member definiert sind, so können Sie die dynamischen Daten nur über das zugehörige Objekt ansprechen. Sie geben den Objektnamen wie eine Strukturqualifikation in einem C-Programm an (siehe Abschnitt „Datennamen“ auf Seite 30); zwischen Objektnamen und Datennamen schreiben Sie einen Punkt. Wenn Sie ein dynamisches Daten-Member aus einem Klassensystem von geschachtelten Klassen ansprechen wollen, müssen Sie im Pfad zu dem gewünschten Datum, ausgehend vom aktuellen Objekt, die Klassennamen der übergeordneten Stufen aufführen. Nach einem Klassennamen werden auch hier zwei Doppelpunkte geschrieben. Innerhalb von Basisklassen und abgeleiteten Klassen gelten dagegen die Scoperegeln von C++, so dass Sie anschließend an den Objektnamen nur die Klassennamen angeben müssen, die zur eindeutigen Ansprache des Datums erforderlich sind.

Beispiele

1. Ansprechen von Daten und Funktionen aus Klassen von verschiedenen Unterbrechungsstellen aus

```
C++-Programm                                SOURCE: VPTR.C
================================================================================
SRC
LIN
...
 20  class X
 21  {
 22    int        a;
 23    static int b;
 24    int        c;
 25    public:
 26    X(int x = 1) : a(x) {c=2; ...; return;}
 27    void        f()     {...; return;}
 28    static void g()     {...; return;}
 29  };
 30  int X::b = 3;
 31
 32  class Y : public X
 33  {
 34    int        a;
 35    static int b;
 36    public:
 37    Y(int x = 4) : a(x) {...; return;}
 38    void        f()     {...; return;}
 39    static void g()     {...; return;}
 40  };
 41  int Y::b = 6;
 42
 43  class Y        y;
...
 68  int  main()
 69  {
 70    y.f();
 71    Y::g();
...
```

```
/%in s'70'
/%in ::Y::n'f()'
/%in ::Y::n'g()'
/%r
```

Zur besseren Lesbarkeit sind im Ablaufprotokoll die Benutzereingaben fettgedruckt.

Zunächst werden mit den %INSERT-Kommandos drei Testpunkte im Programm gesetzt:

– in `main` vor dem Aufruf der Member-Funktion `Y::f()`
– in der Member-Funktion `Y::f()` vor der ersten ausführbaren Anweisung
– in der Member-Funktion `Y::g()` vor der ersten ausführbaren Anweisung

Mit %RESUME wird das Programm gestartet. Es läuft bis zum ersten Testpunkt.

```
STOPPED AT SRC REF: 70 ,  SOURCE: VPTR.C ,  PROC: main
/%d y.X::a, X::b, y.c
y.X::a          =           1
X::b            =           3
y.c             =           2
/%d X::n'f()',X::n'g()'
X::f()      = 01000628
X::g()      = 01000730
/%d y.a, Y::b, Y::n'f()',Y::n'g()'
y.a             =           4
Y::b            =           6
Y::f()      = 01000D48
Y::g()      = 01000E50
/%r
```

Die Unterbrechungsstelle liegt in `main`. Die dynamischen Daten-Member des Objekts `y` der Klasse `Y` können Sie wie in einer Strukturqualifikation über den Objektnamen mit dem nachfolgenden Punkt ansprechen. Daten-Member der Basisklasse `X` werden mit `y.X::`*datenname* angesprochen. Die statischen Variablen aus Basis- und abgeleiteter Klasse (beide heißen `b`) werden über die entsprechende Klassen-Qualifikation adressiert. Die Member-Funktionen werden mit ihren vollständigen Namen und der vorangestellten Klassen-Qualifikation angesprochen, AID gibt die jeweilige Anfangsadresse des Prologs der Funktion aus. Mit %RESUME wird bis zum nächsten Testpunkt fortgefahren.

```
STOPPED AT SRC_REF: 38 ,  SOURCE: VPTR.C ,  PROC: Y::f()
/%d X::a, X::b, c
SRC_REF: 38 SOURCE: VPTR.C    PROC: Y::f()  ****************************************
Y.X::a          =           1
Y.X::b          =           3
Y.X.c           =           2
/%d X::n'f()',X::n'g()'
Y.X::f()    = 01000628
Y.X::g()    = 01000730
/%d a, b, n'f()', n'g()'
Y.a             =           4
b               =           6
f()         = 01000D48
g()         = 01000E50
/%r
```

Die Unterbrechungsstelle liegt diesmal in der dynamischen Member-Funktion `Y::f()` der Klasse `Y`. Die dynamische Variable `a` der Klasse `X` kann nur qualifiziert angesprochen werden, da es in `Y` ebenfalls ein `a` gibt. Aus demselben Grund kann die statische Variable `b` der Klasse `X` nur qualifiziert angesprochen werden. `c` ist dagegen ein-

deutig und benötigt keine Qualifikation. Unmittelbar erreicht werden können auch `a` und `b` aus `Y`. Das statische Daten-Member `b` aus `X` müssen Sie dagegen mit seinem vollständigen Namen angeben.
Die Funktionen `X::f() und X::g()` sind an der Unterbrechungsstelle nicht sichtbar, da sie von den gleichnamigen Funktionen aus `Y` lokal verdeckt werden, und werden daher mit der vorangestellten Klassen-Qualifikation angegeben. Die Funktionen `f()` und `g()` aus `Y` können Sie direkt ansprechen.

```
STOPPED AT SRC_REF: 39 ,  SOURCE: VPTR.C,  PROC: Y::g()
/%d ::y.X::a,X::b,::y.c
SRC_REF: 39 SOURCE: VPTR.C    PROC: Y::g()  ****************************************
y.X::a          =           1
X::b            =           3
y.c             =           2
/%d X::n'f()',X::n'g()'
X::f()          = 01000628
X::g()          = 01000730
/%d ::y.a, b, n'f()',n'g()'
y.a             =           4
b               =           6
f()             = 01000D48
g()             = 01000E50
```

Die Unterbrechungsstelle liegt nun in der statischen Member-Funktion `Y::g()`. Wie im ersten Fall, als das Programm in `main` unterbrochen war, können die dynamischen Daten-Member der Klasse `Y` nur über das zugehörige Objekt adressiert werden. Da der Scope des Objekts `y` jedoch erst nach der Definition von `Y::g()` beginnt, können Sie den Objektnamen `y` nur vollqualifiziert (hier mit den beiden vorangestellten Doppelpunkten, der Qualifikation für den Superblock) ansprechen. Das statische Daten-Member `b` aus `Y` erreicht AID dagegen ohne Qualifikation. Das statische Daten-Member `X::b` wird von `Y::b` verdeckt.

2. Die folgenden Beispiele demonstrieren das Ansprechen von Variablen aus Basisklassen und abgeleiteten Klassen an verschiedenen Konstruktionen von drei Klassen `A`, `B` und `C`. Die Unterbrechungsstelle soll jeweils in der Funktion `func_C()` der Klasse `C` liegen.

   Zunächst die Definitionen der Klassen `A`, `B` und `C`:

```
C++-Programm                                SOURCE: EX1.C
================================================================================
SRC
LIN
...
 42  class A {
 43    int i,j,l;
 44    public:
 45    A(int x=1, int y=2, int z=3) : i(x),j(y),l(z) {...}
 46    void func_A() {...}
 47  };
 48  class B {
 49    int j,k,l;
 50    public:
 51    B(int x=4, int y=5, int z=6) : j(x),k(y),l(z) {...}
 52    void func_B() {...}
 53  };
 54  class C: public A, public B {
 55    int l;
 56    public:
 57    C(int x = 7) : l(x) {...}
 58    void func_C() {...}
 59  };
```

   Testverlauf:

```
...
STOPPED AT SRC_REF: 58 ,  SOURCE: EX1.C ,  PROC: C::func_C()
/%d i, j, k, l
SRC_REF: 58 SOURCE: EX1.C    PROC: C::func_C()  ************************************
C.A.i               =           1
%  AID0376 Ambiguous qualification for SYMBOL j
C.B.k               =           5
C.l                 =           7
/%d A::j, B::j, A::l, B::l
C.A::j              =           2
C.B::j              =           4
C.A::l              =           3
C.B::l              =           6
```

   Wie der Definition zu entnehmen ist, sind `A` und `B` direkte Basisklassen von `C`. Zunächst wird mit dem ersten %DISPLAY-Kommando versucht, alle Variablen über ihren Namen ohne Qualifikation auszugeben.

Die Variablen `i` und `k` kommen jeweils nur einmal vor und können daher direkt angesprochen werden. Mit `l` ohne Qualifikation erreichen Sie `C::l`. Die Variable `C:l` gehört zur Klasse `C` und verdeckt daher die gleichnamigen Variablen `A::l` und `B::l`. Diese werden im zweiten %DISPLAY qualifiziert angegeben. Zu `j` findet AID zwei Definitionen, die auf der gleichen Ebene stehen, nämlich in den Basisklassen `A` und `B`. AID meldet die Mehrdeutigkeit. Mit einer entsprechenden Qualifikation können beide Daten-Member zugeordnet werden, ebenfalls im zweiten %DISPLAY.

3. In diesem Beispiel ist die Beziehung zwischen den Klassen `A`, `B` und `C` aus Beispiel 2 folgendermaßen abgewandelt:

```
C++-Programm                              SOURCE: EX2.C
================================================================================
SRC
LIN
...
 42  class A {
 43    int i,j,l;
 44    public:
 45    A(int x,int y,int z) : i(x), j(y), l(z) {...}
 46    void func_A() {...}
 47  }
 48  class B:  public A {
 49    int j,k,l;
 50    public:
 51    B(int x,int y,int z) : j(x), k(y), l(z), A(1,2,3) {...}
 52    void func_B() {...}
 53  }
 54  class C: public A, public B {
 55    int l;
 56    public:
 57    C(int x=7) : l(x), A(4,5,6), B(8,9,10) {...}
 58    void func_C() {...}
 59  }
```

Testverlauf:

```
...
STOPPED AT SRC REF: 58 ,  SOURCE: EX2.C ,  PROC: C::func_C()
/%d i, j, k, l
SRC_REF: 58 SOURCE: EX2.C    PROC: C::func_C()  ************************************
%  AID0376 Ambiguous qualification for SYMBOL i
%  AID0376 Ambiguous qualification for SYMBOL j
C.B.k               =           9
C.l                 =           7
/%d A::i, B::A::i, A::j, B::j, B::A::j
C.A::i              =           4
C.B::A::i           =           1
C.A::j              =           5
C.B::j              =           8
C.B::A::j           =           2
```

In diesem Fall sind `A` und `B` direkte Basisklassen von `C`, und `A` ist auch indirekte Basisklasse von `C`. `A` ist nicht virtuell.
Die Variable `i` ist jetzt mehrdeutig, da `i` sowohl in der direkten als auch in der indirekten Basisklasse `A` vorkommt. Im zweiten %DISPLAY werden die beiden verschiedenen `i` über die jeweilige Klassen-Qualifikation `A::i` und `B::A::i` ausgegeben. Ebenfalls mehrdeutig ist `j`. Es gibt drei Definitionen auf verschiedenen Ebenen, nämlich in `A`, in `B` und in `B::A`. Auch diese werden mit dem zweiten %DISPLAY ausgegeben.
`k` ist wie in Beispiel 2 eindeutig, da es nur in `B` enthalten ist. `l` kann AID ebenfalls eindeutig zuordnen, da `l` aus `C` die verschiedenen anderen Definitionen in `A`, `B` und `B::A` verdeckt.

5.3.2 Konstruktor und Destruktor

Konstruktoren und Destruktoren sind Member-Funktionen einer Klasse und werden daher beim Testen mit AID genauso angesprochen wie andere Funktionen aus Klassen. Wenn Sie einen Konstruktor oder Destruktor in einer PROC-Qualifikation angeben wollen, müssen Sie den Funktionsnamen mit der Signatur in `n'...'` bzw. `t'...'` setzen, wie es auf Seite 60 beschrieben ist. Auf Seite 63 finden Sie auch ein Beispiel dazu.
Um auf die Anfangsadresse des Konstruktors oder Destruktors zuzugreifen, setzen Sie genau wie in der PROC-Qualifikation den Klassennamen in einer Klassen-Qualifikation vor den Funktionsnamen (siehe Beispiel 1 im Anschluss an diesen Abschnitt).

Unter bestimmten Umständen, wenn z.B. eine Klasse virtuelle Funktionen enthält, aber explizit kein Konstruktor definiert ist, dann generiert der Compiler einen Konstruktor, den AID in der %DISPLAY- oder %SDUMP-Ausgabe dieser Klasse anzeigt.

Konstruktoren globaler Objekte werden bei Programmstart von einer Funktion aufgerufen, die vom Compiler generiert wird und die `__STI__` heißt. Symbolisches Testen ist in `__STI__` nur bedingt möglich. Falls Sie `__STI__` in einem %AID-Kommando angeben, verwenden Sie diesen Namen ebenso wie `main` ohne Signatur.
Nach Beendigung von `main` werden die Destruktoren der globalen Klassen unmittelbar vom Laufzeitsystem aufgerufen. AID gibt die Namen der beteiligten Funktionen des Laufzeitsystems über `%SDUMP %NEST` in der Aufrufhierarchie aus.

Beispiele

1. `%in A_global::n'A_global()'`

   Auf die erste ausführbare Anweisung des Konstruktors der Klasse `A_global` aus dem Beispielprogramm Seite 63 wird ein Testpunkt gesetzt.

2. `%sd %nest`

   AID gibt die aktuelle Aufrufhierarchie aus, beginnend beim Konstruktor `A_global::A_global()` aus demselben Beispiel wie oben. Aufgerufen wurde der Konstruktor von der vom Compiler erzeugten Funktion `__STI__`. Es folgen dann Routinen des Laufzeitsystems.

```
SRC_REF: 7   SOURCE: BSP.C   PROC: A_global::A_global()   *****************
SRC_REF: 12  SOURCE: BSP.C   PROC: __STI__   ******************************
ABSOLUT: V'101CF14'   SOURCE: ICPSINI@   PROC: ICPSINI@   *****************
ABSOLUT: V'101C582'   SOURCE: ICPSINI@   PROC: ICPSINI@   *****************
ABSOLUT: V'100C0FA'   SOURCE: IPPSIN@@   PROC: IPPSINI    *****************
```

5.3.3 Virtuelle Funktionen

Virtuelle Funktionen sprechen Sie in einem AID-Kommando wie in einem C++-Programm an:

```
zeiger->n'funktion([signatur])'
```

zeiger ist der Name einer Zeigervariablen, die auf ein Objekt einer Klasse verweist, die virtuelle Funktionen enthält.

funktion([signatur])
: ist der Name einer virtuellen Funktion aus einer Klasse. Die Signatur muss entfallen, wenn sie `void` ist. Wegen der Sonderzeichen müssen Sie *funktion*`([`*signatur*`])` in `n'...'` setzen.

Wenn Sie die Syntax für die virtuellen Funktionen in einem %DISPLAY-Kommando verwenden, gibt AID die Adresse des Prologs derjenigen Member-Funktion aus, auf die *zeiger* aktuell verweist. Ebenso wird in einem %MOVE- oder %SET-Kommando mit der obigen Syntax die Prologadresse der virtuellen Funktion angesprochen. Dagegen bezeichnen Sie damit in einem %DISASSEMBLE- oder %INSERT-Kommando die Adresse der ersten ausführbaren Anweisung der virtuellen Funktion, die aktuell durch *zeiger* referenziert wird. Wenn Sie anschließend an die obige Syntax einen Pointer-Operator (->) schreiben, so adressieren Sie damit die ersten 4 Bytes ab der Anfangsadresse des Prologs der aktuellen Funktion.

Beispiel

```
C++-Programm                              SOURCE: BCL1.C
================================================================================
SRC
LIN
  1  class A
  2  {
  3    public:
  4    A() { printf ("A::A called\n"); }
  5    virtual void foo1() { printf( "A::foo1 called\n" ); }
  6    virtual void foo2() { printf( "A::foo2 called\n" ); }
  7  } a;
  8
  9  class B : public A
 10  {
 11    int i;
 12    public:
 13    B(int x = 1) : i(x) { printf ("B::B called\n"); }
 14    void foo1() { printf( "B::foo1 called\n" ); }
 15    void foo2() { printf( "B::foo2 called\n" ); }
 16  } b;
...
 30  int main()
 31  {
 32    A* aptr = &a;
 33    A* bptr = &b;
 34    bptr->foo2();
...
```

```
...
STOPPED AT SRC_REF: 34, SOURCE: BCL1.C , PROC: main
/%d bptr->n'foo2()'
SRC_REF: 34    SOURCE: BCL1.C      PROC: main  ****************************
A.foo2()    = 010005E0
/%d B
01          B
 02         A
  03        A()             = 01000000
  03        foo1()          = 01000160
  03        foo2()          = 01000270
 02         B(int)          = 01000360
 02         foo1()          = 010004C0
 02         foo2()          = 010005E0
/%da 5 from bptr->n'foo2()'
BCL1$O&@+67A      MVC   20(4,R11),4(R8)        D2 03 B020 8004
BCL1$O&@+680      L     R14,20(R0,R11)         58 E0 B020
BCL1$O&@+684      ST    R14,88(R0,R13)         50 E0 D088
BCL1$O&@+688      LR    R1,R14                 18 1E
BCL1$O&@+68A      L     R15,0(R0,R9)           58 F0 9000
/%in bptr->n'foo2()'; %r
STOPPED AT SRC_REF: 15, SOURCE: BCL1.C , PROC: B::foo2()
/%d i
SRC_REF: 15 SOURCE: BCL1.C  PROC: B::foo2()  ****************************
B.i             =           1
```

Das Programm wurde an der Source-Referenz S'35' unterbrochen. Der erste %DISPLAY gibt die Prologadresse zu der virtuellen Funktion aus, auf die die Zeigervariable `bptr` aktuell verweist. Der zweite %DISPLAY listet die Klasse `B` auf. Durch Adressenvergleich ergibt sich, dass `bptr` aktuell auf `foo2()` aus `B` zeigt. Der folgende %DISASSEMBLE bereitet die ersten 5 Assembler-Befehle der ersten ausführbaren Anweisung von `B::foo2()` auf.
Mit der Kommandofolge `%INSERT...;%RESUME` setzen Sie den Programmablauf bis zum Erreichen der ersten ausführbaren Anweisung von `B::foo2()` fort. Sie können nun wie gewohnt Daten des zugehörigen Objekts ansprechen.

5.3.4 Zeiger auf Klassen-Member (Pointer-to-Member)

Um dynamisch zur Ablaufzeit auf Daten und Funktionen aus Klassen zugreifen zu können, bietet C++ den Datentyp Pointer-to-Member an. Unterschieden wird zwischen Zeiger auf Daten-Member und Zeiger auf Member-Funktionen. In AID V2.3B können Sie Daten und Funktionen aus Klassen über Pointer-to-Member genauso ansprechen wie in C++. Wie Sie mit Pointer-to-Member beim Testen umgehen, wird in den folgenden Abschnitten ausführlich beschrieben.

5.3.4.1 Pointer-to-Data-Member

Ausgabe

Sie können sich den aktuellen Inhalt eines Zeigers auf ein Daten-Member einer Klasse, im folgenden Pointer-to-Data-Member genannt, mit %DISPLAY oder mit %SDUMP ausgeben lassen. In der Ausgabe wird der Name des Daten-Members angezeigt, auf das der Pointer-to-Data-Member aktuell verweist. Ist das Daten-Member in einer abgeleiteten oder geschachtelten Klasse definiert, so wird vor dem Namen des Datums die vollständige Klassenqualifikation aufgeführt.

```
{%DISPLAY | %SDUMP} [qua•] pointer-to-data-member
```

qua Qualifikation
Der Pointer-to-Data-Member kann wie jeder andere Zeiger über eine entsprechende Qualifikation angesprochen werden, falls er an der Unterbrechungsstelle nicht sichtbar ist.

Pointer-to-Data-Member
ist der Name eines Zeigers auf ein Daten-Member einer Klasse.

Hat ein Pointer-to-Data-Member einen ungültigen Wert, so wird die Fehlermeldung AID0545 ausgegeben.

Modifizierung

Mit %SET können Sie einen Pointer-to-Data-Member überschreiben. Als Sender zugelassen sind andere Pointer-to-Data-Member oder Adressen von Daten-Membern von Klassen. In der folgenden Syntax ist aus Platzgründen Pointer-to-Data-Member mit *ptdm* abgekürzt:

```
                ⎧[objekt•][klasse::][...]ptdm⎫
%SET [qua•]     ⎨                           ⎬ INTO [qua•][objekt•][klasse::]ptdm
                ⎩&[klasse::][...]datenname  ⎭
```

qua Qualifikation
Ist *ptdm*, *klasse* oder *datenname* an der Unterbrechungsstelle nicht sichtbar, müssen Sie eine entsprechende Qualifikation angeben.

objekt Name eines Objekts einer Klasse
Mit *objekt* und einer evtl. nachfolgenden Klassenqualifikation geben Sie den Adressierungspfad zu *ptdm* an

klasse::
Klassenqualifikation
Eine Klassenqualifikation geben Sie wie in C++ nur dann an, wenn *ptdm* oder *datenname* innerhalb einer Klassenhierarchie nicht eindeutig oder lokal verdeckt ist.

datenname
Name eines Daten-Members
Die verschiedenen Möglichkeiten, in C/C++ ein Datum anzusprechen, sind im Abschnitt „Datennamen“ auf Seite 30 beschrieben.

ptdm Name eines Zeigers auf ein Daten-Member einer Klasse

Für Sender und Empfänger muss Folgendes gelten:

– Die Klasse, die mit dem Sender verknüpft ist, muss entweder mit der des Empfängers übereinstimmen oder muss Basisklasse der Klasse des Empfängers sein und dort ein eindeutiges Subobjekt besitzen.

– Der Datentyp, auf den der zu übertragende Pointer-to-Data-Member verweist, bzw. des als Sender angegebenen Datums muss mit dem Datentyp, der vom Empfänger referenziert wird, übereinstimmen. Ist das von Sender und Empfänger referenzierte Datum jedoch vom Typ Pointer oder Pointer-to-Member, so wird nicht weiter überprüft, ob Typgleichheit zwischen den von diesen Zeigern bezeichneten Daten besteht.

Stellt AID fest, dass Sender und Empfänger die obengenannten Bedingungen nicht erfüllen, dann wird die Fehlermeldung `AID0388` ausgegeben: `Types are not convertible`.

Dereferenzierung

Über die Dereferenzierung können Sie auf den Inhalt des Datums, auf das der Pointer-to-Data-Member aktuell verweist, zugreifen. Mit %DISPLAY oder %SDUMP können Sie sich den Wert des zugeordneten Datums ausgeben lassen, mit %SET können Sie diesen Wert verändern. Der dereferenzierte Pointer-to-Data-Member kann auch in Ausdrücken verwendet werden.

Wie in C++ stehen Ihnen auch in AID zwei Möglichkeiten der Dereferenzierung zur Verfügung:

Das Objekt der Klasse wird mit seinem Namen bezeichnet. Als Dereferenzierungsoperator schreiben Sie •*

```
[qua•]objekt•*[objekt•][klasse::][...]pointer-to-data-member
```

Das Objekt der Klasse wird über einen Zeiger angesprochen. Als Dereferenzierungsoperator schreiben Sie **->***

```
[qua•]zeiger->*[objekt•][klasse::][...]pointer-to-data-member
```

qua Qualifikation
Ist das Objekt der Klasse bzw. der Zeiger auf das Objekt an der Unterbrechungsstelle nicht sichtbar, müssen Sie eine entsprechende Qualifikation angeben. In dem mit *qua* bezeichneten Programmbereich muss auch *pointer-to-data-member* sichtbar sein.

objekt Name eines Objekts einer Klasse
Vor dem •*-Operator bezeichnen Sie damit dasjenige Objekt der Klasse, das das gewünschte Datum enthält.
Nach dem •*-Operator oder dem **->***-Operator geben Sie mit *objekt* und einer evtl. nachfolgenden Klassenqualifikation den Adressierungspfad zu *pointer-to-data-member* an

zeiger Name eines Zeigers auf ein Objekt einer Klasse

klasse::
Klassenqualifikation
Eine Klassenqualifikation geben Sie wie in C++ nur dann an, wenn *pointer_to_data_member* innerhalb einer Klassenhierarchie nicht eindeutig oder lokal verdeckt ist.

pointer-to-data-member
Name eines Zeigers auf ein Daten-Member einer Klasse
pointer-to-data-member muss an der Unterbrechungsstelle sichtbar sein, bzw. falls vor dem gesamten Ausdruck eine Qualifikation angegeben wurde, so gilt diese auch für *pointer-to-data-member*.

Zwischen {*objekt* | *zeiger*} und *pointer-to-data-member* muss der folgende Zusammenhang bestehen:

Entweder beziehen sich {*objekt* | *zeiger*} und *pointer-to-data-member* auf dieselbe Klasse oder die Klasse, der *pointer-to-data-member* zugeordnet ist, ist eindeutige Basisklasse in *objekt* bzw. im mit *zeiger* referenzierten Objekt.

Beim Dereferenzieren eines Pointer-to-Data-Member können die folgenden Fehler auftreten:

– Der rechte Operand ist nicht vom Typ Pointer-to-Member (AID0546)

– Der linke Operand bezieht sich nicht auf ein Klassenobjekt (AID0547)

– Der Operand bezieht sich auf unverträgliche Klassentypen (AID0548)

– Die Klasse, auf die sich der Pointer-to-Member bezieht, enthält mehrdeutige Subobjekte (AID0549)

Beispiel

```
C++-Programm                              SOURCE: PTOM01.C
================================================================================
SRC
LIN
  1  class Z
  2  {
  3    public:
  4    int a;               // data member
  5    int Z::*ptr_to_dm_Z;
  6    int gz(int n)        // nonvirtual member function
  7      {
  8           a= 4;
  9           return n+a;
 10      };
 11  };
 12
 13  class X :
 14    public Z
 15  {
 16    public :
 17    int a, bx;           // data member
 18    int g1x(int n)       // nonvirtual member function
 19      {
 20          a= 8;
 21          return n+a;
 22       };
 23     };
 24
 25   class X  x;
 26   class X* px;
 27
 28   int X::* pdmX;
 29   int (X::* ptr_to_fun_mem) (int);
 30
 31   main()
 32   {
 33     px               = &x;
 34     pdmX             = &X::a;
 35     x.ptr_to_dm_Z    = &Z::a;
 36     x.*x.ptr_to_dm_Z = 99;
 37     px->*pdmX        = 2;
 38
 39     ptr_to_fun_mem   = &Z::gz;
 40     x.a              = x.gz(5);
 41     x.Z::a           = (x.*ptr_to_fun_mem)(7);
 42     x.bx             = x.g1x(9);
 43     return 0;
 44   }
```

```
...
STOPPED AT SRC_REF: 39, SOURCE: PTOMO1.C , PROC: main
/%d pdmX,x.ptr_to_dm_Z
SRC_REF: 39 SOURCE: PTOMO1.C      PROC: main  ****************************
pdmX            = X::a
x.ptr_to_dm_Z   = Z::a
/%d px->*pdmX,x.*x.ptr_to_dm_Z
*pdmX           =           2
*x.ptr_to_dm_Z  =          99
/%s x.ptr_to_dm_Z into pdmX
/%d px->*pdmX
*pdmX           =          99
```

Das Programm ist in Zeile 39 unterbrochen. Der erste %DISPLAY gibt die Daten-Member aus, auf die die beiden Pointer-to-Member `pdmX` und `ptr_to_dm_Z` aktuell verweisen. Der zweite %DISPLAY zeigt den Inhalt dieser Daten-Member an. Dann wird mit %SET der Pointer-to-Member `pdmX` überschrieben. Der folgende %DISPLAY gibt den Inhalt des nun durch `pdmX` referenzierten Daten-Members `Z::a` aus.

5.3.4.2 Pointer-to-Function-Member

Ausgabe

Analog zur Ausgabe des aktuellen Inhalts eines Zeigers auf ein Daten-Member einer Klasse können Sie sich den Inhalt eines Zeigers auf eine Member-Funktion, im folgenden Pointer-to-Function-Member genannt, ausgeben lassen. AID gibt den vollständigen Namen der Funktion mit Signatur aus, auf die der Pointer-to-Function-Member aktuell verweist. Ist die Funktion in einer abgeleiteten oder geschachtelten Klasse definiert, so wird vor dem Namen der Funktion die vollständige Klassenqualifikation aufgeführt. Wenn der Pointer-to-Function-Member auf eine virtuelle Funktion verweist, so wird in der Ausgabe der Name der aktuell zugeordneten Funktion ausgegeben.

```
{%DISPLAY | %SDUMP} [qua•]pointer-to-function-member
```

qua Qualifikation
: Der Pointer-to-Function-Member kann wie jede andere Variable über eine entsprechende Qualifikation angesprochen werden, falls er an der Unterbrechungsstelle nicht sichtbar ist.

Pointer-to-Function-Member
: ist der Name eines Zeigers auf eine Member-Funktion einer Klasse.

Hat ein Pointer-to-Function-Member einen ungültigen Wert, so wird die Fehlermeldung AID0545 ausgegeben.

Modifizierung

Mit %SET können Sie einen Pointer-to-Function-Member überschreiben. Als Sender zugelassen sind andere Pointer-to-Function-Member oder Adressen von virtuellen oder nichtvirtuellen Funktionen aus Klassen. In der folgenden Syntax ist aus Platzgründen Pointer-to-Function-Member mit *ptfm* abgekürzt:

```
%SET [qua•]{[objekt•][klasse::][...]ptfm  } INTO [qua•][objekt•][klasse::]ptfm
           {&[klasse::][...]funktion      }
```

qua Qualifikation
Ist *ptfm*, *klasse* oder *funktion* an der Unterbrechungsstelle nicht sichtbar, müssen Sie eine entsprechende Qualifikation angeben.

objekt Name eines Objekts einer Klasse
Mit *objekt* und einer evtl. nachfolgenden Klassenqualifikation geben Sie den Adressierungspfad zu *ptfm* an

klasse::
Klassenqualifikation
Vor *ptfm* geben Sie eine Klassenqualifikation nur dann an, wenn *ptfm* innerhalb einer Klassenhierarchie nicht eindeutig oder lokal verdeckt ist.
Vor *funktion* bezeichnen Sie mit der Klassenqualifikation die Klasse, die die Definition von *funktion* enthält.Sie müssen wie in C++ nur die Stufen der Klassenhierarchie angeben, die notwendig sind, um *funktion* eindeutig zu bezeichnen.

funktion
Name einer Member-Funktion
Wie Sie beim Testen mit AID den Namen einer C++-Funktion angeben können, ist ausführlich auf Seite 60 beschrieben.

ptfm Name eines Zeigers auf eine Member-Funktion einer Klasse

Für Sender und Empfänger muss die folgende Bedingung erfüllt sein:
Die Klasse, die mit Sender verknüpft ist, muss mit der des Empfängers übereinstimmen oder muss Basisklasse der Klasse des Empfängers sein. Andernfalls gibt AID die Fehlermeldung `AID0388` aus.

AID prüft nicht ab, ob die mit Sender und Empfänger referierten Funktionstypen übereinstimmen. Wenn im weiteren Ablauf ein Funktionsaufruf im Programm nicht zu der Funktion passt, die über den modifizierten Pointer-to-Function-Member angesprochen wird, kann dies zu Programmfehlern führen. Es liegt also in der Verantwortung des Anwenders, ob die modifizierten Funktionsaufrufe fehlerfrei ablaufen.

Dereferenzierung

Um eine durch einen Pointer-to-Function-Member bezeichnete Funktion in einem AID-Kommando ansprechen zu können, müssen Sie den Pointer-to Function-Member dereferenzieren.

Wenn Sie in einem %DISPLAY- oder %SDUMP-Kommando einen dereferenzierten Pointer-to-Function-Member angeben, wird die Anfangsadresse des Prologs der zugeordneten Member-Funktion angezeigt. Fügen Sie an den dereferenzierten Pointer-to-Function-Member eine passende Typmodifikation und einen Pointer-Operator (%al4->) an, so erhalten Sie die ersten 4 Bytes des Funktionscodes in sedezimaler Darstellung.
In %INSERT oder %REMOVE können Sie mit einem dereferenzierten Pointer-to-Function-Member einen Testpunkt angeben. Der Testpunkt wird auf die erste ausführbare Anweisung der Funktion gesetzt.
Außerdem kann mit einem dereferenzierten Pointer-to-Function-Member der zu überwachende Bereich in %CONTROL*n* oder %TRACE angegeben werden.

Wie bei der Dereferenzierung der Pointer-to-Data-Member stehen Ihnen auch hier zwei Möglichkeiten zur Verfügung:

Das Objekt der Klasse wird mit seinem Namen bezeichnet. Als Dereferenzierungsoperator schreiben Sie `.*`

```
[qua.]objekt.*[objekt.][klasse::][...]pointer-to-function-member
```

Das Objekt der Klasse wird über einen Zeiger angesprochen. Als Dereferenzierungsoperator schreiben Sie **->***

```
[qua.]zeiger->*[objekt.][klasse::][...]pointer-to-function-member
```

qua Qualifikation
Ist das Objekt der Klasse bzw. der Zeiger auf das Objekt an der Unterbrechungsstelle nicht sichtbar, müssen Sie eine entsprechende Qualifikation angeben. In dem mit *qua* bezeichneten Programmbereich muss auch *pointer-to-function-member* sichtbar sein.

objekt Name eines Objekts einer Klasse
Vor dem •*-Operator bezeichnen Sie damit dasjenige Objekt der Klasse, in der die gewünschte Funktion definiert ist.
Nach dem •*-Operator oder dem **->***-Operator geben Sie mit *objekt* und einer evtl. nachfolgenden Klassenqualifikation den Adressierungspfad zu *pointer-to-function-member* an

zeiger Name eines Zeigers auf ein Objekt einer Klasse

klasse::
Klassenqualifikation
Eine Klassenqualifikation geben Sie wie in C++ nur dann an, wenn *pointer-to-function-member* innerhalb einer Klassenhierarchie nicht eindeutig oder lokal verdeckt ist.

pointer-to-function-member
Name eines Zeigers auf eine Member-Funktion einer Klasse
pointer-to-function-member muss an der Unterbrechungsstelle sichtbar sein, bzw. falls vor dem gesamten Ausdruck eine Qualifikation angegeben wurde, so gilt diese auch für *pointer-to-function-member*

Zwischen {*objekt* | *zeiger*} und *pointer-to-function-member* muss der folgende Zusammenhang bestehen:

Entweder beziehen sich {*objekt* | *zeiger*} und *pointer-to-function-member* auf dieselbe Klasse oder die Klasse, der *pointer-to-function-member* zugeordnet ist, ist eindeutige Basisklasse in *objekt* bzw. im mit *zeiger* referenzierten Objekt.

Analog zum Dereferenzieren eines Pointer-to-Data-Members können die auf Seite 81 beschriebenen Fehler auftreten.

Beispiel

```
...
STOPPED AT SRC_REF: 40, SOURCE: PTOM01.C , PROC: main
/%d ptr_to_fun_mem,x.*ptr_to_fun_mem
SRC_REF: 40 SOURCE: PTOM01.C      PROC: main  ****************************
ptr_to_fun_mem  = Z::gz(int)
*ptr_to_fun_mem = 01000000
/%s &X::n'g1x(int)' into ptr_to_fun_mem
/%in x.*ptr_to_fun_mem;%r
STOPPED AT SRC_REF: 20, SOURCE: PTOM01.C , PROC: X::g1x(int)
```

Das Beispiel bezieht sich auf das C++-Programm, das auf Seite 82 abgebildet ist. Der Programmablauf ist in Zeile 40 unterbrochen. Zunächst wird die Funktion ausgegeben, auf die der Pointer-to-Member `ptr_to_fun_mem` aktuell verweist sowie die Adresse des Prologs

dieser Funktion. Das %SET-Kommando überschreibt den Pointer-to-Member mit der Adresse der Funktion `g1x(int)`. Mit %INSERT wird in diese Funktion ein Testpunkt gesetzt, wobei die Funktion über den dereferenzierten Pointer-to-Member angesprochen wird. %RESUME setzt den Programmablauf fort. Der Funktionsaufruf in Zeile 40 aktiviert nun statt der Funktion `gz(int)`, wie es im Programmcode vorgesehen war, die Funktion `g1x(int)`.

5.3.4.3 Vergleich von Pointer-to-Member

Pointer-to-Member können Sie in einem Subkommando vergleichen, um abhängig vom Ergebnis des Vergleichs eine Kommandofolge ausführen zu lassen. Es sind nur die Operatoren `EQ` und `NE` zugelassen. Pointer-to-Member können mit anderen Pointer-to-Member oder mit Komponenten einer Klasse verglichen werden. Das Ergebnis des Vergleichs ist `TRUE`, wenn sich beide Vergleichsoperanden auf das gleiche Member einer Klasse beziehen.

Für die Zulässigkeit des Vergleichs müssen die beiden Operanden dieselben Bedingungen erfüllen, die auch für die Modifikation gelten:

Beide Operanden müssen sich entweder auf dieselbe Klasse beziehen oder die Klasse des einen Operanden muss eindeutige Basisklasse in der Klasse des anderen Operanden sein.

Beispiel

```
...
STOPPED AT SRC_REF: 33, SOURCE: PTOM01.C , PROC: main
/%in x.n'gz(int)' <(::ptr_to_fun_mem eq &X::n'gz(int)'):                -
/%d 'ptm-Aufruf'; %sd %nest; %stop>
/%r
ptm-Aufruf
SRC_REF:  8  SOURCE: PTOM01.C  PROC: Z::gz(int)  ***************************
SRC_REF: 40  SOURCE: PTOM01.C  PROC: main  *********************************
ABSOLUT: V'113CF88'   SOURCE: IC@RT20A  PROC: IC@RT20A  ********************
ABSOLUT: V'10013D0'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  ********************
STOPPED AT SRC_REF: 8, SOURCE: PTOM01.C , PROC: Z::gz(int)
```

Das Programm, das auf Seite 82 abgebildet ist, ist diesmal am Anfang von `main` unterbrochen. Mit dem %INSERT wird auf die erste ausführbare Anweisung der Funktion `Z::gz(int)` ein Testpunkt gesetzt. Im Subkommando zu diesem %INSERT wird abgeprüft, ob die Funktion über den Pointer-to-Member `ptr_to_fun_mem` aufgerufen wurde. Falls dies zutrifft, wird der Text `'ptm-Aufruf'` sowie die aktuelle Aufrufhierarchie zur Start-Anweisung von `Z::gz(int)` ausgegeben.

5.3.4.4 Null-Setzen von Pointer-to-Member

Mit dem AID-Kommando

`%SET 0 INTO` *pointer-to-member*

erreichen Sie, dass der Inhalt des Pointer-to-Member auf binär Null gesetzt wird. Diese Möglichkeit besteht im übrigen für alle Zeiger.

Beispiel

```
/%AID CHECK=ALL
/%SET 0 INTO ptr_to_fun_mem
OLD CONTENT:
Z::gz(int)
NEW CONTENT:
00000000 00000000
%  AID0274 Change desired? Reply (Y=Yes; N=No)?n
%  AID0342 Nothing changed
```

Mit `%AID CHECK=ALL` wird der Änderungsdialog eingeschaltet:
nach dem %SET-Kommando zeigt AID den alten Inhalt des Pointer-to-Member, den Verweis auf die Funktion `gz(int)` der Klasse `Z`, und den neuen Inhalt, wie er nach der Durchführung des %SET wäre, nämlich binär Null, und fragt an, ob die Änderung ausgeführt werden soll.

5.4 Namespaces

Namespaces sind ein neues Sprachfeature in C++. Sie sind Klassen vergleichbar und dienen dazu, Namenskollisionen im globalen Namensraum zu vermeiden.

Das Ansprechen von Namespaces als Ganzem erfolgt analog dem Ansprechen von ganzen Klassen. Um Elemente von Namespaces zu adressieren, setzen Sie die Namespace-Qualifikation entsprechend ein wie die Klassen-Qualifikation, wenn Sie ein Klassen-Member ansprechen wollen. Sie können jedoch von einer Funktion eines Namespaces aus nur die Daten-Member erreichen, die bereits vor der Funktion definiert sind.
Namespaces können, wie auch Klassen, ineinander verschachtelt sein. Dabei kann der äußerste Namespace nur global definiert sein.

Die in einem Namespace definierten Daten sind stets statisch.

Wenn Sie einen Namespace in einem %DISPLAY- oder %SDUMP-Kommando angeben, wird der gesamte Namespace mit allen darin definierten Variablen und Funktionen ausgegeben. Die Ausgabe wird wie eine Struktur aufbereitet.
Zu den Variablen wird der aktuelle Wert angezeigt, die Funktionen werden mit ihrem vollständigen Funktionsnamen mit Signatur und der Anfangsadresse des Prologs aufgelistet. Im Namespace verschachtelte weitere Namespaces werden ebenfalls vollständig ausgegeben.

Mit einem %ALIAS-Kommando können Sie einem Namespace einen Aliasnamen zuweisen, unter dem Sie den Namespace in darauffolgenden Kommandos ansprechen können. In der Ausgabe eines %SDUMP ohne Operanden wird der Namespace jedoch stets unter seinem Originalnamen aufgeführt. Wird für den Namespace dagegen im C++-Programm ein Aliasname vergeben, so listet AID den Namespace sowohl unter seinem Originalnamen als auch unter dem Aliasnamen auf.
In allen anderen AID-Kommandos, die einen Adressoperanden verlangen, können Sie einen Namespace nur als Qualifikation für einen nachfolgenden Daten- oder Funktionsnamen angeben.

```
[E-qua•][S-qua•][::]namespace[::...][klasse[::...]][{datenname }]
                                                    {funktion  }
```

E-qua Basisqualifikation
Sie wird mit `E=VM` oder `E=D`*n* angegeben und legt fest, ob der AID-Arbeitsbereich im geladenen Programm (`E=VM`) oder in einer Dump-Datei (`E=D`*n*) liegen soll. Die Basisqualifikation wird beim symbolischen Testen und beim maschinennahen Testen gleich verwendet und ist im AID-Basishandbuch, Kapitel „Adressierung in AID“ und im Kommando %BASE auf Seite 132 beschrieben.

S-qua S-Qualifikation
Damit bezeichnen Sie die Übersetzungseinheit, die den Namespace enthält. Die S-Qualifkation geben Sie an, wie es auf Seite 23 beschrieben ist.

namespace
Name eines Namespaces
Um in einem geschachtelten Namespace den Namespace einer inneren Schachtelung anzusprechen, geben Sie im Pfad zu dem gewünschten Namespace von außen nach innen die Namen der äußeren Namespaces, jeweils mit zwei Doppelpunkten abgeschlossen, an.

Namespaces können nur global definiert werden. Daher ist vor *namespace* nur eine Basisqualifikation, eine S-Qualifikation oder eine `::`-Qualifikation für den globalen Namespace möglich, oder Sie geben die Namespace-Qualifikation eines oder mehrerer übergeordneter Namespaces an, um den Pfad zu einem geschachtelten Namespace einer unteren Stufe zu beschreiben.

klasse Name einer Klasse
Bezeichnet *klasse* die Instanz eines Klassentemplates, so müssen Sie die Schreibweise `t'`*k_template*`<`*arg*`[,...]>'` verwenden. Gibt es zu dem Klassentemplate nur eine einzige Instanz, so reicht die Angabe: `t'`*k_template*`'`.
Zwischen Klassennamen und anschließendem Funktionsnamen schreiben Sie wie gewohnt zwei Doppelpunkte (siehe auch Seite 65).

funktion
Name einer Funktion
Ausführliche Informationen, wie Sie einen Funktionsnamen beim Testen von C++-Programmen angeben, finden Sie auf Seite 60.

datenname
Name eines Datums
Die verschiedenen Möglichkeiten, in C/C++ ein Datum anzusprechen, sind im Abschnitt „Datennamen“ auf Seite 30 beschrieben.

5.4.1 Unbenannte Namespaces

Unbenannte Namespaces können Sie mit %DISPLAY oder %SDUMP nicht als Ganzes ansprechen. Nur in der Ausgabe eines %SDUMP ohne Operanden wird ein unbenannter Namespace vollständig aufgelistet. Dabei wird in der ersten Zeile, die normalerweise nach der Stufennummer 1 den Namen des Namespaces enthält, nur die Stufennummer ausgegeben.

Variablen und Funktionen aus unbenannten Namespaces können nicht qualifiziert werden. Daher sind sie für AID nur dort sichtbar, wo sie auch in C++ ohne Qualifikation referenziert werden können, d.h. wo der gesamte Namespace sichtbar ist. Sie verhalten sich wie globale statische Variablen bzw. Funktionen.

Beispiel

Der Namespace sei folgendermaßen definiert:

```
namespace {int i = 1; int j = 10;}
```

Die Ausgabe des unbenannten Namespaces mit dem Kommando %SDUMP wird in der folgenden Form aufbereitet:

```
01
 02        i               =           1
 02        j               =          10
 02        using           = // UNNAMED //
```

5.4.2 Scoperegeln in Namespaces

AID bildet die Scoperegeln, die in C++ für das Arbeiten mit Namespaces gelten, nach. An der Unterbrechungsstelle nicht sichtbare Variablen und Funktionen können Sie wie in C++ jederzeit über die zugehörige Namespace-Qualifikation *namespace*`::` ansprechen.

using-Deklaration / using-Direktive

Ist das Programm nach einer `using`-Deklaration für eine einzelne Variable aus einem Namespace bzw. nach einer `using`-Direktive für einen gesamten Namespace unterbrochen, so können Sie die deklarierte Variable bzw. alle Variablen und Funktionen aus dem Namespace ohne Qualifikation ansprechen.
Die durch eine `using`-Deklaration eingeführte Variable verhält sich wie eine lokal definierte. Eine im aktuellen Gültigkeitsbereich bereits vorhandene gleichnamige Variable wird durch die Definition aus dem Namespace verdeckt und kann nur noch vollqualifiziert angesprochen werden.
Dagegen wird die durch eine `using`-Direktive bekannt gemachte Variable eines Namespaces dem Scope des übergeordneten Namespaces zugeschlagen. In diesem Fall verdeckt eine gleichnamige lokale Variable die Definition aus dem Namespace.
Tritt durch das Arbeiten mit Namespaces Gleichnamigkeit bei Funktionen auf, so werden diese überladen.

Wenn durch die Verwendung von Variablen aus Namespaces Mehrdeutigkeiten entstehen, weil es z.B. gleichnamige globale Definitionen gibt oder weil geschachtelte Namespaces gleichnamige Definitionen auf verschiedenen Stufen enthalten, so können Sie diese Definitionen nicht direkt ansprechen. AID gibt bei Mehrdeutigkeit die folgende Meldung aus:
`AID0529 Symbol` *name* ambiguous because of using directive .
Mit AID können Sie die mehrdeutigen Definitionen über eine entsprechende Qualifikation unterscheiden.

Klassen in Namespaces

Auch wenn Klassen in Namespaces enthalten sind, unterstützt AID die in C++ gültigen Scoperegeln.

Im folgenden wird davon ausgegangen, dass das Programm in einer dynamischen Funktion einer abgeleiteten Klasse unterbrochen ist. Die Definition der Klasse soll in einem Namespace enthalten sein. Wenn Sie von dieser Unterbrechungsstelle aus ein Datum direkt, also ohne Qualifikation, ansprechen, so gilt die folgende Suchreihenfolge:

1. lokaler Scope der dynamischen Funktion, in der das Programm unterbrochen ist, und zwar vor der Unterbrechungsstelle,
2. Scope der abgeleiteten Klasse, die die Funktion enthält,
3. Scope der Basisklassen,
4. Scope des Namespaces, der die Klassendefinition enthält,
5. Scope von übergeordneten Namespaces,
6. globaler Scope vor der Unterbrechungsstelle.

AID durchsucht der Reihe nach sämtliche in Frage kommenden Gültigkeitsbereiche. Werden mehrere Treffer gefunden, prüft AID, ob der erste Treffer die weiteren verdeckt oder ob Mehrdeutigkeit vorliegt. Im zweiten Fall gibt AID eine Fehlermeldung aus (`AID0376` oder `AID0529`).

Beispiele

1. *Namespaces und using*

   Das Beispiel bezieht sich auf folgendes Programmfragment:

```
C++-Programm                              SOURCE: EXNSP1.C
================================================================================
SRC
LIN
  1 namespace PART1
  2 {
  3    int func_1(int) {...}
  4    int j = 88;
  5    int k = 99;
  6 }
  7 namespace SNI
  8 {
  9    void func_1() {...}
 10    using namespace PART1;
 11    int i = 19; int j = 20;
 12 }
 13 int k = 0;
 14
 15 int main()
 16 {
 17    k++;
 18    using SNI::i;
 19    i = k+10;
 20    using namespace PART1;
 21    j = 5; i = j + 10;
 22    ...
```

Testverlauf:

```
STOPPED AT SRC_REF: 19, SOURCE: EXNSP1.C , PROC: main
/%d i, k, SNI
SRC_REF: 19 SOURCE: EXNSP1.C  PROC: main  ****************************************
i               =          19
k               =           1
01          SNI
 02         func_1()        = 01000088
 02         i               =          19
 02         j               =          20
 02         using           = PART1
```

An der Unterbrechungsstelle S‘19‘ sind die Variablen `i` und `k` eindeutig ansprechbar. Mit `i` adressieren Sie wegen der `using`-Deklaration in Zeile 18 die Variable aus dem Namespace `SNI`; mit `k` erreichen Sie das globale `k` aus Zeile 13. In der Ausgabe des Namespaces `SNI` finden Sie alle darin enthaltenen Komponenten sowie den Hinweis, dass Namespace `PART1` über eine `using`-Direktive im Namespace `SNI` angemeldet wurde.

```
/%in S'21';%r
STOPPED AT SRC_REF: 21, SOURCE: EXNSP1.C , PROC: main
/%d i, j, k, n'func_1(int)', PART1
SRC_REF: 21 SOURCE: EXNSP1.C  PROC: main  ****************************************
i               =          11
j               =          88
%  AID0529 Symbol k ambiguous because of using directive.
func_1(int)     = 01000000
01          PART1
 02         func_1(int)     = 01000000
 02         j               =          88
 02         k               =          99
/%d PART1::k, ::k
PART1::k        =          99
k               =           1
```

Mit der Kommandofolge `%INSERT S‘21‘;%RESUME` wird das Programm bis zur Source-Referenz S‘21‘ ausgeführt. An dieser Stelle können Sie `i` und `j` eindeutig ansprechen. Mit `i` adressieren Sie wie zuvor die Variable aus dem Namespace `SNI`; mit `j` sprechen Sie die Variable `PART1::j` an. `k` ist jetzt mehrdeutig, da es außer der globalen Variablen `k` noch ein `k` aus Namespace `PART1` gibt. Die beiden Variablen `k` können jedoch qualifiziert angesprochen werden.

2. *Namespaces und Klassen*

   In diesem Beispiel sind Namespaces und Klassen ineinander verschachtelt:

```
C++-Programm                              SOURCE: EXNSP2.C
================================================================================
SRC
LIN
  1 class A
  2 {
  3    int i,j;
  4    public:
  5    A(int x = 1) : i(x) {j=2; ...}
  6    void func_A() {i=j*i;...}
  7 }a;
  8
  9 class B
 10 {
 11    int j,l;
 12    public:
 13    B(int x = 4) : j(x) {l=6; ...}
 14    void func_B() {...}
 15 };
 16
 17 namespace M
 18 {
 19    int k=1,l=2,m=3;
 20    void func_M() {...}
 21    namespace N
 22    {
 23       int i=4,j=5,k=6;
 24       void func_N() {...}
 25       class X: public A, public B
 26       {
 27          int l;
 28          public:
 29          X(int x = 7) : l(x) {...}
 30          void func_X() {l++;...}
 31       }x;
 32    }
 33 }
 34 int main()
 35 {
 36    using namespace M;
 37    using namespace N;
 38    x.func_X();
...
```

Testverlauf:

```
/%t 1 in proc=main
38                        EXT.PROC START    , BLOCK START, ASSIGN
STOPPED AT SRC_REF: 38, SOURCE: EXNSP2.C , PROC: main , END OF TRACE
/%d M
SRC_REF: 38    SOURCE: EXNSP2.C      PROC: main  ***********************************
 01          M
  02         N
   03        X
    04       A
     05      A(int)          = 01000000
     05      func_A()        = 01000168
    04       B
     05      B(int)          = 010001D0
     05      func_B()        = 01000338
    04       X(int)          = 01000468
    04       func_X()        = 01000600
   03        func_N()        = 01000410
   03        i               =          4
   03        j               =          5
   03        k               =          6
   03        x
    04       A
     05      i               =          1
     05      j               =          2
     05      A(int)          = 01000000
     05      func_A()        = 01000168
    04       B
     05      j               =          4
     05      l               =          6
     05      B(int)          = 010001D0
     05      func_B()        = 01000338
    04       l               =          7
    04       X(int)          = 01000468
    04       func_X()        = 01000600
  02         func_M()        = 010003B8
  02         k               =          1
  02         l               =          2
  02         m               =          3
```

Zunächst wird das Programm mit dem %TRACE-Kommando auf die erste ausführbare Anweisung in `main` positioniert. Mit `%DISPLAY M` lassen Sie sich den gesamten Namespace `M` mit dem darin enthaltenen weiteren Namespace `N` sowie der Klasse `X` auflisten.

```
/%in n'func_X()'; %r
STOPPED AT SRC_REF: 30, SOURCE: EXNSP2.C , PROC: M::N::X::func_X()
/%d i, j, k,l , m
SRC_REF: 30    SOURCE: EXNSP2.C      PROC: M::N::X::func_X()  **********************
X.A.i             =          1
%  AID0376 Ambiguous qualification for SYMBOL j
k                 =          6
X.l               =          7
m                 =          3
/%d M::N::i
M::N::i           =          4
/%d A::j, B::j, M::k
X.A::j        =       2
X.B::j        =       4
M::k          =       1
```

Im nächsten Schritt wird das Programm bis zum Anfang der Funktion `func_X()`, die in der Klasse `X` definiert ist, ausgeführt. Von dieser Stelle aus werden die Variablen `i`, `j`, `k`, `l` und `m` direkt, d.h. ohne explizite Qualifikation, angesprochen:

- Mit `i` erreichen Sie die Variable `i` aus der Basisklasse `A`; das `i` aus Namespace `N` ist davon verdeckt und kann nur über die volle Qualifikation, also mit `M::N::i`, referenziert werden, siehe nächster %DISPLAY.
- `j` ist mehrdeutig, da `j` in beiden Basisklassen vorkommt. Im dritten %DISPLAY werden die beiden Variablen `j` über die entsprechende Klassen-Qualifikation ausgegeben.
- `k` ist eindeutig, da `k` aus Namespace `N` die gleichnamige Variable aus Namespace `M` verdeckt. Die Variable aus `M` wird im dritten %DISPLAY vollqualifiziert, also mit `M::k`, ausgegeben.
- `l` ist in Klasse `X` definiert und verdeckt somit alle übrigen Variablen mit diesem Namen.
- `m` ist eindeutig, da es nur in Namespace `M` enthalten ist.

5.4.3 Aliasnamen für Namespaces

In C++ können Sie für benannte Namespaces weitere Namen, sog. Aliasnamen, vergeben, um nicht jedesmal den vollständigen, u.U. sehr langen Originalnamen eintippen zu müssen. Sie können dabei einem Namespace mehrere Aliasnamen zuordnen. Beim Testen mit AID können Sie den Namespace mit seinem Originalnamen und mit sämtlichen zugeordneten Aliasnamen ansprechen, sofern die Unterbrechungsstelle im Gültigkeitsbereich des Aliasnamen liegt.

Wenn Sie sich mit %SDUMP ohne Operanden eine Liste aller Definitionen des Programms ausgeben lassen, so wird der Namespace vollständig sowohl unter dem Originalnamen als auch unter allen Aliasnamen aufgelistet.

Einen entsprechenden Mechanismus bietet AID mit dem Kommando %ALIAS an (siehe Seite 129).

Beispiel

Aliasvereinbarung im C++-Programm:

```
namespace FJ = Fujitsu;
```

Einer Variablen `i`, die im Namespace `Fujitsu` definiert ist, weisen Sie mit dem folgenden %SET-Kommando einen Wert zu. Dabei wird der Kurzname `FJ` verwendet. Der nachfolgende %DISPLAY gibt den Inhalt von `FJ::i` aus.

```
/%set 20 into FJ::i
/%display FJ::i
Fujitsu::i                =       20
```

5.5 Templates

Templates sind neu in C++. Unterschieden werden Klassentemplates und Funktionstemplates. Ein Template ist dabei lediglich ein Muster, aus dem erst durch Parametrisierung, d.h. durch Angeben von Template-Argumenten, konkrete Klassen oder Funktionen, sog. Template-Instanzen, hervorgehen. Für das Testen von Template-Instanzen mit AID ergibt sich daher die folgende Problematik:

– Zu einer Template-Deklaration kann es mehrere verschiedene Instanzen geben.

– Die Source-Referenzen dieser mehrfachen Instanzen leiten sich von der Template-Deklaration ab und sind daher nicht mehr eindeutig.

– In die Namensbildung der Instanzen gehen die Template-Argumente mit ein; diese können bei AID Literale, Adresskonstanten oder Typparameter sein. Die Namen der Typparameter müssen sich von allen anderen Definitionen im Programm unterscheiden.

5.5.1 Template-Instanziierung

Im Programm entsteht immer dann eine Instanz eines Templates, wenn das Template mit konkreten Argumenten versehen wird. Eine Instanz eines **Funktionstemplates** wird angelegt, wenn der Funktionsname aus der Template-Deklaration mit bestimmten Argumenten aufgerufen wird. Ebenso entsteht aus einem **Klassentemplate** eine konkrete Klasse, wenn die Platzhalter in der Template-Deklaration durch echte Werte oder Typdefinitionen ersetzt werden.

Wenn Sie sich zunächst eine Übersicht über die im Programm angelegten Template-Instanzen verschaffen wollen, so geben Sie den Template-Namen in einem %DISPLAY-Kommando ein: `%DISPLAY` *template*. Wurde zu einem Template nur eine einzige Instanz angelegt, so lassen Sie sich diese mit `%DISPLAY t'`*template*`'` ausgeben (siehe auch Abschnitt Abschnitt „Auflisten von Template-Instanzen“ auf Seite 110). Um jedoch eine konkrete Instanz mit AID anzusprechen, müssen Sie, wie in der Sprache C++, anschließend an den in der Template-Deklaration festgelegten Namen die zugehörigen Template-Argumente schreiben, die wie in C++ in spitze Klammern gesetzt werden müssen. Damit AID erkennt, dass es sich um eine Template-Instanz handelt, müssen Sie den Instanznamen in `t‘...‘` angeben.
Unmittelbar bei der Eingabe prüft AID, ob die Template-Instanz syntaktisch richtig angegeben wurde. Unzulässige Angaben lehnt AID mit Syntaxfehler ab. Außerdem prüft AID, ob im Programm eine Template-Instanz mit dem angegebenen Namen angelegt wurde. Falls die Template-Instanz nicht gefunden werden kann, gibt AID eine entsprechende Fehlermeldung aus.

Syntax für Instanznamen von Klassentemplates:

```
t'k_template<arg[,...]>'
```

Syntax für Instanznamen von Funktionstemplates:

```
t'f_template<arg[,...]>([signatur])'
```

Gibt es nur eine einzige Instanz zu einer Template-Deklaration, so können Sie diese in der Kurzform `t'`*k_template*`' bzw. t'`*f_template*`([`*signatur*`])'` **ohne** Angabe der Template-Argumente ansprechen. Falls mehrere Instanzen existieren, referenziert die Kurzform genau eine dieser Instanzen; welche Instanz ausgewählt wird, ist jedoch nicht vorhersehbar.

template
: Mit *template* geben Sie den Namen der Template-Deklaration an.

arg Für *arg* sind bei AID die folgenden Angaben möglich:

– Literal

  Ein Literal kann eine Zahl sein, z.B. 15, -15, 1.4, ein Character oder eine Zeichenkette. Einzelne Character-Zeichen oder Zeichenketten sind in doppelte Hochkommas (`''`) einzuschließen, z.B. `''a''`, `''ABC''`. Wie in C++ können Sie für ein einzelnes Character-Zeichen auch das numerische Äquivalent angeben. Z.B. sprechen Sie mit `t'CC<''a''>'` und `t'CC<129>'` dieselbe Template-Instanz an. Den numerischen Werten der Zeichen liegt die EBCDIC-Tabelle eines /390-Rechners zugrunde. Ein Beispiel zu diesem Thema finden Sie auf Seite 104.
  Den numerischen Wert eines Character-Zeichens können Sie mit AID über den folgenden Umweg ermitteln:

  `%DISPLAY '`*character*`'%X` gibt den zugehörigen Sedezimalwert aus,

  `%DISPLAY #'`*sedezimal-zahl*`'%F` gibt den zugehörigen Dezimalwert aus.

– Adresskonstanten

  Adresskonstanten, wie z.B. Adressen von Funktionen (`foo`) oder von Variablen (`&var`), etc. werden wie im C++-Programm angegeben.

– elementarer Datentyp

  Den elementaren Datentypen aus C++ sind bestimmte Template-Argumente zugeordnet. Die in der folgenden Tabelle links aufgeführten Namen von Datentypen und die rechts zugeordneten Template-Argumente sind äquivalent, d.h. Sie können bei AID sowohl die Namen der Datentypen als auch die zugeordneten Template-Argumente verwenden und sprechen damit jeweils dieselbe Template-Instanz an:

Datentyp	**Template-Argument**
`char`	`char`
`unsigned char`	`unsigned char`
`signed char`	`signed char`
`bool`	`bool`
`unsigned`	`unsigned int`
`unsigned int`	`unsigned int`
`signed`	`int`
`signed int`	`int`
`int`	`int`
`unsigned short int`	`unsigned short  int`
`unsigned short`	`unsigned short  int`
`unsigned long int`	`unsigned long int`
`unsigned long`	`unsigned long int`
`signed long int`	`long int`
`signed long`	`long int`
`long int`	`long int`
`long`	`long int`
`signed short int`	`short int`
`signed short`	`short int`
`short int`	`short int`
`short`	`short int`
`wchar_t`	`wchar_t`
`float`	`float`

Tabelle 2: elementare Datentypen und zugeordnete Template-Argumente

Datentyp	Template-Argument
`double`	`double`
`long double`	`long double`
`void`	`void`

Tabelle 2: elementare Datentypen und zugeordnete Template-Argumente

– abgeleiteter Datentyp

  Abgeleitete Datentypen werden aus elementaren Typen gebildet, indem diese mit den Operatoren `*`, `&` oder `[]` verknüpft werden.

  Beispiele für abgeleitete Typparameter sind in der folgenden Tabelle zusammengestellt:

Definition im C++-Programm	Typparameter
`int *pi`	`int*`
`int *p[3]`	`int* [3]`
`int (*pi) [3]`	`int (*)[3]`
`int *f()`	`int*()`
`int (*pf) (double)`	`int (*)(double)`

Tabelle 3: Beispiele abgeleiteter Typparameter

– benutzerdefinierter Datentyp

  Sie können bei AID wie in C++ Datentypen verwenden, die Sie selbst im Programm definiert haben.
  Beispiel
  `typedef int (*int_arr_def)[3]` definiert den Datentyp `int(*)[3]` und ordnet ihm den Namen `int_arr_def` zu. Den Datentyp sowie den zugeordneten Namen können Sie bei AID als Template-Argument verwenden und bezeichnen damit dieselbe Template-Instanz.

– nicht zulässig ist es bei AID, statt eines Literals einen Ausdruck anzugeben. Diese Möglichkeit führt bei AID zu einem Syntaxfehler; z.B. würden Sie in C++ durch die Angabe von `foo<40>` und `foo<20*2>` dieselbe Instanz eines Funktionstemplates, nämlich `foo<40>`, bezeichnen, oder es wäre `foo<(1>2)>` gleichbedeutend mit `foo<(false)>`. Die Angaben `foo<20*2>` oder `foo<(1>2)>` weist AID mit Syntaxfehler ab.

Beispiel

```
C++-Programm                                              SOURCE: EXTMP3.C
================================================================================
SRC
LIN
  1 template <class T, T* address> class T1
  2 {
  3    public:
  4    void foo() { (*address)++; }
  5    void bar() { (*address)(); }
  6 };
  7
  8 template <class S, class T, S* address, T (S::*ptm)()> class T2
  9 {
 10 public:
 11    void foo() { (address->*ptm)(); }
 12 };
 13
 14 template <class S, class T, S* address, T (S::*ptm)()> class T3
 15 {
 16 public:
 17    void bar() { (address->*ptm)(); }
 18 };
 19
 20 int i = 0;
 21 void f() { i--; }
 22 struct S
 23 {
 24    int i;
 25    void f() { i--; }
 26 } s;
 27 class X
 28 {
 29 public:
 30    void operator++() {};
 31    void operator++(int) {};
 32    void operator() () {};
 33 } x;
 34
 35 T1<X,&x> t1_x;
 37 T2<S, int, &s, &S::i>  t2_i;
 38 T3<S, void, &s, &S::f> t3_f;
 39 ...
```

Im folgenden werden mit den drei %DISPLAY-Kommandos die Inhalte der Template-Instanzen aufgelistet; zu jeder Funktion der Templates wird die Prologadresse ausgegeben. Die Instanzen wurden hier mit ihrem vollständigen Namen angesprochen. Da zu jedem Template in diesem Beispiel nur je eine Instanz erzeugt wurde, könnten Sie die Instanzen auch mit ihrem Kurznamen ansprechen: `%DISPLAY t'T1', t'T2', t'T3'` würde zum selben Ergebnis führen.

```
/%d t'T1<X,&x>'
01          T1<X,&x>
 02         foo()           = 01000468
 02         bar()           = 01000578
/%d t'T2<S,int,&s,&S::i>'
01          T2<S,int,&s,&S::i>
 02         foo()           = 01000678
/%d t'T3<S,void,&s,&S::f>
01          T3<S,void,&s,&S::f>
 02         bar()           = 010006F8
```

5.5.2 Klassentemplates

Instanzen bzw. Objekte von Instanzen von Klassentemplates können Sie mit AID testen, wie andere Klassen bzw. Objekte von Klassen auch. Die zugeordneten Gültigkeitsbereiche unterscheiden sich nicht von denen anderer Klassen. Durch die Angabe der Template-Argumente ist die Instanz der Klasse eindeutig festgelegt und daher gelten die im Abschnitt „Klassen“ auf Seite 65 beschriebenen Regeln für das Ansprechen von Daten und Funktionen genauso auch für Member aus Instanzen von Klassentemplates. In der Syntax auf Seite 66 müssen Sie den Klassennamen so angeben, wie oben im Abschnitt „Template-Instanziierung“ beschrieben steht.

Beispiel

```
C++-Programm                                              SOURCE: EXTMP2.C
================================================================================
SRC
LIN
  1  #include <iostream.h>
  2  template <class T, int I>
  3  class XX  {
  4  public:
  5     void foo(T& t)  {
  6     --t;
  7     cout << "In XX:foo<>(t=" << t <<", I=" << I << ")\n";
  8     my_t[I-1]=t;
  9     }
 10     T my_t[I];
 11     static char *cptr;
 12  };
 13  template<> char *XX<int,10>::cptr = "XX<int,10>::cptr";
 14  template<> char *XX<int,12>::cptr = "XX<int,12>::cptr";
 15
 16  template <class W> class X {
 17  W x[5];
 18  public:
 19     int foo(int j)  {
 20     cout << "X<>:foo(" << j << ")\n";
 21     return x[j];
 22     }
 23  };
 24
 25  template <typename T> class Y {
 26  T t;
 27  public:
 28     int foo(int i)  {
 29     cout << "Y<>i::foo(" << i << ")\n";
 30     return t.foo(i);
 31     }
 32  };
 33
 34  template <char C>
 35  class CC {
 36  public:
 37     void foo() {
 38     cerr << "CC<" << C << ">::foo()\n";
 39     }
 40  };
 41
```

```
42  int main ()
43  {
44  int i = 0;
45  CC<'c'> c1;
46  CC<195> c2;
47  CC<1>   c3;
48  XX<int,10> xi10;
49  XX<int,12> xi12;
50  Y<X<int> > y;
51
52  c1.foo();
53  c2.foo();
54  c3.foo();
55  xi10.foo(i);
56  xi12.foo(i);
57  y.foo(2);
58  return(0);
59  }
```

Testverlauf:

```
/%t 1 in s=n'extmp2.c'
44                      EXT.PROC START    , BLOCK START, ASSIGN
STOPPED AT SRC_REF: 44, SOURCE: EXTMP2.C , PROC: main , END OF TRACE
/%d CC
01          CC
 02         CC<1>
  03        foo()           = 01005B00
 02         CC<'C'>
  03        foo()           = 010059C0
 02         CC<'c'>
  03        foo()           = 01005880
/%t 1 in t'CC<''c''>'::n'foo()'
49                      EXT.PROC START    , BLOCK START, CALL
STOPPED AT SRC_REF: 49, SOURCE: EXTMP2.C , PROC: CC<'c'>::foo() , END OF
TRACE
/%t 1 in t'CC<195>'::n'foo()'
49                      EXT.PROC START    , BLOCK START, CALL
STOPPED AT SRC_REF: 49, SOURCE: EXTMP2.C , PROC: CC<'C'>::foo() , END OF
TRACE
/%t 1 in t'CC<1>'::n'foo()'
49                      EXT.PROC START    , BLOCK START, CALL
STOPPED AT SRC_REF: 49, SOURCE: EXTMP2.C , PROC: CC<1>::foo() , END OF
TRACE
```

Zunächst wird mit dem %TRACE auf die erste ausführbare Anweisung in `main` positioniert. An dieser Stelle benötigen Sie zum Ansprechen von Daten und Anweisungen nur die dem jeweiligen Scope entsprechenden Qualifikationen und müssen nicht jedesmal die vollständige Qualifikation, beginnend mit der S-Qualifikation, schreiben.

Das nachfolgende %DISPLAY-Kommando liefert die Übersicht über alle zum Klassentemplate `CC` erzeugten Instanzen. Bei der Instanz CC<195> ist zu beachten, dass AID intern 195 in das Character-Zeichen `'C'` umsetzt und die im Programm mit CC<195> definierte Instanz mit CC<'C'> ausgibt. Dessen ungeachtet können Sie im weiteren Testverlauf diese Instanz weiterhin mit CC<195> ansprechen. Umgekehrt könnten Sie die Instanz CC<'c'> auch mit CC<131> ansprechen, da 131 das numerische Äquivalent für 'c' ist.

Anschließend wird durch 3 %TRACE-Kommandos jeweils die erste Anweisung jeder Instanz protokolliert: es ist dies stets die Anweisung in der Zeile 49. Dies hängt damit zusammen, dass die verschiedenen Instanzen aus demselben Template hervorgehen und daher die Source-Referenzen in jeder Instanz gleich sind.

```
/%d XX
01          XX
 02         XX<int,12>
  03        foo(int &)      = 010003B0
  03        cptr            = 0100223D
 02         XX<int,10>
  03        foo(int &)      = 01000208
  03        cptr            = 0100222C
/%in t'XX<int,12>'::n'foo(int &)';%r
STOPPED AT SRC_REF: 10, SOURCE: EXTMP2.C , PROC: XX<int,12>::foo(int &)
/%sd proc=t'XX<int,12>'::n'foo(int &)'
SRC_REF: 10  SOURCE: EXTMP2.C  PROC: XX<int,12>::foo(int &)  **************
this            = 01058838

01          XX<int,12>
 02         my_t( 0: 11)
            (  0)    16813944  (  1)           0  (  2)           0
            (  3)           0  (  4)           0  (  5)           0
            (  6)           0  (  7)           0  (  8)           0
            (  9)           0  ( 10)    17238352  ( 11)    17882652
 02         foo(int &)      = 010003B0
 02         cptr            = 0100223D

t               =           0
```

Im nächsten Schritt werden alle zum Template `XX` angelegten Instanzen aufgelistet. Auf die Funktion `foo(int&)` der Instanz `XX<int,12>` wird ein Testpunkt gesetzt und mit %RESUME das Programm gestartet. Das Programm wird in `foo(int&)` unterbrochen. Der anschließende %SDUMP listet alle Daten der Funktion `XX<int,12>::foo(int &)` auf.

```
/%in t'Y<X<int>>'::n'foo(int)' <%d i;%sd %nest;%d ' '>
/%in t'X<int>'::n'foo(int)' <%d j;%sd %nest;%d ' '>;%r
SRC_REF: 41 SOURCE: EXTMP2.C  PROC: Y<X<int>>::foo(int)  ******************
i               =           2
SRC_REF: 41  SOURCE: EXTMP2.C  PROC: Y<X<int>>::foo(int)  ******************
SRC_REF: 75  SOURCE: EXTMP2.C  PROC: main  ********************************
ABSOLUT: V'113CF88'   SOURCE: IC@RT20A  PROC: IC@RT20A  ********************
ABSOLUT: V'1002E48'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  ********************

SRC_REF: 33 SOURCE: EXTMP2.C  PROC: X<int>::foo(int)  **********************
j               =           2
SRC_REF: 33  SOURCE: EXTMP2.C  PROC: X<int>::foo(int) **********************
SRC_REF: 42  SOURCE: EXTMP2.C  PROC: Y<X<int>>::foo(int)  ******************
SRC_REF: 75  SOURCE: EXTMP2.C  PROC: main  ********************************
ABSOLUT: V'113CF88'   SOURCE: IC@RT20A  PROC: IC@RT20A  ********************
ABSOLUT: V'1002E48'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  ********************
```

Abschließend wird in die beiden Funktionen `foo(int)` der Instanzen `Y<X<int>>` und `X<int>` jeweils ein Testpunkt gesetzt, bei dessen Durchlaufen die Funktionsparameter `i` bzw. `j` sowie die aktuelle Aufrufhierarchie ausgegeben wird.

5.5.3 Funktionstemplates

Durch die Instanziierung entsteht aus einem Funktionstemplate eine konkrete Funktion. Wenn Sie diese Funktion in einer PROC-Qualifikation verwenden, um die Definitionsstelle eines darin enthaltenen Datums anzugeben, oder wenn Sie in einem AID-Kommando den Funktionsnamen einsetzen, um die Anfangsadresse der Funktion zu bezeichnen, sprechen Sie die Funktion über den Funktionsnamen mit den Template-Argumenten, die in spitze Klammern geschrieben werden müssen, und der Signatur an. Diese Angaben setzen Sie in `t'...'`, so wie es im Abschnitt „Template-Instanziierung“ auf Seite 98 beschrieben ist.

Beispiel

```
C++-Programm                                          SOURCE: EXTEMPL1.C
================================================================================
SRC
LIN
  1 // minimum of two objects
  2 template< class T >
  3 const T& minimum( const T& a, const T& b )
  4 {
  5    const T& retval( (a<b) ? a : b );
  6    return retval;
  7 }
  8
  9 // minimum of three objects
 10 template< class T >
 11 const T& minimum( const T& a, const T& b, const T& c )
 12 {
 13    const T& retval( (a<b) ? minimum(a,c) : minimum(b,c) );
 14    return retval;
 15 }
 16
 17 int main()
 18 {
 19    int min;
 20    double xmin;
 21    min  = minimum(2,3,1);
 22    xmin = minimum(2.2,1.1,3.3);
 23 ...
```

Testverlauf:

```
/%in t'minimum<double>(const double &, const double &)'
/%r
STOPPED AT SRC_REF: 5, SOURCE: EXTEMPL1.C ,
 PROC: minimum<double>(const double &, const double &)
/%sd %nest
SRC_REF: 5  SOURCE: EXTEMPL1.C
            PROC: minimum<double>(const double &, const double &)  ********
SRC_REF: 13 SOURCE: EXTEMPL1.C
            PROC: minimum<double>(const double &, const double &,
                  const double &)
SRC_REF: 22  SOURCE: EXTEMPL1.C  PROC: main  ******************************
ABSOLUT: V'113CF88'   SOURCE: IC@RT2OA  PROC: IC@RT2OA  ********************
ABSOLUT: V'10014A8'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  ********************
/%da 3 from t'minimum<double>(const double &, const double &)'
          EXTEMPL1$O&@
EXTEMPL*+5C2      L     R14,0(R0,R9)           58 E0 9000
EXTEMPL*+5C6      LD    R0,0(R0,R14)           68 00 E000
EXTEMPL*+5CA      L     R15,4(R0,R9)           58 F0 9004
```

Das %INSERT-Kommando setzt einen Testpunkt auf die erste ausführbare Anweisung der Instanz `minimum<double>(const double &, const double &)` des Funktionstemplates `minimum`. Mit %RESUME wird der Testpunkt erreicht. Die Aufrufhierarchie an der Unterbrechungsstelle, die Sie sich mit `%SD %NEST` ausgeben lassen können, zeigt, dass die aktuelle Instanz von `minimum` von einer weiteren Instanz desselben Templates, nämlich von `minimum<double>(const double &, const double &, const double &)` aufgerufen wurde. Zum Schluss werden mit dem %DISASSEMBLE die ersten 3 Befehle der Instanz `minimum<double>(const double &, const double &)` rückübersetzt.

5.5.4 Auflisten von Template-Instanzen

Sind an der Unterbrechungsstelle mehrere Instanzen eines Klassen- oder Funktionstemplates sichtbar, so erhalten Sie mit dem folgenden %DISPLAY-Kommando eine Übersicht aller Instanzen:

```
%DISPLAY  [qua]  template
```

qua Qualifikation
Mit *qua* geben Sie den Programmbereich an, der die Template-Deklaration enthält.

template
Name des Klassen- oder Funktionstemplates
Die Template-Argumente müssen weggelassen werden, damit AID die Übersicht ausgibt. Daher brauchen Sie den Template-Namen auch nicht in `t'...'` setzen.

Gibt es jedoch nur eine einzige Template-Instanz, so müssen Sie das folgende %DISPLAY-Kommando verwenden, um sich die Instanz auflisten zu lassen:

```
                 {t'k_template'            }
%DISPLAY  [qua]  {                         }
                 {t'f_template([signatur])'}
```

qua Qualifikation
Mit *qua* geben Sie wie oben den Programmbereich an, der die Template-Deklaration enthält.

k_template
Name des Klassentemplates
Die Template-Argumente können wiederum weggelassen werden, jedoch muss der Template-Name in `t'...' ge`setzt werden.

f_template([signatur])
Name des Funktionstemplates
Die Template-Argumente werden weggelassen; der Template-Name wird in `t'...' ge`setzt. Ist die Signatur ungleich `void`, so muss sie angegeben werden. Ist die Signatur `void`, so schreiben Sie nur die beiden Klammern ().

signatur
: Übergabeparameter der Funktion
Wenn es sich bei dem Template um ein Funktionstemplate handelt, müssen Sie die Signatur angeben. Ist die Signatur jedoch `void`, so wird sie weggelassen

Beispiel

Das Beispiel bezieht sich auf das C++-Programm, das auf Seite 108 abgebildet ist.

```
/%d minimum
01          minimum
 02         minimum<double>(const double &, const double &) = 01000528
 02         minimum<int>(const int &, const int &) = 010004A0
 02         minimum<double>(const double &, const double &,
            const double &) = 01000330
 02         minimum<int>(const int &, const int &, const int &) = 010001D8
```

Das %DISPLAY-Kommando listet alle zum Funktionstemplate `minimum` erzeugten Instanzen auf und gibt die zugehörigen Prologadressen aus.

5.5.5 Ausgaben von Template-Instanznamen

In AID-Ausgaben wie z.B. in der STOP-Meldung oder in der Ausgabe zu `%SDUMP %NEST`, gibt AID den vollständigen Namen der Template-Instanz aus. Für die Template-Argumente verwendet AID die C++-Bezeichnungen für die entsprechenden Datentypen.

Gibt es für einen abgeleiteten Datentyp eine Benutzerdefinition, so setzt AID diese ein.

Beispiele

1. Die Instanz eines Funktionstemplates sei wegen des Funktionsaufrufs `foo(15)` angelegt worden. In einer STOP-Meldung gibt AID als Funktionsnamen `foo<int>` aus. Eine weitere Instanz sei aufgrund des Aufrufs `foo(&i)` erzeugt worden. Der AID-Funktionsname dazu ist `foo<*int>`.

2. ```
   typedef int (*int_arr_def)[3]  int_arr_def;
   int_arr_def int_arr_ptr;
   ...
   foo(int_arr_ptr);
   ```

   In diesem Fall verwendet AID den Funktionsnamen `foo<int_arr_def>` und nicht `foo<int(*)[3]>`.
   ```

5.5.6 Ansprechen von Source-Referenzen aus Template-Instanzen

Durch jede Instanziierung entsteht aus einem Funktionstemplate eine konkrete Funktion, aus einem Klassentemplate eine konkrete Klasse. Ist in dem Klassentemplate eine Funktion definiert, so entsteht durch die Instanziierung nicht nur jedes Mal eine neue Klasse, sondern stets auch eine neue Funktion. Die Source-Referenzen zu den Anweisungen solcher Funktionen werden aus den Zeilennummern der Template-Deklaration gebildet. Sie sind daher für jede Instanz gleich und in der Übersetzungseinheit nicht mehr eindeutig, falls mehr als eine Instanz angelegt wurde. Daraus ergibt sich, dass Sie Source-Referenzen aus Template-Deklarationen, aus denen mehrere Instanzen hervorgegangen sind, mit der Funktion der gewünschten Instanz qualifizieren müssen, damit AID sie richtig zuordnen kann. Andernfalls gibt AID eine entsprechende Meldung aus

Syntax für Source-Referenzen aus Template-Instanzen:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

[E-qua•][S-qua•]PROC=[namespace::[...]][klasse::[...]]funktion•S'[f-]n[:a]'

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

E-qua Basisqualifikation
: Sie wird mit `E=VM` oder `E=D`*n* angegeben und legt fest, ob der AID-Arbeitsbereich im geladenen Programm (`E=VM`) oder in einer Dump-Datei (`E=D`*n*) liegen soll. Die Basisqualifikation wird beim symbolischen Testen und beim maschinennahen Testen gleich verwendet und ist im AID-Basishandbuch, Kapitel „Adressierung in AID“ und im Kommando %BASE auf Seite 132 beschrieben.

S-qua S-Qualifikation
: Damit bezeichnen Sie die Übersetzungseinheit, die die Template-Deklaration enthält. Die S-Qualifkation geben Sie an, wie es auf Seite 23 beschrieben ist.

namespace
: Name des Namespaces, der die Template-Deklaration enthält.

klasse Name der Klasse
: Bezeichnet *klasse* die Instanz eines Klassentemplates, so müssen Sie die Schreibweise `t'`*k_template*`<`*arg*`[,...]>'` verwenden. Gibt es zu dem Klassentemplate nur eine einzige Instanz, so reicht die Angabe: `t'`*k_template*`'`.
Zwischen Klassennamen und anschließendem Funktionsnamen schreiben Sie wie gewohnt zwei Doppelpunkte.

funktion
: Name der Funktion
funktion geben Sie mit `n'`*funktion*`([`*signatur*`])'` an, falls die Source-Referenz in einer gewöhnlichen C++-Funktion liegt, die in einem Klassentemplate definiert ist

oder mit `t'`*f_template*`<`*arg*`[,...]>([`*signatur*`])'`, falls es sich um eine Instanz eines Funktionstemplates handelt; gibt es zu dem Funktionstemplate nur eine einzige Instanz, so reicht die Angabe : `t'`*f_template*`([`*signatur*`])'`.
Zwischen dem Funktionsnamen und der Source-Referenz schreiben Sie einen Punkt.
Ausführliche Informationen, wie Sie einen Funktionsnamen beim Testen von C++-Programmen angeben, finden Sie auf Seite 60.

S'[f-]n[:a]'
: ist eine Source-Referenz und bezeichnet eine ausführbare Anweisung in einer Funktion, die in einem Klassentemplate definiert ist oder die in der Dekaration eines Funktionstemplates enthalten ist.

Die Verwendung einer unqualifizierten Source-Referenz aus einer Template-Instanz in einem AID-Kommando lehnt AID mit der folgenden Fehlermeldung ab:
`AID0376 Ambiguous qualification for SRC_REF`

Beispiel

Das Beispiel bezieht sich auf das C++-Programm, das auf Seite 108 abgebildet ist.

```
/%in proc=t'minimum<int>(const int &, const int &)'.s'6'
/%in proc=t'minimum<double>(const double &, const double &)'.s'6'
```

In den beiden Instanzen `minimum<int>(const int &, const int &)` und `minimum<double>(const double &, const double &)` wird jeweils auf die Source-Referenz S'6' ein Testpunkt gesetzt.

5.6 Überladene Funktionen

Überladene Funktionen können eindeutig über ihre Signatur, die ja in die Funktionsbezeichnung mit eingeht (siehe Abschnitt „Qualifikationen“ auf Seite 59), angesprochen werden. Um sich einen Überblick über im Programm definierte überladene Funktionen zu verschaffen, können Sie mit %DISPLAY eine Übersicht aller überladenen Funktionen anfordern, die an der Unterbrechungsstelle oder im angegebenen Programmbereich denselben Namen haben:

```
%DISPLAY  [qua•]  funktion
```

qua bezeichnet die Qualifikation des Programmbereichs, in dem die überladenen Funktionen gesucht werden sollen.

funktion
ist der Name einer überladenen Funktion. Damit AID die Übersicht ausgibt, müssen Sie die Signatur weglassen. Ansonsten geben Sie den Funktionsnamen an, wie im Abschnitt „Qualifikationen“ auf Seite 59 beschrieben, also ggfs. mit vorangestellter Namespace- und/oder Klassen-Qualifikation.

Aufbau der Tabelle der überladenen Funktionen:

```
01 funktion
 02 funktion(signatur1)   = adresse1
 02 funktion(signatur2)   = adresse2
 ...
```

Der Tabelle können Sie den Namen der gewünschten überladenen Funktion in C++-Standard-Schreibweise entnehmen. Zu jeder im Gültigkeitsbereich vorhandenen überladenen Funktion wird die Signatur und die zugehörige Prologadresse aufgeführt.

Beispiel

Das Beispiel bezieht sich auf den Programmausschnitt des Beispiels auf Seite 63. Zur besseren Lesbarkeit sind die Benutzerkommandos fettgedruckt.

```
/%d FOO
SRC_REF: 70 SOURCE: BSP.C   PROC: main  ***********************************
01          FOO
 02         FOO(long)      = 01000070
 02         FOO(int)       = 01000000
/%in n'FOO(int)'
/%in n'FOO(long)'
```

Das %DISPLAY-Kommando gibt die beiden Prologadressen der überladenen Funktionen mit dem Namen `FOO` aus. Die anschließenden %INSERT-Kommandos setzen auf die jeweils erste ausführbare Anweisung von `FOO(int)` und `FOO(long)` einen Testpunkt.

5.7 Überladene Operatoren

Überladene Operatoren können Sie mit AID genauso testen wie jede andere von Ihnen geschriebene Funktion. Um AID den gewünschten überladenen Operator zu bezeichnen, verwenden Sie den Funktionsnamen aus Ihrem C++-Programm in C++-Standard- Schreibweise, also mit Signatur und evtl. vorangestellter Klassen-Qualifikation. Das Ganze müssen Sie dann noch wie gewohnt in `n'...'` setzen. Innerhalb von `n'...'` dürfen keine zusätzlichen Leerzeichen geschrieben werden.

Ist ein Operator mehrfach überladen, so verfahren Sie wie im Abschnitt „Überladene Funktionen“ auf Seite 114 beschrieben, um eine eindeutige Zuordnung zur gewünschten Funktion herzustellen.

Beim Testen eines Programms, das mit überladenen Operatoren arbeitet, müssen Sie jedoch beachten, dass AID stets mit den Standard-Operatoren arbeitet, also das Überladen für seine eigenen Berechnungen nicht nachvollzieht.

Beispiel

```
%insert X1::n'operator+(float&)' <%d *this>
```

In der Klasse `X1` ist eine Funktion definiert, mit der der Operator + überladen wird. Mit dem %INSERT setzen Sie einen Testpunkt auf die erste ausführbare Anweisung von `X1::operator+(float&)`. Das Subkommando veranlasst, dass nach jedem Aufruf der Funktion der Inhalt des zugehörigen Objekts ausgegeben wird.

5.8 Referenzvariablen

Eine Referenzvariable hat bis auf den Namen dieselben Attribute wie die dadurch referenzierte Variable. Sie stimmen also in Adresse, Länge, Inhalt, Speichertyp und Ausgabetyp überein. Beim Ansprechen von Referenzvariablen mit AID gibt es keine Besonderheiten.

Beispiel

```
C++-Programm                              SOURCE: BSP1.C
================================================================================
SRC
LIN
  1  int    i;
  2  int&   p = i;
  3
  4  int main()
  5  {
  6     i = 17;
  7     ...
```

```
/%d i, p, %@(i), %@(p), %l(i), %l(p)
SRC_REF: 7   SOURCE: BSP1.C   PROC: main  **********************************
i               =          17
p               =          17
010547C8
010547C8
          +4
          +4
```

Die Variablen `i` und `p` unterscheiden sich nur in ihrem Namen. Wert, Adresse und Länge stimmen überein.

6 AID-Kommandos

%AID

Mit %AID können Sie globale Einstellungen vereinbaren oder bis zu diesem Kommando geltende Einstellungen wieder zurücknehmen.

– Mit *C* legen Sie fest, wie AID C-String-Literale und C-String-Vektoren der Sprache C/C++ behandeln soll.

– Mit *CCS* treffen Sie eine Voreinstellung, in welchem Zeichensatz Zeichen interpretiert werden, falls keine explizite Angabe im %DISPLAY-Kommando gemacht wird. Unicode-Zeichensätze sind nicht erlaubt.

– Mit *CHECK* legen Sie fest, ob vor der Ausführung von %MOVE oder %SET ein Änderungsdialog durchgeführt werden soll.

– Mit *DELIM* legen Sie die Begrenzer (Delimiter) für die AID-Ausgabe alphanumerischer Daten fest. Der senkrechte Strich ist der Standard-Begrenzer.

– Mit *EXEC* legen Sie fest, ob nach dem Laden durch `exec()` der Testmodus aktiviert wird.

– Mit *FORK* legen Sie fest, ob eine durch `fork()` erzeugte Task unmittelbar nach ihrer Entstehung unterbrochen und in den Testmodus versetzt wird.

– Mit *LANG* legen Sie fest, ob AID die Informationen des %HELP-Kommandos auf Deutsch oder Englisch ausgeben soll.

– Mit *LEV* legen Sie fest, ob beim AID-Kommando %SDUMP %NEST die Aufruftiefe ausgegeben werden soll.

– Mit *LOW* legen Sie fest, ob AID Kleinbuchstaben aus Character-Literalen und Namen in Großbuchstaben konvertieren soll oder nicht.

– Mit *OV* legen Sie fest, ob AID die Überlagerungsstruktur eines Programms (Overlay) berücksichtigen soll.

– Mit *REP* legen Sie fest, ob die Speicheränderungen eines %MOVE-Kommandos als REP abgelegt werden sollen.

– Mit *SYMCHARS* legen Sie fest, ob AID den Bindestrich „-“ in Programm-, Daten- und Anweisungsnamen als Bindestrich oder als Minuszeichen interpretieren soll.

Kommando	Operand
%AID	C = {YES| NO} CCS = {<coded-character-set> | *USRDEF} CHECK [= {ALL| NO}] DELIM [= {C'x'|'x'C|'x' | '|'}] EXEC [= {OFF| ON}] FORK [= {OFF| NEXT| ALL}] LANG [= {D | E}] LEV [= {ON|OFF}] LOW [= {ON| OFF| ALL}] OV [= {YES| NO}] REP [= {YES| NO}] SYMCHARS [= {STD| NOSTD}]

Für die Gültigkeitsdauer der mit %AID getroffenen Vereinbarungen gilt Folgendes:

– In der LOGON-Task gelten die Einstellungen mit %AID solange, bis sie durch ein neues %AID-Kommando geändert werden oder bis /LOGOFF.

– In der Fork-Task sind alle Einstellungen, die mit %AID in der Vater-Task festgelegt wurden, zurückgesetzt. Die einzige Ausnahme bildet `FORK=ALL`.

– Ein `exec()`–Aufruf beeinflusst mit %AID getroffene Vereinbarungen nicht.

– In der POSIX-Shell sind nach dem Laden mit `debug` *progname* (siehe Seite 303) alle mit %AID getroffenen Vereinbarungen außer `FORK=ALL` zurückgesetzt. War `FORK=ALL` in der LOGON-Task gesetzt, so gilt dies weiter. Nach jedem Laden mit `debug` *progname* ist `EXEC=ON` eingeschaltet.

%AID darf nur als Einzelkommando eingegeben werden, es darf nicht in einer Kommandofolge oder einem Subkommando stehen.

%AID verändert den Programmzustand nicht.

C

YES AID unterstützt die Stringnotation von C/C++. C-String-Literale in der Form "*zeichenfolge*" werden in den Kommandos %DISPLAY und %SET sowie in Vergleichen in Subkommandos von AID akzeptiert und wie von C/C++ her gewohnt verarbeitet. Einen Kommentar können Sie dann nur noch in /*...*/ eingeschlossen angeben. Außerdem fasst AID die Elemente eines `char`-Vektors, die über den hintersten Index adressiert werden, zu C-Strings zusammen, wobei das erste Vektorelement dieser Indexstufe mit dem Wert X'00' das Ende des C-Strings kennzeichnet. Ausführliche Informationen zu diesem Thema erhalten Sie im Abschnitt „C-Strings“ auf Seite 37.

Laufen AID-Kommandos in einer Prozedur ab, so müssen Sie beachten, dass auch bei eingeschaltetem `C=YES` innerhalb von C-String-Literalen keine Parameter ersetzt werden, da der BS2000-Kommandointerpreter C-String-Literale stets als Kommentare betrachtet.

Mit `%AID C=YES` werden gleichzeitig auch `%AID LOW=ALL` (siehe Seite 123) und `%AID SYMCHARS=NOSTD` (siehe Seite 125) eingeschaltet.

NO Die Interpretation von "*zeichenfolge*" als C-String-Literal wird abgeschaltet. In "..." eingeschlossene Zeichen werden als Kommentar angesehen. `char`-Vektoren behandelt AID als Vektoren einzelner Character-Zeichen.

War zuvor `%AID C=YES` vereinbart, bleiben die dadurch implizit gesetzten Einstellungen `%AID LOW=ALL` und `%AID SYMCHARS=NOSTD` erhalten und müssen bei Bedarf gesondert zurückgesetzt werden.

Wenn in einer Testsitzung noch kein *C*-Operand eingegeben wurde, gilt die Voreinstellung NO.

CCS

<coded-character-set>
Name des Zeichensatzes (`<name 1..8>`), in dem AID Daten interpretiert werden. Der angegebene Zeichensatz muss XHCS bekannt sein, andernfalls weist AID die Anweisung mit der Meldung `AID0555` ab.

*USRDEF
CCSNAME des Zeichensatzes, der der Benutzer-Id zugewiesen ist. *USRDEF ist der Standardwert von *CCS*.

Wenn Sie den *CCS*-Operand in einem %AID-Kommando eingeben, prüft AID mit Hilfe von XHCS, ob der CCSNAME zulässig ist. Wenn XHCS den CCSNAME nicht kennt, wird das Kommando zurückgewiesen und der aktuelle *CCS*-Wert wird beibehalten.

Mit folgendem AID-Kommando können Sie eine vollständige Liste der CCSNAMEs, die XHCS unterstützt, anzeigen:

```
%SHOW %CCSN
```

CHECK

ALL Vor der Ausführung eines %MOVE oder %SET führt AID folgenden Änderungsdialog:

```
OLD CONTENT:
AAAAAAAA
NEW CONTENT:
BBBBBBBB
%  AID0274 Change desired? Reply (Y=Yes; N=No)?n
%  AID0342 Nothing changed
```

Nach der Eingabe **y** wird der alte Speicherinhalt ohne weitere Meldung überschrieben. In Prozeduren im Stapelbetrieb kann AID keinen Dialog führen und nimmt immer **y** an.
Der alte bzw. neue Inhalt wird auf SYSOUT ausgegeben. Wird SYSOUT umgewiesen, dann sind diese Ausgaben am Terminal nicht zu sehen. Das gilt auch, wenn das %MOVE- bzw. %SET-Kommando mit dem CMD-Makro gegeben wurde und eine Ausgabe auf SYSOUT vereinbart ist. Dagegen geht die Meldung AID0274 und gegebenenfalls AID0342 immer auf das Medium Terminal.

NO %MOVE und %SET werden ohne Änderungsdialog ausgeführt.

Wird der *CHECK*-Operand ohne Wertangabe eingegeben, setzt AID den Standardwert (NO) ein.

DELIM

C'x'|'x'C|'x'
: Mit diesem Operanden legen Sie ein Zeichen als linke und rechte Begrenzung (Delimiter) für die AID-Ausgabe von symbolischen Daten vom Typ Character (Kommandos %DISPLAY und %SDUMP bzw. beim Änderungsdialog im Kommando %SET) fest.

| : Der Standard-Begrenzer ist der senkrechte Strich.

Wird der *DELIM*-Operand ohne Wertangabe eingegeben, setzt AID den Standardwert (|) ein.

EXEC

OFF : Programme, die mit einem `exec()`-Aufruf geladen werden, werden nach dem Laden nicht unterbrochen und gehen nicht in den Testmodus.

ON : Unmittelbar nach dem Laden mit `exec()` wird das Programm unterbrochen und in den Testmodus versetzt. Alle mit %AID bisher vereinbarten Einstellungen bleiben erhalten.

`%AID EXEC` ohne Wertangabe ist gleichbedeutend mit `%AID EXEC=OFF` (Standardwert).

FORK

OFF : Fork-Tasks werden nach ihrer Erzeugung nicht unterbrochen und gehen nicht in den Testmodus. Dies ist der Standardwert. Wurde in einer Task die Einstellung `%AID FORK` noch nicht gesetzt, so zeigt `%SHOW %AID` für FORK die Angabe `NOT_USED` an.

NEXT : Alle Fork-Tasks der ersten Generation werden unmittelbar, nachdem sie erzeugt werden, unterbrochen und in den Testmodus versetzt. In diesen Fork-Tasks ist jedoch `FORK=OFF` gesetzt, d.h. durch `fork()` erzeugte Tasks zweiter und höherer Generationen können Sie ohne weitere Maßnahmen nicht testen. In diesem Fall

können Sie eine solche Fork-Task höherer Generation nur in den Testmodus versetzen, indem Sie von der POSIX-Shell aus mit `debug -p` *pid* (siehe Seite 303) die Fork-Task unterbrechen bzw. von einer anderen Task derselben Task-Familie aus ein %STOP-Kommando mit Angabe der entsprechenden *TSN* oder *pid* (siehe Seite 287) an die gewünschte Fork-Task schicken.

ALL Alle Fork-Tasks, die in beliebiger Generation aus der aktuellen Task hervorgehen, werden nach ihrer Erzeugung unterbrochen und in den Testmodus versetzt. In den Fork-Tasks ist FORK=ALL gesetzt. Diese Einstellung ist die einzige Vereinbarung mit %AID, die vererbt wird.

Eine Änderung dieses Schalters wirkt sich nur auf Tasks aus, die nach der Änderung in direkter Linie aus der Task erzeugt werden, in der der Schalter gesetzt wurde.

`%AID FORK` ohne Wertangabe ist gleichbedeutend mit `%AID FORK=OFF` (Standardwert).

LANG

D AID gibt die Informationen, die mit %HELP angefordert wurden, in Deutsch aus.

E AID gibt die Informationen, die mit %HELP angefordert wurden, in Englisch aus.

Wird der *LANG*-Operand ohne Wertangabe eingegeben, setzt AID den Standardwert (D) ein.
Mit dem SDF-Kommando `MODIFY-MSG-ATTRIBUTES TASK-LANGUAGE=D` bekommen Sie auch die AID-Meldungen auf Deutsch. Der Änderungsdialog (siehe CHECK-Operand) wird dadurch nicht beeinflusst.

LEV

ON Ausgabe der Aufruftiefe einschalten.

Wird die Ausgabe der Aufruftiefe eingeschaltet, dann gibt `%SDUMP %NEST` für jede Prozedur in der Aufrufhierarchie (Funktion oder Block in C/C++) zusätzlich zwei Werte aus:

– Die einfache Aufruftiefe (Counter) mit einer rückläufigen Nummerierung, d.h. von der aktuellen Prozedur zur Haupt-Prozedur. Diese Aufruftiefe kann in der *NESTLEV*-Qualifikation verwendet werden.

– Die Rekursionstiefe (RLEV) oder einen individuellen Zähler für jede Prozedur mit einer rückläufigen Nummerierung, beginnend bei 0. Die Rekursionstiefe dient nur zur Information.

OFF Ausgabe der Aufruftiefe ausschalten.

LOW

ON Kleinbuchstaben in Character-Literalen und in Programm-, Daten- und Anweisungsnamen werden nicht in Großbuchstaben konvertiert. Beim Testen von C/C++-Programmen sollten Sie `%AID LOW` zu Beginn jeder Testsitzung eingeben. Erst dann kann AID die Unterscheidung von Groß- und Kleinschreibung in C/C++ nachvollziehen. Nur bei der S-Qualifikation unterscheidet AID nicht zwischen Groß- und Kleinschreibung. Angaben in der S-Qualifikation werden immer in Großbuchstaben umgesetzt.

OFF Alle Kleinbuchstaben aus Benutzereingaben werden in Großbuchstaben umgesetzt.

ALL Zusätzlich zu allen Eingaben, auf die sich die Einstellung `LOW=ON` auswirkt, wird bei `LOW=ALL` auch die Klein-/Großschreibung bei der Eingabe von allen BLS-Namen berücksichtigt. Diese Einstellung benötigen Sie immer dann, wenn Sie ein Programm testen, das in der POSIX-Shell übersetzt wurde und dessen zugehörige Quelldatei Kleinbuchstaben im Namen enthält.

Außerdem wird, wie bei der Angabe von %AID LOW=ON, die Klein-/Großschreibung in Character-Literalen und in Programm-, Daten- und Anweisungsnamen beibehalten.

BLS-Namen, die von AID verwendet werden, sind:
– Kontextnamen der CTX-Qualifikation
– Namen von Ladeeinheiten der L-Qualifikation
– Bindemodulnamen der O-Qualifikation
– CSECT-Namen der C-Qualifikation
– COMMON-Namen der COM-Qualifikation
– Namen von Übersetzungseinheiten der S-Qualifikation

Die Angabe von `%AID C=YES` impliziert das Setzen von `LOW=ALL`. Jedoch wird dies von `%AID C=NO` nicht zurückgenommen; `LOW=ALL` muss gesondert zurückgesetzt werden, falls dies gewünscht wird.

Beim Operanden *LOW* stimmen Voreinstellung und Standardwert nicht überein. Wenn in einer Testsitzung noch kein *LOW*-Operand eingegeben wurde, gilt die Voreinstellung OFF. Wird der *LOW*-Operand jedoch ohne Wertangabe eingegeben, gilt der Standardwert ON. Um wieder die Umsetzung in Großbuchstaben einzuschalten, müssen Sie das vollständige Kommando `%AID LOW=OFF` eingeben.

OV

YES müssen Sie angeben, wenn Sie ein Programm mit Überlagerungsstruktur (Overlay) testen. Dies gilt auch für Programme, die Programmteile dynamisch ent- und nachladen (`BIND` / `UNBIND`). AID überprüft dann jedesmal, ob der angesprochene Programmteil eventuell aus einem nachgeladenen Segment stammt.

NO AID geht davon aus, dass das zu testende Programm ohne Überlagerungsstruktur gebunden ist. AID benutzt die einmal geladenen LSD-Sätze, ohne zu prüfen, ob der angesprochene Programmteil in einem nachgeladenen Segment liegt.

Wird der *OV*-Operand ohne Wertangabe eingegeben, setzt AID den Standardwert (NO) ein.

REP

YES Zu Änderungen im Speicher mit %MOVE werden LMS-Korrekturanweisungen im SDF-Format (REPs) erstellt. Wenn die Objekt-Strukturliste nicht zur Verfügung steht, erstellt AID keine Korrektursätze und gibt eine Fehlermeldung aus.

AID hinterlegt die Korrekturen in einer Datei mit dem Linknamen F6. Für den LMS-Lauf muss dann noch die `MODIFY-ELEMENT`-Anweisung eingefügt werden. Achten Sie darauf, dass Sie in die Datei mit dem Linknamen F6 keine anderen Ausgaben schreiben lassen. Ist keine Datei mit dem Linknamen F6 angemeldet (siehe Beschreibung des Kommandos %OUTFILE), so wird der REP in der von AID angelegten Datei `AID.OUTFILE.F6` abgelegt.
Benutzerspezifische REP-Dateien müssen mit Zugriffsmethode SAM angelegt sein. Von AID angelegte REP-Dateien werden ebenfalls mit Zugriffsmethode SAM, Satzformat V und Eröffnungsmodus EXTEND angelegt.
Die Datei bleibt geöffnet, bis sie mit %OUTFILE geschlossen wird oder bis /LOGOFF bzw. /EXIT-JOB.

NO Es werden keine REPs erstellt.

Wird der *REP*-Operand ohne Wertangabe eingegeben, setzt AID den Standardwert (NO) ein. Der *REP*-Operand des %MOVE-Kommandos kann die mit %AID getroffene Vereinbarung für dieses eine %MOVE-Kommando ersetzen. Für nachfolgende %MOVE-Kommandos ohne *REP*-Operand gilt dann wieder die Vereinbarung mit %AID.

SYMCHARS

STD Der Bindestrich (-) wird als alphanumerisches Zeichen interpretiert und kann somit in Programm-, Daten- und Anweisungsnamen verwendet werden. Er wird nur dann als Minuszeichen interpretiert, wenn vor dem Bindestrich ein Leerzeichen steht.

NOSTD

Der Bindestrich (-) wird immer als Minuszeichen interpretiert und kann in Namen nicht verwendet werden. Da Namen in C und C++ keinen Bindestrich enthalten dürfen, sollten Sie zu Beginn jeder Testsitzung NOSTD einstellen. Sie brauchen dann nicht darauf zu achten, wann Sie vor dem Bindestrich ein Leerzeichen eingeben müssen.

Die Angabe von `%AID C=YES` impliziert das Setzen von `SYMCHARS=NOSTD`. Jedoch wird dies von `%AID C=NO` nicht zurückgenommen; `SYMCHARS=NOSTD` muss gesondert zurückgesetzt werden, falls dies gewünscht wird.

Wird der *SYMCHARS*-Operand ohne Wertangabe eingegeben, setzt AID den Standardwert (STD) ein.

%AINT

Mit %AINT legen Sie fest, ob AID bei indirekter Adressierung mit 24-, 31- oder 32-Bit-Adressen arbeiten soll. Die Adresse vor dem Pointer-Operator (->) besteht für AID dann entsprechend aus 24, 31 oder 32 Bits.
Der Adressierungsmodus des Testobjekts wird damit nicht beeinflusst.

– Mit *aid-mode* legen Sie die Adressinterpretation für indirekte Adressierung innerhalb eines AID-Arbeitsbereiches fest.

Kommando	Operand
%AINT	[aid-mode] [,...]

Standardmäßig interpretiert AID indirekte Adressangaben entsprechend dem aktuellen Adressierungsmodus des Testobjekts. Mit einem %AINT mit dem Schlüsselwort %MODE*n* schalten Sie diese automatische Anpassung aus. Der implizite Adressierungsmodus ist auf /390-Anlagen 24 oder 31 und kann mit `%DISPLAY %AMODE` abgefragt werden. Mit %MOVE können Sie ihn verändern (siehe Handbuch „Testen auf Maschinencode-Ebene“ [2]). Mit `%SHOW %AID` oder `%SHOW %BASE` erhalten Sie neben anderen Informationen auch den für den aktuellen AID-Arbeitsbereich geltenden Adressierungsmodus.

Ohne die Angabe einer Qualifikation gilt %AINT für AID-Kommandos, die indirekt Adressen des aktuellen AID-Arbeitsbereichs ansprechen oder benutzen.
Bei angegebener Qualifikation gilt %AINT nur für AID-Kommandos, die indirekt Adressen des qualifizierten Bereichs ansprechen oder benutzen.

Für die Gültigkeitsdauer des mit %AINT festgelegten Adressierungsmodus gilt Folgendes:

– In der LOGON-Task gilt der Adressierungsmodus solange, bis mit einem %AINT ohne Operanden oder mit einem %AINT mit Basisqualifikation und ohne %MODE*n* auf die Standard-Adressinterpretation zurückgeschaltet wurde. Sonst gilt der vereinbarte Adressierungsmodus bis /LOGOFF bzw. /EXIT-JOB.

– In einer durch `fork()` erzeugten Task sowie in einem durch `exec()` geladenen Programm wird der Adressierungsmodus auf die Standard-Adressinterpretation zurückgesetzt.

– In der POSIX-Shell ist nach dem Laden mit `debug` *progname* (siehe Seite 303) stets die Standard-Adressinterpretation eingeschaltet.

%AINT verändert den Programmzustand nicht.

aid-mode

legt für den aktuellen oder den mit der angegebenen Basisqualifikation bezeichneten AID-Arbeitsbereich fest, wie indirekte Adressen in später folgenden AID-Kommandos interpretiert werden sollen.

Geben Sie ein Schlüsselwort für die Adressinterpretation und keine Qualifikation an, so gilt das %AINT-Kommando für die Bearbeitung des aktuellen AID-Arbeitsbereichs.

Geben Sie eine Basisqualifikation und kein Schlüsselwort für die Adressinterpretation an, gilt für den entsprechenden AID-Arbeitsbereich die AID-Standard-Adressinterpretation.

```
aid-mode-OPERAND  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

       ⎧VM⎫       ⎧%M[ODE]32⎫
[•][E=⎨  ⎬[•]] [⎨%M[ODE]31⎬]
       ⎩Dn⎭       ⎩%M[ODE]24⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

- Steht der Punkt an führender Stelle, ist er das Kennzeichen für eine *vorqualifikation*. Sie muss mit einem vorhergehenden %QUALIFY-Kommando definiert worden sein.
  Zwischen Basisqualifikation und dem Schlüsselwort zur Adressinterpretation muss ein Punkt gesetzt werden. Geben Sie nur eine Basisqualifikation an, so dürfen Sie keinen abschließenden Punkt eingeben.

E={VM | Dn}
: geben Sie an, wenn die Umschaltung der Adressinterpretation nicht für den aktuellen AID-Arbeitsbereich gelten soll. Geben Sie nur eine Basisqualifikation an, so gilt für den damit angesprochenen Bereich wieder die Standard-Adressinterpretation.

{%M[ODE]32 | %M[ODE]31 | %M[ODE]24}
: Schlüsselwort, das angibt, wie viele Bits bei indirekter Adressierung in AID-Kommandos berücksichtigt werden sollen.

  ```
  %M[ODE]32   32-Bit-Adressierung
  %M[ODE]31   31-Bit-Adressierung
  %M[ODE]24   24-Bit-Adressierung
  ```

Beispiele

Die Adresse V’100’ hat den Inhalt: 1200000C
Register 5 hat den Inhalt: 010001A0

1. ```
   %AINT %MODE24
   %DISPLAY V'100'->
   %MOVE %5-> INTO %5G
   ```

   Mit %AINT stellen Sie auf 24-Bit-Adressinterpretation um. Die Umstellung gilt für den aktuellen AID-Arbeitsbereich.
   Der %DISPLAY gibt 4 Bytes ab Adresse V’00000C’ aus.
   Der %MOVE überträgt 4 Bytes ab Adresse V’0001A0’ in das AID-Register 5.

2. ```
   %AINT %MODE31
   %DISPLAY V'100'->
   %MOVE %5-> INTO %5G
   ```

   Sie stellen die Adressinterpretation für den aktuellen AID-Arbeitsbereich auf 31-Bit-Interpretation um.
   Der %DISPLAY gibt 4 Bytes ab Adresse V’1200000C’ aus.
   Der %MOVE überträgt 4 Bytes ab Adresse V’010001A0’ in das AID-Register 5.

%ALIAS

Mit %ALIAS können Sie für lange Variablennamen bzw. lange Klassen- oder Namespace-Qualifikationen kurze Namen, sog. Aliasnamen, vereinbaren, um sich in nachfolgenden Kommandos Schreibarbeit zu sparen.

– Mit *aliasname* legen Sie einen verkürzten Namen fest, den Sie in nachfolgenden Kommandos statt des Originalnamens verwenden möchten.

– Mit *originalname* geben Sie den Namen eines Datums, einer Klasse, eines Objekts einer Klasse, eines Namespaces, einer Instanz eines Templates oder einer Funktion an. Vor den Namen können Sie eine Klassen- oder Namespace-Qualifikation oder die beiden Doppelpunkte (`::`) für den globalen Namespace schreiben.

Kommando	Operand
%ALIAS	aliasname [= originalname]

Ein mit %ALIAS vereinbarter *aliasname* gilt, bis er durch ein %ALIAS-Kommando ohne *originalname*-Operanden gelöscht wird, oder bis /LOGOFF bzw. /EXIT-JOB.
In der POSIX-Shell sind alle Aliasnamen nach einem `fork()`-Aufruf, also auch nach `debug` *progname* (siehe Kapitel „POSIX-Kommando debug“ auf Seite 303), gelöscht. In einem Programm, das durch `exec()` geladen wurde, gelten mit %ALIAS vereinbarte Namen weiter.

Die Vereinbarungen des %ALIAS werden nur von nachfolgend eingegebenen Kommandos übernommen. Auf die Subkommandos in %CONTROL*n*, %INSERT und %ON, die vorher eingegeben wurden, hat ein neuer %ALIAS keine Auswirkungen, auch wenn die Subkommandos erst danach ausgeführt werden.

Wenn Sie einen Aliasnamen eingeben, müssen Sie darauf achten, dass die Behandlung der Groß-/Kleinschreibung (`%AID LOW={ON|OFF|ALL}`) richtig eingestellt ist.
Einem Originalnamen können mehrere Aliasnamen zugewiesen werden.
In Meldungen gibt AID stets den Originalnamen aus.
Je nach Länge der zugewiesenen Originalnamen können Sie bis zu 40 Aliasnamen vergeben.

%ALIAS darf nur als Einzelkommando eingegeben werden, es darf nicht in einer Kommandofolge oder einem Subkommando stehen.

%ALIAS verändert den Programmzustand nicht.

aliasname

legt die Bezeichnung fest, die Sie in nachfolgenden AID-Kommandos statt *originalname* verwenden können.
aliasname kann bis zu 32 Zeichen lang sein. Es steht der folgende Zeichenvorrat zur Verfügung: a-z, A-Z, 0-9, $, #, @, Unterstrich (_) oder Bindestrich (-).
Wenn Sie mehrere Aliasnamen vergeben, so müssen diese sämtlich verschieden sein. Gleiche Aliasnamen weist AID mit folgender Meldung ab:
`AID0531 Alias name is ambiguous.`

Ein von Ihnen vergebener Aliasname sollte nicht mit dem Namen einer Definition aus Ihrem Programm übereinstimmen, da Sie die Definition anschließend nicht mehr ansprechen können. Aus dem gleichen Grund sollte der Aliasname nicht mit einem AID-Schlüsselwort identisch sein; Sie können das Schlüsselwort dann nicht mehr verwenden.

originalname

bezeichnet die Definition aus dem Quellprogramm, die im weiteren Testverlauf mit dem verkürzten Aliasnamen angesprochen werden soll.

Geben Sie keinen *originalname*-Operanden an, so wird *aliasname* aus der Liste der Aliasnamen gelöscht. Gab es gar keine Vereinbarung mit *aliasname*, so werden Sie mit der Meldung `AID0530 Alias name is undeclared` darauf hingewiesen.

```
originalname-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - -

= [::]name [::name...]

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

name ist der im Quellprogramm definierte Name eines Datums, einer Funktion, einer Instanz eines Funktionstemplates, einer Klasse, einer Instanz eines Klassentemplates, eines Objekts einer Klasse oder eines Namespaces.

Einen Datennamen können Sie angeben, wie es im Abschnitt „Datennamen“ auf Seite 30 beschrieben steht.
Funktionen können Sie in der Schreibweise *funktion*, `n'`*funktion*`([`*signatur*`])'` oder `t'`*f_template*`<`*arg*`[,...]>([`*signatur*`])'` angeben, je nachdem, um welchen Typ von Funktion es sich handelt (siehe Seite 60).
Bei abgeleiteten oder geschachtelten Klassen oder bei geschachtelten Namespaces können Sie für *originalname* den gesamten Pfad angeben, mit dem die gewünschte Klasse bzw. der Namespace angesprochen wird. Von außen nach

innen geben Sie die Namen aller übergeordneten Klassen bzw. Namespaces, jeweils durch `::` getrennt, an.
Instanzen von Klassentemplates sind in der Form `t'`*k_template*`<`*arg*`[,...]>'` anzugeben.

Wenn Sie für ein AID-Schlüsselwort einen Aliasnamen vergeben, macht Sie AID mit einer Warnung darauf aufmerksam, akzeptiert jedoch die Zuordnung und ersetzt in folgenden Kommandos das AID-Schlüsselwort durch den Aliasnamen.

Ein Aliasname, dem bereits ein Originalname zugeordnet ist, darf nicht als Originalname angegeben werden.

Beispiel

```
/%alias FJ=Fujitsu
/%display FJ::i
Fujitsu::i    =    32
```

Dem Namespace `Fujitsu` weisen Sie mit dem %ALIAS-Kommando den Aliasnamen `FJ` zu. Im folgenden %DISPLAY wird eine Variable `i` aus diesem Namespace referenziert und zwar mit der Kurzform `FJ::i`. AID zeigt in der Ausgabe wieder den Originalnamen an.

%BASE

Mit %BASE legen Sie die Basisqualifikation fest. Alle nachfolgend eingegebenen Speicherreferenzen ohne eigene Basisqualifikation übernehmen die mit %BASE vereinbarte. Mit %BASE wird zugleich festgelegt, wo sich der AID-Arbeitsbereich befinden soll.

– Mit *basis* bezeichnen Sie den virtuellen Speicherbereich des geladenen Programms oder einen Speicherabzug in einer Dump-Datei.

Kommando	Operand
%BASE	[basis]

Beim Testen von C/C++-Programmen entspricht der AID-Arbeitsbereich dem Bereich, den die Ladeeinheit im virtuellen Speicher oder in einer Dump-Datei belegt. Geben Sie in einer Testsitzung kein %BASE oder geben Sie ein %BASE ohne Operanden ein, gilt die Basisqualifikation E=VM (Standardwert) und der AID-Arbeitsbereich entspricht dem nicht privilegierten Teil im virtuellen Speicher, der vom geladenen Programm mit allen konnektierten Subsystemen belegt ist (AID-Standard-Arbeitsbereich).

Ein %BASE gilt bis zum nächsten %BASE, bis /LOGOFF bzw. /EXIT-JOB oder bis zum Schließen der Dump-Datei (siehe Beschreibung des Kommandos %DUMPFILE), die als Basisqualifikation vereinbart war.

Unmittelbar bei der Eingabe werden alle Speicherreferenzen in einem Kommando, auch in einem Subkommando, mit der aktuellen Basisqualifikation ergänzt, d.h. ein %BASE hat keine Auswirkungen auf Subkommandos, die vorher vereinbart wurden.

%BASE darf nur als Einzelkommando eingegeben werden, es darf nicht in einer Kommandofolge oder einem Subkommando stehen.

%BASE verändert den Programmzustand nicht.

basis

legt die Basisqualifikation fest. Alle nachfolgend eingegebenen Speicherreferenzen ohne eigene Basisqualifikation übernehmen die mit %BASE vereinbarte.

basis-OPERAND

E = { VM | Dn }

E=VM

Der virtuelle Speicherbereich des geladenen Programms ist als Basisqualifikation vereinbart. VM ist der Standardwert.

E=Dn

Ein Speicherabzug in einer Dump-Datei mit dem Linknamen Dn ist als Basisqualifikation vereinbart.
n ist eine Zahl mit einem Wert $0 \leq n \leq 7$.

Bevor Sie eine Dump-Datei als Basisqualifikation vereinbaren, müssen Sie mit %DUMPFILE die entsprechende Dump-Datei einem Linknamen zuweisen und öffnen.

%CONTINUE

Mit %CONTINUE starten Sie das geladene Programm oder setzen es an der Unterbrechungsstelle fort.
Die Fortsetzungsadresse für den Programmablauf kann mit %CONTINUE nicht beeinflusst werden. Eine andere Fortsetzungsadresse können Sie nur festlegen, indem Sie mit %SET den Befehlszähler (%PC) ändern (siehe Beschreibung des Kommandos %SET, *schlüsselwort,* Seite 275).

Im Gegensatz zu %RESUME wird ein aktiver %TRACE durch %CONTINUE nicht beendet, sondern entsprechend den Vereinbarungen ausgeführt.

Kommando	Operand
%CONT[INUE]	

Ein %TRACE gilt als aktiv, sobald er eingegeben wurde.

In den folgenden Fällen ist der %TRACE nur unterbrochen und kann mit %CONTINUE fortgesetzt werden:

1. Ein Subkommando wurde ausgeführt, und in diesem Subkommando ist ein %STOP enthalten.
2. Die K2-Taste wurde gedrückt (siehe Abschnitt „Kommandos zu Beginn einer Testsitzung“ auf Seite 19).

Steht in einem Subkommando nur das Kommando %CONTINUE, wird nur der Durchlaufzähler erhöht.

Steht %CONTINUE in einer Kommandofolge oder in einem Subkommando, werden nachfolgende Kommandos nicht mehr ausgeführt.

%CONTINUE verändert den Programmzustand.

%CONTROLn

Mit %CONTROL*n* können Sie nacheinander bis zu sieben Ablaufüberwachungs-Funktionen vereinbaren, die dann gleichzeitig wirken. Es gibt %CONTROL1 bis %CONTROL7.

– Mit *kriterium* wählen Sie verschiedene Typen von Programmanweisungen aus. Steht eine Anweisung des gewählten Typs zur Ausführung an, unterbricht AID das Programm und bearbeitet *subkdo*.

– Mit *control-bereich* legen Sie den Programmbereich fest, in dem *kriterium* überwacht werden soll.

– Mit *subkdo* definieren Sie ein Kommando oder eine Kommandofolge und eventuell eine Bedingung. Bei zutreffendem *kriterium* und erfüllter Bedingung wird *subkdo* ausgeführt. Wenn *subkdo* nicht explizit angegeben ist, wird standardmäßig <%STOP> eingesetzt.

Kommando	Operand
%C[ONTROL]n	[kriterium][,...] [IN control-bereich] [<subkdo>]

Mehrere %CONTROL*n*-Kommandos mit unterschiedlichen Nummern beeinflussen einander nicht, so dass Sie mehrere Kommandos mit demselben *kriterium* für verschiedene Bereiche oder mit unterschiedlichen *kriterien* für denselben Bereich aktivieren können. Treffen an einer Anweisung mehrere %CONTROL*n* zusammen, so werden die zugehörigen Subkommandos in der Reihenfolge %C1 bis %C7 bearbeitet.

Der einzelne Operandenwert eines %CONTROL*n* gilt solange, bis Sie ihn durch neue Angaben in einem späteren %CONTROL*n* mit derselben Nummer überschreiben, bis Sie den %CONTROL*n* löschen oder bis zum Programmende. Außerdem gilt, dass in einer durch `fork()` erzeugten Task und in einem mit `exec()` geladenen Programm alle Vereinbarungen mit %CONTROL*n* zurückgesetzt sind.

Mit %REMOVE löschen Sie einen bestimmten %CONTROL*n* oder alle aktiven %CONTROL*n*-Vereinbarungen.

%CONTROL*n* kann nur im laufenden Programm eingesetzt werden, deshalb muss die Basisqualifikation E=VM eingestellt sein (siehe %BASE) oder explizit angegeben werden.

%CONTROL*n* verändert den Programmzustand nicht.

kriterium

Schlüsselwort, das den Typ der Programmanweisungen festlegt, vor deren Ausführung AID *subkdo* bearbeiten soll.
Sie können mehrere Schlüsselwörter gleichzeitig angeben, die dann gemeinsam wirken. Zwischen zwei Schlüsselwörtern muss ein Komma stehen.
Wird kein *kriterium* vereinbart, arbeitet AID mit dem Standardwert %STMT, falls nicht noch aus einem vorhergehenden %CONTROL*n* eine *kriterium*-Vereinbarung gültig ist.

kriterium	*subkdo* **wird bearbeitet vor**
%STMT	jeder Anweisung
%ASSGN	jeder Zuweisungsanweisung
%CALL	jedem Funktionsaufruf
%COND	jeder if- und switch-Anweisung, jedem else-Zweig der if-Anweisung und jedem Bedingungsteil der do-, while- oder for-Anweisung
%EH %EXCEPTION	jeder catch- und throw-Anweisung
%GOTO	jeder goto-, break- und continue-Anweisung
%LAB	jeder Anweisung mit einer Marke, gilt jedoch nicht für case- und default-Marken
%PROC	der ersten und der letzten Anweisung einer Funktion

Tabelle 4: Werte des Operanden *kriterium* und ihre Bedeutung

control-bereich

legt den Programmbereich fest, in dem die Überwachungsfunktion wirksam wird. Beim Verlassen des festgelegten Programmbereichs wird die Überwachungsfunktion inaktiv, bis wieder eine Anweisung ausgeführt wird, die in dem zu überwachenden Programmbereich liegt.

Eine *control-bereich*-Definition gilt bis zum nächsten %CONTROL*n* derselben Nummer mit neuer Definition, bis zum entsprechenden %REMOVE oder bis Programmende. Auch in einer durch `fork()` erzeugten Task und in einem mit `exec()` geladenen Programm sind alle *control-bereich*-Definitionen zurückgesetzt.
Ein %CONTROL*n* ohne eigenen *control-bereich*-Operanden übernimmt eine wirksame Bereichsdefinition. Ein wirksamer *control-bereich* muss in einem %CONTROL*n* mit derselben

Nummer definiert sein, und die aktuelle Unterbrechungsstelle muss innerhalb dieses Bereichs liegen. Gibt es keine wirksame Bereichsdefinition, so umfasst der *control-bereich* standardmäßig die aktuelle Übersetzungseinheit.

```
control-bereich-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - -

                ⎧S=srcname                                              ⎫
                ⎪                                                       ⎪
IN   [•][E=VM•]⎨              ⎧[qua•][PROC=]funktion                 ⎫ ⎬
                ⎪[S=srcname•]⎨BLK='[f-]n[:b]'                        ⎬ ⎪
                ⎩              ⎩([PROC=funktion•]src-ref:src-ref)    ⎭ ⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

- Steht der Punkt an führender Stelle, so ist er das Kennzeichen für eine *vorqualifikation*. Sie muss mit einem vorhergehenden %QUALIFY-Kommando definiert worden sein.
  Aufeinanderfolgende Qualifikationen werden durch einen Punkt getrennt. Außerdem muss zwischen der letzten Qualifikation und dem anschließenden Operandenteil ein Punkt stehen.

E=VM

Da *control-bereich* nur im virtuellen Speicher des geladenen Programms liegen kann, geben Sie E=VM nur an, wenn als aktuelle Basisqualifikation eine Dump-Datei vereinbart ist (siehe %BASE).

S=srcname

geben Sie an, wenn *control-bereich* nicht in der aktuellen Übersetzungseinheit liegen soll oder wenn eine vereinbarte Bereichseinschränkung nicht mehr gelten soll. Endet *control-bereich* mit einer S-Qualifikation, so umfasst er die gesamte angegebene Übersetzungseinheit.

Die mit *srcname* bezeichnete Übersetzungseinheit muss zum Zeitpunkt der Eingabe des %CONTROL*n* bzw. bei Abarbeitung des Subkommandos, in dem der %CONTROL*n* enthalten ist, geladen sein.

[qua•][PROC=]funktion

geben Sie an, wenn *control-bereich* nicht in der aktuellen Funktion liegen soll oder um eine bisher geltende *control-bereich*-Vereinbarung zu überschreiben.
Bei Funktionen aus C-Programmen ist *funktion* der im Quellprogramm vergebene Name der Funktion ohne Klammern und ohne Signatur.
Funktionen aus C++-Programmen müssen Sie, je nachdem, um welchen Typ es sich handelt, in der Schreibweise `n'...'` oder `t'...'` angeben. Ist die Funktion in einem Namespace oder in einer Klasse definiert, so wird dem Funktionsnamen

die Namespace- oder Klassen-Qualifikation vorangestellt.
Die Signatur `void` darf nicht mehr geschrieben werden. Wie auch in C++ möglich geben Sie in diesem Fall nur die beiden Klammern nach dem Funktionsnamen an.

Für *funktion* ergibt sich die folgende Syntax:

```
[namespace::[...]][klasse::[...]]{n'funktion([signatur])'                }
                                 {t'f_template<arg[,...]>([signatur])'   }
```

Abweichend davon werden die Funktionen `main` und `__STI__` sowie alle Funktionen mit C-Linkage auch beim Testen in C++-Programmen nur mit dem Funktionsnamen bezeichnet (siehe Seite 60).

Virtuelle Funktionen sprechen Sie mit der folgenden Syntax an:
`p->n'funktion([signatur])'`

p ist eine Zeigervariable, die auf das Objekt einer Klasse verweist, das die gewünschte Member-Funktion enthält. Wenn *p* von der aktuellen Unterbrechungsstelle aus nicht zu erreichen ist, muss dem Gültigkeitsbereich entsprechend qualifiziert werden. Liegt die Unterbrechungsstelle in der virtuellen Funktion selbst, dann können Sie die Prologadresse der aktuellen Funktion ansprechen, indem Sie statt *p* den this-Zeiger einsetze*n* (siehe Beschreibung von *this* auf Seite 66 und Abschnitt „Virtuelle Funktionen“ auf Seite 76).

Um als Programmbereich eine Funktion über Pointer-to-Member anzugeben, stehen Ihnen die folgenden beiden Möglichkeiten zur Verfügung:

Das Objekt der Klasse, das die gewünschte Funktion enthält, wird mit seinem Namen bezeichnet. Als Dereferenzierungsoperator schreiben Sie `.*`

```
[qua.]objekt.*[objekt.][klasse::][...]pointer-to-function-member
```

Das Objekt der Klasse wird über einen Zeiger angesprochen. Als Dereferenzierungsoperator schreiben Sie **->***

```
[qua.]zeiger->*[objekt.][klasse::][...]pointer-to-function-member
```

Das Objekt der Klasse bezeichnen Sie mit dem Operanden, der auf der linken Seite der Dereferenzierungsoperatoren •* bzw. **->*** steht:
mit *objekt* bezeichnen Sie das Objekt der Klasse mit seinem Namen,
mit *zeiger* adressieren Sie das Objekt über einen Zeiger.
Rechts der Deferenzierungsoperatoren schreiben Sie den Namen des Pointer-to-Function-Member. Evtl. müssen Sie das Objekt, das die Definition des Pointer-to-Function-Member enthält, und innerhalb des Objekts die zur eindeutigen Ansprache nötige Klassenqualifikation davorschreiben, falls der Pointer-to-Member von der Unterbrechungsstelle aus anders nicht zu erreichen ist. Näheres zum Arbeiten mit Pointer-to-Function-Member finden Sie auf Seite 83.

qua
: Ist die Funktion in einer lokalen Klasse definiert, so müssen Sie vor dem Namen der gewünschten Funktion eine PROC-Qualifikation für die übergeordnete Funktion angeben, die die Definition der lokalen Klasse enthält. Für Funktionen, die in einem inneren Block der übergeordneten Funktion definiert sind, schließen Sie an die PROC-Qualifikation für die übergeordnete Funktion eine oder ggf. mehrere BLK-Qualifikationen, jeweils durch einen Punkt getrennt, an, um den Pfad zu der lokalen Klasse zu beschreiben (siehe Seite 62).

  Syntax für *qua*:

```
PROC=übergeordnete_fkt•[BLK='[f-]n[:b]'•[...]]
```

  Das Ansprechen von Funktionen, die in lokalen Klassen innerer Blöcke definiert sind, wird nur bei Programmen unterstützt, die mit einem C/C++-Compiler ab V3.0B übersetzt wurden.

BLK='[f-]n[:b]'
: *control-bereich* wird durch eine BLK-Qualifikation festgelegt und umfasst den gesamten angegebenen Block. Der Name für einen Block wird aus Zeilennummer (*n*) und gegebenenfalls FILE-Nummer (*f*) und relativer Blocknummer (*b*) gebildet

  Die BLK-Qualifikation kann nicht zusammen mit dem *kriterium* %PROC verwendet werden.

([PROC=funktion•]src-ref : src-ref)
: Mit Source-Referenzen können Sie *control-bereich* durch Angabe einer Anfangs- und Endadresse festlegen. Beide müssen innerhalb derselben Übersetzungseinheit liegen, und es gilt:
  Anfangsadresse ≤ Endadresse.

Es ist zu beachten, dass aufsteigenden Source-Referenzen nur innerhalb eines Funktionsblocks aufsteigende Adressen zugeordnet sind. Falls die Bedingung Anfangsadresse ≤ Endadresse nicht erfüllt ist, weist AID das Kommando mit einer entsprechenden Fehlermeldung ab.

Soll *control-bereich* nur eine Anweisung umfassen, müssen Anfangs- und Endadresse gleich sein.

PROC=funktion•
: Die PROC-Qualifikation müssen Sie nur dann schreiben, wenn die angegebenen Source-Referenzen in der Übersetzungseinheit nicht eindeutig sind. Dies ist dann der Fall, wenn die Source-Referenzen in einer Funktion liegen, die durch Instanziierung aus einem Funktionstemplate hervorgegangen ist, oder wenn die Funktion, die die Source-Referenzen enthält, in einem Klassentemplate definiert ist, und wenn zu dem Template mindestens zwei Instanzen existieren (siehe oben sowie Seite 112).

src-ref
: wird mit `S'[`*f*`-]`*n*`[:`*a*`]'` angegeben und bezeichnet die Adresse einer ausführbaren Anweisung. Dabei ist *n* die Zeilennummer, *f* die FILE-Nummer, falls >0 und *a* die relative Anweisungsnummer innerhalb der Zeile, falls >1.

subkdo

wird immer dann bearbeitet, wenn im *control-bereich* eine Anweisung zur Ausführung ansteht, die *kriterium* entspricht. *subkdo* wird vor der Ausführung der *kriterium*-Anweisung bearbeitet.
Wird der *subkdo*-Operand nicht angegeben, so setzt AID ein <%STOP> ein.

Vollständig beschrieben finden Sie *subkdo* im Basishandbuch, Kapitel „Subkommando“ [1].

```
subkdo-OPERAND  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

                                      ⎧AID-kommando    ⎫
<[subkdoname:] [(bedingung):] [       ⎨                ⎬ {;...}]>
                                      ⎩BS2000-kommando ⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Ein Subkommando kann einen Namen, eine Bedingung und einen Kommandoteil enthalten. Zu jedem Kommando gehört ein Durchlaufzähler. Der Kommandoteil kann aus einem einzelnen Kommando oder einer Kommandofolge bestehen, er kann AID- und BS2000-Kommandos und Kommentare enthalten.

Wenn das Subkommando aus einem Namen oder einer Bedingung besteht, aber der Kommandoteil fehlt, erhöht AID beim Erreichen einer Anweisung vom Typ *kriterium* nur den Durchlaufzähler.

Im *subkdo* eines %CONTROL*n* sind zusätzlich zu den Kommandos, die in allen Subkommandos nicht zugelassen sind, die AID-Kommandos %CONTROL*n*, %INSERT und %ON nicht erlaubt.

Die Kommandos in einem *subkdo* werden nacheinander ausgeführt. Danach wird das Programm fortgesetzt. Die Kommandos zur Ablaufsteuerung verändern auch in einem Subkommando sofort den Programmzustand. Sie brechen *subkdo* ab und starten das Programm (%CONTINUE, %RESUME, %TRACE) oder halten es an (%STOP). Sie sind nur als letztes Kommando in einem *subkdo* sinnvoll, da im *subkdo* nachfolgende Kommandos nicht mehr ausgeführt werden. Auch ein Löschen des gerade aktiven Subkommandos mit %REMOVE ist nur als letztes Kommando in *subkdo* sinnvoll.

Adressoperanden in Subkommandos werden bei der Eingabe nicht automatisch mit den Qualifikationen ergänzt, die der aktuellen Unterbrechungsstelle entsprechen. Wenn im weiteren Testverlauf eine Anweisung vom Typ *kriterium* auftritt und AID das Programm unterbricht, um *subkdo* zu bearbeiten, können mit AID-Kommandos aus *subkdo* ohne Qualifikation nur die Daten und Funktionen angesprochen werden, die an der Adresse der Anweisung, die die Unterbrechung verursacht hat, sichtbar sind.

Beispiele

1. ```
   %control1 %call, %proc    in (s'123':s'250') <%display count; %stop>
   %c1 %call,%proc in (s'123':s'250') <%d count;%stop>
   ```

   Die beiden AID-Kommandos unterscheiden sich nur in der Schreibweise.
   Das erste Beispiel ist voll ausgeschrieben und enthält unterschiedlich viele Leerzeichen an den zulässigen Stellen, das zweite ist abgekürzt.

   Das %CONTROL1-Kommando gilt für die Kriterien %CALL und %PROC und soll zwischen den Anweisungszeilen 123 bis einschließlich 250 wirken.

   Tritt im Programmablauf im genannten Bereich eine Anweisung auf, die den Kriterien %CALL oder %PROC entspricht, wird aus *subkdo* der %DISPLAY für die Variable `count` ausgeführt. Anschließend wird durch %STOP der Programmablauf unterbrochen, und es können AID- oder BS2000-Kommandos eingegeben werden.

2. ```
   %control1 %call <%display 'call'; %stop>
   ```

   Vor jedem Funktionsaufruf führt AID den %DISPLAY aus *subkdo* aus und unterbricht dann das Programm aufgrund des %STOP-Kommandos.

3. `%control2 %goto <%sdump %nest p=max; %remove %c1; %stop>`

   Bevor eine `goto`-, `break`- oder `continue`-Anweisung ausgeführt wird, gibt AID die Aufrufhierarchie aus; wegen der Angabe `p=max` wird in die Systemdatei SYSLST geschrieben. Danach führt AID das %REMOVE-Kommando aus, mit dem die Vereinbarungen des %CONTROL1 gelöscht werden. Das Programm wird angehalten (%STOP).

4. `%c3 %proc in nread`

   Mit dem %C3 wird vereinbart, dass AID das Programm unterbrechen soll, bevor die erste oder die letzte Anweisung der Funktion `nread` ausgeführt wird.

5. `%c4 %assgn <(z1 le 10): %d tab[0])>`

   Mit dem %C4 wird vereinbart, dass AID das erste Element des Vektors `tab` vor jeder Zuweisungsanweisung ausgeben soll, aber nur wenn der Wert in `z1` kleiner oder gleich 10 ist.

6. *Ausnahmebehandlung*

   Im unten abgebildeten Programm `EXMEM.C` wird versucht, für zwei Zeiger `p` und `q` Speicherplatz zu allokieren. Die Größe des angeforderten Speichers ist so ausgelegt, dass durch die Allokierung die Ausnahmebehandlung angestoßen wird.

```
C++-Programm                                                    EXMEM.C
=============================================================================
SRC
LIN
  1  #include <iostream.h>
  2  #include <cstdlib>
  3  #include <new>
  4
  5  void main()
  6  {
  7    char* p;
  8    try
  9    {
 10      p = new char[0x1000000];
 11    }
 12    catch(bad_alloc)
 13    {
 14      cerr << "No more memory!\n";
 15      return;
 16    }
 17    char *q;
 18    try
 19    {
 20      q = new char[0x10000000];
 21    }
 22    catch(bad_alloc)
```

```
23    {
24      cerr << "No memory for second pointer!\n";
25      delete[] p;
26      try
27      {
28        q = new char[0x10000000];
29      }
30      catch(bad_alloc)
31      {
32        cerr << "No more memory!\n";
33        return;
34      }
35    }
36  }
```

Ablaufprotokoll:

```
/LOAD-PROG   *MOD(MYLIB,EXMEM,RUN-MODE=ADVANCED,PROGRAM-MODE=ANY),
TEST-OPTIONS=AID
%  BLS0523 ELEMENT 'EXMEM', VERSION '@' FROM LIBRARY '$TEST.MYLIB'
IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$EXMEM$', VERSION ' ' OF '1999-01-07
10:27:02' LOADED
/%c1 %eh in s=n'exmem.c' <%t 1 r %stmt>
/%r
10                        EXT.PROC START   , BLOCK START, , BLOCK START,
                          ASSIGN
20                        BLOCK END,  , BLOCK START, ASSIGN
24                         , BLOCK START, CALL
No memory for second pointer!
28                         , BLOCK START, ASSIGN
32                         , BLOCK START, CALL
No more memory!
%  CCM0998 CPU TIME USED: 0.1531 SECONDS
```

Unmittelbar nach dem Laden des Programms wird ein %CONTROL1-Kommando eingegeben, das den Ablauf der Ausnahmebehandlung protokollieren soll. Danach wird das Programm mit %RESUME gestartet.
In der Ausgabe sind die jeweils ersten Anweisungen der `try`- und `catch`-Blöcke in der Reihenfolge aufgelistet, wie sie vom Programm durchlaufen werden.

%DISASSEMBLE

Mit %DISASSEMBLE können Sie Speicherinhalte in symbolische Assembler-Notation rückübersetzen und ausgeben lassen. Die Ausgabe erfolgt über SYSOUT, SYSLST oder in eine katalogisierte Datei.

– Mit *ausgabe-menge* legen Sie den Umfang der Speicherinhalte fest, die ausgegeben werden sollen.

– Mit *start* bestimmen Sie die Adresse, bei der AID mit der Rückübersetzung beginnen soll.

Kommando	Operand
{ %DISASSEMBLE | %DA }	[ausgabe-menge] [FROM start]

Für Speicherinhalt, der nicht als Befehl interpretiert werden kann, wird eine Ausgabezeile erzeugt, die die sedezimale Darstellung des Speicherinhalts und den Hinweis `INVALID OPCODE` enthält. Die Suche nach gültigem Befehlscode geht dann in 2-Byte-Schritten vorwärts.

Mit einem %DISASSEMBLE ohne *start*-Operanden können Sie ein vorher eingegebenes %DISASSEMBLE-Kommando solange fortsetzen, bis Sie mit einem BS2000- oder AID-Kommando (LOAD-EXECUTABLE-PROGRAM, START-EXECUTABLE-PROGRAM, %BASE) oder mit einem `fork()`- oder `exec()`-Aufruf das Testobjekt wechseln oder einen neuen Operandenwert vereinbaren. AID setzt die Rückübersetzung an der Speicheradresse fort, die an die Adresse anschließt, die mit dem vorhergehenden %DISASSEMBLE-Kommando zuletzt bearbeitet wurde. Ist auch *ausgabe-menge* nicht angegeben, so erzeugt AID dieselbe Menge von Ausgabezeilen wie bisher vereinbart.

Haben Sie in einer Testsitzung noch kein %DISASSEMBLE-Kommando eingegeben, oder haben Sie das Testobjekt gewechselt und geben nun im %DISASSEMBLE-Kommando keine aktuellen Werte für einen oder beide Operanden an, dann arbeitet AID mit den Standardwerten 10 für *ausgabe-menge* und V'0' für *start*. Der Standardwert V'0' für *start* kann für C/C++-Programme nicht genutzt werden, da diese Programme in den oberen Adressraum des Speichers geladen werden. Sie müssen daher beim ersten Aufruf des Kommandos %DISASSEMBLE einen expliziten Wert für *start* angeben.

Mit %OUT können Sie steuern, wie die aufbereitete Speicherinformation dargestellt wird und ob sie auf SYSOUT, SYSLST oder in eine katalogisierte Datei ausgegeben werden soll. Der Aufbau der möglichen Ausgabezeilen ist im Anschluss an die Beschreibung des *start*-Operanden nachzulesen.

%DISASSEMBLE verändert den Programmzustand nicht.

ausgabe-menge

legt den Umfang der Speicherinhalte fest, die ausgegeben werden sollen. Wenn Sie *ausgabe-menge* nicht angeben, setzt AID beim ersten %DISASSEMBLE nach dem Laden des Programms den Standardwert 10 ein.

Bei jedem weiteren %DISASSEMBLE wird die zuletzt angegebenen *ausgabe-menge* genutzt.

```
ausgabe-menge-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - -

{anzahl }
{laenge }
{ALL    }

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

anzahl

gibt an, wieviele Assembler-Befehle rückübersetzt und ausgegeben werden sollen.

ist eine Ganzzahl mit einem Wert:
$1 \leq anzahl \leq 2^{31}-1$

laenge

gibt die Größe des Speicherinhalts an, der innerhalb eines einzelnen, eingegebenen %DISASSEMBLE-Kommandos interpretiert und ausgegeben werden soll.

ist eine Sedezimalzahl #'f..f' mit einem Wert:
$1 \leq laenge \leq 2^{31}-1$

ALL gibt an, dass die Assembler-Befehle bis zum Ende der CSECT rückübersetzt und ausgegeben werden sollen, in der der *start*-Wert liegt. Wenn *start* nicht angegeben ist, bestimmt die aktuelle %DA-Position die CSECT.

Wenn der *start*-Wert nicht innerhalb einer CSECT liegt, wird das Kommando mit einer Fehlermeldung abgewiesen.

start

legt die Adresse fest, an der die Rückübersetzung von Speicherinhalt in Assembler-Befehle beginnen soll. Wird *start* nicht angegeben, setzt AID beim ersten %DISASSEMBLE nach dem Laden des Programms den Standardwert V'0' ein. Wenn ein Programm nicht ab V'0' geladen wurde, gibt AID eine Fehlermeldung aus.
Bei jedem weiteren %DISASSEMBLE wird hinter dem zuletzt rückübersetzten Assembler-Befehl fortgefahren.

```
start=OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

                  ⎧ funktion[->]          ⎫
                  ⎪ L'label'              ⎪
FROM  [.][qua.] ⎨                        ⎬
                  ⎪ S'[f=]n[:a]'          ⎪
                  ⎩ kompl=speicherref     ⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

- Steht der Punkt an führender Stelle, ist er das Kennzeichen für eine *vorqualifikation*. Sie muss mit einem vorhergehenden %QUALIFY-Kommando definiert worden sein. Aufeinanderfolgende Qualifikationen werden durch einen Punkt getrennt. Außerdem muss zwischen der letzten Qualifikation und dem anschließenden Operandenteil ein Punkt stehen.

qua Eine oder mehrere Qualifikationen geben Sie an, wenn Sie eine Funktion, eine Marke oder eine Source-Referenz ansprechen wollen, die von der aktuellen Unterbrechungsstelle aus anders nicht zu erreichen ist.

E={VM | Dn}
: geben Sie nur an, wenn für eine Funktion, eine Marke oder eine Source-Referenz die aktuelle Basisqualifikation nicht gelten soll (siehe %BASE).

S=srcname
: geben Sie nur an, wenn eine Funktion, eine Marke oder eine Source-Referenz nicht in der aktuellen Übersetzungseinheit liegt (siehe Kapitel „Adressierung in C- und C++-Programmen“ auf Seite 21).

:: Die beiden Doppelpunkte für den globalen Namespace geben Sie an, wenn der Name eines Namespaces oder einer globalen Klasse oder Funktion durch eine gleichnamige Defintion an der Unterbrechungsstelle verdeckt ist. Vor den beiden Doppelpunkten kann nur eine E- oder S-Qualifikation stehen.

namespace::
: geben Sie an, wenn Sie eine Funktion ansprechen wollen, die in einem Namespace definiert ist und der Namespace ist bis zur Unterbrechungsstelle noch nicht durch eine `using`-Direktive bekannt gemacht worden, oder die Funktion wurde noch nicht durch eine `using`-Deklaration angemeldet, oder an der Unterbrechungsstelle besteht Mehrdeutigkeit.
Vor einer Namespace-Qualifikation kann nur eine E- oder S-Qualifikation stehen (siehe Abschnitt „Qualifikationen“ auf Seite 59).

klasse::
: geben Sie an, wenn die gewünschte Funktion in einer Klasse definiert ist und die Unterbrechungsstelle nicht zum Scope des Funktionsnamens gehört. Falls *klasse* eine Instanz eines Klassentemplates ist, müssen Sie für *klasse* die Schreibweise `t'`*k_template*`<`*arg*`[,...]>'` verwenden (siehe Abschnitt „Qualifikationen“ auf Seite 59).

BLK='[f-]n[:b]'
: Eine BLK-Qualifikation müssen Sie angeben, wenn Sie mit einer nachfolgenden PROC-Qualifikation eine Funktion aus einer lokalen Klasse bezeichnen und die Definition dieser lokalen Klasse steht im angegebenen Block (siehe unten PROC=*funktion*).
Der Name für einen Block wird aus Zeilennummer (*n*) und gegebenenfalls FILE-Nummer (*f*) und relativer Blocknummer (*b*) gebildet

PROC=funktion
: geben Sie nur an, wenn Sie eine Marke aus einer anderen als der aktuellen Funktion ansprechen wollen (siehe Kapitel „Adressierung in C- und C++-Programmen“ auf Seite 21) oder wenn die Rückübersetzung bei einer Source-Referenz beginnen soll, die in einer Instanz eines Funktionstemplates liegt oder die einer Funktion zugeordnet ist, die in einer Instanz eines Klassentemplates definiert ist (siehe Abschnitt „Templates“ auf Seite 98), und es existieren mindestens zwei Instanzen.
Bei Funktionen aus C-Programmen ist *funktion* der im Quellprogramm vergebene Name der Funktion ohne Klammern und ohne Signatur.
Funktionen aus C++-Programmen müssen Sie, je nachdem, um welchen Typ es sich handelt, in der Schreibweise `n'...'` oder `t'...'` angeben. Ist die Funktion in einem Namespace oder in einer Klasse definiert, so wird dem Funktionsnamen die Namespace- oder Klassen-Qualifikation vorangestellt. Die Signatur `void` darf nicht mehr geschrieben werden. Wie auch in C++ möglich geben Sie in diesem Fall nur die beiden Klammern nach dem Funktionsnamen an.

Es ergibt sich die folgende Syntax (*f-template* sowie *signatur* sind in der Syntax aus Platzgründen abgekürzt):

```
PROC=[namespace::[...]][klasse::[...]]{n'funktion([sign])'          }
                                      {t'f_templ<arg[,...]>([sign])'}
```

Abweichend davon werden die Funktionen `main` und `_ _STI_ _` sowie alle Funktionen mit C-Linkage auch beim Testen in C++-Programmen nur mit dem Funktionsnamen bezeichnet (siehe Seite 60).

Ist die Funktion in einer lokalen Klasse definiert, so müssen Sie vor der PROC-Qualifikation noch eine weitere PROC-Qualifikation für die übergeordnete Funktion angeben, die die Definition der lokalen Klasse enthält. Zwischen die beiden PROC-Qualifikationen schreiben Sie bei Funktionen, die in einem inneren Block der übergeordneten Funktion definiert sind, eine oder ggfs. mehrere BLK-Qualifikationen (siehe Seite 62).
Das Ansprechen von Funktionen, die in lokalen Klassen innerer Blöcke definiert sind, wird nur bei Programmen unterstützt, die mit einem C/C++-Compiler ab V3.0B übersetzt wurden.

funktion[->]

Damit legen Sie *start* auf die erste ausführbare Anweisung in einer Funktion bzw. auf den ersten Befehl in einer Bibliotheksfunktion.
funktion ist der Name einer Funktion, der im Quellprogramm vergeben wurde bzw. der Name einer Bibliotheksfunktion (s.o. PROC=*funktion*).

Für virtuelle Funktionen gilt die folgende Syntax:

```
p->n'funktion([signatur])'
```

p ist eine Zeigervariable, die auf das Objekt einer Klasse verweist, das die gewünschte Member-Funktion enthält. Wenn *p* von der aktuellen Unterbrechungsstelle aus nicht zu erreichen ist, muss sie ihrem Gültigkeitsbereich entsprechend qualifiziert werden. Liegt die Unterbrechungsstelle in der virtuellen Funktion selbst, dann können Sie die erste ausführbare Anweisung der aktuellen Funktion ansprechen, indem Sie statt *p* den `this`-Zeiger einsetzen.

Wenn Sie für *start* eine Funktion angeben wollen, die über Pointer-to-Member adressiert wird, stehen Ihnen die folgenden beiden Möglichkeiten zur Verfügung:

Das Objekt der Klasse, das die gewünschte Funktion enthält, wird mit seinem Namen bezeichnet. Als Dereferenzierungsoperator schreiben Sie `.*`

```
[qua•]objekt•*[objekt•][klasse::][...]pointer-to-function-member
```

Das Objekt der Klasse wird über einen Zeiger angesprochen. Als Dereferenzierungsoperator schreiben Sie **->***

```
[qua•]zeiger->*[objekt•][klasse::][...]pointer-to-function-member
```

Das Objekt der Klasse bezeichnen Sie mit dem Operanden, der auf der linken Seite der Dereferenzierungsoperatoren `•*` bzw. **->*** steht:
mit *objekt* bezeichnen Sie das Objekt der Klasse mit seinem Namen,
mit *zeiger* adressieren Sie das Objekt über einen Zeiger.
Rechts der Deferenzierungsoperatoren schreiben Sie den Namen des Pointer-to-Function-Member. Evtl. müssen Sie das Objekt, das die Definition des Pointer-to-Function-Member enthält, und innerhalb des Objekts die zur eindeutigen Ansprache nötige Klassenqualifikation davorschreiben, falls der Pointer-to-Member von der Unterbrechungsstelle aus anders nicht zu erreichen ist.

Die Disassemblierung beginnt mit der ersten ausführbaren Anweisung der aktuell dem Pointer-to-Function-Member zugeordneten Funktion.

Wollen Sie an die mit dem dereferenzierten Pointer-to-Function-Member bezeichneten Adresse eine Adressrechnung anschließen, damit die Disassemblierung an einer von Ihnen berechneten Adresse mitten in der Funktion beginnt, so müssen Sie beachten, dass Sie nicht unmittelbar an eine der obigen Syntaxen den Pointer-Operator anfügen können. Sie müssen vielmehr zunächst durch eine Typmodifikation, nämlich %al4, angeben, dass die durch den dereferenzierten Pointer-to-Function-Member bezeichnete Adresse Ausgangspunkt für eine Adressrechnung sein soll. Es ergibt sich die folgende Syntax:

```
dereferenzierter-pointer-to-function-member %al4->•offset
```

Die Adressrechnung beginnt in diesem Fall jedoch **nicht** mit der Adresse der ersten ausführbaren Anweisung, sondern mit der Prologadresse der Funktion.

Näheres zum Arbeiten mit Pointer-to-Function-Member finden Sie auf Seite 83.

Wenn Sie mit *start* eine Bibliotheksfunktion bezeichnen, müssen Sie *funktion* mit dem Pointer-Operator abschließen, da zu den Bibliotheksfunktionen keine LSD vorliegt. Die Rückübersetzung beginnt in diesen Fällen mit dem ersten Befehl des Prologs der Funktion.

L'label'

Damit legen Sie *start* auf die erste ausführbare Anweisung nach einer Marke. *label* ist eine Marke, die im Quellprogramm vergeben wurde.
In diesem Kommando können Sie *label* auch ohne L'...' angeben, da eine Verwechslung mit einem Datennamen nicht möglich ist.

S'[f-]n[:a]'

ist eine Source-Referenz und bezeichnet eine ausführbare Anweisung. Sie wird aus Zeilennummer (*n*) und gegebenenfalls der FILE-Nummer (*f*) sowie der relativen Anweisungsnummer innerhalb der Zeile (*a*) gebildet.
Liegt die Source-Referenz in einer Funktion, die durch Instanziierung aus einem Funktionstemplate hervorgegangen ist, oder ist die Funktion, die die Source-Referenz enthält, in einer Instanz eines Klassentemplates definiert, so müssen Sie bei Mehrdeutigkeit vor die Source-Referenz die entsprechende PROC-Qualifikation schreiben.

kompl-speicherref

sollte die Anfangsadresse eines Maschinenbefehls sein, andernfalls erhalten Sie eine unsinnige Disassemblierung. Folgende Operationen können darin vorkommen (siehe AID-Basishandbuch, Kapitel „Schlüsselwörter" [1]):

- Adressversatz (•)
- indirekte Adressierung (->)
- Typmodifikation (%A, %S, %SX)
- Längenmodifikation (%L=(ausdruck), %Ln)
- Adressselektion (%@(...))

Beginnt eine *kompl-speicherref* mit einer Source-Referenz, einer Marke oder einem Funktionsnamen, muss anschließend der Pointer-Operator geschrieben werden. Sie müssen jedoch beachten, dass durch den Pointer die symbolische Ebene verlassen wird. Wenn Sie nach einem Funktionsnamen einen Pointer schreiben, bezeichnen Sie damit nicht mehr die erste ausführbare Anweisung der Funktion, sondern den ersten Befehl des Prologs, den der Compiler zu dieser Funktion erzeugt hat. Wenn Sie eine Marke in einer komplexen Speicherreferenz verwenden, müssen Sie diese stets in `L'...'` setzen.
Ohne den Pointer-Operator können Source-Referenz, Marke oder Funktionsname überall da verwendet werden, wo auch Sedezimalzahlen stehen können.

Eine Typmodifikation ist nur sinnvoll, wenn der Inhalt einer Variablen als Adresse eingesetzt werden kann oder wenn Sie die Adresse aus einem Register entnehmen.

Beispiel: `%3g.2%al2->`
Die letzten beiden Bytes aus AID-Register %3G werden als Adresse benutzt.

Ausgabe des %DISASSEMBLE-Protokolls

Das %DISASSEMBLE-Protokoll wird standardmäßig mit Zusatzinformationen über SYSOUT ausgegeben (T=MAX). Mit %OUT können Sie die Ausgabemedien wählen und festlegen, ob AID Zusatzinformationen ausgeben soll oder nicht.

AID berücksichtigt die Modi XMAX und XFLAT für die Ausgabe des %DISASSEMBLE-Protokolls nicht. Statt dessen generiert es die Standardausgabe (T=MAX).

Eine %DA-Ausgabezeile enthält folgende Elemente, wenn der Standardwert T=MAX gilt:

– CSECT-relative Speicheradresse
– in symbolische Assembler-Notation rückübersetzter Speicherinhalt, wobei Distanzen im Gegensatz zum Assembler-Format als Sedezimalzahlen dargestellt werden.
– für Speicherinhalt, der nicht mit einem gültigen Operationscode beginnt, wird eine Assembler-Anweisung DC im Sedezimal-Format mit der Länge von 2 Bytes aufgebaut, der der Hinweis `INVALID OPCODE` folgt.
– sedezimale Darstellung des Speicherinhalts (Maschinencode).

Beispiel zum Zeilenaufbau mit T=MAX

Ab der ersten Anweisung der Funktion facul (siehe Beschreibung des Kommandos %SDUMP, Beispiel 4 auf Seite 259) sollen acht Maschinenbefehle rückübersetzt werden.

```
/%disassemble 8 from facul
         EXAMP$O&@
 EXAMP$O*+20A      TM    O(R9),X'80'           91 80 9000
 EXAMP$O*+20E      BC    B'1000',3A(R0,R10)    47 80 A03A
 EXAMP$O*+212      L     R1,8(R0,R8)           58 10 8008
 EXAMP$O*+216      L     R15,4C(R0,R13)        58 F0 D04C
 EXAMP$O*+21A      ST    R13,18(R0,R15)        50 D0 F018
 EXAMP$O*+21E      L     R13,4(R0,R13)         58 D0 D004
 EXAMP$O*+222      ST    R13,C(R0,R15)         50 D0 F00C
 EXAMP$O*+226      L     R14,C(R0,R13)         58 E0 D00C
```

Beispiel zum Zeilenaufbau mit T=MIN

Mit dem %OUT-Operandenwert T=MIN baut AID verkürzte Ausgabezeilen auf, in denen die CSECT-relative Adresse durch die virtuelle Adresse ersetzt wird und die sedezimale Darstellung des Speicherinhalts entfällt.

```
/%out %da t=min
/%disassemble 8 from facul
 0100020A  TM    0(R9),X'80'
 0100020E  BC    B'1000',3A(R0,R10)
 01000212  L     R1,8(R0,R8)
 01000216  L     R15,4C(R0,R13)
 0100021A  ST    R13,18(R0,R15)
 0100021E  L     R13,4(R0,R13)
 01000222  ST    R13,C(R0,R15)
 01000226  L     R14,C(R0,R13)
```

Beispiele

1. `%disassemble from s=n'examp.c'.::facul`

   Das Kommando veranlasst die Rückübersetzung von zehn Befehlen (Standardwert) ab der Adresse der ersten Anweisung in der Funktion `facul`, die in der Übersetzungseinheit `EXAMP.C` liegt.

2. `%da 2 from e=d1.s'27'`

   In der Dump-Datei mit dem Linknamen `D1` sollen zwei Befehle des Programmcodes, der zu der ersten Anweisung in Zeile 27 erzeugt wurde, disassembliert werden.

3. `%da from s'27:2'`

   Da für *ausgabe-menge* kein Wert angegeben wurde, setzt AID entweder den Standardwert 10 ein, wenn es das erste %DISASSEMBLE für dieses Programm ist, oder übernimmt den Wert aus dem vorherigen %DISASSEMBLE. Die Rückübersetzung beginnt mit dem ersten Befehl, der zu der zweiten Anweisung in Zeile 27 erzeugt wurde.

4. `%disassemble from l'output'->.4`

   Vom ersten Befehl, der nach der Marke `output` steht, wird um 4 Bytes weiterpositioniert, und ab dort wird rückübersetzt.

5. `%da #'18' from facul`

   Es werden 24 Bytes am Anfang der Funktion `facul` disassembliert, d.h. die Befehle auf den Adressen 0100020A bis 0100021E. Der Befehl auf Adresse 0100022 wird nicht mehr angezeigt, da er außerhalb des Bereichs liegt.

%DISPLAY

Mit %DISPLAY veranlassen Sie die Ausgabe von Speicherinhalten, Adressen, Längen, Systeminformationen und AID-Literalen, und Sie können damit den Vorschub nach SYSLST steuern. Daten bereitet AID entsprechend der Definition im Quellprogramm auf, wenn Sie nicht mit Typmodifikation einen anderen Ausgabetyp wählen.
Mit %DISPLAY können Sie sich eine Übersicht über alle überladenen Funktionen ausgeben lassen, die im aktuellen oder explizit angegebenen Gültigkeitsbereich denselben Namen haben. Des weiteren können Sie das %DISPLAY-Kommando dazu verwenden, sich alle Instanzen eines Templates auflisten zu lassen.
Die Ausgabe erfolgt über SYSOUT, SYSLST oder in eine katalogisierte Datei.

– Mit *daten* bezeichnen Sie Namespaces, Templates, Klassen, Objekte von Klassen, Daten, deren Adressen und Längen, Anweisungen, Register, Durchlaufzähler von Subkommandos und Systeminformationen. Sie definieren AID-Literale, oder Sie steuern den Vorschub nach SYSLST.
– Mit *medium-u-menge* geben Sie an, welche Ausgabe-Medien AID verwenden soll und ob Zusatzinformationen ausgegeben werden sollen. Mit diesem Operanden setzen Sie eine mit %OUT getroffene Vereinbarung für das aktuelle %DISPLAY-Kommando außer Kraft.

Kommando	Operand
%D[ISPLAY]	daten {,...} [medium-u-menge][,...]

Ohne Qualifikation zu *daten* sprechen Sie Daten des aktuellen Blocks, bei geschachtelten Blöcken auch die Daten der äußeren Blöcke, außerdem die Daten der aktuellen Funktion, sofern sie nicht erst nach der Unterbrechungsstelle definiert sind, und die globalen Daten der zugehörigen Übersetzungseinheit an. Bei Namensgleichheit innerhalb der aktuellen Aufrufhierarchie gibt AID das Datum aus, das an der Unterbrechungsstelle auch vom Programm angesprochen worden wäre.

Mit einer Qualifikation können Sie *daten* in einer Dump-Datei oder in einer anderen geladenen Übersetzungseinheit, Funktion oder in einem anderen Block ansprechen, jedoch nur dann, wenn sich der Gültigkeitsbereich des angesprochenen Datums in der aktuellen Aufrufhierarchie befindet. Außerhalb der aktuellen Aufrufhierarchie können Sie nur Daten der Speicherklasse static oder extern ansprechen.

Wollen Sie eine überladene Funktion ansprechen, so können Sie sich mit
`%DISPLAY [`*qua*`]` *funktion*
eine Übersicht ausgeben lassen, die alle Funktionen mit dem angegebenen Namen in C++-Standard-Schreibweise mit Signatur und evtl. vorangehender Blocknummer und Klassennamen auflistet, die AID im aktuellen bzw. im explizit angegebenen Gültigkeitsbereich gefunden hat (siehe Abschnitt „Überladene Funktionen“ auf Seite 114).

Ebenso können Sie sich mit
`%DISPLAY [`*qua*`]` *template*
eine Übersicht über alle die Instanzen eines Klassen- oder Funktionstemplates ausgeben lassen, die an der aktuellen Unterbrechungsstelle sichtbar sind. Die Deklaration des Templates sucht AID von der Unterbrechungsstelle aus gemäß den Scoperegeln für Daten bzw. in dem mit *qua* bezeichneten Programmteil (siehe Seite 110). Wurde zu dem Template nur eine einzige Instanz angelegt, so können Sie sich diese mit
`%DISPLAY [`*qua*`] { t'`*k_template*`'|t'`*f_template*`([`*signatur*`])' }`
auflisten lassen.

AID ab V3.4B10 unterstützt auch die Ausgabe von Daten in unterschiedlichen EBCDIC- und ASCII-Zeichensätzen. Da BS2000-Terminals nur ausgewählte EBCDIC-Zeichensätze direkt unterstützen, muss zwischen folgenden Zeichensätzen unterschieden werden:

- Datenzeichensatz: Zeichensatz, in dem die Daten vorliegen bzw. interpretiert werden
- Ausgabezeichensatz: Zeichensatz, in dem die Daten dargestellt werden

AID interpretiert die Daten in dem Zeichensatz der beim %DISPLAY-Kommando angegeben ist. Wenn dort keiner angegeben ist, wird der Zeichensatz aus dem Operanden `CCS` des %AID-Kommandos verwendet.

Zuvor müssen Sie den Ausgabezeichensatz mit dem Kommando MODIFY-TERMINAL-OPTIONS einstellen. Es muss ein EBCDIC-Zeichensatz sein, der vom Terminal unterstützt wird. UTFE ist nicht zulässig. Außerdem muss der Ausgabezeichensatz in der gleichen Gruppe sein, wie der Datenzeichensatz. Wenn beispielsweise der Datenzeichensatz ISO88592 ist, stellen Sie zunächst mit /MOD-TERM-OPT CODE=EDF042 den entsprechenden Ausgabezeichensatz ein (siehe Benutzerhandbuch XHCS).

`%DISPLAY <`*data-start*`> { %C|%X }[Lddd] ['<`*coded-character-set*`>']`

Wenn Sie das Kommando %DISPLAY mit dem Speichertyp `%C` oder `%X` eingeben, gibt AID Zeichen in Übereinstimmung mit dem explizit definierten Zeichensatz `<coded-character-set>` aus, oder in Übereinstimmung mit dem aktuellen Zeichensatz `CCS`, falls '`<coded-character-set>`' nicht angegeben ist. `%C` and `%X` legen verschiedene Ausgabelayouts fest.

```
%DISPLAY <char-variable> ['<coded-character-set>']
```

Wenn char-Variablen ausgegeben werden sollen, gibt AID diese in Übereinstimmung mit dem explizit definierten Zeichensatz <coded-character-set> aus, oder in Übereinstimmung mit dem aktuellen Zeichensatz CCS. Das Ausgabelayout unterscheidet sich jedoch von den Layouts, die durch %C bzw. %X festgelegt sind.

Den aktuellen Zeichensatz CCS zeigen Sie mit folgendem AID-Kommando an:
`%SHOW %AID`

Den aktuellen Zeichensatz CCS können Sie mit folgendem AID-Kommando ändern:
`%AID CCS = {<coded-character-set>|*USRDEF}`

Ohne den *medium-u-menge*-Operanden gibt AID die Daten entweder gemäß den Vereinbarungen im %OUT-Kommando oder standardmäßig mit Zusatzinformationen über SYSOUT aus (siehe AID-Basishandbuch, Kapitel „Operand Medium-und-Menge“ [1]).

Unmittelbar nach dem Laden können Sie nur globale und static vereinbarte Daten ansprechen. Zum Zugriff benötigt AID die entsprechenden Qualifikationen.

Neben den hier beschriebenen Operandenwerten können Sie auch die im Handbuch für das Testen auf Maschinencode-Ebene [2] beschriebenen Operandenwerte einsetzen.

Sie können mit diesem Kommando im geladenen Programm und in einer Dump-Datei arbeiten.

%DISPLAY verändert den Programmzustand nicht.

daten

beschreibt, welche Informationen AID ausgeben soll. Sie können sich Inhalt, Adresse und Länge von Variablen, Vektoren, Vektorelementen, Strukturen, Unionen, Namespaces und deren Komponenten, Objekten von Klassen und deren Komponenten und die Adressen von Anweisungen und Funktionen ausgeben lassen. Den Inhalt von Registern und Durchlaufzählern sowie für Ihr Programm relevante Systeminformationen können Sie über Schlüsselwörter adressieren. Um die Protokolle Ihrer Tests übersichtlicher zu gestalten, können Sie AID-Literale definieren oder den Vorschub nach SYSLST steuern.

Daten bereitet AID entsprechend der Definition im Quellprogramm auf, wenn Sie nicht mit Typmodifikation einen anderen Ausgabetyp festlegen (siehe AID-Basishandbuch, Abschnitt „Typmodifikation“ [1]).

Geben Sie in einem %DISPLAY mehrere *daten*-Operanden an, so können Sie von Operand zu Operand wechseln zwischen den hier beschriebenen symbolischen Angaben und den nicht-symbolischen, wie sie im Handbuch für das Testen auf Maschinencode-Ebene [2] beschrieben sind. Auch innerhalb einer komplexen Speicherreferenz können Sie symbolische und maschinennahe Angaben mischen.

Falls Sie mit überladenen Operatoren arbeiten, müssen Sie beachten, dass AID diesen Vorgang nicht nachvollzieht, sondern stets mit den Standard-Operatoren arbeitet.

Geben Sie für *daten* einen Namen an, der nicht in den LSD-Sätzen verzeichnet ist, gibt AID eine Fehlermeldung aus. Die anderen *daten* desselben Kommandos werden ordnungsgemäß bearbeitet.

- Steht der Punkt an führender Stelle, ist er das Kennzeichen für eine *vorqualifikation*. Sie muss mit einem vorhergehenden %QUALIFY definiert worden sein. Aufeinanderfolgende Qualifikationen werden durch einen Punkt getrennt. Außerdem muss zwischen der letzten Qualifikation und dem anschließenden Operandenteil ein Punkt stehen.

qua Eine oder mehrere Qualifikationen geben Sie an, wenn die Unterbrechungsstelle nicht im Gültigkeitsbereich von *daten* liegt oder wenn *daten* an der Unterbrechungsstelle nicht sichtbar ist. Sie geben nur die Qualifikationen an, die zur eindeutigen Ansprache des Speicherobjekts genügen.

E={VM | Dn}
: geben Sie nur an, wenn für einen Datennamen, einen Anweisungsnamen, eine Source-Referenz oder ein Schlüsselwort die aktuelle Basisqualifikation nicht gelten soll (siehe Beschreibung des Kommandos %BASE).

S=srcname
: geben Sie nur an, wenn Sie einen Datennamen, einen Namespace, eine Klasse oder ein Objekt einer Klasse, einen Anweisungsnamen oder eine Source-Referenz ansprechen, die nicht in der aktuellen Übersetzungseinheit liegen (siehe Kapitel „Adressierung in C- und C++-Programmen“ auf Seite 21).

:: Die beiden vorangestellten Doppelpunkte verwenden Sie, um ein globales Datum anzusprechen, das durch eine gleichnamige Definition an der Unterbrechungsstelle lokal verdeckt ist. Außerdem müssen Sie die beiden Doppelpunkte vor den Namen eines globalen Datums oder einer Funktion setzen, weil entweder das Datum oder die Funktion nicht in der Aufrufhierarchie liegen oder weil deren Definition erst nach der Unterbrechungsstelle steht.
Im Gegensatz zu den übrigen Qualifikationen wird zwischen den beiden Doppelpunkten und dem anschließenden Operanden kein Punkt geschrieben.

PROC=funktion
: geben Sie nur an, wenn Sie einen Datennamen ansprechen wollen, der zwar in der aktuellen Funktion definiert ist, aber von einer gleichnamigen Definition an der Unterbrechungsstelle verdeckt wird. Außerdem geben Sie eine PROC-Qualifikation an, wenn Sie eine Marke oder einen static vereinbarten Datennamen ansprechen wollen, der in einer Funktion außerhalb der aktuellen Aufrufhierarchie definiert ist (siehe Kapitel „Adressierung in C- und C++-Programmen“ auf Seite 21). Wenn Sie eine Source-Referenz angeben, die in einer Instanz eines Funktionstemplates liegt oder die einer Funktion zugeordnet ist, die in einer Instanz eines Klassentemplates definiert ist (siehe Abschnitt „Templates“ auf Seite 98), müssen Sie bei Mehrdeutigkeit ebenfalls die entsprechende PROC-Qualifikation davorschreiben.
Bei Funktionen aus C-Programmen ist *funktion* der im Quellprogramm vergebene Name der Funktion ohne Klammern und ohne Signatur.
Funktionen aus C++-Programmen müssen Sie, je nachdem, um welchen Typ es sich handelt, in der Schreibweise `n'...'` oder `t'...'` angeben. Ist die Funktion in einem Namespace oder in einer Klasse definiert, so wird dem Funktionsnamen die Namespace- oder Klassen-Qualifikation vorangestellt. Die Signatur `void` darf nicht mehr geschrieben werden. Wie es auch in C++ mög-

lich ist, geben Sie in diesem Fall nach dem Funktionsnamen nur die beiden Klammern an. Es ergibt sich die folgende Syntax (*f-template* sowie *signatur* sind in der Syntax aus Platzgründen abgekürzt):

```
PROC=[namespace::[...]][klasse::[...]]{n'funktion([sign])'          }
                                      {t'f_templ<arg[,...]>([sign])'}
```

Abweichend davon werden die Funktionen `main` und `_ _STI_ _` sowie alle Funktionen mit C-Linkage auch beim Testen in C++-Programmen nur mit dem Funktionsnamen bezeichnet (siehe Seite 60).

Ist die Funktion in einer lokalen Klasse definiert, so müssen Sie vor der PROC-Qualifikation noch eine weitere PROC-Qualifikation für die übergeordnete Funktion angeben, die die Definition der lokalen Klasse enthält. Zwischen die beiden PROC-Qualifikationen schreiben Sie bei Funktionen, die in einem inneren Block der übergeordneten Funktion definiert sind, eine oder ggfs. mehrere BLK-Qualifikationen (siehe Seite 62).
Das Ansprechen von Funktionen, die in lokalen Klassen innerer Blöcke definiert sind, wird nur bei Programmen unterstützt, die mit einem C/C++-Compiler ab V3.0B übersetzt wurden.

BLK='[f-]n[:b]'

geben Sie an, wenn Sie einen Datennamen ansprechen wollen, der einem Block innerhalb der aktuellen Aufrufhierarchie zugeordnet ist und von einer gleichnamigen Definition an der Unterbrechungsstelle verdeckt wird oder wenn Sie einen static vereinbarten Datennamen ansprechen wollen, der einem Block außerhalb der aktuellen Aufrufhierarchie zugeordnet ist (siehe Kapitel „Adressierung in C- und C++-Programmen“ auf Seite 21).

Des weiteren müssen Sie eine BLK-Qualifikation angeben, wenn Sie mit einer nachfolgenden PROC-Qualifikation eine Funktion aus einer lokalen Klasse bezeichnen und die Definition dieser lokalen Klasse steht im angegebenen Block (siehe oben PROC=*funktion*).

NESTLEV= level-nummer

level-nummer Nummer einer Ebene in der aktuellen Aufrufhierarchie

Auf *level-nummer* muss *datenname* folgen.
Das %DISPLAY-Kommando gibt das Datenelement *datenname* aus, das auf der Ebene *level-nummer* der aktuellen Aufrufhierarchie definiert wurde.

namespace
: ist der im Quellprogramm deklarierte Name eines Namespaces.

  Endet der *daten*-Operand mit dem Namen eines Namespaces, so listet AID alle darin definierten Daten und Funktionen auf. Die Funktionen werden in C++-Standard-Schreibweise aufgeführt; die Anfangsadresse des zugehörigen Prologs wird ausgegeben. Bei geschachtelten Namespaces wird der Inhalt der inneren Schachtelungen mit ausgegeben. Enthält ein Namespace eine `using`-Direktive auf einen weiteren Namespace, so wird dieser nur mit seinem Namen aufgeführt.

  Im Adressierungspfad zu Klassen, Daten oder Funktionen, die im Namespace definiert sind, geben Sie die Namespace-Qualifikation nur dann an, wenn die gewünschte Komponente des Namespaces an der Unterbrechungsstelle nicht sichtbar ist.

  Vor der Namespace-Qualifikation sind nur die E- oder S-Qualifikation oder die beiden Doppelpunkte (`::`) für den globalen Namespace möglich.
  Weitere Informationen zu den Namespaces finden Sie im Abschnitt „Namespaces“ auf Seite 89.

{ klasse | this-> | objekt }
: ist der im Quellprogramm deklarierte Name einer Klasse, der `this`-Zeiger oder der Name eines Objekts einer Klasse.
  Klassennamen, den `this`-Zeiger mit nachfolgendem Pointer-Operator sowie die Namen von Objekten von Klassen geben Sie an, um den Adressierungspfad zu Daten-Membern aus Klassen zu beschreiben (siehe Abschnitt „Klassen“ auf Seite 65).

  Liegt die aktuelle Unterbrechungsstelle in einer dynamischen Member-Funktion, können Sie die Daten der Klasse gemäß den aus C++ bekannten Scoperegeln ansprechen.

  Unabhängig von der Unterbrechungsstelle erreichen Sie die dynamischen Daten eines Objekts über den Objektnamen mit nachfolgendem Punkt, falls das Objekt in der aktuellen Aufrufhierarchie liegt.

  Statische Daten-Member erreichen Sie von jeder Stelle des Programms über den zugehörigen Klassennamen mit den beiden nachfolgenden Doppelpunkten. Bei geschachtelten Klassen enthält der Pfad zum Datum alle Klassennamen von außen nach innen, jeweils durch zwei Doppelpunkte getrennt. Der äußerste Klassennamen benötigt die dem Gültigkeitsbereich entsprechende Qualifikation. Ist das Programm in einer Member-Funktion der Klasse unterbrochen, so gelten für das Ansprechen von statischen Daten-Member die Regeln des Klassenscopes, d.h. wenn das Daten-Member nicht durch eine gleichnamige Defintion verdeckt ist, so kann es unqualifiziert angesprochen werden.

Handelt es sich bei der Klasse um eine Instanz eines Klassentemplates, müssen Sie die folgende Schreibweise verwenden: `t'`*k_template*`<`*arg*`[,...]>'`. Existiert nur eine einzige Instanz des Templates, genügt die Angabe: `t'`*k_template*`'`.

Besteht der *daten*-Operand aus einem oder mehreren Klassennamen und die Unterbrechungsstelle liegt außerhalb des Klassensystems, werden die statischen Daten-Member und alle Member-Funktionen außer den virtuellen aufgelistet. Zu den Daten wird der aktuelle Inhalt ausgegeben. Die Member-Funktionen werden in C++-Standard-Schreibweise aufgeführt; die Anfangsadresse des zugehörigen Prologs wird ausgegeben. Bei abgeleiteten Klassen gibt AID auch die Basisklassen aus. Bei geschachtelten Klassen wird der Inhalt der inneren Schachtelungen mit ausgegeben.
Liegt die Unterbrechungsstelle dagegen in einer Member-Funktion der Klasse, so listet AID zusätzlich die dynamischen Daten-Member und die virtuellen Funktionen auf. Zum Namen der virtuellen Funktion wird die Prologadresse der aktuell gültigen Member-Funktion angezeigt. Die gleiche Ausgabe erhalten Sie mit `%DISPLAY *this`.

Endet der *daten*-Operand mit einem Objektnamen, dann gibt AID ebenfalls das vollständige Objekt aus, also mit den dynamischen Daten-Membern und mit der vollständigen Information zu den virtuellen Funktionen. Falls es für die eindeutige Ansprache eines Datums oder einer Funktion aus einer Basisklasse erforderlich ist, geben Sie anschließend an den Objektnamen, getrennt durch einen Punkt, den Namen der gewünschten Basisklasse an. Innerhalb des Klassensystems gelten dann wiederum die von C++ bekannten Scoperegeln.

objekt benötigt die seinem Gültigkeitsbereich entsprechende Qualifikation. Vor `this` ist nur eine Basisqualifikation sinnvoll.
Endet der *daten*-Operand auf `this->`, so gibt AID 4 Bytes ab der Anfangsadresse des aktuellen Objekts im Dump-Format (Sedezimal und Character) aus.

datenname
: ist der im Quellprogramm definierte Name eines Datums. *datenname* wird wie im Quellprogramm angegeben.

  Daten können Sie bis auf folgende Ausnahmen wie in C/C++ ansprechen:
  Mit einem Vektornamen ohne Index gibt AID alle Vektorelemente aus.
  Vektorelemente können Sie nur über Index ansprechen, nicht über Zeiger.
  Es können auch Indexbereiche ausgegeben werden.
  Wenn `%AID C=YES` gesetzt ist (siehe Seite 119), fasst AID die Vektorelemente eines `char`-Vektors, die über den am weitesten rechts stehenden Index adressiert werden, zu C-Strings zusammen und gibt den Vektorinhalt in Form von C-String-Literalen aus.
  Zum Arbeiten mit Vektoren siehe auch die Abschnitte „Indexschreibweise“ auf Seite 31 und „C-Strings“ auf Seite 37.

Bei Variablen vom Typ `long double` wertet AID nur die ersten 8 Bytes aus. Variablen vom Typ `char` gibt AID im Ausgabetyp %C aus. Mit Hilfe einer Typmodifikation (%A oder %F) können Sie sich den entsprechenden numerischen Wert ausgeben lassen (siehe auch Seite 30). Die Datentypen `unsigned char` und `signed char` behandelt AID dagegen wie Integer-Variable.
Vektoren, die als Parameter an eine Funktion übergeben wurden, und Zeiger werden als Sedezimalzahl ausgegeben.

Bezeichnet *datenname* einen Pointer-to-Member, so erhalten Sie in der Ausgabe den Namen desjenigen Klassen-Members, auf das der Pointer-to-Member aktuell verweist. Den Inhalt des aktuellen Daten-Members bzw. die Anfangsadresse der aktuellen Member-Funktion gibt AID aus, wenn Sie im %DISPLAY-Kommando den dereferenzierten Pointer-to-Member angeben. Das Dereferenzieren von Pointer-to-Member können Sie auf Seite 79 nachschlagen.

datenname kann folgendermaßen angegeben werden. Die Formate können auch kombiniert werden (siehe Abschnitt „Datennamen“ auf Seite 30).

Indexschreibweise:	*datenname* `[`*index*`]` `{...}`
Zeigerschreibweise:	*datenname1* `->` *datenname2*
Strukturqualifizierung:	*übergeordneter datenname*`.` `{...}` *datenname*
Dereferenzierung:	`[(]*{...}` *datenname*`[)]`
Pointer-to-Member-Dereferenzierung:	*datenname1*`.*`*datennname2* oder *datenname1*`->*`*datenname2*

funktion

ist der Name einer Funktion, der im Quellprogramm vergeben wurde bzw. der Name einer Bibliotheksfunktion (siehe PROC=*funktion* auf Seite 157 und Kapitel „Adressierung in C- und C++-Programmen“ auf Seite 21). Ohne nachfolgenden Pointer-Operator gibt AID die Anfangsadresse des Prologs der Funktion aus. Mit Pointer-Operator werden die ersten 4 Bytes ab dieser Adresse ausgegeben.

Virtuelle Funktionen sprechen Sie mit der folgenden Syntax an:
`p->n'funktion([signatur])'`

p ist eine Zeigervariable, die auf das Objekt einer Klasse verweist, das die gewünschte Member-Funktion enthält. Wenn *p* von der aktuellen Unterbrechungsstelle aus nicht zu erreichen ist, muss dem Gültigkeitsbereich entsprechend qualifiziert werden. Liegt die Unterbrechungsstelle in der virtuellen Funktion selbst, dann können Sie die Prologadresse der aktuellen Funktion ansprechen, indem Sie statt *p* den `this`-Zeiger einsetzen (siehe Beschreibung von `this` auf Seite 66 und Abschnitt „Virtuelle Funktionen“ auf Seite 76).
Auf den Befehlscode ab der Anfangsadresse des Prologs können Sie mit `p->n‘funktion([signatur])‘->` zugreifen.

Wenn Sie sich die Anfangsadresse des Prologs einer Funktion ausgeben lassen wollen, die über Pointer-to-Member angesprochen wird, stehen Ihnen zur Dereferenzierung des Pointer-to-Member die folgenden beiden Möglichkeiten zur Verfügung:

Das Objekt der Klasse, das die gewünschte Funktion enthält, wird mit seinem Namen bezeichnet. Als Dereferenzierungsoperator schreiben Sie `.*`

```
[qua.]objekt.*[objekt.][klasse::][...]pointer-to-function-member
```

Das Objekt der Klasse wird über einen Zeiger angesprochen. Als Dereferenzierungsoperator schreiben Sie **->***

```
[qua.]zeiger->*[objekt.][klasse::][...]pointer-to-function-member
```

Das Objekt der Klasse bezeichnen Sie mit dem Operanden, der auf der linken Seite der Dereferenzierungsoperatoren `.*` bzw. **->*** steht:
mit *objekt* bezeichnen Sie das Objekt der Klasse mit seinem Namen,
mit *zeiger* adressieren Sie das Objekt über einen Zeiger.

Rechts der Deferenzierungsoperatoren schreiben Sie den Namen des Pointer-to-Function-Member. Evtl. müssen Sie das Objekt, das die Definition des Pointer-to-Function-Member enthält, und innerhalb des Objekts die zur eindeutigen Ansprache nötige Klassenqualifikation davorschreiben, falls der Pointer-to-Member von der Unterbrechungsstelle aus anders nicht zu erreichen ist.

Wenn Sie sich den Befehlscode ab der Prologadresse der Funktion ausgeben lassen wollen, so müssen Sie beachten, dass Sie nicht unmittelbar an eine der obigen Syntaxen den Pointer-Operator anfügen können. Vielmehr muss zunächst durch eine Typmodifikation, nämlich %al4, der Übergang auf die Maschinencode-Ebene ausgelöst werden, sodass sich die folgende Syntax ergibt:

`%DISPLAY` *dereferenzierter-pointer-to-function-member* `%al4->`

Näheres zum Arbeiten mit Pointer-to-Function-Member finden Sie auf Seite 83.

L'label'

bezeichnet die Adresse der ersten ausführbaren Anweisung nach einer Marke. *label* ist der Name einer Marke, der im Quellprogramm vergeben wurde.

S'[f-]n[:a]'

ist eine Source-Referenz und bezeichnet eine ausführbare Anweisung. Sie wird aus Zeilennummer (*n*) und gegebenenfalls der FILE-Nummer (*f*) sowie der relativen Anweisungsnummer innerhalb der Zeile (*a*) gebildet. Liegt die Source-Referenz in einer Funktion, die durch Instanziierung aus einem Funktionstemplate hervorgegangen ist, oder ist die Funktion, die die Source-Referenz enthält, in einer Instanz eines Klassentemplates definiert, so müssen Sie bei Mehrdeutigkeit vor die Source-Referenz die entsprechende PROC-Qualifikation schreiben.

Ohne nachfolgenden Pointer-Operator gibt AID die Adresse des Befehlscodes aus, der zu der Anweisung erzeugt wurde. Mit Pointer-Operator gibt AID 4 Bytes des Befehlscodes an dieser Adresse aus.

schlüsselwort

Sie können alle Schlüsselwörter für Programmregister, AID-Register, Systemtabellen und das für den Durchlaufzähler oder die symbolische Lokalisierungsinformation verwenden (siehe AID-Basishandbuch, Kapitel „Schlüsselwörter" [1]).
Vor *schlüsselwort* können Sie nur eine Basisqualifikation angeben.

%n	Mehrzweckregister, 0 ≤ n ≤ 15
%nD|E	Gleitpunktregister, n = 0,2,4,6
%nQ	Gleitpunktregister, n = 0,4
%nG	AID-Mehrzweckregister, 0 ≤ n ≤ 15
%nGD	AID-Gleitpunktregister, n = 0,2,4,6
%MR	alle 16 Mehrzweckregister in Tabellenform
%FR	alle 4 Gleitpunktregister mit doppelter Genauigkeit in Tabellenform aufbereitet
%PC	Befehlszähler (Program Counter)
%CC	Condition Code
%PM	Program Mask
%AMODE	Adressierungsmodus des Testobjekts; entweder 24 oder 31. Der Adressierungsmodus wird beim Laden des Programms festgelegt.
%PCB	Process Control Block
%PCBLST	Liste aller Process Control Blocks
%AUD1	P1-Audit-Tabelle beginnend mit dem jüngsten Eintrag: nur wenn bei Systemgenerierung angelegt.
%SORTEDMAP	Liste aller CSECTs des Benutzerprogramms (namen- und adresssortiert). Lange Namen werden abgeschnitten.
%MAP [(CTX=kontext)]	Liste der CSECTs und COMMONs aller Kontexte des Benutzerprogramms oder des mit der Kontext-Qualifikation bezeichneten Kontextes; die Namen werden ungekürzt ausgegeben (weitere Operanden siehe AID-Basishandbuch, Abschnitt „Systeminformationen" [1])

```
%LINK                 Name des zuletzt nachgeladenen Segments
                      (siehe %ON, Ereignis %LPOV)
%HLLOC(speicherref)   Lokalisierungsinformation auf symbolischer Ebene
                      für eine Speicherreferenz im ausführbaren Teil
                      des Programms (High-Level-Location)
%LOC(speicherref)     Lokalisierungsinformation auf Maschinencode-Ebene
                      für eine Speicherreferenz im ausführbaren Teil
                      des Programms (Low-Level-Location)
%•[subkdoname]        Durchlaufzähler
%•                    Durchlaufzähler des gerade aktiven Subkommandos
```

kompl-speicherref

Folgende Operationen können darin vorkommen (siehe Basishandbuch, Abschnitt „Komplexe Speicherreferenzen“ [1]):

- Adressversatz (•)
- indirekte Adressierung (->)
- Typmodifikation (%X, %C, %E, %D, %F, %A, %S, %SX)
- Längenmodifikation (%L(...), %L=(ausdruck), %Ln)
- Adressselektion (%@(...))

Soll ein Anweisungsname oder eine Source-Referenz als Speicherreferenz verwendet werden, muss anschließend der Pointer-Operator (->) folgen. Sie sprechen damit den ersten Maschinenbefehl des Prologs an. Ohne den Pointer-Operator können Anweisungsname und Source-Referenz überall da verwendet werden, wo auch Sedezimalzahlen geschrieben werden können.
Mit der Typmodifikation können Sie sich Daten in einer anderen Aufbereitung ausgeben lassen (siehe Basishandbuch, Abschnitt „Typmodifikation“ [1]).

Vor einer Verwechslung der AID-Ausgabetypen mit den Formatangaben bei `printf` wird gewarnt:

AID-Ausgabetyp		entsprechende Formatangabe bei printf	
%C[l-mod]	char	%c	ein einzelnes char-Zeichen
		%s	Character-String
%D[l-mod]	float	%f	float, double
%F[l-mod]	signed int	%d	signed int
%A[l-mod]	unsigned int	%u	unsigned int
%X[l-mod]	sedezimal	%x, %X	sedezimal

Mit der Längenmodifikation können Sie die Ausgabelänge selbst bestimmen, z.B. wenn Sie nur Teile einer Variablen oder eine Variable in der Länge einer anderen Variablen ausgeben lassen wollen. Mit Typ- oder Längenmodifikation dürfen die impliziten Bereichsgrenzen einer Adresse nur überschritten werden, wenn Sie mit %@(...)-> auf die Maschinencode-Ebene gewechselt haben, auf der der Bereich den vom geladenen Programm belegten virtuellen Speicher umfasst.

& ist der Adressoperator. Sie können sich damit die Anfangsadresse eines Datums, eines Objekts einer Klasse oder einer Funktion anzeigen lassen.

Außerdem besteht die Möglichkeit, die relative Adresse eines dynamischen Daten-Members einer Klasse ausgeben zu lassen. Dabei ist Folgendes zu beachten:

Wenn die Unterbrechungsstelle außerhalb der Klasse liegt, die das Daten-Member enthält, schreiben Sie nach dem Adressoperator die entsprechende Klassenqualifikation und dann den Namen des Datums. Liegt die Unterbrechungsstelle dagegen in einer dynamischen Member-Funktion der Klasse, müssen Sie vor den Adressoperator eine Basis- oder Bereichsqualifikation (S-, PROC- oder ::-Qualifikation) setzen, damit AID von außen auf die Klasse zugreift.

Der Adressoperator ist im Gegensatz zum Adressselektor %@(...) (siehe Seite 165) eine reine High-Level-Funktion; daher kann er auf komplexe Speicherreferenzen nicht angewandt werden.

Weitergehende Informationen zum Adressoperator finden Sie im Abschnitt „Adressoperator & und Adressselektor %@(...)“ auf Seite 44.

sizeof()

ist der Längenoperator. Die Länge eines Datums oder einer Klasse wird ausgegeben.
Um die Länge einer Klasse zu ermitteln, können Sie sowohl den Namen der Klasse selbst als auch den Namen eines Objekts der Klasse als Operanden angeben. Sie erhalten die Anzahl Bytes, die die dynamischen Daten-Member der Klasse und eventuell vom Compiler generierte Hilfsvariablen belegen.
Den Namen eines Namespaces können Sie hier nur im Pfad zu einer Komponente des Namespaces angegeben.
Bitfelder und Registervariablen können Sie nicht angeben.

Ausführlich beschrieben ist der Längenoperator im Abschnitt „Längenoperator sizeof() und Längenselektor %L(...)“ auf Seite 49.

%@(...)

Mit dem Adressselektor (siehe Basishandbuch, Abschnitt „Adress-, Typ- und Längenselektor“ [1]) können Sie sich die Anfangsadresse eines Datums, des Objekts einer Klasse oder einer komplexen Speicherreferenz ausgeben lassen. Einen Klassennamen können Sie nur im Pfad zur Basisklasse eines Objekts einer abgeleiteten Klasse angeben, um sich die Anfangsadresse der dynamischen Daten-Member der Basisklasse ausgeben zu lassen.
Den Namen eines Namespaces können Sie hier nur im Pfad zu einer Komponente des Namespaces angegeben.
Der Adressselektor lässt sich nicht auf Konstanten anwenden, dazu gehören auch die Marken, die Source-Referenzen und alle Funktionen.

%L(...)

Mit dem Längenselektor (siehe Basishandbuch, Abschnitt „Komplexe Speicherreferenzen“ [1]) können Sie sich die Länge eines Datums oder einer Klasse ausgeben lassen. Wenn Sie den Längenselektor auf eine Klasse oder ein Objekt einer Klasse anwenden, entspricht das Ergebnis dem von sizeof() in C++; Sie erhalten also die Länge der dynamischen Daten-Member und eventuell vom Compiler generierter Hilfsvariablen.
Den Namen eines Namespaces können Sie hier nur im Pfad zu einer Komponente des Namespaces angegeben.
AID gibt immer die Länge in Bytes aus. Für Bitfelder gibt AID die Anzahl der Bytes aus, über die sich das Bitfeld erstreckt.
Beispiel: `%l(var1)`
Die Länge von `var1` wird ausgegeben.

%L=(ausdruck)

Mit der Längenfunktion können Sie einen Wert errechnen lassen.
ausdruck wird gebildet aus Speicherreferenzen und arithmetischen Operatoren (siehe Basishandbuch, Kapitel „Adressierung in AID“ [1]).

AID verwendet für seine Berechnungen die Standard-Operatoren und vollzieht das Überladen von Operatoren nicht nach.

Beispiel: `%l=(var1)`
Wenn `var1` vom Typ int ist (Typ %F), wird der Inhalt von `var1` ausgegeben. Andernfalls gibt AID eine Fehlermeldung aus.

literal

Alle AID-Literale, die im Basishandbuch, Kapitel „AID-Literale“ [1] beschrieben sind, können Sie im %DISPLAY verwenden:

```
{C'x...x' | 'x...x'C | 'x...x'}   Character-Literal
{X'f...f' | 'f...f'X}             Sedezimal-Literal
{B'b...b' | 'b...b'B}             Binär-Literal
[{±}]n                            Ganzzahl
#'f...f'                          Sedezimalzahl
[{±}]n.m                          Dezimalpunktzahl
[{±}]mantisseE[{±}]exponent       Gleitpunktzahl
```

Ist `%AID C=YES` gesetzt, können Sie auch ein C-String-Literal ("*x...x*") angeben (siehe auch Seite 37).

vorschubsteuerung

Für das Ausgabemedium SYSLST kann die Druckaufbereitung durch die folgenden beiden Schlüsselwörter gesteuert werden:

%NP bewirkt einen Seitenvorschub

%NL[(n)] bewirkt einen Zeilenvorschub um n Leerzeilen. $1 \leq n \leq 255$. Standardwert für n ist 1.

medium-u-menge

legt fest, über welches oder über welche Medien die Ausgabe erfolgen soll und ob AID Zusatzinformationen ausgeben soll. Ohne diesen Operanden und ohne eine Vereinbarung mit dem %OUT-Kommando arbeitet AID mit der Voreinstellung T = MAX.

```
medium-u-menge-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - -

{T  }   {MIN }
{H  } = {MAX }
{Fn }   {XMAX}
{P  }   {XFLAT}

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

medium-u-menge ist ausführlich im Basishandbuch, Kapitel „Operand Medium-und-Menge“ [1] beschrieben.

T Terminal-Ausgabe
H Hardcopy-Ausgabe (schließt die Terminal-Ausgabe mit ein und kann nicht gemeinsam mit T angegeben werden)
Fn Datei-Ausgabe
P Ausgabe nach SYSLST

MAX Ausgabe mit Zusatzinformationen.

MIN Ausgabe ohne Zusatzinformationen.

XMAX Im Kommando %DISPLAY wird der Operandenwert XMAX derzeit nicht berücksichtigt, so dass das Verhalten identisch zum Standardwert MAX ist.

XFLAT Im Kommando %DISPLAY wird der Operandenwert XFLAT derzeit nicht berücksichtigt, so dass das Verhalten identisch zum Standardwert MAX ist.

Beispiele

1. ```
   %df d1=dump.test1
   %base e=d1
   %display s=n'test1.c'.int_var,'dump-inhalt'
   ```

   Es handelt sich hier um die Auswertung eines Dumps. Zusätzlich zum Inhalt von `int_var` gibt AID eine Kopfzeile mit dem Namen der Dump-Datei aus.

   ```
   ** D1: DUMP.TEST1 ******************************************************
   int_var     =            -53
   dump-inhalt
   ```

2. ```
   %display %l=(s'13'-s'12')
   ```

   AID gibt die Länge der Maschinencode-Sequenz aus, die für die Anweisung in der Zeile 12 erzeugt wurde.

   ```
           +52
   ```

3. ```
   %base
   %display scanf
   ```

   Mit %BASE wird zurückgeschaltet auf den AID-Standard-Arbeitsbereich. AID gibt zunächst zwei Kopfzeilen aus, die TID und TSN enthalten sowie die Source-Referenz, an der der Programmablauf unterbrochen wurde. Danach gibt AID die Adresse des ersten Befehls der Funktion `scanf` als Sedezimalzahl aus.

   ```
   *** TID: 00010266 *** TSN: 069R ****************************************
   SRC_REF: 6 SOURCE: EXAMP.C   PROC: main  ********************************
   scanf           = 01001B94
   ```
   ```

4. `%display scanf->`

   AID gibt 4 Bytes des Maschinencodes aus, der ab der Adresse der Funktion `scanf` steht. Der Pointer-Operator bewirkt den Übergang zur Maschinencode-Ebene, so dass AID zusätzlich eine entsprechende Kopfzeile ausgibt.

```
CURRENT PC: 01000098    CSECT: EXAMP$O&@  ******************************
V'01001B94' = IC@PCON  + #'00000634'
01001B94 (00000634) 58F0FF70                              .0~.
```

5. `%display var.4`
   `%display var.(z)`

   Im ersten Fall zählt AID zur Anfangsadresse der Variablen `var` vier Bytes dazu und gibt ab dieser Stelle vier Bytes im Dump-Format aus. Auch im zweiten Fall führt AID einen Adressversatz durch, da `z` in Klammern geschrieben ist. AID rechnet zur Adresse von `var` den Inhalt von `z` dazu und gibt wie oben ab der errechneten Adresse vier Bytes aus. In C/C++ hingegen ist `var.(z)` und `var.z` gleichbedeutend; die Klammern des zweiten Operanden sind für C/C++ redundant. Es könnte also die Strukturkomponente `z` der Struktur `var` gemeint sein. Bei AID darf jedoch bei einer Strukturqualifikation die letzte Komponente nie in Klammern angegeben werden.

6.
```
/LOAD-PROG *M(MYLIB,EXAMP,...),TEST-OPT = AID
%  BLS0523 ELEMENT 'EXAMP', VERSION '@' FROM LIBRARY 'MYLIB' IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$EXAMP$', VERSION ' ' OF '1999-01-07
11:47:57' LOADED
/%r
%  IDA0N51 PROGRAM INTERRUPT AT LOCATION '010000BC (EXAMP$O&), (CDUMP),
EC=58'
%  IDA0N45 DUMP DESIRED? REPLY (Y=USER/AREA DUMP; Y,SYSTEM=SYSTEM DUMP;
N = NO)? N
%  EXC0077 PROGRAM STILL LOADED AND IN 'COMMAND-MODE'. PROGRAM RUN MAY
BE CONTINUED WITH /RESUME-PROGRAM
```

In Ihrem Programm ist ein Fehler aufgetreten. Nun interessiert es Sie, welche Anweisung den Fehler verursacht hat. Dazu geben Sie `%DISPLAY %HLLOC` zu der Adresse ein, an der das Programm durch den Fehler unterbrochen wurde. Diese Adresse ist im Befehlszähler (%PC) enthalten. Weitere Informationen erhalten Sie mit `%DISPLAY %LOC`.

```
/%display %hlloc(%pc->)
*** TID: 00010266 *** TSN: 069R ****************************************
CURRENT PC: 010000BC    CSECT: EXAMP$O&@   ******************************
V'010000BC' = CONTEXT  :  LOCAL#DEFAULT
              SMOD     :  EXAMP.C
              BLOCK    :  *root*
              PROC     :  main
              SRC-REF  :  11
/%display %loc(%pc->)
V'010000BC' = CONTEXT:LOCAL#DEFAULT
              LMOD :  %UNIT
              SMOD :  EXAMP.C
              OMOD :  EXAMP$O&@
              CSECT : EXAMP$O&@ (01000000) + 000000BC (/390)
```

7. `%d abc_arr`

   Der Vektor `abc_arr` enthält 27 Elemente vom Typ `char` und ist folgendermaßen definiert:
   `char abc_arr[27] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";`

   Da im %DISPLAY kein Index angegeben wurde, gibt AID den gesamten Vektor aus:

```
abc_arr( 0: 26)
(  0) |A|  (  1) |B|  (  2) |C|  (  3) |D|  (  4) |E|  (  5) |F|  (  6) |G|
(  7) |H|  (  8) |I|  (  9) |J|  ( 10) |K|  ( 11) |L|  ( 12) |M|  ( 13) |N|
( 14) |O|  ( 15) |P|  ( 16) |Q|  ( 17) |R|  ( 18) |S|  ( 19) |T|  ( 20) |U|
( 21) |V|  ( 22) |W|  ( 23) |X|  ( 24) |Y|  ( 25) |Z|  ( 26) |.|
```

   Ein ausführliches Beispiel zur Bearbeitung von Vektoren mit AID finden Sie auf Seite 34.

8. `%d abc_arr[n]`

   `abc_arr` ist definiert, wie in Beispiel 7 beschrieben. `n` enthält den Wert 4.
   Es wird das 5. Element des Vektors ausgegeben:

```
abc_arr( 4)     = ?E?
```

9. Der folgende Programmausschnitt zeigt die Definition einer abgeleiteten Klasse `B` mit Basisklasse `A`. Die Klasse enthält zwei virtuelle Funktionen `foo1(void)` und `foo2(void)`.

```
C++-Programm                                                    BCL1.C
================================================================================
SRC
LIN
  1  class A
  2  {
  3    public:
  4    A() { printf ("A::A called\n"); }
  5    virtual void foo1() { printf( "A::foo1 called\n" ); }
  6    virtual void foo2() { printf( "A::foo2 called\n" ); }
  7  } a;
  8
  9  class B : public A
 10  {
 11    int i;
 12    public:
 13    B(int x = 1) : i(x) { printf ("B::B called\n"); }
 14    void foo1() { printf( "B::foo1 called\n" ); }
 15    void foo2() { printf( "B::foo2 called\n" ); }
 16  } b;
 ...


%in s'13'; %r
%d this, *this
```

Durch den %INSERT mit nachfolgendem %RESUME wird der Programmablauf im Konstruktor für Klasse `B` an der Source-Referenz 13 unterbrochen. Das %DISPLAY-Kommando gibt den Inhalt des Zeigers `this` aus, also die Adresse des zugehörigen Objekts und mit der Dereferenzierung (`*this`) den Inhalt des Objekts.

```
SRC_REF: 13 SOURCE: BCL1.C PROC: B::B(int)  ****************************
this             = 01001138

01          *
 02         A
  03        A()             = 01000000
  03        foo1()          = 010004C0
  03        foo2()          = 010005E0
 02         i               =          1
 02         B(int)          = 01000360
 02         foo1()          = 010004C0
 02         foo2()          = 010005E0
```

10. Das C-Programm `OUTPUT.C` gibt einige einfache unstrukturierte Datentypen aus, die in C definiert werden können.

```
*** SOURCE - ERROR - LISTING ** BS2000 C/C++ COMPILER 03.2E21 DATE:2015-02-27     PAGE:    1
                    SOURCENAME:*LIB-ELEM(MYLIB,OUTPUT.C(*HIGHEST-EXISTING),S)
_____________________________________________________________________________________________

  EXP  INC  FILE    SRC BLOCK
  LIN  LEV   NO     LIN  LEV

    1    0     0      1    0  #include <stdio.h>
 1746    0     0      2    0  int main(void)
 1747    0     0      3    0  {
 1748    0     0      4    1
 1749    0     0      5    1     short int1 = -32768;
 1750    0     0      6    1     int   int2 = 234;
 1751    0     0      7    1     long  int3 = -567;
 1752    0     0      8    1
 1753    0     0      9    1     unsigned short un1 = 65535;
 1754    0     0     10    1     unsigned int   un2 = 78900;
 1755    0     0     11    1     unsigned long  un3 = 90123;
 1756    0     0     12    1
 1757    0     0     13    1     signed long long sll   = -9223372036854775808;
 1758    0     0     14    1     unsigned long long ull = 18446744073709551615;
 1759    0     0     15    1
 1760    0     0     16    1     float       fl1 = 123.456;
 1761    0     0     17    1     double      fl2 = 567.89;
 1762    0     0     18    1     long double fl3 = 333.444;
 1763    0     0     19    1
 1764    0     0     20    1     char        char1 = 'A';
 1765    0     0     21    1     signed char char2 = -63;
 1766    0     0     22    1
 1767    0     0     23    1     char *chstr     = "Character-String";
 1768    0     0     24    1     char chvek[17]  = "Character-Vektor";
 1769    0     0     25    1
 1770    0     0     26    1     char *c_aus  = "Entsprechende C-Ausgabe:";
 1771    0     0     27    1
 1772    0     0     28    1     printf ("%s\n",c_aus);
 1773    0     0     29    1     printf ("int1 = %d\n", int1);
 1774    0     0     30    1     printf ("int2 = %d\n", int2);
 1775    0     0     31    1     printf ("int3 = %d\n", int3);
 1776    0     0     32    1
 1777    0     0     33    1     printf ("%s\n",c_aus);
 1778    0     0     34    1     printf ("un1 = %u\n", un1);
 1779    0     0     35    1     printf ("un2 = %u\n", un2);
 1780    0     0     36    1     printf ("un3 = %u\n", un3);
 1781    0     0     37    1
 1782    0     0     38    1     printf ("%s\n",c_aus);
 1783    0     0     39    1     printf ("sll = %lld\n", sll);
 1784    0     0     40    1     printf ("ull = %llu\n", ull);
 1785    0     0     41    1
 1786    0     0     42    1     printf ("%s\n",c_aus);
 1787    0     0     43    1     printf ("fl1 = %f\n", fl1);
 1788    0     0     44    1     printf ("fl2 = %f\n", fl2);
 1789    0     0     45    1     printf ("fl3 = %f\n", fl3);
 1790    0     0     46    1
 1791    0     0     47    1     printf ("%s\n",c_aus);
 1792    0     0     48    1     printf ("char1 als Zeichen = %c und als Wert = %d\n",
 1793    0     0     49    1                                    char1, char1);
 1794    0     0     50    1     printf ("char2 als Zeichen = %c und als Wert = %d\n",
 1795    0     0     51    1                                    char2, char2);
 1796    0     0     52    1     printf ("%s\n",c_aus);
 1797    0     0     53    1     printf ("*chstr = %s\n", chstr);
 1798    0     0     54    1     printf ("chvek = %s\n", chvek);
 1799    0     0     55    1
 1800    0     0     56    1     return 0;
 1801    0     0     57    1  }
```

```
/START-CPLUS-COMPILER
%  BLS0523 ELEMENT 'SDFCC', VERSION '03.2E21', TYPE 'L' FROM LIBRARY
        ':2OSH:$TSOS.SYSLNK.CPP.032' IN PROCESS
%  BLS0524 LLM 'SDFCC', VERSION '03.2E21' OF '2015-02-24 07:17:56' LOADED
%  BLS0551 COPYRIGHT (C) 2015 Fujitsu Technology Solutions GmbH.
        ALL RIGHTS RESERVED
%  CDR9992 : BEGIN C/C++ VERSION 03.2E21
//MODIFY-TEST-PROPERTIES TEST-SUPPORT=YES
//MODIFY-SOURCE-PROPERTIES LANGUAGE=*C(*ANSI)
...
//MODIFY-BIND-PROPERTIES ...  RUNTIME-LANGUAGE = *C,  TEST-SUPPORT = *YES
...
//END
...
%  BLS0524 LLM '$LIB-ELEM$MYLIB$OUTPUT', VERSION ' ' OF '2015-03-05 11:15:23' LOADED
/%aid c=yes
/%in s'28' <%d 'AID-AUSGABE int1, int2 und int3', int1, int2, int3>
/%in s'33' <%d 'AID-AUSGABE un1, un2 und un3', un1, un2, un3>
/%in s'38' <%d 'AID-AUSGABE sll und ull', sll, ull>
/%in s'42' <%d 'AID-AUSGABE fl1, fl2 und fl3', fl1, fl2, fl3>
/%in s'47' <%d 'AID-AUSGABE char2 als Zeichen und als Wert', char2%c, char2>
/%in s'47' <%d 'AID-AUSGABE char1 als Zeichen und als Wert', char1, char1%a>
/%in s'52' <%d 'AID-AUSGABE *chstr und chvek', chstr->%cl16, chvek>
/%resume
```

Programm `OUTPUT.C` wurde fehlerfrei übersetzt und mit LSD-Sätzen gebunden und geladen. `%AID C=YES` wurde gesetzt, damit AID den `char`-Vektor `chvek` als C-String ausgibt. Gleichzeitig wurde damit die Unterscheidung der Groß-/Kleinschreibung eingeschaltet.
Mit den %INSERT-Kommandos wurden Testpunkte gesetzt, so dass jeweils auf ein %DISPLAY-Kommando die entsprechende C-Ausgabe folgt. Zur besseren Lesbarkeit sind die Textzeilen `"AID-AUSGABE"` und `"ENTSPRECHENDE C-AUSGABE"` fettgedruckt.

```
AID-AUSGABE int1, int2 und int3
*** TID: 000E01D8 *** TSN: 7313 ****************************************************
SRC_REF: 28 SOURCE: OUTPUT.C  PROC: main  ******************************************
int1             =      -32768
int2             =         234
int3             =        -567
Entsprechende C-Ausgabe:
int1 = -32768
int2 = 234
int3 = -567

AID-AUSGABE un1, un2 und un3
SRC_REF: 33 SOURCE: OUTPUT.C  PROC: main  ******************************************
un1              =    65535
un2              =      78900
un3              =      90123
Entsprechende C-Ausgabe:
un1 = 65535
un2 = 78900
un3 = 90123

AID-AUSGABE sll und ull
SRC_REF: 38 SOURCE: OUTPUT.C  PROC: main  ******************************************
sll              =   -9223372036854775808
ull              =   18446744073709551615
Entsprechende C-Ausgabe:
sll = -9223372036854775808
ull = 18446744073709551615
```

Alle Variablen vom Typ `signed` und `unsigned int` wurden ausgegeben. Mit `printf` können Sie Daten des Typs `long long` in CRTE ausgeben.

```
AID-AUSGABE fl1, fl2 und fl3
SRC_REF: 42 SOURCE: OUTPUT.C  PROC: main  ******************************************
fl1              = +.1234559 E+003
fl2              = +.5678899999999999 E+003
fl3              = +.3334439999999999 E+003
Entsprechende C-Ausgabe:
fl1 = 123.455994
fl2 = 567.890000
fl3 = 333.444000
```

Daten vom Typ `float`, `double` und `long double` wurden ausgegeben. AID gibt Gleitpunkt-Variablen stets in Exponentendarstellung aus. Zu Variablen mit einfacher Genauigkeit gibt AID sieben signifikante Stellen aus, zu `double` und `long double` Datentypen werden jeweils 16 Stellen ausgegeben.

```
AID-AUSGABE char1 als Zeichen und als Wert
SRC_REF: 47 SOURCE: OUTPUT.C  PROC: main  ****************************************
char1           = |A|
CURRENT PC: 010002D4    CSECT: OUTPUT$O&@  ****************************************
V'0104A7E8' = char1    + #'00000000'
0104A7E8 (00000000) 193
AID-AUSGABE char2 als Zeichen und als Wert
V'0104A7E9' = char2    + #'00000000'
0104A7E9 (00000000) A
SRC_REF: 47 SOURCE: OUTPUT.C  PROC: main  ****************************************
char2           =  -63
Entsprechende C-Ausgabe:
char1 als Zeichen = A und als Wert = 193
char2 als Zeichen = A und als Wert = -63
```

AID behandelt den Datentyp `char` anders als `signed char`. Zur Variablen `char1` gibt AID den Character-Wert, nämlich A, aus. Den entsprechenden Dezimalwert erhalten Sie nur über die explizite Typmodifikation %A. Zur `signed char`-Variablen `char2` gibt AID ohne Typmodifikation den Dezimalwert, nämlich -63, aus. Um auch `char2` als Character darzustellen, müssen Sie die explizite Typmodifikation %C verwenden.

```
AID-AUSGABE *chstr und chvek
CURRENT PC: 0100035C    CSECT: OUTPUT$O&@  ****************************************
V'01001188 = OUTPUT$O&#                          + #'00000118'
01001188 (00000188) Character-String
SRC_REF: 52 SOURCE: OUTPUT.C  PROC: main  ****************************************
chvek = "Character-Vektor"
Entsprechende C-Ausgabe:
*chstr = Character-String
chvek = Character-Vektor
```

Die Ausgabe von Character-Strings - im ersten Fall über einen Zeiger angesprochen und im zweiten Fall als Vektor von Zeichen abgelegt - schließt die Gegenüberstellung der Aufbereitung der einzelnen Datentypen bei AID und C ab.
Der String, auf den der Zeiger `chstr` verweist, kann mit AID nur über einen anschließenden Pointer-Operator mit Typ- und Längenmodifikation ausgegeben werden. Der Vektor `chvek` wird dagegen in Form eines C-String-Literals aufbereitet, weil `%AID C=YES` gesetzt wurde.

11. Zeichenausgabe mit beliebigen Coded Character Sets (CCS)

```
void main(void)
{
    char unsigned data[256];
    char ALPHA[28];
    int i;

    for (i=1; i<=255; i++)
                     data[i-1] = i;
    data[255] = 0x00;
    strncpy(ALPHA, data+64, 26);
    ALPHA[26]= 0x00;
```

```
        STOP: ;
    }
```

Terminal-Zeichensatz einstellen:

```
/mod-term-opt coded-character-set=edf041
```

Nach dem Laden des Programms:

```
%AID C=YES
%INSERT STOP
%RESUME
```

Datenausgabe mit der Standardinterpretation `*USRDEF` (=EDF03IRV):

```
/%D data.32 %XL80
V'0100112C' = data     + #'00000020'
0100112C (00000020) 21222324 25262728 292A2B2C 2D2E2F30    ................
0100113C (00000030) 31323334 35363738 393A3B3C 3D3E3F40    ................
0100114C (00000040) 41424344 45464748 494A4B4C 4D4E4F50    .........`.<(+|&
0100115C (00000050) 51525354 55565758 595A5B5C 5D5E5F60    .........!$*);.-
0100116C (00000060) 61626364 65666768 696A6B6C 6D6E6F70    /........^,%_>?.
```

Den aktuellen Zeichensatz auf ISO88591ändern:

```
%AID CCS=ISO88591
```

Datenausgabe mit der definierten ISO88591-Interpretation:

```
/%D data.32 %XL80
V'0100112C' = data     + #'00000020'
0100112C (00000020) 21222324 25262728 292A2B2C 2D2E2F30    !"#$%&'()*+,-./0
0100113C (00000030) 31323334 35363738 393A3B3C 3D3E3F40    123456789:;<=>?@
0100114C (00000040) 41424344 45464748 494A4B4C 4D4E4F50    ABCDEFGHIJKLMNOP
0100115C (00000050) 51525354 55565758 595A5B5C 5D5E5F60    QRSTUVWXYZ[\]^_`
0100116C (00000060) 61626364 65666768 696A6B6C 6D6E6F70    abcdefghijklmnop
```

Ausgabe der char-Variable ALPHA (ISO88591-Interpretation):

```
/%D ALPHA
ALPHA           =         "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

%DUMPFILE

Mit %DUMPFILE weisen Sie einem der Linknamen eine Dump-Datei zu und veranlassen AID, diese Datei zu öffnen oder zu schließen.

– Mit *link* wählen Sie den Linknamen für die Dump-Datei aus, die geöffnet oder geschlossen werden soll.

– Mit *datei* bezeichnen Sie die Dump-Datei, die geöffnet werden soll.

Kommando	Operand
%DUMPFILE %DF	[link [=datei]]

Ohne den *datei*-Operanden veranlassen Sie AID, die Datei zu schließen, die dem angegebenen Linknamen zugewiesen ist.

Mit einem %DUMPFILE ohne Operanden veranlassen Sie AID, alle offenen Dump-Dateien zu schließen. Lag bis dahin der AID-Arbeitsbereich in der nun geschlossenen Dump-Datei, gilt anschließend wieder der AID-Standard-Arbeitsbereich (siehe Beschreibung des Kommandos %BASE).

War zu einer mit %DUMPFILE geschlossenen Datei eine Bibliothek zum Nachladen von LSD mit dem Kommando %SYMLIB angemeldet, so wird die Bibliothek für den zugehörigen Linknamen *Dn* abgemeldet.

%DUMPFILE darf nur als Einzelkommando eingegeben werden, es darf nicht in einer Kommandofolge oder in einem Subkommando stehen.

%DUMPFILE verändert den Programmzustand nicht.

link

bezeichnet einen der AID-Linknamen für Dump-Dateien und hat das Format Dn, wobei n eine Zahl ist mit einem Wert $0 \leq n \leq 7$.

datei

gibt den vollqualifizierten Dateinamen an, unter dem die Dump-Datei katalogisiert ist, die AID öffnen soll.
Ohne diesen Operanden wird die Dump-Datei mit dem Linknamen *link* geschlossen.
Eine offene Dump-Datei muss erst mit einem eigenen %DUMPFILE geschlossen worden sein, bevor eine andere demselben Linknamen zugewiesen werden kann.

Beispiele

1. `%dumpfile d3=dump.1234.00001`

   Die Datei `DUMP.1234.00001` mit dem Linknamen `D3` wird geöffnet.

2. `%df d3`

   Die Datei, die dem Linknamen `D3` zugewiesen ist, wird geschlossen.

3. `%df`

   Alle offenen Dump-Dateien werden geschlossen.

%FIND

Mit %FIND können Sie ein Literal im Datenteil oder im ausführbaren Teil eines Programms suchen und Treffer auf Terminal (SYSOUT) ausgeben lassen. Außerdem werden in den AID-Registern %0G und %1G Trefferadresse und Fortsetzungsadresse abgelegt. Mit %FIND können Sie sowohl im virtuellen Speicher als auch in einer Dump-Datei suchen.

– *suchbegriff* ist ein Character- oder Sedezimal-Literal, das gesucht werden soll.

– Mit *ALL* geben Sie an, dass die Suche nicht nach der Ausgabe des ersten Treffers abgebrochen werden soll, sondern dass der gesamte *find-bereich* durchsucht und alle Treffer ausgegeben werden sollen. Die Suche kann dann nur mit der K2-Taste abgebrochen werden.

– Mit *find-bereich* geben Sie an, in welchem Datum oder in welchem Bereich des ausführbaren Programmteils AID *suchbegriff* suchen soll. Ohne Angabe von *find-bereich* durchsucht AID den gesamten Speicherbereich zur aktuell eingestellten Basisqualifikation (siehe Beschreibung des Kommandos %BASE).

– Mit *alignment* geben Sie an, ob *suchbegriff* an Doppelwort-, Wort-, Halbwort- oder Byte-Grenze gesucht werden soll. Ohne Angabe von *alignment* wird an Byte-Grenze gesucht.

Kommando	Operanden
%F[IND]	[[ALL] suchbegriff [IN find-bereich] [alignment]]

Ein %FIND ohne *ALL*-Operanden können Sie hinter der Adresse des letzten Treffers fortsetzen, bis das Ende von *find-bereich* erreicht ist, indem Sie ein neues %FIND-Kommando ohne eigene Operandenwerte eingeben.

In einem %FIND mit eigenem *suchbegriff* und ohne weitere Operanden setzt AID für *find-bereich* und *alignment* die entsprechenden Standardwerte. Hier werden also keine Operanden aus einem vorhergehenden %FIND übernommen.

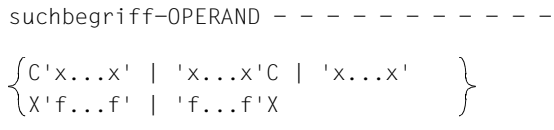
Im Trefferfall erfolgt eine Ausgabe in der Länge von maximal 12 Bytes vom Treffer an bis zum Ende von *find-bereich* auf das Medium Terminal (SYSOUT) im Ausgabetyp DUMP (Sedezimal- und Character-Darstellung). Zusätzlich zum Treffer wird seine Adresse und (soweit möglich) der Name der CSECT, in der der Treffer gefunden wurde, und die relative Adresse des Treffers zum Anfang der CSECT ausgegeben. Bei der Suche in globalen Daten wird die relative Adresse zum Anfang des Datenmoduls ausgegeben. In allen übrigen Fällen wird die absolute Adresse ausgegeben.

Im Trefferfall wird die Trefferadresse im AID-Register %0G und die Fortsetzungsadresse (Trefferadresse + Suchstringlänge) im AID-Register %1G abgespeichert. Bei der Angabe von *ALL* wird die Adresse des letzten Treffers in %0G und die Fortsetzungsadresse des letzten Treffers in %1G abgespeichert. Falls *suchbegriff* nicht gefunden wurde, setzt AID %0G auf –1; %1G bleibt unverändert.
Die beiden Registerinhalte ermöglichen es Ihnen, das %FIND-Kommando auch in Prozeduren oder Subkommandos einzusetzen und mit den Ergebnissen weiterzuarbeiten.

%FIND verändert den Programmzustand nicht.

suchbegriff

ist ein Character- oder Sedezimal-Literal. *suchbegriff* kann Wildcard-Symbole enthalten. Diese Symbole sind immer Treffer. Sie werden durch '%' dargestellt.



`{C'x...x' | 'x...x'C | 'x...x'}`

Character-Literal

mit einer maximalen Länge von 80 Zeichen. Kleinbuchstaben können nur nach Eingabe von %AID LOW[=ON] als Character-Literal gesucht werden.

x kann jedes darstellbare Zeichen annehmen, insbesondere das Wildcard-Symbol '%', welches immer einen Treffer darstellt. Das Zeichen '%' selbst kann in dieser Form nicht gesucht werden, da es als C'%' in einem Character-Literal stets zu einem Treffer führt. Es muss deshalb als Sedezimal-Literal X'6C' gesucht werden.

Bitte beachten Sie, dass der CCS von *find-bereich* mit dem CCS des Eingabemediums (SYSCMD) übereinstimmen muss, damit die Character-Literale gefunden werden können. Legen Sie daher den CCS von *find-bereich* fest, bevor Sie in *find-bereich* nach einem Character-Literal suchen:

`%AID CCS=` *CCS-name*

Eine komplette Liste der von XHCS unterstützten CCS-Namen und den aktuellen CCS von SYSCMD können Sie mit dem folgenden AID-Kommando ausgeben:

```
%SHOW %CCSN
```

Den CCS von SYSCMD können Sie mit dem folgenden SDF-Kommando ändern:

```
MODIFY-TERMINAL-OPTION CODED-CHARACTER-SET= {EBCDIC-CCS-name | UTFE}
```

Den aktuellen CCS von *find-bereich* können Sie mit dem folgenden AID-Kommando ausgeben:

```
%SHOW %AID
```

Beachten Sie bitte, dass das %DISPLAY-Kommando seit der AID-Version V3.4B11 als Voreinstellung den CCS-Wert von `%AID` verwendet, wenn kein CCS-Wert angegeben wurde.

```
%D char-data ['CCS-name']
```

Siehe Basishandbuch, Abschnitt „Character-Literal“ [1]) für ein Beispiel zur Suche nach Character-Literalen in unterschiedlichen Coded Character Sets.

`{X'f...f' | 'f...f'X}`

Sedezimal-Literal

mit einer maximalen Länge von 80 Sedezimal-Stellen bzw. 40 Zeichen. Ein Literal mit ungerader Stellenzahl wird rechts mit X’0’ ergänzt.

f kann jeden Wert zwischen 0 und F sowie das Wildcard-Symbol X’%’ annehmen. Das Wildcard-Symbol stellt für jede Sedezimal-Stelle zwischen 0 und F einen Treffer dar.

find-bereich

legt einen Speicherbereich fest, in dem *suchbegriff* gesucht werden soll. *find-bereich* kann ein Objekt einer Klasse, ein Datum oder ein Bereich im ausführbaren Teil des geladenen Programms oder einer Dump-Datei sein.

Ist kein *find-bereich* angegeben, so setzt AID den Standardwert %CLASS6 ein (siehe Basishandbuch, Abschnitt „Speicherklassen“ [1]), d.h. es wird der Klasse-6-Speicher zur aktuell eingestellten Basisqualifikation (siehe Beschreibung des Kommandos %BASE) durchsucht. Der Standardwert für *find-bereich* kann für C/C++-Programme nicht genutzt werden, da diese Programme in den oberen Adressraum des Speichers geladen werden. Sie müssen daher beim ersten Aufruf des Kommandos %FIND einen expliziten Wert für *find-bereich* angeben (z.B. %CLASS6ABOVE).

- Steht der Punkt an führender Stelle, ist er das Kennzeichen für eine *vorqualifikation*. Sie muss mit einem vorhergehenden %QUALIFY definiert worden sein. Aufeinanderfolgende Qualifikationen werden durch einen Punkt getrennt. Außerdem muss zwischen der letzten Qualifikation und dem anschließenden Operandenteil ein Punkt stehen.

qua Eine oder mehrere Qualifikationen geben Sie an, wenn *find-bereich* von der aktuellen Unterbrechungsstelle aus anders nicht zu erreichen ist oder um einen Datennamen anzusprechen, der an der Unterbrechungsstelle durch eine gleichnamige Definition lokal verdeckt ist. Sie geben nur die Qualifikationen an, die zur eindeutigen Ansprache des Speicherobjekts genügen.

E={VM | Dn}
: geben Sie nur an, wenn für *find-bereich* nicht die aktuelle Basisqualifikation gelten soll (siehe %BASE).

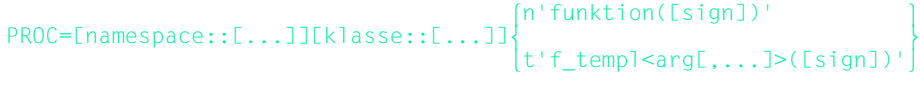
S=srcname
: geben Sie nur an, wenn *find-bereich* nicht in der aktuellen Übersetzungseinheit liegt (siehe Kapitel „Adressierung in C- und C++-Programmen“ auf Seite 21).

:: Die beiden vorangestellten Doppelpunkte verwenden Sie, um ein globales Datum anzusprechen, das durch eine gleichnamige Definition an der Unterbrechungsstelle lokal verdeckt ist. Außerdem müssen Sie die beiden

Doppelpunkte vor den Namen eines globalen Datums oder einer Funktion setzen, weil entweder das Datum oder die Funktion nicht in der Aufrufhierarchie liegen oder weil deren Definition erst nach der Unterbrechungsstelle steht.
Im Gegensatz zu den übrigen Qualifikationen wird zwischen den beiden Doppelpunkten und dem anschließenden Operanden kein Punkt geschrieben.

PROC=funktion

geben Sie nur an, wenn Sie einen Datennamen ansprechen wollen, der zwar in der aktuellen Funktion definiert ist, aber von einer gleichnamigen Definition an der Unterbrechungsstelle verdeckt wird. Außerdem geben Sie eine PROC-Qualifikation an, wenn Sie eine Marke oder einen static vereinbarten Datennamen ansprechen wollen, der einer Funktion außerhalb der aktuellen Aufrufhierarchie zugeordnet ist (siehe Kapitel „Adressierung in C- und C++-Programmen“ auf Seite 21). Wenn die Startadresse von *find-bereich* durch eine Source-Referenz bezeichnet wird, die in einer Instanz eines Funktionstemplates liegt oder die einer Funktion zugeordnet ist, die in einer Instanz eines Klassentemplates definiert ist (siehe Abschnitt „Templates“ auf Seite 98), müssen Sie bei Mehrdeutigkeit ebenfalls die entsprechende PROC-Qualifikation davorschreiben.
Bei Funktionen aus C-Programmen ist *funktion* der im Quellprogramm vergebene Name der Funktion ohne Klammern und ohne Signatur.
Funktionen aus C++-Programmen müssen Sie, je nachdem, um welchen Typ es sich handelt, in der Schreibweise `n'...'` oder `t'...'` angeben. Ist die Funktion in einem Namespace oder in einer Klasse definiert, so wird dem Funktionsnamen die Namespace- oder Klassen-Qualifikation vorangestellt. Die Signatur `void` darf nicht mehr geschrieben werden. Wie auch in C++ möglich geben Sie in diesem Fall nur die beiden Klammern nach dem Funktionsnamen an.
Es ergibt sich die folgende Syntax (*f_template* sowie *signatur* sind in der Syntax aus Platzgründen abgekürzt):



Abweichend davon werden die Funktionen `main` und `_ _STI_ _` sowie alle Funktionen mit C-Linkage auch beim Testen in C++-Programmen nur mit dem Funktionsnamen bezeichnet (siehe Seite 60).

Ist die Funktion in einer lokalen Klasse definiert, so müssen Sie vor der PROC-Qualifikation noch eine weitere PROC-Qualifikation für die übergeordnete Funktion angeben, die die Definition der lokalen Klasse enthält. Zwischen die beiden PROC-Qualifikationen schreiben Sie bei Funktionen, die in einem

inneren Block der übergeordneten Funktion definiert sind, eine oder ggfs. mehrere BLK-Qualifikationen (siehe Seite 62).
Das Ansprechen von Funktionen, die in lokalen Klassen innerer Blöcke definiert sind, wird nur bei Programmen unterstützt, die mit einem C/C++-Compiler ab V3.0B übersetzt wurden.

BLK='[f-]n[:b]'
geben Sie an, wenn Sie einen Datennamen ansprechen wollen, der einem Block innerhalb der aktuellen Aufrufhierarchie zugeordnet ist und von einer gleichnamigen Definition an der Unterbrechungsstelle verdeckt wird oder wenn Sie einen static vereinbarten Datennamen ansprechen wollen, der einem Block außerhalb der aktuellen Aufrufhierarchie zugeordnet ist (siehe Kapitel „Adressierung in C- und C++-Programmen“ auf Seite 21).

Des weiteren müssen Sie eine BLK-Qualifikation angeben, wenn Sie mit einer nachfolgenden PROC-Qualifikation eine Funktion aus einer lokalen Klasse bezeichnen und die Definition dieser lokalen Klasse im angegebenen Block steht (siehe oben PROC=*funktion*).

namespace
ist der im Quellprogramm deklarierte Name eines Namespaces.

Den Namen eines Namespaces geben Sie nur dann an, wenn die gewünschte Komponente des Namespaces an der Unterbrechungsstelle nicht sichtbar ist. Sie beschreiben damit den Adressierungspfad zu Klassen, Daten oder Funktionen, die in dem Namespace definiert sind (siehe Abschnitt „Namespaces“ auf Seite 89).

Vor der Namespace-Qualifikation sind nur die E- oder S-Qualifikation oder die beiden Doppelpunkte (`::`) für den globalen Namespace möglich.

{ klasse | this-> | objekt }
ist der im Quellprogramm deklarierte Name einer Klasse, der `this`-Zeiger oder der Name eines Objekts einer Klasse.
Klassennamen, den `this`-Zeiger mit nachfolgendem Pointer-Operator sowie die Namen von Objekten von Klassen geben Sie an, um den Adressierungspfad zu Daten zu beschreiben, die Klassen zugeordnet sind (siehe Abschnitt „Klassen“ auf Seite 65).

Liegt die aktuelle Unterbrechungsstelle in einer dynamischen Member-Funktion, können Sie die Daten der Klasse gemäß den aus C++ bekannten Scoperegeln ansprechen.

Unabhängig von der Unterbrechungsstelle erreichen Sie die dynamischen Daten eines Objekts über den Objektnamen mit nachfolgendem Punkt, falls das Objekt in der aktuellen Aufrufhierarchie liegt.

Statische Daten-Member können Sie nur einzeln ansprechen. Sie erreichen sie von jeder Stelle des Programms über den zugehörigen Klassennamen mit den beiden nachfolgenden Doppelpunkten. Bei geschachtelten Klassen enthält der Pfad zum Datum alle Klassennamen von außen nach innen, jeweils durch zwei Doppelpunkte getrennt. Der äußerste Klassennamen benötigt die dem Gültigkeitsbereich entsprechende Qualifikation. Ist das Programm in einer Member-Funktion der Klasse unterbrochen, so gelten für das Ansprechen von statischen Daten-Member die Regeln des Klassenscopes, d.h. wenn das Daten-Member nicht durch eine gleichnamige Defintion verdeckt ist, so kann es unqualifiziert angesprochen werden.

Handelt es sich bei der Klasse um eine Instanz eines Klassentemplates, müssen Sie die folgende Schreibweise verwenden: `t'`*k_template*`<`*arg*`[,...]>'`. Existiert nur eine einzige Instanz des Templates, so genügt die Angabe: `t'`*k_template*`'`.

Endet *find-bereich* mit einem Objektnamen, so bezeichnen Sie damit, unabhängig von der aktuellen Unterbrechungsstelle, die dynamischen Daten-Member und, falls vorhanden, vom Compiler generierte Hilfsvariablen und die Adresse der Tabelle der virtuellen Funktionen. Bei abgeleiteten Klassen umfasst *find-bereich* auch die Basisklassen. Den gleichen Bereich bezeichnen Sie mit `*this`, falls das Programm in einer dynamischen Member-Funktion der Klasse unterbrochen ist. Um in einer abgeleiteten Klasse eine Basisklasse zu bezeichnen, geben Sie ausgehend vom Objektnamen bzw. von `this->` im Pfad den Namen der gewünschten Basisklasse an.
Falls die Unterbrechungsstelle nicht im Gültigkeitsbereich von *objekt* liegt, müssen Sie entsprechend qualifizieren. Vor `*this` ist nur eine Basisqualifikation sinnvoll. Endet der *find-bereich*-Operand auf `this->`, so bezeichnen Sie damit 4 Bytes ab der Anfangsadresse des aktuellen Objekts.

datenname

ist der im Quellprogramm definierte Name eines Datums. *datenname* wird wie im Quellprogramm angegeben.

Daten können Sie bis auf folgende Ausnahmen wie in C/C++ ansprechen:
Mit einem Vektornamen ohne Index sprechen Sie alle Vektorelemente an.
Vektorelemente können Sie nur über Index ansprechen, nicht über Zeiger; zum Arbeiten mit Vektoren siehe auch Abschnitt „Indexschreibweise“ auf Seite 31.

datenname kann folgendermaßen angegeben werden. Die Formate können auch kombiniert werden (siehe Abschnitt „Datennamen“ auf Seite 30):

Indexschreibweise:	*datenname* `[`*index*`]` `{...}`
Zeigerschreibweise:	*datenname1* `->` *datenname2*
Strukturqualifizierung:	*übergeordneter datenname*`.` `{...}` *datenname*
Dereferenzierung:	`[(]*{...}` *datenname*`[)]`

```
{funktion[%al4]  }
{L'label'        }->
{S'[f-]n[:a]'    }
```

bezeichnet 4 Bytes des Maschinencodes ab der Adresse, die in einer der Adresskonstanten hinterlegt ist. Soll eine andere Anzahl von Bytes durchsucht werden, müssen Sie eine entsprechende Längenmodifikation angeben.

funktion

ist der Name einer Funktion, der im Quellprogramm vergeben wurde bzw. der Name einer Bibliotheksfunktion. Sie bezeichnen damit den ersten Befehl des vom Compiler erzeugten Prologs einer Funktion (siehe PROC=*funktion* auf Seite 183 und Kapitel „Adressierung in C- und C++-Programmen“ auf Seite 21).

Virtuelle Funktionen sprechen Sie mit der folgenden Syntax an:
`p->n'funktion([signatur])'`

p ist eine Zeigervariable, die auf das Objekt einer Klasse verweist, das die gewünschte Member-Funktion enthält. Wenn *p* von der aktuellen Unterbrechungsstelle aus nicht zu erreichen ist, muss dem Gültigkeitsbereich entsprechend qualifiziert werden. Liegt die Unterbrechungsstelle in der virtuellen Funktion selbst, dann können Sie die Prologadresse der aktuellen Funktion ansprechen, indem Sie statt *p* den `this`-Zeiger einsetze*n* (siehe Beschreibung von `this` auf Seite 66 und Abschnitt „Virtuelle Funktionen“ auf Seite 76).

Wenn *find-bereich* in einer Funktion liegen soll, die über Pointer-to-Member adressiert wird, stehen Ihnen die folgenden beiden Möglichkeiten zur Verfügung:

Das Objekt der Klasse, das die gewünschte Funktion enthält, wird mit seinem Namen bezeichnet. Als Dereferenzierungsoperator schreiben Sie `.*`

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

[qua.]objekt.*[objekt.][klasse::][...]pointer-to-function-member

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Das Objekt der Klasse wird über einen Zeiger angesprochen. Als Dereferenzierungsoperator schreiben Sie **->***

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

[qua.]zeiger->*[objekt.][klasse::][...]pointer-to-function-member

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Das Objekt der Klasse bezeichnen Sie mit dem Operanden, der auf der linken Seite der Dereferenzierungsoperatoren `.*` bzw. **->*** steht:
mit *objekt* bezeichnen Sie das Objekt der Klasse mit seinem Namen,
mit *zeiger* adressieren Sie das Objekt über einen Zeiger.
Rechts der Deferenzierungsoperatoren schreiben Sie den Namen des Pointer-to-Function-Member. Evtl. müssen Sie das Objekt, das die Definition des Pointer-to-Function-Member enthält, und innerhalb des Objekts die zur eindeutigen Ansprache nötige Klassenqualifikation davorschreiben, falls der Pointer-to-Member von der Unterbrechungsstelle aus anders nicht zu erreichen ist.

An eine der oben beschriebenen Syntaxen können Sie nicht unmittelbar den Pointer-Operator anfügen. Vielmehr muss zunächst durch eine Typmodifikation, nämlich %al4, der Übergang auf die Maschinencode-Ebene ausgelöst werden, sodass sich die folgende Syntax ergibt:

`%DISPLAY` *dereferenzierter-pointer-to-function-member* `%al4->`

Näheres zum Arbeiten mit Pointer-to-Function-Member finden Sie auf Seite 83.

L'label'
bezeichnet die Adresse der ersten ausführbaren Anweisung nach einer Marke. *label* ist der Name einer Marke, der im Quellprogramm vergeben wurde.

S'[f-]n[:a]'
ist eine Source-Referenz und bezeichnet die Adresse einer ausführbaren Anweisung. Sie wird aus Zeilennummer (*n*) und gegebenenfalls der FILE-Nummer (*f*) sowie der relativen Anweisungsnummer innerhalb der Zeile (*a*) gebildet. Liegt die Source-Referenz in einer Funktion, die durch Instanziierung aus einem Funktionstemplate hervorgegangen ist, oder ist die Funktion, die die Source-Referenz enthält, in einer Instanz eines Klassentemplates definiert, so müssen Sie bei Mehrdeutigkeit vor die Source-Referenz die entsprechende PROC-Qualifikation schreiben.

schlüsselwort
legt einen Speicherbereich durch Angabe eines der folgenden Schlüsselwörter fest (siehe AID-Basishandbuch, Kapitel „Schlüsselwörter" [1]).

Wenn Sie eines der Schlüsselwörter für den Klasse-5-Speicher angeben, werden die nichtpriviligierten Bereiche, die Ihr Programm im Klasse-5-Speicher belegt, durchsucht.

Vor *schlüsselwort* können Sie nur eine Basisqualifikation angeben.

```
%CLASS6              Klasse-6-Speicher, unterhalb der 16MB-Grenze
%CLASS6BELOW         Klasse-6-Speicher, unterhalb der 16MB-Grenze
%CLASS6ABOVE         Klasse-6-Speicher, oberhalb der 16MB-Grenze
```

```
%CLASS5              Klasse-5-Speicher, unterhalb der 16MB-Grenze
%CLASS5BELOW         Klasse-5-Speicher, unterhalb der 16MB-Grenze
%CLASS5ABOVE         Klasse-5-Speicher, oberhalb der 16MB-Grenze

%n                   Mehrzweckregister, 0 ≤ n ≤ 15
%nD|E                Gleitpunktregister, n = 0,2,4,6
%nQ                  Gleitpunktregister, n = 0,4
%nG                  AID-Mehrzweckregister, 0 ≤ n ≤ 15
%nGD                 AID-Gleitpunktregister, n = 0,2,4,6
%MR                  alle 16 Mehrzweckregister in Tabellenform
%PC                  Befehlszähler (Program Counter)
```

kompl-speicherref

Folgende Operationen können in *kompl-speicherref* vorkommen (siehe AID-Basishandbuch, Abschnitt „Komplexe Speicherreferenzen“ [1]):

- Adressversatz (•)
- indirekte Adressierung (->)
- Typmodifikation (%A, %S, %SX)
- Längenmodifikation (%L(...), %L=(ausdruck), %Ln)
- Adressselektion (%@(...))

Beginnt *kompl-speicherref* mit Anweisungsname oder Source-Referenz, muss anschließend der Pointer-Operator geschrieben werden. Ohne den Pointer-Operator können Anweisungsname und Source-Referenz überall da verwendet werden, wo auch Sedezimalzahlen geschrieben werden können. Eine Marke muss in *kompl-speicherref* stets in `L'...'` gesetzt werden.
kompl-speicherref bezeichnet einen Bereich von 4 Bytes ab der errechneten Adresse. Soll eine andere Anzahl von Bytes durchsucht werden, muss *kompl-speicherref* mit der entsprechenden Längenmodifikation enden. Bei der Längenmodifikation von Daten müssen Sie die Bereichsgrenzen beachten oder mit %@(*datenname*)-> auf Maschinencode-Ebene wechseln. Mit einer Längenmodifikation können Sie nicht mehr als 65 535 Bytes vereinbaren.

alignment

legt fest, dass *suchbegriff* nur an bestimmten ausgerichteten Adressen gesucht werden soll.

```
alignment-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

ALIGN [=] { 1 }
          { 2 }
          { 4 }
          { 8 }

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

suchbegriff wird gesucht an:

1 Byte-Grenze (Standardwert)

2 Halbwort-Grenze

4 Wort-Grenze

8 Doppelwort-Grenze

Beispiele

1. `%find x'f0' in arr1`

   Das Sedezimal-Literal X'F0' wird im Vektor `arr1` gesucht. Ein Treffer wird auf SYSOUT ausgegeben.

2. `%f x'd2' in s'12'->%l=(s'13'-s'12') align=2`

   Im für die Anweisung S'12' erzeugten Maschinencode wird an einer Halbwortgrenze das Sedezimal-Literal X'D2' gesucht.

3. `%f`

   Die Suche wird mit den Parametern des letzten %FIND-Kommandos hinter dem letzten Treffer fortgesetzt.

%HELP

Mit %HELP können Sie sich über die Bedienung von AID informieren. Auf das gewählte Medium werden ausgegeben: entweder alle AID-Kommandos oder das gewählte Kommando und seine Operanden.

– Mit *info-ziel* geben Sie das Kommando an, über das Sie weitere Angaben brauchen.

– Mit *medium-u-menge* geben Sie an, über welche Ausgabemedien AID die angeforderten Informationen ausgeben soll. Mit diesem Operanden setzen Sie vorübergehend eine mit %OUT getroffene Vereinbarung außer Kraft.

Kommando	Operand	
%H[ELP]	[info-ziel]	[medium-u-menge][,...]

%HELP informiert Sie über alle Operanden des gewählten Kommandos, d.h. sowohl über alle sprachspezifischen Operanden für das symbolische Testen, als auch über alle Operanden für das maschinennahe Testen. Was für die Sprache, in der Ihr Programm geschrieben wurde, erlaubt ist, können Sie dem jeweiligen sprachspezifischen Handbuch entnehmen.

Die AID-Meldungen haben im Meldungsschlüssel den Aufbau AID0*n*, die AIDSYS-Meldungen den Aufbau IDA0*n*. Beide werden mit `/HELP-MSG-INFORMATION` abgefragt.

%HELP darf nur als Einzelkommando eingegeben werden, es darf nicht in einer Kommandofolge oder in einem Subkommando stehen.

%HELP verändert den Programmzustand nicht.

info-ziel

bezeichnet ein Kommando, zu dem Informationen ausgegeben werden sollen.
Ohne den *info-ziel*-Operanden gibt das Kommando eine Übersicht über die AID-Kommandos mit einer Kommando-Kurzbeschreibung aus.

Ein %HELP-Kommando mit falschem *info-ziel*-Operanden beantwortet AID mit einer Fehlermeldung. Daran schließt sich die vorher beschriebene Übersicht an. Diese Übersicht erhalten Sie auch, wenn Sie %?, %H? oder %H %? angeben.

```
info-ziel-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

( %AID | %AINT | %ALIAS | %BASE | %CONT[INUE] | %C[ONTROL]    )
| %DISASSEMBLE | %DA | %D[ISPLAY] | %DUMPFILE | %DF          |
| %F[IND] | %H[ELP] | %IN[SERT] | %JUMP | %M[OVE]           |
{ %ON | %OUT | %OUTFILE | %Q[UALIFY]                        }
| %REM[OVE] | %R[ESUME] | %SD[UMP] | %S[ET]                 |
( %SH[OW] | %STOP | %SYMLIB | %TITLE | %T[RACE]              )

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Die AID-Kommandonamen können auch in der zulässigen Abkürzung angegeben werden.

medium-u-menge

legt fest, über welche Medien die Informationen zu *info-ziel* ausgegeben werden sollen.

Ohne diesen Operanden und ohne eine Vereinbarung mit dem %OUT-Kommando arbeitet AID mit dem Standardwert T=MAX. Die Angabe {MIN | MAX | XMAX | XFLAT} hat bei %HELP keine Auswirkungen, eine der Angaben ist aber syntaktisch erforderlich.

```
medium-u-menge-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - -

{ T    }   { MIN   }
{ H    } = { MAX   }
{ Fn   }   { XMAX  }
{ P    }   { XFLAT }

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

medium-u-menge ist ausführlich im AID-Basishandbuch, Kapitel „Operand Medium-und-Menge“ [1] beschrieben.

T Terminal-Ausgabe

H Hardcopy-Ausgabe (schließt die Terminal-Ausgabe mit ein und kann nicht gemeinsam mit *T* angegeben werden)

Fn Datei-Ausgabe

P Ausgabe nach SYSLST

%INSERT

Mit %INSERT legen Sie einen Testpunkt fest und definieren ein Subkommando. Wenn der Programmablauf den Testpunkt erreicht, bearbeitet AID das zugehörige Subkommando.

– Mit *testpunkt* bezeichnen Sie die Adresse eines Befehls im Programm, vor dessen Ausführung AID den Programmablauf unterbrechen soll, um *subkdo* zu bearbeiten.

– Mit *subkdo* definieren Sie ein Kommando oder eine Kommandofolge und eventuell eine Bedingung. Wird *testpunkt* erreicht und ist die Bedingung erfüllt, wird *subkdo* ausgeführt.

Kommando	Operand
%IN[SERT]	testpunkt [<subkdo>]

Ein *testpunkt* wird in folgenden Fällen gelöscht:

– Das Programmende wird erreicht.
– Der *testpunkt* wird mit %REMOVE gelöscht.

Ohne *subkdo*-Operanden setzt AID das *subkdo* <%STOP> ein.

Das *subkdo* eines %INSERT für einen bereits gesetzten *testpunkt* überschreibt nicht das bestehende *subkdo*, sondern das neue *subkdo* wird vor das bestehende gekettet. Die geketteten Subkommandos werden somit nach dem LIFO-Prinzip abgearbeitet.

Mit %REMOVE löschen Sie ein Subkommando, einen Testpunkt oder alle eingetragenen Testpunkte.

testpunkt kann nur eine Adresse im geladenen Programm sein, deshalb muss die Basisqualifikation E=VM eingestellt sein (siehe Beschreibung des Kommandos %BASE) oder explizit angegeben werden.

Auf die Anweisungen `throw` und `catch` eines C++-Programms können Sie erst dann einen Testpunkt setzen, wenn das Programm initialisiert ist, d.h. wenn es bis zur ersten ausführbaren Anweisung abgelaufen ist. Wenn Sie nach dem Laden das folgende Kommando eingeben, wird das Programm unmittelbar nach der Initialisierung unterbrochen:

```
/%trace 1 in s=srcname
```

Wenn Sie ein Programm testen, das Klassen mit virtuellen Funktionen oder Konstruktoren enthält, hält das Programm nicht in `main`, sondern in einer vom Compiler erzeugten Funktion mit Namen `_ _STI_ _`. Diese Funktion ruft die Konstruktoren auf und legt die Tabellen der virtuellen Funktionen an. AID gibt den Namen `_ _STI_ _` in der STOP-Meldung aus.

%INSERT verändert den Programmzustand nicht.

testpunkt

muss die Adresse eines ausführbaren Maschinenbefehls sein, der entweder für eineC/C++-Anweisung erzeugt wurde oder, falls `throw`-/`catch`-Anweisungen überwacht werden sollen, der in einer Routine des Laufzeitsystems steht, die bei jeder `throw`- oder `catch`-Anweisung aufgerufen wird.
testpunkt wird sofort durch das gezielte Überschreiben der adressierten Speicherstelle eingetragen und muss deshalb zum Zeitpunkt der %INSERT-Eingabe bzw. bei der Abarbeitung des Subkommandos, in dem der %INSERT enthalten ist, im virtuellen Speicher geladen sein. Da durch das Eintragen von *testpunkt* der Code des Programms verändert wird, führt ein falsch gesetzter Testpunkt zu Fehlern im Programmablauf (z.B. Daten- oder Adressierungsfehler).

Kommt der Programmablauf an den *testpunkt*, unterbricht AID das Programm und startet *subkdo*.

```
testpunkt-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

  ⎧              ⎧funktion[->]        ⎫ ⎫
  ⎪              ⎪L'label'            ⎪ ⎪
  ⎪[•][qua•]     ⎨                    ⎬ ⎪
  ⎪              ⎪S'[f-]n[:a]'        ⎪ ⎪
  ⎨              ⎩kompl-speicherref   ⎭ ⎬
  ⎪                                     ⎪
  ⎪ ⎧%EXCEPTION⎫  ⎧throw⎫               ⎪
  ⎪ ⎨          ⎬( ⎨     ⎬)              ⎪
  ⎩ ⎩%EH       ⎭  ⎩catch⎭               ⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

• Steht der Punkt an führender Stelle, ist er das Kennzeichen für eine *vorqualifikation*. Sie muss mit einem vorhergehenden %QUALIFY-Kommando definiert worden sein.
Aufeinanderfolgende Qualifikationen werden durch einen Punkt getrennt. Außerdem muss zwischen der letzten Qualifikation und dem anschließenden Operandenteil ein Punkt stehen.

qua Eine oder mehrere Qualifikationen geben Sie an, wenn Sie eine Funktion, eine Marke oder eine Source-Referenz ansprechen wollen, die von der aktuellen Unterbrechungsstelle aus anders nicht zu erreichen ist.
Sie geben nur die Qualifikationen an, die zur eindeutigen Ansprache genügen.

E=VM
Da *testpunkt* nur im virtuellen Speicher des geladenen Programms eingetragen werden kann, geben Sie E=VM nur an, wenn als aktuelle Basisqualifikation eine Dump-Datei vereinbart ist (siehe Beschreibung des Kommandos %BASE).

S=srcname
: geben Sie nur an, wenn Sie einen Anweisungsnamen oder eine Source-Referenz ansprechen, die nicht in der aktuellen Übersetzungseinheit liegen (siehe Kapitel „Adressierung in C- und C++-Programmen“ auf Seite 21).

BLK='[f-]n[:b]'
: Eine BLK-Qualifikation müssen Sie angeben, wenn Sie mit einer nachfolgenden PROC-Qualifikation eine Funktion aus einer lokalen Klasse bezeichnen und die Definition dieser lokalen Klasse steht im angegebenen Block (siehe unten PROC=*funktion*).
Der Name für einen Block wird aus Zeilennummer (*n*) und gegebenenfalls FILE-Nummer (*f*) und relativer Blocknummer (*b*) gebildet

PROC=funktion
: geben Sie nur an, wenn Sie eine Marke aus einer anderen als der aktuellen Funktion ansprechen wollen (siehe Kapitel „Adressierung in C- und C++-Programmen“ auf Seite 21) oder wenn *testpunkt* auf eine Source-Referenz gesetzt werden soll, die in der Übersetzungseinheit mehrdeutig ist, weil sie in einer Instanz eines Funktionstemplates liegt oder weil sie einer Funktion zugeordnet ist, die in einer Instanz eines Klassentemplates definiert ist (siehe Abschnitt „Templates“ auf Seite 98).
Bei Funktionen aus C-Programmen ist *funktion* der im Quellprogramm vergebene Name der Funktion ohne Klammern und ohne Signatur.
Funktionen aus C++-Programmen müssen Sie, je nachdem, um welchen Typ es sich handelt, in der Schreibweise `n'...'` oder `t'...'` angeben. Ist die Funktion in einem Namespace oder in einer Klasse definiert, so wird dem Funktionsnamen die Namespace- oder Klassen-Qualifikation vorangestellt. Die Signatur `void` darf nicht mehr geschrieben werden. Wie auch in C++ möglich geben Sie in diesem Fall nur die beiden Klammern nach dem Funktionsnamen an.
Es ergibt sich die folgende Syntax (*f_template* und *signatur* sind in der Syntax aus Platzgründen abgekürzt):

Abweichend davon werden die Funktionen `main` und `_ _STI_ _` sowie alle Funktionen mit C-Linkage auch beim Testen in C++-Programmen nur mit dem Funktionsnamen bezeichnet (siehe Seite 60).

Ist die Funktion in einer lokalen Klasse definiert, so müssen Sie vor der PROC-Qualifikation noch eine weitere PROC-Qualifikation für die übergeordnete Funktion angeben, die die Definition der lokalen Klasse enthält. Zwischen die

beiden PROC-Qualifikationen schreiben Sie bei Funktionen, die in einem inneren Block der übergeordneten Funktion definiert sind, eine oder ggfs. mehrere BLK-Qualifikationen (siehe Seite 62).
Das Ansprechen von Funktionen, die in lokalen Klassen innerer Blöcke definiert sind, wird nur bei Programmen unterstützt, die mit einem C/C++-Compiler ab V3.0B übersetzt wurden.

funktion[->]

Damit legen Sie *testpunkt* auf die erste ausführbare Anweisung in einer Funktion bzw. auf den ersten Befehl in einer Bibliotheksfunktion.
funktion ist der Name einer Funktion, der im Quellprogramm vergeben wurde bzw. der Name einer Bibliotheksfunktion (siehe oben PROC=*funktion* und Kapitel „C++-spezifische Adressierung“ auf Seite 59).

Für virtuelle Funktionen gilt die folgende Syntax:

```
p->n'funktion([signatur])'
```

p ist eine Zeigervariable, die auf das Objekt einer Klasse verweist, das die gewünschte Member-Funktion enthält. Wenn *p* von der aktuellen Unterbrechungsstelle aus nicht zu erreichen ist, muss sie ihrem Gültigkeitsbereich entsprechend qualifiziert werden. Liegt die Unterbrechungsstelle in der virtuellen Funktion selbst, dann können Sie die erste ausführbare Anweisung der aktuellen Funktion ansprechen, indem Sie statt *p* den `this`-Zeiger einsetzen.

Wenn Sie den Testpunkt auf die erste ausführbare Anweisung einer Funktion setzen wollen, die über Pointer-to-Member adressiert wird, stehen Ihnen die folgenden beiden Möglichkeiten zur Verfügung:

Das Objekt der Klasse, das die gewünschte Funktion enthält, wird mit seinem Namen bezeichnet. Als Dereferenzierungsoperator schreiben Sie •*

```
[qua•]objekt•*[objekt•][klasse::][...]pointer-to-function-member
```

Das Objekt der Klasse wird über einen Zeiger angesprochen. Als Dereferenzierungsoperator schreiben Sie **->***

```
[qua•]zeiger->*[objekt•][klasse::][...]pointer-to-function-member
```

Das Objekt der Klasse bezeichnen Sie mit dem Operanden, der auf der linken Seite der Dereferenzierungsoperatoren •* bzw. **->*** steht:
mit *objekt* bezeichnen Sie das Objekt der Klasse mit seinem Namen,
mit *zeiger* adressieren Sie das Objekt über einen Zeiger.
Rechts der Deferenzierungsoperatoren schreiben Sie den Namen des Pointer-to-Function-Member. Evtl. müssen Sie das Objekt, das die Definition des Pointer-to-Function-Member enthält, und innerhalb des Objekts die zur eindeutigen Ansprache nötige Klassenqualifikation davorschreiben, falls der Pointer-to-Member von der Unterbrechungsstelle aus anders nicht zu erreichen ist. Näheres zum Arbeiten mit Pointer-to-Function-Member finden Sie auf Seite 83.

Wenn Sie mit *testpunkt* eine Bibliotheksfunktion bezeichnen, müssen Sie *funktion* mit dem Pointer-Operator abschließen. Der Testpunkt wird in diesem Fall auf den ersten Befehl des Prologs der Funktion gesetzt. Wenn Sie sich mit %SDUMP %NEST von dieser Stelle aus die Aufrufhierarchie ansehen, so fehlt u.U. der unmittelbare Aufrufer der Bibliotheksfunktion. AID kann die Aufrufhierarchie erst dann vollständig ermitteln, wenn der Prolog durchlaufen und die erste ausführbare Anweisung der Funktion erreicht ist. Falls zu der Funktion LSD zur Verfügung steht, können Sie mit %TRACE 1 %STMT auf die erste ausführbare Anweisung der Funktion positionieren.

Es wird darauf hingewiesen, dass AID i.a. die Prologadresse nicht der entsprechenden Funktion zuordnen kann. Diesen Effekt müssen Sie beachten, wenn Sie bei Funktionen aus C++-Programmen der Einfachheit halber im %INSERT die Funktionsadressen verwenden, die AID zu %DISPLAY{*namespace*|*objekt*|*klasse*} bzw. in der %SDUMP-Ausgabe auflistet, um sich die Eingabe der in der Regel sehr langen Namen zu ersparen. Wenn Sie sich dann im weiteren Testverlauf mit %SHOW %INSERT über

die bis dahin gesetzten Testpunkte informieren wollen, gibt AID zu den Prologadressen der Funktionen den Namen einer davorliegenden Funktion aus.

L'label'

Damit legen Sie *testpunkt* auf die erste ausführbare Anweisung nach einer Marke. *label* ist der Name einer Marke, der im Quellprogramm vergeben wurde. In diesem Kommando können Sie *label* auch ohne `L'...'` angeben, da eine Verwechslung mit einem Datennamen nicht möglich ist.

S'[f-]n[:a]'

ist eine Source-Referenz. Damit legen Sie *testpunkt* auf eine ausführbare Anweisung. Die Source-Referenz wird aus Zeilennummer (*n*) und gegebenenfalls der FILE-Nummer (*f*) sowie der relativen Anweisungsnummer innerhalb der Zeile (*a*) gebildet.
Liegt die Source-Referenz in einer Funktion, die durch Instanziierung aus einem Funktionstemplate hervorgegangen ist, oder ist die Funktion, die die Source-Referenz enthält, in einer Instanz eines Klassentemplates definiert, so müssen Sie bei Mehrdeutigkeit vor die Source-Referenz die entsprechende PROC-Qualifikation schreiben.

kompl-speicherref

Das Ergebnis von *kompl-speicherref* muss die Anfangsadresse eines ausführbaren Maschinenbefehls sein. *kompl-speicherref* kann folgende Operationen enthalten (siehe AID-Basishandbuch, Abschnitt „Komplexe Speicherreferenzen“ [1]):

- Adressversatz (•)
- indirekte Adressierung (->)
- Typmodifikation (%A, %S, %SX)
- Längenmodifikation (%Ln)
- Adressselektion (%@(...))

Beginnt eine *kompl-speicherref* mit Anweisungsname oder Source-Referenz, muss anschließend der Pointer-Operator geschrieben werden. Marken müssen Sie in einer *kompl-speicherref* stets in L'...' setzen. Dabei ist zu beachten, dass durch den Pointer die symbolische Ebene verlassen wird. Wenn Sie nach einem Funktionsnamen einen Pointer schreiben, bezeichnen Sie damit nicht mehr die erste ausführbare Anweisung der Funktion, sondern den ersten Befehl des Prologs, den der Compiler zu dieser Funktion erzeugt hat.
Ohne den Pointer-Operator können Anweisungsname und Source-Referenz überall da verwendet werden, wo auch Sedezimalzahlen geschrieben werden können.

Eine Typmodifikation ist nur sinnvoll, wenn der Inhalt eines Datums als Adresse eingesetzt werden kann oder wenn Sie die Adresse aus einem Register entnehmen.

Beispiel: `%3g.2 %al2 ->`
Die letzten beiden Bytes aus AID-Register %3G werden als Adresse benutzt.

{%EXCEPTION|%EH} ({throw|catch})

Mit dieser Angabe können Sie die `throw`– bzw. `catch`–Anweisungen eines C++-Programms überwachen. Dazu wird ein Testpunkt in eine Routine des Laufzeitsystems gesetzt, die bei jeder Ausführung einer `throw`- bzw. `catch`-Anweisung aufgerufen wird. Daraus ergibt sich, dass bei der Ausführung einer `throw`- bzw. `catch`-Anweisung das Programm nicht an der entsprechenden Anweisung im Benutzerprogramm unterbrochen wird, sondern vor dem ersten Befehl dieser Laufzeitroutine. Wenn Sie von dieser Stelle aus die näheren Umstände untersuchen wollen, die zum Auslösen der Ausnahmebehandlung geführt haben, so müssen Sie beachten, dass Sie bei Unterbrechung im Laufzeitsystem Daten und Anweisungen Ihres Programms nur vollqualifiziert ansprechen können.
Um herauszufinden, welche Anweisung Ihres Programms die Unterbrechung verursacht hat, müssen Sie bei `throw`- und bei `catch`-Anweisungen unterschiedlich verfahren:

– `throw` (Ausnahmebehandlung wird ausgelöst):

  Sie lassen sich mit `%SDUMP %NEST` die Aufrufhierarchie an der Unterbrechungsstelle ausgeben. Das Kommando `%SDUMP %NEST` können Sie direkt in das Subkommando des %INSERT schreiben, dann gibt AID automatisch bei jedem Auftreten von `throw` die Aufrufhierarchie aus.

– `catch` (Ausnahme wird bearbeitet):

  AID stellt die Prologadresse des `catch`-Handlers Ihres Programms in Register 15 bereit. Mit der Kommandofolge
  `%INSERT %15-> <%TRACE 1 %STMT>; %RESUME`
  wird das Programm bis zur ersten Anweisung des `catch`-Handlers ausgeführt, diese Anweisung wird protokolliert und das Programm angehalten.

Um `%INSERT %EH({throw|catch})` eingeben zu können, müssen Sie die Unterscheidung von Klein-/Großbuchstaben eingeschaltet haben (Kommando `%AID LOW={ON|ALL}`). Andernfalls meldet AID einen Syntaxfehler.

Beispiel 5 auf Seite 202 illustriert die Anwendung von `%INSERT %EH(...)`.

subkdo

wird immer dann bearbeitet, wenn der Programmablauf an die mit *testpunkt* bezeichnete Adresse kommt.
Wird der *subkdo*-Operand nicht angegeben, so setzt AID ein <%STOP> ein.

Vollständig beschrieben finden Sie *subkdo* im Basishandbuch, Kapitel „Subkommando“ [1].

```
subkdo-OPERAND  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

                                      ⎧AID-kommando    ⎫
<[subkdoname:] [(bedingung):] [ {     ⎨                } {;...}]>
                                      ⎩BS2000-kommando⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Ein Subkommando kann einen Namen, eine Bedingung und einen Kommandoteil enthalten. Zu jedem Subkommando gehört ein Durchlaufzähler. Der Kommandoteil kann aus einem einzelnen Kommando oder einer Kommandofolge bestehen, er kann AID- und BS2000-Kommandos und Kommentare enthalten.

Wenn das Subkommando aus einem Namen oder einer Bedingung besteht, aber der Kommandoteil fehlt, erhöht AID beim Erreichen des Testpunkts nur den Durchlaufzähler.

subkdo überschreibt nicht ein bestehendes Subkommando zu demselben *testpunkt*, sondern das neue Subkommando wird vor das bestehende gekettet. Im *subkdo* eines %INSERT sind die Kommandos %CONTROL*n*, %INSERT und %ON zugelassen .

Die Kommandos in einem *subkdo* werden nacheinander ausgeführt. Danach wird das Programm fortgesetzt. Die Kommandos zur Ablaufsteuerung verändern auch in einem Subkommando sofort den Programmzustand. Sie brechen *subkdo* ab und starten das Programm (%CONTINUE, %RESUME, %TRACE) oder halten es an (%STOP). Sie sind nur als letztes Kommando sinnvoll, da im *subkdo* nachfolgende Kommandos nicht mehr ausgeführt werden. Auch ein Löschen des gerade aktiven Subkommandos mit %REMOVE ist nur als letztes Kommando in *subkdo* sinnvoll.

Adressoperanden in Subkommandos werden bei der Eingabe nicht automatisch mit den Qualifikationen ergänzt, die der aktuellen Unterbrechungsstelle entsprechen. Wenn im weiteren Testverlauf *testpunkt* erreicht wird und AID das Programm unterbricht, um *subkdo* zu bearbeiten, können mit den AID-Kommandos aus *subkdo* ohne Qualifikation nur die Daten und Funktionen angesprochen werden, die am Testpunkt sichtbar sind.

Beispiele

1. `%in s'48'`

   testpunkt wird mit einer Source-Referenz angegeben und auf die Speicherstelle gesetzt, an der der Befehlscode steht, der zu der ersten Anweisung in Zeile 48 erzeugt wurde.

2. `%in facul <%display %.,n>`

   testpunkt wird mit einem Funktionsnamen bezeichnet. Da `facul` ein C-Programm ist, wird der Funktionsname ohne Signatur und ohne Klammern angegeben. Immer wenn der Programmablauf zur ersten Anweisung in der Funktion `facul` kommt, wird der %DISPLAY aus *subkdo* ausgeführt.

3. ```
   %in A::n'out1(int)' <%display var1, 'function A::out1'>
   %in A::n'out2(int)' <%display 'INSERT1', var1; %in A::n'out3(int)' –
       <%d 'INSERT2',i,j,k, i_arr[i]; %in s'172' <%d 'INSERT3' ,i,j; –
       %remove A::n'out3(int)'>>>
   ```

   Mit dem ersten %INSERT wird als *testpunkt* die erste Anweisung der Member-Funktion `A::out1(int)` festgelegt. Kommt nach der Beendigung der Kommandoeingabe der Programmablauf an diese Adresse, wird das Subkommando ausgeführt. Es besteht aus einem %DISPLAY für den Datennamen `var1` und dem Literal `'function A::out1'`. Anschließend läuft das Programm weiter.

   Mit dem zweiten %INSERT wird der *testpunkt* `A::n'out2(int)'` vereinbart. Dieser %INSERT enthält noch zwei geschachtelte %INSERT-Kommandos. Ihre *testpunkt*-Werte sind für AID noch nicht wirksam. Sie können erst aktiv werden, wenn der *testpunkt* des %INSERT erreicht wird, in dessen *subkdo* sie definiert sind.

   Kommt der Programmablauf zur ersten ausführbaren Anweisung in der Member-Funktion `A::out2(int)`, wird das zugehörige *subkdo* ausgeführt, d.h. das %DISPLAY-Kommando für das Literal `'INSERT1'` und den Datennamen `var1` wird ausgeführt und der *testpunkt* `A::n'out3(int)'` wird eingetragen.
   Das *subkdo* zum *testpunkt* `A::n'out3(int)'` ist noch nicht wirksam. Im zu testenden Programm sind also bis zu dieser Stelle des Programmablaufs drei *testpunkte* gesetzt: die jeweils erste ausführbare Anweisung in den Member-Funktionen `A::out1(int)`, `A::out2(int)` und `A::out3(int)`.

   Da auch das *subkdo* zum *testpunkt* `A::n'out2(int)'` kein %STOP-Kommando enthält, wird das Programm nach Ausführung von *subkdo* fortgesetzt. Wird der Programmablauf nicht aus einem anderen Grund unterbrochen, z.B. einem Fehler oder dem Eintreten eines mit %ON vereinbarten Ereignisses, und erreicht schließlich die erste Anweisung in `A::out3(int)`, wird nun der `%D 'INSERT2', i, j, k, i_arr[i]` ausgeführt. Außerdem enthält *subkdo* wieder ein %INSERT-Kommando, dessen *testpunkt* diesmal mit der Source-Referenz S'172' bezeichnet ist.
   ```

Wenn der Programmablauf die Anweisung in Zeile 172 erreicht, führt AID den %DISPLAY für das Literal 'INSERT3' und die Inhalte der Variablen `i` und `j` aus. Mit dem zweiten Kommando in diesem *subkdo*, dem `%REMOVE A::n'out3(int)'`, wird der *testpunkt* `A::n'out3(int)'` gelöscht. Das ist z.B. dann erforderlich, wenn ein *testpunkt* in einer Schleife liegt und es dadurch zur unerwünschten Kettung geschachtelter *subkdo's* kommen würde. Ohne das %REMOVE-Kommando würde beim zweiten Durchlaufen von `A::n'out3(int)'` zum *testpunkt* S'172' folgendes *subkdo* entstehen:
`<%d 'insert3',i,j; %d 'insert3',i,j>`

4.
```
%out %display p=max
%in s'73' <%d 'i GE 10',i,c_str[i],k,num[i][k]>
%in s'73' <(i lt 10): %d 'i LT 10',i,c_str[i]; %cont>
```

Zunächst wird vereinbart, dass alle Ausgaben des Kommandos %DISPLAY nach SYSLST erfolgen.
Durch die danach eingegebenen beiden %INSERTs entsteht am *testpunkt* S'73' das folgende Subkommando:

```
<(i lt 10): %d 'I LT 10',i,c_str[i]; %cont; %d 'i GE 10',i,c_str[i], -
k,num[i][k]>
```

Jedesmal, wenn der Programmablauf an die Anweisung in der Zeile 73 kommt, wird geprüft, ob der Index `i` einen Wert < 10 enthält. Falls die Bedingung zutrifft, schreibt AID den Kommentar `'i LT 10'` sowie den Inhalt von `i` und `c_str[i]` nach SYSLST und setzt wegen des %CONTINUE den Programmablauf fort, evtl. mit Ablaufverfolgung, falls durch das Subkommando ein %TRACE unterbrochen wurde.
Ist der Wert von `i` ≥ 10, dann schreibt AID den Kommentar `'i GE 10'` und zusätzlich zu `i` und `c_str[i]` auch die Werte von `k` und dem Vektorelement `num[i][k]` nach SYSLST und setzt ebenfalls das Programm fort. Auch in diesem Fall läuft das Programm mit Ablaufverfolgung weiter, wenn noch ein %TRACE aktiv ist.

5. *Ausnahmebehandlung*

   Das Beispielprogramm zum folgenden Ablaufprotokoll steht auf Seite 142.

```
/LOAD-PROG   *MOD(LIB.23A,EXMEM,RUN-MODE=ADVANCED,PROGRAM-MODE=ANY),
TEST-OPTIONS=AID
%  BLS0523 ELEMENT 'EXMEM', VERSION '@' FROM LIBRARY '$TEST.MYLIB' IN
PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$EXMEM$', VERSION ' ' OF '1999-01-07
10:27:02' LOADED
/%aid low
/%t 1 in s=n'exmem.c'
10                        EXT.PROC START   , BLOCK START,  , BLOCK START,
                          ASSIGN
STOPPED AT SRC_REF: 10, SOURCE: EXMEM.C , BLK: 9 , END OF TRACE
/%in %eh(throw) <%sd %nest; %stop>
/%in %eh(catch) <sub1: %in %15-> <%t 1 %stmt>; %r>
```

Nach dem Laden des Programms wird zunächst mit dem Kommando %AID die Unterscheidung von Groß-/Kleinschreibung eingeschaltet und mit dem %TRACE bis zur ersten ausführbaren Anweisung des Programms vorgesetzt. Die beiden %INSERTs setzen Testpunkte für den `throw`- und für den `catch`-Fall. Am Testpunkt zu `throw` wird mit `%SD %NEST` die Aufrufhierarchie angefordert, so dass die Stelle im Programm angezeigt wird, an der die Ausnahmebehandlung ausgelöst wurde. Das Subkommando `sub1` zum Testpunkt zu `catch` bewirkt, dass das Programm nach dem Erreichen des Testpunkts in der Laufzeitroutine `AIDITO@` bis zur ersten Anweisung des `catch`-Handlers im Benutzerprogramm ausgeführt wird.

```
/%r
SRC_REF:  1336  SOURCE: AIDITO@  PROC: AIDITO@  *************************
ABSOLUT: V'1021458'   SOURCE: IPP_THROW&@  PROC: unwind_stack  *********
ABSOLUT: V'1025C6A'   SOURCE: IPP_THROW&@  PROC: IPPTHR  ***************
ABSOLUT: V'102E1F6'   SOURCE: IPP_NEW&@  PROC: operator new  ***********
ABSOLUT: V'101E96E'   SOURCE: IPP_ARRAY_NEW&@  PROC: operator new[]  ***
SRC_REF: 20  SOURCE: EXMEM.C  BLK : 19  ******************************
SRC_REF: 36  SOURCE: EXMEM.C  PROC: main  ****************************
ABSOLUT: V'113CF88'   SOURCE: IC@RT20A  PROC: IC@RT20A  ****************
ABSOLUT: V'1001568'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  ****************
STOPPED AT LABEL: CPPTHROW , SRC_REF: 1336, SOURCE: AIDITO@ ,
PROC: AIDITO@
```

Das Programm wird mit %RESUME gestartet und läuft bis zum Eintreten der Ausnahmebehandlung durch. Der Aufrufhierarchie können Sie entnehmen, dass die Anweisung in der Zeile 20 des Programms (`q = new char[0x10000000];`) nicht ausgeführt werden konnte und dadurch die Ausnahmebehandlung verursacht wurde.

```
/%resume
 24                          , BLOCK START, CALL
STOPPED AT SRC_REF: 24, SOURCE: EXMEM.C , BLK: 23 , END OF TRACE
/%sh %in
> CTX: LOCAL#DEFAULT  SRC-REF: 20  SOURCE: EXMEM.C  PROC: main
  ( V'0100016C' )
> CTX: LOCAL#DEFAULT  SRC-REF: 1336  SOURCE: AIDITO@  PROC: AIDITO@
  LABEL: CPPTHROW
> CTX: LOCAL#DEFAULT  SRC-REF: 1345  SOURCE: AIDITO@  PROC: AIDITO@
  LABEL: CPPCATCH
(SUB1                          )
/%rem V'01000016C'
```

Mit %RESUME wird der Programmablauf bis zur Unterbrechung im `catch`-Handler fortgesetzt. Damit der %INSERT auf die im Register 15 gespeicherte Prologadresse des `catch`-Handlers im weiteren Programmverlauf nicht zu unerwünschten Unterbrechungen führt, empfiehlt es sich, diesen Testpunkt mit %REMOVE zu löschen. Das Kommando `%SHOW %INSERT` zeigt Ihnen die entsprechende Adresse an.

%MOVE

Mit %MOVE übertragen Sie Speicherinhalte oder AID-Literale auf Speicherstellen im geladenen Programm. Die Übertragung erfolgt byteweise, linksbündig, ohne Überprüfung und ohne Anpassung des Speichertyps von Sender an Empfänger.

– Mit *sender* bezeichnen Sie eine Variable, ein Objekt einer Klasse oder eine Komponente daraus, eine Struktur oder eine Strukturkomponente, einen Vektor oder ein Vektorelement, eine Länge, eine Adresse, einen Durchlaufzähler, ein Register, eine komplexe Speicherreferenz oder ein AID-Literal.
sender kann im virtuellen Speicher des geladenen Programms oder in einer Dump-Datei liegen.

– Mit *empfänger* bezeichnen Sie eine Variable, ein Objekt einer Klasse oder eine Komponente daraus, eine Struktur oder eine Strukturkomponente, einen Vektor oder ein Vektorelement, eine komplexe Speicherreferenz, einen Durchlaufzähler oder ein Register, das überschrieben werden soll. *empfänger* kann nur im virtuellen Speicher des geladenen Programms liegen.

– Mit *REP* geben Sie an, ob AID zu einer durchgeführten Änderung einen REP-Satz erzeugen soll. Mit diesem Operanden setzen Sie eine mit dem Kommando %AID vereinbarte Voreinstellung für das aktuelle %MOVE-Kommando außer Kraft.

Kommando	Operand
%M[OVE]	sender INTO empfänger [REP]

Im Gegensatz zu %SET überprüft AID beim %MOVE nicht die Verträglichkeit der Speichertypen von *sender* und *empfänger* und konvertiert *sender* nicht in den Speichertyp von *empfänger*.

AID überträgt binär linksbündig in der Länge von *sender*. Ist *sender* länger als *empfänger*, weist AID die Übertragung mit einer Fehlermeldung ab.

Wenn *empfänger* ein Objekt einer Klasse ist, überschreibt der %MOVE möglicherweise vom Compiler generierte Hilfsvariablen. Dies führt zu einem inkonsistenten Zustand des *empfänger*-Objektes.

Unmittelbar nach dem Laden können Sie nur globale und static vereinbarte Daten ansprechen. Zum Zugriff benötigt AID die entsprechende Qualifikation.

Neben den hier beschriebenen Operandenwerten können Sie auch die im Handbuch für das Testen auf Maschinencode-Ebene [2] beschriebenen Operandenwerte einsetzen.

Mit %AID CHECK=ALL können Sie zur Kontrolle einen Änderungsdialog einschalten, der Ihnen vor Durchführung der Übertragung den alten und neuen Inhalt des *empfängers* zeigt und Ihnen die Möglichkeit zum Abbruch des %MOVE-Kommandos bietet.

%MOVE verändert den Programmzustand nicht.

sender INTO empfänger

Für *sender* oder *empfänger* können Sie eine Variable, ein Objekt einer Klasse oder eine Komponente daraus, eine Struktur oder eine Strukturkomponente, einen Vektor oder ein Vektorelement, einen Durchlaufzähler, ein Register oder eine komplexe Speicherreferenz angeben. Konstanten, Adressen, Längen und AID-Literale können Sie nur als *sender* einsetzen.

sender kann sowohl im virtuellen Speicherbereich des geladenen Programms als auch in einer Dump-Datei liegen; *empfänger* kann dagegen nur im virtuellen Speicherbereich des geladenen Programms liegen. Übertragen bzw. überschreiben Sie Programmbereiche mit Befehlscode, kann das zu unerwünschten Ergebnissen führen, wenn Adressen betroffen sind, die zu einem *control-* oder *trace-bereich* gehören oder auf die mit %INSERT ein *testpunkt* gesetzt wurde (siehe AID-Basishandbuch, Abschnitt „Wechselwirkungen“ [1]).

Mehr als 3900 Bytes können mit einem %MOVE nicht übertragen werden. Wenn Sie einen größeren Bereich übertragen wollen, müssen Sie daher mehrere %MOVE-Kommandos verwenden.

sender-OPERAND

- Steht der Punkt an führender Stelle, ist er das Kennzeichen für eine *vorqualifikation*. Sie muss mit einem vorhergehenden %QUALIFY definiert worden sein. Aufeinanderfolgende Qualifikationen werden durch einen Punkt getrennt. Außerdem muss zwischen der letzten Qualifikation und dem anschließenden Operandenteil ein Punkt stehen.

qua
: Eine oder mehrere Qualifikationen geben Sie an, wenn *sender* oder *empfänger* von der aktuellen Unterbrechungsstelle aus anders nicht zu erreichen sind oder um einen Datennamen anzusprechen, der an der Unterbrechungsstelle durch eine gleichnamige Definition lokal verdeckt ist. Sie geben nur die Qualifikationen an, die zur eindeutigen Ansprache des Speicherobjekts genügen.

  {E={VM | Dn} für *sender* | E=VM für *empfänger*}
  : geben Sie nur an, wenn für einen Datennamen, eine Klasse, ein Objekt einer Klasse, einen Anweisungsnamen, eine Source-Referenz oder ein Schlüsselwort die aktuelle Basisqualifikation nicht gelten soll (siehe %BASE). *sender* kann sowohl im virtuellen Speicher als auch in einer Dump-Datei liegen. *empfänger* kann dagegen nur im virtuellen Speicher liegen.

  S=srcname
  : geben Sie nur an, wenn Sie einen Datennamen, eine Klasse, ein Objekt einer Klasse, einen Anweisungsnamen oder eine Source-Referenz ansprechen, die nicht in der aktuellen Übersetzungseinheit liegen (siehe Kapitel „Adressierung in C- und C++-Programmen“ auf Seite 21).

  :: Die beiden vorangestellten Doppelpunkte verwenden Sie, um ein globales Datum anzusprechen, das durch eine gleichnamige Definition an der Unterbrechungsstelle lokal verdeckt ist. Außerdem müssen Sie die beiden Doppelpunkte vor den Namen eines globalen Datums oder einer Funktion setzen, weil

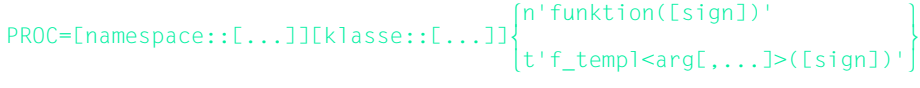
entweder das Datum oder die Funktion nicht in der Aufrufhierarchie liegen oder weil deren Definition erst nach der Unterbrechungsstelle steht.
Im Gegensatz zu den übrigen Qualifikationen wird zwischen den beiden Doppelpunkten und dem anschließenden Operanden kein Punkt geschrieben.

PROC=funktion

geben Sie an, wenn Sie eine Klasse oder ein Objekt einer Klasse oder einen Datennamen ansprechen wollen, der zwar in der aktuellen Funktion definiert ist, aber von einer gleichnamigen Definition an der Unterbrechungsstelle verdeckt wird. Außerdem geben Sie eine PROC-Qualifikation an, wenn Sie eine Marke oder einen static vereinbarten Datennamen ansprechen wollen, der einer Funktion außerhalb der aktuellen Aufrufhierarchie zugeordnet ist (siehe Kapitel „Adressierung in C- und C++-Programmen“ auf Seite 21). Wenn Sie eine Source-Referenz ansprechen wollen, die in einer Instanz eines Funktionstemplates liegt oder die einer Funktion zugeordnet ist, die in einer Instanz eines Klassentemplates definiert ist (siehe Abschnitt „Templates“ auf Seite 98), müssen Sie bei Mehrdeutigkeit ebenfalls die entsprechende PROC-Qualifikation davorschreiben.

Bei Funktionen aus C-Programmen ist *funktion* der im Quellprogramm vergebene Name der Funktion ohne Klammern und ohne Signatur.

Funktionen aus C++-Programmen müssen Sie, je nachdem, um welchen Typ es sich handelt, in der Schreibweise `n'...'` oder `t'...'` angeben. Ist die Funktion in einem Namespace oder in einer Klasse definiert, so wird dem Funktionsnamen die Namespace- oder Klassen-Qualifikation vorangestellt. Die Signatur `void` darf nicht mehr geschrieben werden. Wie auch in C++ möglich geben Sie in diesem Fall nur die beiden Klammern nach dem Funktionsnamen an. Es ergibt sich die folgende Syntax (*f_template* und *signatur* sind in der Syntax aus Platzgründen abgekürzt):



Abweichend davon werden die Funktionen `main` und `_ _STI_ _` sowie alle Funktionen mit C-Linkage auch beim Testen in C++-Programmen nur mit dem Funktionsnamen bezeichnet (siehe Seite 60).

Ist die Funktion in einer lokalen Klasse definiert, so müssen Sie vor der PROC-Qualifikation noch eine weitere PROC-Qualifikation für die übergeordnete Funktion angeben, die die Definition der lokalen Klasse enthält. Zwischen die beiden PROC-Qualifikationen schreiben Sie bei Funktionen, die in einem inne-

ren Block der übergeordneten Funktion definiert sind, eine oder ggfs. mehrere BLK-Qualifikationen (siehe Seite 62).
Das Ansprechen von Funktionen, die in lokalen Klassen innerer Blöcke definiert sind, wird nur bei Programmen unterstützt, die mit einem C/C++-Compiler ab V3.0B übersetzt wurden.

BLK='[f-]n[:b]'

geben Sie an, wenn Sie einen Datennamen ansprechen wollen, der einem Block innerhalb der aktuellen Aufrufhierarchie zugeordnet ist und von einer gleichnamigen Definition an der Unterbrechungsstelle verdeckt wird oder wenn Sie einen static vereinbarten Datennamen ansprechen wollen, der einem Block außerhalb der aktuellen Aufrufhierarchie zugeordnet ist (siehe Kapitel „Adressierung in C- und C++-Programmen“ auf Seite 21).

Des weiteren geben Sie eine BLK-Qualifikation an, wenn Sie mit einer nachfolgenden PROC-Qualifikation eine Funktion aus einer lokalen Klasse bezeichnen und die Definition dieser lokalen Klasse steht im angegebenen Block (siehe oben PROC=*funktion*).

Der Name für einen Block wird aus Zeilennummer (*n*) und gegebenenfalls FILE-Nummer (*f*) und relativer Blocknummer (*b*) gebildet

NESTLEV= level-nummer

level-nummer Nummer einer Ebene in der aktuellen Aufrufhierarchie

Auf *level-nummer* muss *datenname* folgen.
NESTLEV= *level-nummer* geben sie an, wenn Sie einen Datennamen in einer bestimmten Ebene der aktuellen Aufrufhierachie ansprechen wollen. Diese Qualifikation kann nur mit E= kombiniert werden, nicht mit anderen Qualifikationen.

namespace

ist der im Quellprogramm deklarierte Name eines Namespaces.

Den Namen eines Namespaces geben Sie an, um den Adressierungspfad zu Klassen, Daten oder Funktionen zu beschreiben, die in dem Namespace definiert sind (siehe Abschnitt „Namespaces“ auf Seite 89), wenn die gewünschte Komponente des Namespaces an der Unterbrechungsstelle nicht sichtbar ist.

Vor der Namespace-Qualifikation sind nur die E- oder S-Qualifikation oder die beiden Doppelpunkte (`::`) für den globalen Namespace möglich.

{ klasse | this-> | objekt }

ist der im Quellprogramm deklarierte Name einer Klasse, der `this`-Zeiger oder der Name eines Objekts einer Klasse.
Klassennamen, den `this`-Zeiger mit nachfolgendem Pointer-Operator sowie die Namen von Objekten von Klassen geben Sie an, um den Adressierungspfad zu Daten zu beschreiben, die Klassen zugeordnet sind (siehe Abschnitt „Klassen“ auf Seite 65).

Liegt die aktuelle Unterbrechungsstelle in einer dynamischen Member-Funktion, können Sie dynamische Daten-Member genauso ansprechen wie in C++, d.h. wenn das Datum an der Unterbrechungsstelle sichtbar ist, können Sie es mit AID direkt, also ohne Qualifikation, ansprechen. Lokal verdeckte Daten benötigen wie in C++ die entsprechende Klassenqualifikation. Außerdem können Sie dynamische Daten-Member über den `this`-Zeiger mit anschließendem Pointer-Operator ansprechen. Dies ist gleichbedeutend mit der Verwendung des Objektnamens und einem nachfolgenden Punkt.

Statische Daten-Member können Sie nur einzeln ansprechen. Sie erreichen sie von jeder Stelle des Programms über den zugehörigen Klassennamen mit den beiden nachfolgenden Doppelpunkten. Bei geschachtelten Klassen enthält der Pfad zum Datum alle Klassennamen von außen nach innen, jeweils durch zwei Doppelpunkte getrennt. Der äußerste Klassennamen benötigt die dem Gültigkeitsbereich entsprechende Qualifikation. Ist das Programm in einer Member-Funktion der Klasse unterbrochen, so gelten für das Ansprechen von statischen Daten-Member die Regeln des Klassenscopes, d.h. wenn das Daten-Member nicht durch eine gleichnamige Defintion verdeckt ist, so kann es unqualifiziert angesprochen werden.

Handelt es sich bei der Klasse um eine Instanz eines Klassentemplates, müssen Sie die folgende Schreibweise verwenden: `t'`*k_template*`<`*arg*`[,...]>'`. Existiert nur eine einzige Instanz des Templates, genügt die Angabe: `t'`*k_template*`'`.

Endet *sender* oder *empfänger* mit einem Objektnamen, so bezeichnen Sie damit, unabhängig von der aktuellen Unterbrechungsstelle, die dynamischen Daten und, falls vorhanden, vom Compiler generierte Hilfsvariablen und die Adresse der Tabelle der virtuellen Funktionen. Bei abgeleiteten Klassen umfasst *sender* oder *empfänger* auch die Basisklassen. Den gleichen Bereich bezeichnen Sie mit `*this`, falls das Programm in einer dynamischen Member-Funktion der Klasse unterbrochen ist. Um in einer abgeleiteten Klasse eine Basisklasse zu bezeichnen, geben Sie ausgehend vom Objektnamen bzw. von `this->` im Pfad den Namen der gewünschten Basisklasse an. Der mit *sender* oder *empfänger bezeichnete Bereich wird ohne Berücksichtigung der Einteilung in Komponenten übertragen bzw. linksbündig binär in der Länge von sender überschrieben.*
Falls *objekt* nicht zum Gültigkeitsbereich der Unterbrechungsstelle gehört, müssen Sie entsprechend qualifizieren. Vor `*this` ist nur eine Basisqualifikation sinnvoll.

Endet *sender* oder *empfänger* auf `this->`, so bezeichnen Sie damit 4 Bytes ab der Anfangsadresse des aktuellen Objekts.

datenname

ist der im Quellprogramm definierte Name eines Datums. *datenname* wird wie im Quellprogramm angegeben.
Daten können Sie bis auf folgende Ausnahmen wie in C/C++ ansprechen:
Mit einem Vektornamen ohne Index sprechen Sie alle Vektorelemente an. Vektorelemente können Sie nur über Index ansprechen, nicht über Zeiger; zum Arbeiten mit Vektoren siehe auch Abschnitt „Indexschreibweise“ auf Seite 31.

datenname kann folgendermaßen angegeben werden. Die Formate können auch kombiniert werden (siehe Abschnitt „Datennamen“ auf Seite 30):

Indexschreibweise:	*datenname* `[`*index*`]` `{...}`
Zeigerschreibweise:	*datenname1* `->` *datenname2*
Strukturqualifizierung:	*übergeordneter datenname*`.` `{...}` *datenname*
Dereferenzierung:	`[(]*{...}` *datenname*`[)]`

Eine gesamte Struktur können Sie übertragen, wenn Sie als *sender* den Strukturnamen angeben.
Einen ganzen Vektor können Sie übertragen, wenn Sie als *sender* einen Vektornamen ohne Index angeben. Eine Ausnahme bilden die Namen von formalen Vektorparametern; damit sprechen Sie nicht den Vektor, sondern nur seine Adresse an.

Wenn Sie einen Struktur- oder Vektornamen als *empfänger* angeben, wird die Struktur bzw. der Vektor, beginnend bei der Anfangsadresse, in der Länge von *sender* überschrieben, ohne dass die Unterteilung in Komponenten bzw. Elemente beachtet wird.

```
{funktion    }
{L'label'    }
{S'[f-]n[:a]'}
```

Anweisungsnamen und Source-Referenzen sind Adresskonstanten. Sie können nur als *sender* angegeben werden. Es wird die in der Adresskonstanten hinterlegte Adresse übertragen.
Mit nachfolgendem Pointer-Operator (->) bezeichnen Sie 4 Bytes des an der entsprechenden Adresse stehenden Maschinencodes. Die Maschinenbefehle können Sie mit %DISASSEMBLE ausgeben lassen, um eventuell eine Längenmodifikation vorzunehmen.
funktion`[%a14]->`, `L'`*label*`'->` und `S'[`*f*`-]`*n*`[:`*a*`]'->` können Sie als *sender* und als *empfänger* verwenden (siehe Beispiele 4 auf Seite 217 und 6 auf Seite 217).

funktion

ist der Name einer Funktion, der im Quellprogramm vergeben wurde bzw. der Name einer Bibliotheksfunktion. Sie bezeichnen damit die Anfangsadresse des vom Compiler erzeugten Prologs einer Funktion (siehe PROC=*funktion* auf Seite 207 und Kapitel „C++-spezifische Adressierung“ auf Seite 59).

Virtuelle Funktionen sprechen Sie mit der folgenden Syntax an:
`p->n'funktion([signatur])'`

p ist eine Zeigervariable, die auf das Objekt einer Klasse verweist, das die gewünschte Member-Funktion enthält. Wenn *p* von der aktuellen Unterbrechungsstelle aus nicht zu erreichen ist, muss dem Gültigkeitsbereich entsprechend qualifiziert werden. Liegt die Unterbrechungsstelle in der virtuellen Funktion selbst, dann können Sie die Prologadresse der aktuellen Funktion ansprechen, indem Sie statt *p* den `this`-Zeiger einsetze*n* (siehe Beschreibung von `this` auf Seite 66 und Abschnitt „Virtuelle Funktionen“ auf Seite 76).

Wenn Sie eine Funktion ansprechen wollen, die über Pointer-to-Member adressiert wird, stehen Ihnen die folgenden beiden Möglichkeiten zur Verfügung:

Das Objekt der Klasse, das die gewünschte Funktion enthält, wird mit seinem Namen bezeichnet. Als Dereferenzierungsoperator schreiben Sie `.*`

```
[qua.]objekt.*[objekt.][klasse::][...]pointer-to-function-member
```

Das Objekt der Klasse wird über einen Zeiger angesprochen. Als Dereferenzierungsoperator schreiben Sie **->***

```
[qua.]zeiger->*[objekt.][klasse::][...]pointer-to-function-member
```

Das Objekt der Klasse bezeichnen Sie mit dem Operanden, der auf der linken Seite der Dereferenzierungsoperatoren `.*` bzw. **->*** steht:
mit *objekt* bezeichnen Sie das Objekt der Klasse mit seinem Namen,
mit *zeiger* adressieren Sie das Objekt über einen Zeiger.
Rechts der Deferenzierungsoperatoren schreiben Sie den Namen des Pointer-to-Function-Member. Evtl. müssen Sie das Objekt, das die Definition des Pointer-to-Function-Member enthält, und innerhalb des Objekts die zur eindeutigen Ansprache nötige Klassenqualifikation davorschreiben, falls der Pointer-to-Member von der Unterbrechungsstelle aus anders nicht zu erreichen ist.

Wenn Sie mit %MOVE den Befehlscode einer über Pointer-to-Member angesprochenen Funktion übertragen oder überschreiben wollen, so müssen Sie beachten, dass Sie nicht unmittelbar an eine der obigen Syntaxen den Pointer-Operator anfügen können. Sie müssen vielmehr zunächst durch eine Typmodifikation, nämlich %al4, den Übergang auf die Maschinencode-Ebene auslösen. Es ergibt sich die folgende Syntax:

dereferenzierter-pointer-to-function-member `%al4->`

Sie bezeichnen damit die ersten 4 Bytes des Befehlscodes, der an der Prologadresse steht.

Näheres zum Arbeiten mit Pointer-to-Function-Member finden Sie auf Seite 83.

L'label'
: bezeichnet die Adresse der ersten ausführbaren Anweisung nach einer Marke. *label* ist der Name einer Marke, der im Quellprogramm vergeben wurde.

S'[f-]n[:a]'
: ist eine Source-Referenz und bezeichnet die Adresse einer ausführbaren Anweisung. Sie wird aus Zeilennummer (*n*) und gegebenenfalls der FILE-Nummer (*f*) sowie der relativen Anweisungsnummer innerhalb der Zeile (*a*) gebildet. Liegt die Source-Referenz in einer Funktion, die durch Instanziierung aus einem Funktionstemplate hervorgegangen ist, oder ist die Funktion, die die Source-Referenz enthält, in einer Instanz eines Klassentemplates definiert, so müssen Sie bei Mehrdeutigkeit vor die Source-Referenz die entsprechende PROC-Qualifikation schreiben.

schlüsselwort
: ist ein Durchlaufzähler, der Befehlszähler oder ein Register. Vor *schlüsselwort* können Sie nur eine Basisqualifikation angeben.

```
%•subkdoname        Durchlaufzähler
%•                  Durchlaufzähler des gerade aktiven Subkommandos
%PC                 Befehlszähler (Program Counter)
%n                  Mehrzweckregister, 0 ≤ n ≤ 15
%nD|E               Gleitpunktregister, n = 0,2,4,6
%nQ                 Gleitpunktregister, n = 0,4
%nG                 AID-Mehrzweckregister, 0 ≤ n ≤ 15
%nGD                AID-Gleitpunktregister, n = 0,2,4,6
```

kompl-speicherref

Folgende Operationen können in *kompl-speicherref* vorkommen (siehe AID-Basishandbuch, Abschnitt „Komplexe Speicherreferenzen“ [1]):

– Adressversatz (•)
– indirekte Adressierung (->)
– Typmodifikation (%A, %E, %S, %SX)
– Längenmodifikation (%L(...), %L=(ausdruck), %Ln)
– Adressselektion (%@(...))

Beginnt eine *kompl-speicherref* mit einer Adresskonstanten (z.B. eine Source-Referenz, eine Marke oder ein Funktionsname), muss anschließend der Pointer-Operator geschrieben werden. In diesem Fall muss eine Marke stets in `L'...'` geschrieben werden. Ohne den Pointer-Operator können Adresskonstanten innerhalb der *kompl-speicherref* überall da stehen, wo auch Sedezimalzahlen geschrieben werden können.

Nach Adressversatz (•) oder Pointer-Operation (->) gehen impliziter Speichertyp und implizite Länge der Ausgangsadresse verloren. An der errechneten Adresse gilt beim Sender der Speichertyp %X in der Länge 4, falls Sie Typ und Länge nicht explizit angeben. Geben Sie als Empfänger eine komplexe Speicherreferenz an, so reicht der Bereich, der überschrieben werden kann, von der Anfangsadresse von *kompl-speicherref* an bis zum Ende des von Ihrem Programm belegten Speichers. Jedoch können Sie mit einem %MOVE nie mehr als 3900 Bytes übertragen. Eine abschließende Typmodifikation ist bei *kompl-speicherref* sinnlos, da mit %MOVE stets binär übertragen wird. Allerdings kann eine Typmodifikation vor einer Pointer-Operation (->) notwendig sein.

Beispiel: `%3g.2%al2->`
Die letzten beiden Bytes des AID-Registers %3G werden als Adresse benutzt.

Für keinen Operanden in einer komplexen Speicherreferenz darf der zugeordnete Speicherbereich durch einen Adressversatz oder eine Längenmodifikation überschritten werden, sonst führt AID das Kommando nicht aus und schreibt eine Fehlermeldung. Durch die Verbindung von Adressselektion (%@) mit Pointer-Operation (->) verlassen Sie die symbolische Ebene. Nun können Sie die Adresse eines Datums verwenden, ohne auf dessen Bereichsgrenzen achten zu müssen.

Beispiel
Die Vektoren `name` und `name1` sind vom Typ char und belegen je 5 Bytes. Die letzten 2 Bytes von `name` sowie die 3 anschließenden Bytes sollen nach `name1` übertragen werden.

```
%move name.3%l5 into name1
```

Dieses Kommando wird von AID wegen Bereichsverletzung von `name` abgelehnt. Richtig müsste das Kommando lauten:

```
%move %@(name)->.3%l5 into name1
```

& ist der Adressoperator. Damit können Sie die Anfangsadresse eines Datums, eines Objekts einer Klasse oder einer Funktion als *sender* verwenden.

Außerdem besteht die Möglichkeit, mit dem Adressoperator & die relative Adresse eines dynamischen Daten-Members einer Klasse zu ermitteln. Dabei ist Folgendes zu beachten:

– Wenn die Unterbrechungsstelle außerhalb der Klasse liegt, die das Daten-Member enthält, schreiben Sie nach dem Adressoperator die entsprechende Klassenqualifikation und dann den Namen des Datums.

– Liegt die Unterbrechungsstelle dagegen in einer dynamischen Member-Funktion der Klasse, müssen Sie vor den Adressoperator eine Basis- oder Bereichsqualifikation (S-, PROC- oder ::-Qualifikation) setzen, damit AID gewissermaßen von außen auf die Klasse zugreift.

Der Adressoperator ist im Gegensatz zum Adressselektor %@(...) (siehe Seite 277) eine reine High-Level-Funktion; daher kann er auf komplexe Speicherreferenzen nicht angewandt werden.

Weitergehende Informationen zum Adressoperator finden Sie im Abschnitt „Adressoperator & und Adressselektor %@(...)“ auf Seite 44.

sizeof()

ist der Längenoperator. Die Länge eines Datums oder einer Klasse wird übertragen.
Um die Länge einer Klasse zu ermitteln, können Sie sowohl den Namen der Klasse selbst als auch den Namen eines Objekts der Klasse als Operanden angeben. Sie erhalten die Anzahl Bytes, die die dynamischen Daten-Member der Klasse und eventuell vom Compiler generierte Hilfsvariablen belegen.
Den Namen eines Namespaces können Sie hier nur im Pfad zu einer Komponente des Namespaces angegeben.

Bitfelder und Registervariablen können Sie hier nicht angeben.

Ausführlich beschrieben ist der Längenoperator im Abschnitt „Längenoperator sizeof() und Längenselektor %L(...)“ auf Seite 49.

%@(...)

Mit dem Adressselektor können Sie die Anfangsadresse eines Datums, des Objekts einer Klasse oder einer komplexen Speicherreferenz als *sender* verwenden.
Einen Klassennamen können Sie nur im Pfad zur Basisklasse eines Objekts einer abgeleiteten Klasse angeben; damit bezeichnen Sie die Anfangsadresse der dynamischen Daten der Basisklasse.
Den Namen eines Namespaces können Sie hier nur im Pfad zu einer Komponente des Namespaces angegeben.

Der Adressselektor liefert als Ergebnis eine Adresskonstante (siehe AID-Basishandbuch, Abschnitt „Adress-, Typ- und Längenselektor“ [1]).
Der Adressselektor lässt sich nicht auf Konstanten anwenden, dazu gehören auch die Marken, die Source-Referenzen und die Funktionen.

%L(...)

Mit dem Längenselektor können Sie die Länge eines Datums oder einer Klasse als *sender* verwenden. Wenn Sie den Längenselektor auf eine Klasse oder ein Objekt einer Klasse anwenden, entspricht das Ergebnis dem von sizeof() in C++; Sie erhalten also die Länge der dynamischen Daten und eventuell vom Compiler generierter Hilfsvariablen.
Den Namen eines Namespaces können Sie hier nur im Pfad zu einer Komponente des Namespaces angegeben.

Der Längenselektor liefert als Ergebnis eine Ganzzahl (siehe AID-Basishandbuch, Abschnitt „Komplexe Speicherreferenzen“ [1]). Für Bitfelder wird als Länge die Anzahl der Bytes ermittelt, über die sich das Bitfeld erstreckt.
Beispiel: `%move %l(var1) into %3g`
Die Länge von `var1` wird übertragen.

%L=(ausdruck)

Mit der Längenfunktion können Sie einen Wert errechnen und in *empfänger* abspeichern lassen. In *ausdruck* können Sie Speicherinhalte, Konstanten und Ganzzahlen mit den arithmetischen Operatoren (+,-,*,/) verknüpfen. Nur ganzzahlige Speicherreferenz-Inhalte (Typ %F oder %A) sind zugelassen. Die Längenfunktion liefert als Ergebnis eine Ganzzahl (siehe AID-Basishandbuch, Abschnitt „Längenmodifikation“ [1]).

Falls Sie in Ihrem Programm mit überladenen Operatoren arbeiten, müssen Sie beachten, dass AID das Überladen der Operatoren nicht nachvollzieht, sondern stets die Standard-Operatoren verwendet.

Beispiel: `%move %l=(var1) into %3g`
Der Inhalt von `var1` wird übertragen, wenn er ganzzahlig ist (Datentyp int). Sonst gibt AID eine Fehlermeldung aus.

AID-Literal

Die folgenden AID-Literale (siehe AID-Basishandbuch, Kapitel „AID-Literale“ [1]) können mit %MOVE übertragen werden:

```
{C'x...x' | 'x...x'C | 'x...x'}    Character-Literal
{X'f...f' | 'f...f'X}              Sedezimal-Literal
{B'b...b' | 'b...b'B}              Binär-Literal
```

```
[{±}]n                          Ganzzahl
#'f...f'                        Sedezimalzahl
```

REP

gibt an, ob AID zu einer durchgeführten Änderung einen REP-Satz erzeugen soll. Mit *REP* setzen Sie eine mit dem Kommando %AID getroffene Vereinbarung vorübergehend außer Kraft. Wird *REP* nicht angegeben, und gibt es keine gültige Vereinbarung im %AID-Kommando, so wird kein REP-Satz erstellt.

```
rep-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

REP = {Y[ES] | NO}

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

REP=Y[ES]
: Zu der durch das aktuelle %MOVE-Kommando durchgeführten Änderung werden LMS-Korrekturanweisungen (REPs) im SDF-Format erstellt. Wenn die Objekt-Strukturliste nicht zur Verfügung steht, erstellt AID keine Korrekturanweisungen und gibt eine Fehlermeldung aus.

  Auch wenn *empfänger* nicht vollständig innerhalb einer CSECT liegt oder *sender* länger als 3900 Bytes ist, gibt AID eine Fehlermeldung aus und schreibt keinen REP. Um dennoch REP-Sätze zu erhalten, müssen Sie die Übertragung auf mehrere %MOVE-Kommandos verteilen (siehe Handbuch „Testen auf Maschinencode-Ebene“ [2]).

  AID hinterlegt die Korrekturen mit den nötigen LMS-Korrekturanweisungen in einer Datei mit dem Linknamen F6. Für den LMS-Lauf muss dann noch die `MODIFY-ELEMENT`-Anweisung eingefügt werden. Sie sollten darauf achten, dass Sie in die Datei mit dem Linknamen F6 keine anderen Ausgaben schreiben lassen.

  Ist keine Datei mit dem Linknamen F6 angemeldet (siehe %OUTFILE), so wird der REP in der von AID angelegten Datei `AID.OUTFILE.F6` abgelegt.

REP=NO
: Zum aktuellen %MOVE-Kommando werden keine REPs erstellt.

Beispiele

In einem C-Programm sind die folgenden Variablen und Vektoren definiert:

```
C-Programm
==========================================
      int           i_arr_1[10];
      int           i_arr_2[20];
      unsigned long number;
      float         x_arr[10];
==========================================
```

1. `%move i_arr_1 into i_arr_2`

   Bei beiden Vektoren ist kein Index angegeben; AID überträgt daher den gesamten Vektor `i_arr_1`, ohne die Unterteilung in einzelne Elemente zu beachten, sedezimal linksbündig nach `i_arr_2`.

2. `%move 20 into number`

   In die Variable `number`, die auch im C-Programm 4 Bytes belegt, schreibt AID ein Wort, das den Wert 20 (X'00000014') enthält.

3. `%move 20 into x_arr[5]`

   Achtung: Wie im Beispiel 2 wird auch hier ein Wort mit dem Inhalt X'00000014' nach `x_arr[5]` geschrieben, was natürlich bei einem Vektorelement vom Typ float keinen Sinn ergibt. Um den Wert 20 nach `x_arr[5]` zu übertragen, müssten Sie ein %SET-Kommando eingeben (siehe %SET auf Seite 265), das vor der Übertragung eine Konvertierung durchführt.

4. `%move x'58f0c160' into s'21:2'->.16 rep=yes`

   Der zur Anweisung S'21:2' erzeugte Code wird geändert: ab der 16. Stelle nach der Anfangsadresse von S'21:2' werden 4 Bytes mit dem Sedezimal-Literal X'58F0C160' überschrieben. Zur Korrektur wird ein REP erstellt und in der Datei AID.OUTFILE.F6 bzw. der dem Linknamen F6 zugewiesenen Datei abgelegt.

5. `%move s'12' into %2g`

   Die Adresse der ersten Anweisung in Zeile 12 wird in das AID-Register %2G geschrieben.

6. `%move s'12'->%l=(s'13'-s'12') into Y::n'f()'->.#'20'`

   Der zur Anweisung S'12' erzeugte Maschinencode wird übertragen. Die Angabe `%l=(s'13'-s'12')` bestimmt die Länge dieser Anweisung. In dieser Länge wird der Maschinencode zur Member-Funktion `Y::f()` überschrieben, und zwar ab der Adresse, die sich aus der Prologadresse und dem Adressversatz (`#'20'` = 32 Bytes) ergibt.

7. `%move ::a into *this.4`

   Das Programm wurde in einer dynamischen Funktion einer Klasse unterbrochen. Mit dem %MOVE übertragen Sie den Inhalt der globalen Variablen `a` in das aktuelle Objekt, dem die Funktion zugeordnet ist. Das Objekt wird ab dem fünften Byte überschrieben.

8. `%move ::A::j into z.X.4`

   Der %MOVE überträgt die statische Variable `j` aus der globalen Klasse `A` linksbündig binär ab der fünften Stelle in die Basisklasse `X` des Objekts `z`.

%ON

Mit %ON legen Sie Ereignisse fest und definieren Subkommandos. Wenn ein ausgewähltes *ereignis* eintritt, wird das zugehörige *subkdo* von AID bearbeitet.

- Mit *write-ereignis* beschreiben Sie das Ereignis des schreibenden Zugriffs auf einen Speicherbereich. Immer wenn das Programm den angegebenen Speicherbereich verändert, soll AID den Programmablauf unterbrechen, um *subkdo* zu bearbeiten.
- Mit *ereignis* beschreiben Sie eines der anderen Ereignisse (normale oder abnormale Programmbeendigung, Supervisor-Call (SVC), Programmfehler, etc.), bei dem AID den Programmablauf unterbrechen soll, um *subkdo* zu bearbeiten.
- Mit *subkdo* definieren Sie ein Kommando oder eine Kommandofolge und eventuell eine Bedingung. Bei zutreffendem *ereignis* und erfüllter Bedingung wird *subkdo* ausgeführt.

Kommando	Operand	
%ON	{write-ereignis / ereignis}	[<subkdo>]

Ohne *subkdo*-Operanden setzt AID das *subkdo* <%STOP> ein.

Das *subkdo* eines %ON für ein bereits angemeldetes *ereignis* überschreibt nicht das bestehende *subkdo*, sondern das neue *subkdo* wird vor das bestehende gekettet. Das bedeutet, dass gekettete Subkommandos nach dem LIFO-Prinzip abgearbeitet werden. Dies gilt nicht für *write-ereignis*. Die Eingabe von einem neuen *write-ereignis* überschreibt ein bereits bestehendes.

Ein eingetragenes Ereignis gilt, bis es mit %REMOVE gelöscht wird oder bis zum Programmende. Außerdem sind alle Vereinbarungen mit %ON nach einem `fork()`-Aufruf sowie in einem Programm, das durch `exec()` geladen wurde, zurückgesetzt.

Für %ON muss die Basisqualifikation E=VM eingestellt sein (siehe Beschreibung des Kommandos %BASE).

%ON verändert den Programmzustand nicht.

write-ereignis

Mit dem Schlüsselwort %WRITE schalten Sie die Schreibüberwachung ein. Dahinter setzen Sie in Klammern den zu überwachenden Speicherbereich. Wenn das Programm ein Byte innerhalb des festgelegten Bereichs verändert, wird der Programmablauf unterbrochen und *subkdo* ausgeführt. Die Unterbrechung erfolgt nach dem Befehl, der die Speicherstelle verändert hat.

Es kann immer nur ein *write-ereignis* definiert sein. Die Eingabe von einem neuen *write-ereignis* überschreibt ein bestehendes. Andere Ereignisse können jedoch gleichzeitig angemeldet sein. Trifft ein *ereignis* gleichzeitig mit *write-ereignis* ein, so bearbeitet AID das Subkommando zu *write-ereignis* als erstes.
Das *write-ereignis* löschen Sie mit `%REMOVE %WRITE` ohne Angabe der Speicherreferenz.

Folgende Wechselwirkungen bestehen zwischen %ON *write-ereignis* und anderen AID-Kommandos:

– Wenn ein %CONTROL*n* oder ein %TRACE mit maschinennahem *kriterium* angemeldet ist, wird die Eingabe von %ON *write-ereignis* mit einer Fehlermeldung abgewiesen.

– Wenn ein Maschinenbefehl durch einen %CONTROL*n* oder %TRACE mit symbolischem *kriterium* mit der AID-internen Markierung (X'0A81') überschrieben wurde, bemerkt AID den schreibenden Zugriff dieses Befehls nicht.

– Wenn ein Maschinenbefehl durch den mit %INSERT vereinbarten Testpunkt mit der AID-internen Markierung überschrieben wurde, erkennt AID auch hier den schreibenden Zugriff dieses Befehls nicht.

Für eine lückenlose Schreibüberwachung empfiehlt es sich, alle %CONTROL*n*- und %INSERT-Kommandos mit %REMOVE zu löschen und einen eventuell noch eingetragenen %TRACE zu löschen, indem Sie nach dem %ON mit %RESUME fortfahren.

Der zu überwachende Speicherbereich kann jedes, wie auch immer angesprochene Speicherobjekt sein. Er wird durch Anfangsadresse und implizite oder explizite Länge festgelegt. Der Bereich kann maximal 64KB lang sein, sonst wird eine Fehlermeldung ausgegeben.

Wenn bei einem Programm mit Überlagerungsstruktur (Overlay) die Adresse des angegebenen Speicherobjekts überladen wird, wird der entsprechende Bereich des neu geladenen Programmteils überwacht.

- Steht der Punkt an führender Stelle, ist er das Kennzeichen für eine *vorqualifikation*. Sie muss mit einem vorhergehenden %QUALIFY definiert worden sein. Aufeinanderfolgende Qualifikationen werden durch einen Punkt getrennt. Außerdem muss zwischen der letzten Qualifikation und dem anschließenden Operandenteil ein Punkt stehen.

qua Eine oder mehrere Qualifikationen geben Sie an, wenn der zu überwachende Bereich von der aktuellen Unterbrechungsstelle aus anders nicht zu erreichen ist oder um einen Datennamen anzusprechen, der an der Unterbrechungsstelle durch eine gleichnamige Definition lokal verdeckt ist. Sie geben nur die Qualifikationen an, die zur eindeutigen Ansprache des Speicherobjekts genügen.

S=srcname
ist nur erforderlich, wenn der zu überwachende Speicherbereich nicht in der aktuellen Übersetzungseinheit liegt (Abschnitt „Qualifikationen“ auf Seite 21).

:: Die beiden vorangestellten Doppelpunkte verwenden Sie, um ein globales Datum anzusprechen, das durch eine gleichnamige Definition an der Unterbrechungsstelle lokal verdeckt ist. Außerdem müssen Sie die beiden Doppelpunkte vor den Namen eines globalen Datums oder einer Funktion setzen, weil entweder das Datum oder die Funktion nicht in der Aufrufhierarchie liegen oder weil deren Definition erst nach der Unterbrechungsstelle steht.
Im Gegensatz zu den übrigen Qualifikationen wird zwischen den beiden Doppelpunkten und dem anschließenden Operanden kein Punkt geschrieben.

PROC=funktion

geben Sie nur an, wenn Sie einen Datennamen ansprechen wollen, der zwar in der aktuellen Funktion definiert ist, aber von einer gleichnamigen Definition an der Unterbrechungsstelle verdeckt wird. Außerdem geben Sie eine PROC-Qualifikation an, wenn Sie eine Marke oder einen static vereinbarten Datennamen ansprechen wollen, der einer Funktion außerhalb der aktuellen Aufrufhierarchie zugeordnet ist (siehe Kapitel „Adressierung in C- und C++-Programmen“ auf Seite 21). Wenn Sie eine Source-Referenz angeben, die in einer Instanz eines Funktionstemplates liegt oder die einer Funktion zugeordnet ist, die in einer Instanz eines Klassentemplates definiert ist (siehe Abschnitt „Templates“ auf Seite 98), müssen Sie bei Mehrdeutigkeit ebenfalls die entsprechende PROC-Qualifikation davorschreiben.

Bei Funktionen aus C-Programmen ist *funktion* der im Quellprogramm vergebene Name der Funktion ohne Klammern und ohne Signatur.

Funktionen aus C++-Programmen müssen Sie in der Schreibweise `n'...'` oder `t'...'` angeben, je nachdem um welchen Typ es sich handelt. Ist die Funktion in einem Namespace oder in einer Klasse definiert, so wird dem Funktionsnamen die Namespace- oder Klassen-Qualifikation vorangestellt. Die Signatur `void` darf nicht mehr geschrieben werden. Wie auch in C++ möglich geben Sie in diesem Fall nur die beiden Klammern nach dem Funktionsnamen an. Es ergibt sich die folgende Syntax (*f_template* und *signatur* sind in der Syntax aus Platzgründen abgekürzt):

```
PROC=[namespace::[...]][klasse::[...]]{n'funktion([sign])'          }
                                      {t'f_templ<arg[,...]>([sign])'}
```

Abweichend davon werden die Funktionen `main` und `_ _STI_ _` sowie alle Funktionen mit C-Linkage auch beim Testen in C++-Programmen nur mit dem Funktionsnamen bezeichnet (siehe Seite 60).

Ist die Funktion in einer lokalen Klasse definiert, so müssen Sie vor der PROC-Qualifikation noch eine weitere PROC-Qualifikation für die übergeordnete Funktion angeben, die die Definition der lokalen Klasse enthält. Zwischen die beiden PROC-Qualifikationen schreiben Sie bei Funktionen, die in einem inneren Block der übergeordneten Funktion definiert sind, eine oder ggfs. mehrere BLK-Qualifikationen (siehe Seite 62).

Das Ansprechen von Funktionen, die in lokalen Klassen innerer Blöcke definiert sind, wird nur bei Programmen unterstützt, die mit einem C/C++-Compiler ab V3.0B übersetzt wurden.

BLK='[f-]n[:b]'

geben Sie nur an, wenn Sie einen Datennamen ansprechen wollen, der einem Block innerhalb der aktuellen Aufrufhierarchie zugeordnet ist und von einer gleichnamigen Definition an der Unterbrechungsstelle verdeckt wird oder wenn Sie einen static vereinbarten Datennamen ansprechen wollen, der einem Block außerhalb der aktuellen Aufrufhierarchie zugeordnet ist (siehe Kapitel „Adressierung in C- und C++-Programmen“ auf Seite 21).

Des weiteren geben Sie eine BLK-Qualifikation an, wenn Sie mit einer nachfolgenden PROC-Qualifikation eine Funktion aus einer lokalen Klasse bezeichnen und die Definition dieser lokalen Klasse steht im angegebenen Block (siehe oben PROC=*funktion*).

Der Name für einen Block wird aus Zeilennummer (*n*) und gegebenenfalls FILE-Nummer (*f*) und relativer Blocknummer (*b*) gebildet

namespace

ist der im Quellprogramm deklarierte Name eines Namespaces.

Den Namen eines Namespaces können Sie nur verwenden, um den Adressierungspfad zu Klassen, Daten oder Funktionen zu beschreiben, die in dem Namespace definiert sind, falls die gewünschte Komponente des Namespaces an der Unterbrechungsstelle nicht sichtbar ist (siehe Abschnitt „Namespaces“ auf Seite 89).

Vor der Namespace-Qualifikation sind nur die E- oder S-Qualifikation oder die beiden Doppelpunkte (`::`) für den globalen Namespace möglich.

{ klasse | this-> | objekt }

ist der im Quellprogramm deklarierte Name einer Klasse, der `this`-Zeiger oder der Name eines Objekts einer Klasse.
Klassennamen, den `this`-Zeiger mit nachfolgendem Pointer-Operator sowie die Namen von Objekten von Klassen geben Sie an, um den Adressierungspfad zu Daten-Membern zu beschreiben (siehe Abschnitt „Klassen“ auf Seite 65).

Liegt die aktuelle Unterbrechungsstelle in einer dynamischen Member-Funktion, können Sie die Daten der Klasse gemäß den aus C++ bekannten Scoperegeln ansprechen.

Unabhängig von der Unterbrechungsstelle erreichen Sie die dynamischen Daten eines Objekts über den Objektnamen mit nachfolgendem Punkt, falls das Objekt in der aktuellen Aufrufhierarchie liegt.

Statische Daten-Member können Sie nur einzeln ansprechen. Sie erreichen sie von jeder Stelle des Programms über den zugehörigen Klassennamen mit den beiden nachfolgenden Doppelpunkten. Bei geschachtelten Klassen enthält der Pfad zum

Datum alle Klassennamen von außen nach innen, jeweils durch zwei Doppelpunkte getrennt. Der äußerste Klassennamen benötigt die dem Gültigkeitsbereich entsprechende Qualifikation. Ist das Programm in einer Member-Funktion der Klasse unterbrochen, so gelten für das Ansprechen von statischen Daten-Member die Regeln des Klassenscopes, d.h. wenn das Daten-Member nicht durch eine gleichnamige Defintion verdeckt ist, so kann es unqualifiziert angesprochen werden.

Handelt es sich bei der Klasse um eine Instanz eines Klassentemplates, müssen Sie die folgende Schreibweise verwenden: `t'`*k_template*`<`*arg*`[,...]>'`. Existiert nur eine einzige Instanz des Templates, genügt die Angabe: `t'`*k_template*`'`.

Endet der Bereichsoperand mit einem Objektnamen, so bezeichnen Sie damit, unabhängig von der aktuellen Unterbrechungsstelle, die dynamischen Daten und, falls vorhanden, vom Compiler generierte Hilfsvariablen und die Adresse der Tabelle der virtuellen Funktionen. Bei abgeleiteten Klassen umfasst der zu überwachende Bereich auch die Basisklassen. Gleichbedeutend dazu ist die Verwendung von `*this`, falls das Programm in einer dynamischen Member-Funktion der Klasse unterbrochen ist. Um in einer abgeleiteten Klasse eine Basisklasse zu bezeichnen, geben Sie ausgehend vom Objektnamen bzw. von `this->` im Pfad den Namen der gewünschten Basisklasse an. Sie müssen nur die Klassennamen angeben, die zur eindeutigen Ansprache des gewünschten Members erforderlich sind.
Falls *objekt* an der Unterbrechungsstelle nicht sichtbar ist, müssen Sie entsprechend qualifizieren. Vor `this` ist nur eine Basisqualifikation sinnvoll.
Endet der Bereichsoperand auf `this->`, so bezeichnen Sie damit 4 Bytes ab der Anfangsadresse des aktuellen Objekts.

datenname

ist der im Quellprogramm definierte Name eines Datums. *datenname* wird wie im Quellprogramm angegeben.
Daten können Sie bis auf folgende Ausnahmen wie in C/C++ ansprechen:
Mit einem Vektornamen ohne Index sprechen Sie alle Vektorelemente an.
Vektorelemente können Sie nur über Index ansprechen, nicht über Zeiger.
Indexbereiche können nicht überwacht werden.
Es ist zu beachten, dass AID zu einer indizierten Angabe in einem `%ON %WRITE(...)` sofort bei der Eingabe Anfangsadresse und Länge des zu überwachenden Bereichs berechnet. Wenn sich also während des Programmablaufs der Wert des Index ändert und sich daraus eine Änderung der Anfangsadresse des mit *datenname*`[`*index*`]{...}` bezeichneten Bereichs ergibt, so wird mit `%ON %WRITE(...)` weiterhin der bei der Eingabe des Kommandos gültige Bereich überwacht.
Wenn `%AID C=YES` gesetzt ist (siehe Seite 119), fasst AID die Vektorelemente eines `char`-Vektors, die über den am weitesten rechts stehenden Index adressiert werden, zu C-Strings zusammen. Wird ein solcher C-String angegeben, so überwacht `%ON %WRITE(...)` den gesamten, dem C-String zugrunde liegenden Vektor

und nicht nur den C-String bis zum Endekriterium X'00'.
Vektoren, die als Parameter an eine Funktion übergeben wurden, sind dort als Zeiger auf den Vektor im aufrufenden Programm realisiert. Mit `%ON %WRITE(...)` wird dann dieser Zeiger überwacht, aber nicht der zugehörige Vektor.
Zum Arbeiten mit Vektoren siehe auch „Indexschreibweise“ auf Seite 31 und „C-Strings“ auf Seite 37.

Wenn Sie im %ON %WRITE(...) ein Daten-Member angeben, das über Pointer-to-Member referenziert wird, so überwacht AID stets den bei der Eingabe bezeichneten Speicherbereich, auch wenn sich im weiteren Verlauf des Programms die im Pointer-to-Member eingetragene Adresse ändert und der Pointer-to-Member mittlerweile auf ein anderes Daten-Member verweist.

datenname kann folgendermaßen angegeben werden. Die Formate können auch kombiniert werden (siehe Abschnitt „Datennamen“ auf Seite 30).

Indexschreibweise:	*datenname* `[`*index*`]` `{...}`
Zeigerschreibweise:	*datenname1* `->` *datenname2*
Strukturqualifizierung:	*übergeordneter datenname*`.` `{...}` *datenname*
Dereferenzierung:	`[(]*{...}` *datenname*`[)]`
Pointer-to-Member-Dereferenzierung:	*datenname1*`.*`*datennname2* oder *datenname1*`->*`*datenname2*

```
{funktion[%al4]}
{L'label'      }->
{S'[f-]n[:a]'  }
```

bezeichnet 4 Bytes des Maschinencodes ab der Adresse, die in einer der Adresskonstanten hinterlegt ist. Soll eine andere Anzahl von Bytes überwacht werden, müssen Sie eine entsprechende Längenmodifikation angeben.

funktion
: ist der Name einer Funktion, der im Quellprogramm vergeben wurde bzw. der Name einer Bibliotheksfunktion. Sie bezeichnen damit die Anfangsadresse des vom Compiler erzeugten Prologs einer Funktion (siehe PROC=*funktion* auf Seite 222 und Kapitel „C++-spezifische Adressierung“ auf Seite 59).

  Virtuelle Funktionen sprechen Sie mit der folgenden Syntax an:
  `p->n'funktion([signatur])'`

  p ist eine Zeigervariable, die auf das Objekt einer Klasse verweist, das die gewünschte Member-Funktion enthält. Wenn *p* von der aktuellen Unterbrechungsstelle aus nicht zu erreichen ist, muss dem Gültigkeitsbereich entsprechend qualifiziert werden. Liegt die Unterbrechungsstelle in der virtuellen Funk-

tion selbst, dann können Sie die Prologadresse der aktuellen Funktion ansprechen, indem Sie statt *p* den `this`-Zeiger einsetze*n* (siehe Beschreibung von `this` auf Seite 66 und Abschnitt „Virtuelle Funktionen“ auf Seite 76).

Wenn Sie eine Funktion ansprechen wollen, die über Pointer-to-Member adressiert wird, stehen Ihnen die folgenden beiden Möglichkeiten zur Verfügung:

Das Objekt der Klasse, das die gewünschte Funktion enthält, wird mit seinem Namen bezeichnet. Als Dereferenzierungsoperator schreiben Sie `.*`

```
[qua.]objekt.*[objekt.][klasse::][...]pointer-to-function-member
```

Das Objekt der Klasse wird über einen Zeiger angesprochen. Als Dereferenzierungsoperator schreiben Sie **->***

```
[qua.]zeiger->*[objekt.][klasse::][...]pointer-to-function-member
```

Das Objekt der Klasse bezeichnen Sie mit dem Operanden, der auf der linken Seite der Dereferenzierungsoperatoren `.*` bzw. **->*** steht:
mit *objekt* bezeichnen Sie das Objekt der Klasse mit seinem Namen,
mit *zeiger* adressieren Sie das Objekt über einen Zeiger.
Rechts der Deferenzierungsoperatoren schreiben Sie den Namen des Pointer-to-Function-Member. Evtl. müssen Sie das Objekt, das die Definition des Pointer-to-Function-Member enthält, und innerhalb des Objekts die zur eindeutigen Ansprache nötige Klassenqualifikation davorschreiben, falls der Pointer-to-Member von der Unterbrechungsstelle aus anders nicht zu erreichen ist.

Wenn Sie mit %ON %WRITE(...) den Befehlscode einer über Pointer-to-Member angesprochenen Funktion überwachen wollen, so müssen Sie beachten, dass Sie nicht unmittelbar an eine der obigen Syntaxen den Pointer-Operator anfügen können. Sie müssen vielmehr zunächst durch eine Typmodifikation, nämlich %al4, den Übergang auf die Maschinencode-Ebene auslösen. Es ergibt sich die folgende Syntax:

dereferenzierter-pointer-to-function-member `%al4->`

Sie bezeichnen damit die ersten 4 Bytes des Befehlscodes, der an der Prologadresse steht.

Näheres zum Arbeiten mit Pointer-to-Function-Member finden Sie auf Seite 83.

L'label'
: bezeichnet die Adresse der ersten ausführbaren Anweisung nach einer Marke. *label* ist der Name einer Marke, der im Quellprogramm vergeben wurde.

S'[f-]n[:a]'
: ist eine Source-Referenz und bezeichnet die Adresse einer ausführbaren Anweisung. Sie wird aus Zeilennummer (*n*) und gegebenenfalls der FILE-Nummer (*f*) sowie der relativen Anweisungsnummer innerhalb der Zeile (*a*) gebildet. Liegt die Source-Referenz in einer Funktion, die durch Instanziierung aus einem Funktionstemplate hervorgegangen ist, oder ist die Funktion, die die Source-Referenz enthält, in einer Instanz eines Klassentemplates definiert, so müssen Sie bei Mehrdeutigkeit vor die Source-Referenz die entsprechende PROC-Qualifikation schreiben.

kompl-speicherref
: Folgende Operationen können in *kompl-speicherref* vorkommen (siehe AID-Basishandbuch, Abschnitt „Komplexe Speicherreferenzen" [1]):
  – Adressversatz (•)
  – indirekte Adressierung (->)
  – Typmodifikation (%A, %S, %SX)
  – Längenmodifikation (%L(...), %L=(ausdruck), %Ln)
  – Adressselektion (%@(...))

  kompl-speicherref bezeichnet einen Bereich von 4 Bytes ab der errechneten Adresse. Soll eine andere Anzahl von Bytes überwacht werden, muss *kompl-speicherref* mit der entsprechenden Längenmodifikation enden. Bei der Längenmodifikation von Daten müssen Sie die Bereichsgrenzen beachten oder mit %@(...)-> auf Maschinencode-Ebene wechseln. Beginnt *kompl-speicherref* mit einem Funktionsnamen, einer Marke oder einer Source-Referenz, muss anschließend der Pointer-Operator (->) geschrieben werden. Eine Marke muss in diesem Fall mit `L'...'` angegeben werden. Ohne den Pointer-Operator können Funktionsnamen, Marken und Source-Referenzen überall da verwendet werden, wo auch Sedezimalzahlen geschrieben werden können.

ereignis

Ein Schlüsselwort legt fest, bei welchem Ereignis (Programmfehler, Programmbeendigung, Supervisor-Call etc.) AID das angegebene *subkdo* bearbeiten soll. Auf einen Ereigniscode, der mit einer STXIT-Routine bearbeitet wurde, kann anschließend nicht noch mit einem zu diesem *ereignis* vereinbarten *subkdo* reagiert werden.
Wenn mehrere %ON-Kommandos mit unterschiedlichen *ereignis*-Vereinbarungen gleichzeitig aktiv sind und auch zutreffen, bearbeitet AID die zugehörigen Subkommandos in der Reihenfolge, in der die Schlüsselwörter in der folgenden Tabelle aufgeführt sind. Treffen

verschiedene %TERM-Ereignisse zu, werden die zugehörigen Subkommandos entgegen der Reihenfolge abgearbeitet, in der die %TERM-Ereignisse erklärt wurden (LIFO-Prinzip wie bei der Verkettung der Subkommandos). Wenn ein *write-ereignis* gleichzeitig mit einem anderen *ereignis* eintritt, wird erst das Subkommando zum *write-ereignis* abgearbeitet.
Zur Auswahl der SVC-Nummern und Ereigniscodes siehe „Makroaufrufe an den Ablaufteil“ (Seite 353).

Es wird darauf hingewiesen, dass ein `exec()`-Aufruf nicht mit `%ON %TERM` überwacht werden kann. Durch den `exec()`-Aufruf wird zwar das Programm überladen und dadurch beendet. Dies ist jedoch kein Programmende im Sinne von `%ON %TERM`. Um einen `exec()`-Aufruf zu überwachen, müssen Sie `%AID EXEC=ON` setzen. Diese Vereinbarung bewirkt, dass das Programm unmittelbar nach dem `exec()` angehalten wird.

ereignis	*subkdo* `wird bearbeitet`
%ERRFLG (z)	nach Auftreten eines Fehlers mit dem angegebenen Ereigniscode und vor Abbruch des Programms.
%INSTCHK	nach Auftreten eines Adressierungsfehlers, eines unzulässigen Systemaufrufs (SVC), nicht decodierbaren Operations-Codes, Seitenwechsel-Fehlers oder einer priviligierten Operation und vor Abbruch des Programms.
%ARTHCHK	nach Auftreten eines Datenfehlers, Divisionsfehlers, Exponenten-Überlaufs oder einer Mantisse Null und vor Abbruch des Programms.
%ABNORM	nach Auftreten eines der Fehler, die mit den vorher beschriebenen Ereignissen erfasst werden, sowie eines DMS-Errors (ab BS2000 V10).
%ERRFLG	nach Auftreten eines Fehlers mit beliebigem Ereigniscode.
%SVC(z) %SVC	vor Ausführung des Systemaufrufs (SVC) mit der angegebenen Nummer. vor Ausführung eines beliebigen Systemaufrufs (SVC).
%LPOV(name) %LPOV	nach dem Laden des Segments mit dem angegebenen Namen nach dem Laden eines beliebigen Segments (der Name wird mit %D %LINK ausgegeben)
%TERM(N[ORMAL]) %TERM(A[BNORMAL])	vor normaler Beendigung des Programms vor abnormaler Beendigung des Programms, jedoch nach der Ausgabe eines Speicherabzugs

Tabelle 5: Ereignisse des %ON-Kommandos und ihre Bedeutung

%TERM(D[UMP])	vor	Ausgabe eines Speicherabzugs mit anschließender Programmbeendigung
%TERM(S[TEP])	vor	Beendigung eines Programms mit anschließender Verzweigung innerhalb von Prozeduren
%TERM	vor	Beendigung eines Programms durch alle vorher beschriebenen %TERM-Ereignisse. Ein exec()-Aufruf kann damit nicht überwacht werden.
%ANY	vor	der Beendigung des Programms aufgrund eines Programmfehlers bzw. durch die oben beschriebenen %TERM-Ereignisse oder aufgrund eines DMS-Errors (ab BS2000 V10).

Tabelle 5: Ereignisse des %ON-Kommandos und ihre Bedeutung

z ist eine Ganzzahl mit: $1 \leq z \leq 255$. *z* kann als maximal dreistellige vorzeichenlose Dezimalzahl oder als zweistellige Sedezimalzahl (#'ff') angegeben werden.
Es wird nicht überprüft, ob der angegebene Ereigniscode oder die SVC-Nummer sinnvoll oder zulässig ist.

subkdo

wird immer dann bearbeitet, wenn im Programmablauf das vereinbarte *ereignis* eintritt. Wird der Operand *subkdo* nicht angegeben, so setzt AID ein <%STOP> ein.
Vollständig beschrieben finden Sie *subkdo* im AID-Basishandbuch, Kapitel „Subkommando“ [1].

```
subkdo-OPERAND  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

                                  {AID-kommando    }
<[subkdoname:] [(bedingung):] [ {                  } {;...}]>
                                  {BS2000-kommando}

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Das Subkommando kann einen Namen, eine Bedingung und einen Kommandoteil enthalten. Zu jedem Subkommando gehört ein Durchlaufzähler. Der Kommandoteil kann aus einem einzelnen Kommando oder einer Kommandofolge bestehen, er kann AID- und BS2000-Kommandos und Kommentare enthalten.

Wenn das Subkommando aus einem Namen oder einer Bedingung besteht, aber der Kommandoteil fehlt, erhöht AID beim Eintreten des vereinbarten Ereignisses nur den Durchlaufzähler.
subkdo überschreibt nicht ein bestehendes Subkommando zu demselben *ereignis*, sondern

das neue Subkommando wird vor das bestehende gekettet. Nur die Kettung von mehreren Write-Ereignissen ist nicht möglich.
Beim %ON sind in *subkdo* auch die Kommandos %CONTROL*n*, %INSERT und %ON zugelassen. Sie können also mehrere Überwachungskommandos ineinander schachteln. Ein Beispiel dazu finden Sie in der %INSERT-Beschreibung.

Die Kommandos in einem *subkdo* werden nacheinander ausgeführt. Danach wird das Programm fortgesetzt. Die Kommandos zur Ablaufsteuerung verändern auch in einem Subkommando sofort den Programmzustand. Sie brechen *subkdo* ab und setzen das Programm fort (%CONTINUE, %RESUME, %TRACE) oder halten es an (%STOP). Sie sind nur als letztes Kommando in einem *subkdo* sinnvoll, da im *subkdo* nachfolgende Kommandos nicht mehr ausgeführt werden. Auch ein %REMOVE für das gerade aktive Subkommando oder für das %ON-Kommando selbst oder für das zugehörige *ereignis* ist aus demselben Grund nur als letztes Kommando in *subkdo* sinnvoll.

Adressoperanden in Subkommandos werden bei der Eingabe nicht automatisch mit den Qualifikationen ergänzt, die der aktuellen Unterbrechungsstelle entsprechen. Wenn im weiteren Testverlauf das vereinbarte *ereignis* eintritt und AID das Programm unterbricht, um *subkdo* zu bearbeiten, können mit den AID-Kommandos aus *subkdo* ohne Qualifikation nur die Daten und Funktionen angesprochen werden, die an der Adresse sichtbar sind, an der *ereignis* eingetreten ist. Die Unterbrechungsstelle kann auch in einer Routine des Laufzeitsystems sein, was bedeutet, dass in diesem Fall alle Daten und Funktionen des Benutzerprogramms nur qualifiziert angesprochen werden können.

Beispiele

1.
```
%on %errflg (108)
%on %errflg (#'6c')
```

   Beide Angaben bezeichnen den gleichen Programmfehler (Mantisse gleich Null).

2.
```
%on %errflg (107) <%d 'error'>
```

   Diesen Ereigniscode gibt es nicht; deshalb wird das für dieses *ereignis* definierte *subkdo* nie gestartet.

3.

```
/%on %any
/%resume
STOPPED AT SRC REF: 59, SOURCE: VPTR.C , PROC: g(void) ,
EVENT: ADDRESS ERROR
```

Ihr Programm ist aufgrund eines Fehlers unterbrochen worden. Zusätzlich zur Art des Fehlers gibt AID die Source-Referenz aus, an der der Fehler aufgetreten ist sowie die Namen der Übersetzungseinheit und der Funktion, in der die Unterbrechung aufgetreten ist.

4.

```
/%on %write(page) <wr1: %d 'Schreibender Zugriff auf page'>
/%r

Schreibender Zugriff auf page
Schreibender Zugriff auf page
Schreibender Zugriff auf page
Schreibender Zugriff auf page
Schreibender Zugriff auf page
```

Fünfmal wurde vom Programm auf `page` schreibend zugegriffen.

5.

```
/%on %write(page) <wr1:>
/%on %term <%display %.wr1; %stop>
/%r
%  CCM0998 CPU TIME USED: 0.0323 SECONDS
CURRENT PC: 01015846    CSECT: ITOTRM@@  ******************************
%.WR1                          =              5
STOPPED AT V'1015846' = ITOTRM@@ + #'2E' , EVENT: TERM (NORMAL,PROGRAM,
 NODUMP)
```

Diesmal wird nur der Durchlaufzähler `%.WR1` als Subkommando zum %WRITE angelegt. Er wird bei jedem Durchlauf um eins erhöht. Bei Programmende, Ereignis %TERM, wird das Programm angehalten und der Durchlaufzähler und eine STOP-Meldung ausgegeben.

%OUT

Mit %OUT können Sie für die Ausgabe-Kommandos %DISASSEMBLE, %DISPLAY, %HELP, %SDUMP und %TRACE festlegen, über welche Medien die Daten ausgegeben werden und ob in der Ausgabe Zusatzinformationen enthalten sein sollen.

– Mit *ziel-kommando* bezeichnen Sie das Ausgabe-Kommando, für das Sie *medium-u-menge* festlegen wollen.

– Mit *medium-u-menge* geben Sie an, welche Ausgabemedien verwendet und ob Zusatzinformationen ausgegeben werden sollen. –

Kommando	Operand
%OUT	[ziel-kommando [medium-u-menge][,...]]

Bei den Kommandos %DISPLAY, %HELP und %SDUMP können Sie einen *medium-u-menge*-Operanden angeben, der für diese Kommandos die Vereinbarungen des %OUT-Kommandos vorübergehend außer Kraft setzt. %DISASSEMBLE und %TRACE haben keinen eigenen *medium-u-menge*-Operanden, ihre Ausgaben können Sie nur über %OUT steuern.

Bevor Sie mit %OUT das Ausgabemedium Datei wählen, müssen Sie die Datei mit %OUTFILE einem Linknamen zuweisen und öffnen; ansonsten legt AID eine Standard-Ausgabedatei mit dem Namen AID.OUTFILE.Fn an.

Die Vereinbarungen mit %OUT gelten, bis sie durch ein neues %OUT-Kommando überschrieben werden oder bis /LOGOFF bzw. /EXIT-JOB.

Ein %OUT-Kommando ohne Operanden setzt für alle *ziel-kommandos* den Standardwert T=MAX ein.

%OUT darf nur als Einzelkommando eingegeben werden, es darf nicht in einer Kommandofolge oder einem Subkommando stehen.

%OUT verändert den Programmzustand nicht.

ziel-kommando

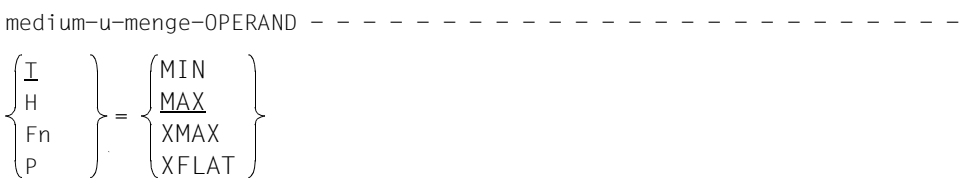
bezeichnet das Kommando, für das die Vereinbarungen gelten sollen. Jeweils eines der folgenden Kommandos kann hier angegeben werden:

```
{%D[IS]A[SSEMBLE]}
{%D[ISPLAY]      }
{%H[ELP]         }
{%SD[UMP]        }
{%T[RACE]        }
```

medium-u-menge

legt für *ziel-kommando* fest, über welches oder über welche Medien die Ausgabe erfolgen und ob AID Zusatzinformationen ausgeben soll über den AID-Arbeitsbereich, die aktuelle Unterbrechungsstelle und die auszugebenden Daten.

Wird der *medium-u-menge*-Operand nicht angegeben, so gilt für *ziel-kommando* der Standardwert T=MAX.



medium-u-menge ist ausführlich im AID-Basishandbuch, Kapitel „Operand Medium-und-Menge“ [1] beschrieben.

T Terminal-Ausgabe
H Hardcopy-Ausgabe (schließt die Terminal-Ausgabe mit ein und kann nicht gemeinsam mit *T* angegeben werden)
Fn Datei-Ausgabe
P Ausgabe nach SYSLST

AID berücksichtigt die Modi XMAX und XFLAT für die Ausgabe des %OUT-Protokolls nicht. Statt dessen generiert es die Standardausgabe (T=MAX).

MAX Ausgabe mit Zusatzinformationen.

MIN Ausgabe ohne Zusatzinformationen.

XMAX Festlegung des Modus XMAX für das entsprechende Kommando %DISASSEMBLE, %DISPLAY, %HELP, %SDUMP oder %TRACE.

XFLAT Festlegung des Modus XFLAT für das entsprechende Kommando %DISASSEMBLE, %DISPLAY, %HELP, %SDUMP oder %TRACE.

Beispiele

1. ```
   %outfile f1=output
   %out %sdump t=min,f1=max
   ```

   Datenausgaben des Kommandos %SDUMP sollen auf dem Terminal in Kurzform und parallel dazu in die Datei mit dem Linknamen `F1` mit Zusatzinformationen ausgegeben werden. Zuvor wurde dem Linknamen `F1` die Datei `OUTPUT` zugewiesen.

2. ```
   %out %trace f1=max
   ```

   Das TRACE-Protokoll mit Zusatzinformationen wird nur in die Datei mit dem Linknamen `F1` ausgegeben.

3. ```
   %out %trace
   ```

   Für das Kommando %TRACE wird festgelegt, dass bisherige Vereinbarungen zur Ausgabe von Daten gelöscht werden und dass der Standardwert T=MAX gilt.
   ```

%OUTFILE

Mit %OUTFILE können Sie den AID-Linknamen F0 bis F7 Ausgabedateien zuweisen oder Ausgabedateien schließen. In diese Dateien können Sie die Ausgaben der Kommandos %DISASSEMBLE, %DISPLAY, %HELP, %SDUMP und %TRACE schreiben lassen, indem Sie im *medium-u-menge*-Operanden von %OUT, %DISPLAY , %HELP oder %SDUMP den entsprechenden Linknamen angeben. Falls eine Datei noch nicht existiert, wird sie durch AID katalogisiert und geöffnet.

– Mit *link* wählen Sie den Linknamen für die Datei aus, die katalogisiert und geöffnet oder geschlossen werden soll.
– Mit *datei* bezeichnen Sie die Ausgabedatei.

Kommando	Operand
%OUTFILE	[link [= datei]]

Ohne den *datei*-Operanden veranlassen Sie AID, die mit *link* bezeichnete Datei zu schliessen. So können Sie auch während des Testverlaufs einen Zwischenstand der Datei ausdrucken.

Ein %OUTFILE ohne Operanden schließt alle offenen AID-Ausgabedateien.
Wenn Sie eine AID-Ausgabedatei nicht explizit mit %OUTFILE schließen, bleibt sie geöffnet bis LOGOFF bzw. EXIT-JOB. Außerdem werden die AID-Ausgabedateien durch einen `fork()`- oder einen `exec()`-Aufruf geschlossen,

Ohne Verwendung von %OUTFILE haben Sie zwei Möglichkeiten, AID-Ausgabedateien einzurichten und zuzuweisen:

1. Sie geben ein `ADD-FILE-LINK`-Kommando für einen noch nicht belegten Linknamen F*n*. Dann öffnet AID diese Datei beim ersten Ausgabekommando für diesen Linknamen.
2. Sie überlassen AID das Einrichten, Zuweisen und Öffnen. Dann verwendet AID Standard-Datei-Namen mit folgendem Aufbau: `AID.OUTFILE.F`*n* entsprechend dem Linknamen *Fn*.

%OUTFILE verändert den Programmzustand nicht.

link

bezeichnet einen der AID-Linknamen für Ausgabedateien und hat das folgende Format: F*n*.
n eine Zahl mit einem Wert $0 \leq n \leq 7$.

Die mit %MOVE erzeugten REPs werden stets in die Ausgabedatei mit dem Linknamen F6 geschrieben (siehe Beschreibung der Kommandos %AID und %MOVE). Achten Sie deshalb darauf, dass Sie in die Datei mit dem Linknamen F6 keine anderen Ausgaben schreiben lassen.

datei

gibt den vollqualifizierten Dateinamen an, mit dem AID die Ausgabedatei katalogisiert und öffnet.
Mit einem %OUTFILE ohne *datei*-Operand wird die dem Linknamen F*n* zugewiesene Datei geschlossen.

%QUALIFY

Mit %QUALIFY definieren Sie Qualifikationen, auf die Sie sich im Adressoperanden eines anderen Kommandos durch Voranstellen eines Punktes beziehen können.
Diese verkürzte Schreibweise ist immer dann sinnvoll, wenn Sie mehrfach Adressen ansprechen wollen, die dieselbe Qualifikation erfordern.

– Mit *vorqualifikation* legen Sie die Qualifikationen fest, die Sie in nachfolgenden Kommandos durch Voranstellen eines Punktes übernehmen möchten.

Kommando	Operand
%Q[UALIFY]	[vorqualifikation]

Eine mit %QUALIFY vereinbarte *vorqualifikation* gilt, bis sie durch einen %QUALIFY mit neuer *vorqualifikation* überschrieben wird, bis sie durch einen %QUALIFY ohne Operanden aufgehoben wird oder bis /LOGOFF bzw. /EXIT-JOB. Außerdem werden Vereinbarungen mit %QUALIFY durch einen `fork()`- oder einen `exec()`-Aufruf zurückgesetzt.

Bei der Eingabe eines %QUALIFY wird das Kommando nur syntaktisch überprüft. Ob dem angegebenen Linknamen eine Dump-Datei zugewiesen bzw. ob die angegebene Übersetzungseinheit geladen oder in den LSD-Sätzen verzeichnet ist, wird erst bei der Ausführung darauffolgender Kommandos geprüft, wenn die Angaben aus *vorqualifikation* in die Adressierung einbezogen werden.

Die Vereinbarungen des %QUALIFY werden nur von nachfolgend eingegebenen Kommandos übernommen. Auf die Subkommandos in %CONTROL, %INSERT und %ON, die vorher eingegeben wurden, hat ein neuer %QUALIFY keine Auswirkungen, auch wenn die Subkommandos erst danach ausgeführt werden.

Bei der Eingabe des %QUALIFY müssen Sie die aktuelle Einstellung für die Behandlung der Groß-/Kleinschreibung (`%AID LOW={ON|OFF|ALL}`) beachten, damit die *vorqualifikation* richtige Ergänzungen in den Adressoperanden nachfolgender Kommandos erzeugt.

%QUALIFY darf nur als Einzelkommando eingegeben werden, es darf nicht in einer Kommandofolge oder einem Subkommando stehen.

%QUALIFY verändert den Programmzustand nicht.

vorqualifikation

bezeichnet eine Basisqualifikation oder eine bzw. mehrere Bereichsqualifikationen. Mehrere Qualifikationen müssen durch einen Punkt getrennt werden.

In den Adressoperanden nachfolgender AID-Kommandos können Sie sich durch Voranstellen eines Punktes auf die im %QUALIFY definierte *vorqualifikation* beziehen.

```
vorqualifikation-OPERAND  - - - - - - - - - - - - - - - - - - - - - - - - - - - -

                  ⎧VM⎫                    ⎧[[[qua]•]PROC=funktion]  ⎫
[E=⎨  ⎬] [[•]S=srcname] ⎨                  ⎬
                  ⎩Dn⎭                    ⎩[[•]BLK='[f-]n[:b]']     ⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

E={VM|Dn}
: geben Sie an, wenn Sie eine andere als die aktuelle Basisqualifikation verwenden wollen (siehe %BASE).

S=srcname
: bezeichnet eine Übersetzungseinheit (siehe Abschnitt „Qualifikationen“ auf Seite 21).

[qua•]PROC=funktion
: bezeichnet eine Funktion.

  Bei Funktionen aus C-Programmen ist *funktion* der im Quellprogramm vergebene Name der Funktion ohne Klammern und ohne Signatur.

  Funktionen aus C++ - Programmen müssen Sie, je nachdem, um welchen Typ es sich handelt, in der Schreibweise `n'...'` oder `t'...'` angeben. Ist die Funktion in einem Namespace oder in einer Klasse definiert, so wird dem Funktionsnamen die Namespace- oder Klassen-Qualifikation vorangestellt. Die Signatur `void` darf nicht mehr geschrieben werden. Wie in C++ geben Sie in diesem Fall nur die beiden Klammern nach dem Funktionsnamen an. Es ergibt sich die folgende Syntax (*f-template* sowie *signatur* sind in der Syntax aus Platzgründen abgekürzt):

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

                                                ⎧n'funktion([sign])'           ⎫
PROC=[namespace::[...]][klasse::[...]]          ⎨                              ⎬
                                                ⎩t'f_templ<arg[,...]>([sign])'⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Abweichend davon werden die Funktionen `main` und `_ _STI_ _` sowie alle Funktionen mit C-Linkage auch beim Testen in C++-Programmen nur mit dem Funktionsnamen bezeichnet (siehe Seite 60).

qua
: Ist die Funktion in einer lokalen Klasse definiert, so müssen Sie vor der PROC-Qualifikation noch eine weitere PROC-Qualifikation für die übergeordnete Funktion angeben, die die Definition der lokalen Klasse enthält. Zwischen die beiden PROC-Qualifikationen schreiben Sie bei Funktionen, die in einem inneren Block der übergeordneten Funktion definiert sind, eine oder ggfs. mehrere BLK-Qualifikationen (siehe Seite 62).

  Syntax für *qua*:

```
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

PROC=übergeordnete_fkt.[BLK='[f-]n[:b]'•[...]]

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

  Das Ansprechen von Funktionen, die in lokalen Klassen innerer Blöcke definiert sind, wird nur bei Programmen unterstützt, die mit einem C/C++-Compiler ab V3.0B übersetzt wurden.

BLK='[f-]n[:b]'
: bezeichnet einen Block. Der Name für einen Block wird wie bei den Source-Referenzen aus Zeilennummer (*n*) und gegebenenfalls FILE-Nummer (*f*) und relativer Blocknummer (*b*) gebildet.

Beispiele

1.
```
%qualify e=d1.s=n'parr.c'.proc=foo
%d .str[1]
```

   Durch die *vorqualifikation* hat der %DISPLAY dieselbe Bedeutung wie das folgende, ausgeschriebene %DISPLAY-Kommando:

```
%d e=d1.s=n'parr.c'.proc=foo.str[1]
```

2.
```
%qualify s=n'bcl.c'
%set .::a into b
```

   Mit dieser Qualifikation bezeichnen Sie die Übersetzungseinheit `BCL.C`. Durch die *vorqualifikation* hat der %SET dieselbe Bedeutung wie das folgende, ausgeschriebene %SET-Kommando:

```
%set s=n'bcl.c'.::a into b
```

   Die globale Variable `a` aus der Übersetzungseinheit `BCL.C` wird nach `b` übertragen.

3.
```
%qualify s=n'examp.c'.proc=main
%d .x_arr[i]
```

   Im %DISPLAY-Kommando wird wie in Beispiel 1 und 2 vor den Punkt die Vorqualifikation aus dem %QUALIFY geschrieben.
   Die Vorqualifikation gilt hier nicht nur für den Vektor `x_arr`, sondern auch für den angegebenen Index `i`. Sie sprechen damit das `i`-te Element des Vektors `x_arr` aus der Funktion `main` in der Übersetzungseinheit `EXAMP.C` an.

%REMOVE

Mit %REMOVE heben Sie die Testvereinbarungen der Kommandos %CONTROL*n*, %INSERT oder %ON auf.

– Mit *ziel* legen Sie fest, ob AID für ein angegebenes Kommando alle wirksamen Vereinbarungen aufheben soll oder ob nur ein bestimmter Testpunkt, ein bestimmtes Ereignis oder ein Subkommando gelöscht werden soll.

Kommando	Operand
%REM[OVE]	ziel

Steht ein %REMOVE in einem Subkommando, der dieses Subkommando oder die zugehörige Überwachungsbedingung (*testpunkt*, *ereignis* oder *kriterium*) löscht, werden nachfolgende Kommandos in *subkdo* nicht mehr ausgeführt. Diese Angabe ist deshalb nur als letztes Kommando in einem Subkommando sinnvoll.

%REMOVE verändert den Programmzustand nicht.

ziel

bezeichnet entweder ein Kommando, für das alle Vereinbarungen gelöscht werden sollen, oder einen *testpunkt*, der gelöscht werden soll, oder ein *ereignis*, das nicht mehr überwacht werden soll, oder das zu löschende Subkommando. Liegt *ziel* in einem geschachtelten Subkommando und ist somit noch nicht eingetragen, kann es auch nicht gelöscht werden.

```
ziel-OPERAND  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

{%C[ONTROL] | %C[ONTROL]n    }
{%IN[SERT] | testpunkt       }
{%ON | %WRITE | ereignis     }
{%•[subkdoname]              }

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

%C[ONTROL]
Die Vereinbarungen aller eingetragenen %CONTROL*n* werden gelöscht.

%C[ONTROL]n
: Der %CONTROL*n* mit der angegebenen Nummer ($1 \leq n \leq 7$) wird gelöscht.

%IN[SERT]
: Alle eingetragenen Testpunkte werden gelöscht.

testpunkt
: Der angegebene *testpunkt* wird gelöscht. *testpunkt* wird wie bei %INSERT angegeben. Innerhalb des eigenen Subkommandos kann der Testpunkt auch mit `%REMOVE %PC->` gelöscht werden, da der Befehlszähler (%PC) zu diesem Zeitpunkt die Adresse des Testpunkts enthält.

%ON Alle eingetragenen Ereignisse werden gelöscht.

%WRITE
: Das *write-ereignis* wird gelöscht.

ereignis
: Das angegebene *ereignis* wird gelöscht. *ereignis* wird wie bei %ON mit einem Schlüsselwort angegeben. Die *ereignis*-Tabelle mit den Schlüsselwörtern und den Erläuterungen der einzelnen Ereignisse steht in der %ON-Beschreibung.

  Für die Ereignisse %ERRFLG(*z*), %SVC(*z*) und %LPOV(*name*) gilt:

  – %REMOVE *ereignis(z* | *name)* löscht nur das Ereignis mit der angegebenen Nummer bzw. dem angegebenen Namen.

  – %REMOVE *ereignis* ohne Angabe einer Nummer löscht alle Ereignisse der entsprechenden Gruppe.

%•[subkdoname]
: löscht das Subkommando eines %CONTROL*n* oder %INSERT mit *subkdoname*.

  %• ist die Kurzform für einen Subkommandonamen, die nur innerhalb des Subkommandos verwendet werden kann. %REMOVE %• löscht folglich das gerade ausgeführte Subkommando.

  Da %CONTROL*n* nicht gekettet werden kann, wird auch der zugehörige %CONTROL*n* gelöscht. Das Löschen des Subkommandos entspricht folglich einer Löschung des %CONTROL*n* mit Angabe der Nummer.

  An einem *testpunkt* des Kommandos %INSERT können dagegen mehrere Subkommandos gekettet sein. Mit `%REMOVE %•[`*subkdoname*`]` löschen Sie ein einzelnes Subkommando aus einer Kette, weitere Subkommandos zum selben *testpunkt* bleiben dagegen bestehen (siehe AID-Basishandbuch, Kapitel „Subkommando“ [1]). War zu dem *testpunkt* nur das Subkommando mit *subkdoname* eingetragen, so wird auch der *testpunkt* gelöscht.

  Für %ON ist `%REMOVE %•[`*subkdoname*`]` nicht zugelassen.

Beispiele

1.
```
%c1 %call <call: %d %.>
%rem %c1
%rem %.call
```

   Die beiden %REMOVE-Kommandos haben dieselbe Wirkung: `%C1` wird gelöscht.

2.
```
%in s'58' <sub1: %d char1, char2, numb>
%in s'58' <sub2: %d result; %rem %.>
%r
...
%rem s'58'
```

   Wenn der Testpunkt S’58’ erreicht wird, wird `result` ausgegeben und das Subkommando `SUB2` gelöscht. Dieses Subkommando wird also nur ein einziges Mal ausgeführt. Dann werden `char1`, `char2` und `numb` ausgegeben, und das Programm wird fortgesetzt. Immer wenn danach der Programmablauf an den Testpunkt S’58’ kommt, wird Subkommando SUB1 ausgeführt. Mit `%rem s'58'` wird später der Testpunkt gelöscht. Ein `%rem %.sub1` würde dasselbe bewirken, denn zum Testpunkt S’58’ ist nur noch dieses Subkommando eingetragen.

%RESUME

Mit %RESUME starten Sie das geladene Programm oder setzen es an der Unterbrechungsstelle fort. Das Programm läuft ohne Ablaufverfolgung.
Die Fortsetzungsadresse für den Programmablauf kann mit %RESUME nicht beeinflusst werden. Eine andere Fortsetzungsadresse können Sie festlegen, indem Sie mit %SET den Befehlszähler (%PC) ändern (siehe Beschreibung des Kommandos %SET, *schlüsselwort* auf Seite 275).

%RESUME beendet alle aktiven %TRACE-Kommandos, %CONTINUE hingegen hat keine Auswirkungen auf %TRACE.

Kommando	Operand
%R[ESUME]	

Steht %RESUME in einer Kommandofolge oder in einem Subkommando, werden nachfolgende Kommandos nicht mehr ausgeführt.
Steht in einem Subkommando nur das Kommando %RESUME, wird der Durchlaufzähler erhöht und ein eventuell aktiver %TRACE gelöscht.

%RESUME verändert den Programmzustand.

%SDUMP

Mit %SDUMP geben Sie einen symbolischen Dump aus. Sie können einzelne Daten, alle Daten der aktuellen Aufrufhierarchie oder die aktuelle Aufrufhierarchie selbst ausgeben lassen. Die aktuelle Aufrufhierarchie reicht von der Funktion bzw. dem Block, in dem das Programm unterbrochen wurde, bis zum Hauptprogramm. Die Ausgabe erfolgt über SYSOUT, SYSLST oder in eine katalogisierte Datei.

– Mit *dump-bereich* bezeichnen Sie die Daten oder Datenbereiche, die AID ausgeben soll, oder Sie geben an, dass AID die aktuelle Aufrufhierarchie ausgeben soll.

– Mit *medium-u-menge* geben Sie an, welche Ausgabemedien AID verwenden und ob Zusatzinformationen ausgegeben werden sollen. Mit diesem Operanden setzen Sie eine mit %OUT getroffene Vereinbarung für das aktuelle %SDUMP-Kommando außer Kraft.

Kommando	Operand
%SD[UMP]	[[dump-bereich][,...] [medium-u-menge][,...]]

Befinden sich in der Hierarchie Übersetzungseinheiten, für die es keine LSD-Sätze gibt, auch nicht in einer PLAM-Bibliothek, so können Sie das Kommando %SDUMP nur einzeln für die Übersetzungseinheiten eingeben, für die LSD-Sätze geladen wurden oder aus einer PLAM-Bibliothek nachladbar sind (siehe Beschreibung des Kommandos %SYMLIB).

%SDUMP ohne Operanden gibt alle Daten der aktuellen Aufrufhierarchie aus, soweit AID auf die zugehörigen LSD-Sätze zugreifen kann. Daten, die mehrfach definiert sind, werden auch mehrfach ausgegeben. Daten, denen mit %ALIAS ein Aliasname zugeordnet wurde, werden unter ihrem Originalnamen aufgelistet.

%SDUMP %NEST gibt die aktuelle Aufrufhierarchie aus, das sind die Source-Referenz der Unterbrechungsstelle, die Nummern der Blöcke, die Namen der Funktionen und des Programms, die an der Unterbrechungsstelle aktiv sind. Auch für Programme, zu denen keine LSD-Sätze erzeugt wurden, gibt AID die aktuelle Aufrufhierarchie aus (siehe auch Beispiel 3 auf Seite 258).

Liegt die aktuelle Unterbrechungsstelle in einer rekursiven Funktion, so zählt für %SDUMP jeder Aufruf dieser Funktion, d.h. die Daten der Funktion werden zu jedem Aufruf ausgegeben, und bei %SDUMP %NEST wird die Funktion so oft aufgeführt, wie sie bis zur Unterbrechungsstelle aufgerufen wurde.

Unmittelbar nach dem Laden können Sie %SDUMP nicht verwenden. Erst wenn das Programm vor der ersten ausführbaren Anweisung steht, ist eine Aufrufhierarchie vorhanden, und AID kann die entsprechenden Namensräume zuordnen. Mit einem der folgenden Kommandos unterbrechen Sie das Programm vor der ersten ausführbaren Anweisung:

`/%insert main;%r` oder

`/%trace 1 in s=srcname`

Wenn Sie ein C++-Programm testen, das Klassen mit virtuellen Funktionen oder Konstruktoren enthält, und den %TRACE verwenden, dann hält das Programm nicht in `main`, sondern in einer vom Compiler erzeugten Funktion mit Namen `_ _STI_ _`. Diese Funktion ruft die Konstruktoren von globalen Objekten auf und legt die Tabellen der virtuellen Funktionen an. Den Funktionsnamen `_ _STI_ _` gibt AID in der Stop-Meldung und in der aktuellen Aufrufhierarchie aus.
Die Destruktoren der globalen Objekte werden nach Beendigung von `main` von Routinen des Laufzeitsystems aufgerufen. Deren Namen können Sie in der Aufrufhierarchie sehen.

dump-bereich können Sie bis zu siebenmal wiederholen. Geben Sie für *dump-bereich* einen Namen an, der nicht in den LSD-Sätzen verzeichnet ist, gibt AID eine Fehlermeldung aus. Die anderen *dump-bereich*-Operanden desselben Kommandos werden ordnungsgemäß bearbeitet.

Sie können mit diesem Kommando im geladenen Programm oder in einer Dump-Datei arbeiten.

%SDUMP verändert den Programmzustand nicht.

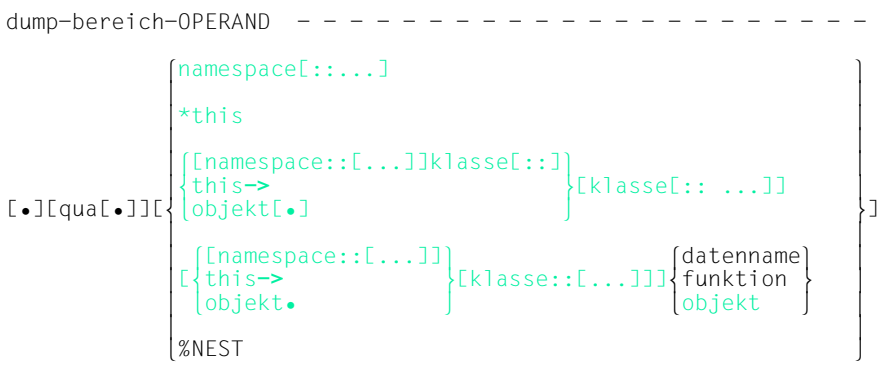
dump-bereich

beschreibt, welche Informationen AID ausgeben soll. AID kann die aktuelle Aufrufhierarchie, alle Daten der aktuellen Aufrufhierarchie, alle Daten einer Übersetzungseinheit, einer Funktion, eines Blocks oder einzelne Daten ausgeben.

Daten bereitet AID entsprechend der Definition im Quellprogramm auf.

datenname, *namespace*, *klasse, objekt* oder *funktion*, die innerhalb der aktuellen Aufrufhierarchie mehrfach definiert sind, werden auch mehrmals ausgegeben, es sei denn, *dump-bereich* wurde mit einer Qualifikation eingeschränkt. Dabei ist zu beachten, dass der globale Bereich nicht zur Aufrufhierarchie zählt. Um sich ein globales Datum gleichen Namens ausgeben zu lassen, müssen Sie `%DISPLAY [S=`*srcname*`•]::`*datenname* schreiben. Dasselbe gilt für *namespace, klasse, objekt* oder *funktion*.

Bei einem `%SDUMP S=`*srcname*, mit dem alle Daten einer Übersetzungseinheit ausgegeben werden, gibt AID dagegen auch alle globalen Daten und die Adressen aller in dieser Übersetzungseinheit definierten oder deklarierten und benutzten Funktionen aus. Explizit mit einer ::-Qualifikation können Sie den globalen Bereich bei %SDUMP nicht ansprechen.



- Steht der Punkt an führender Stelle, ist er das Kennzeichen für eine *vorqualifikation*. Sie muss mit einem vorhergehenden %QUALIFY definiert worden sein. Aufeinanderfolgende Qualifikationen werden durch einen Punkt getrennt. Außerdem muss zwischen der letzten Qualifikation und dem anschließenden Operandenteil ein Punkt stehen. Endet die Adressierung mit einer Qualifikation, entfällt der abschließende Punkt.

qua Eine oder mehrere Qualifikationen geben Sie an, wenn die Unterbrechungsstelle nicht im Gültigkeitsbereich des adressierten Objekts liegt oder wenn das Speicherobjekt an der Unterbrechungsstelle nicht sichtbar ist. Sie geben nur die Qualifikationen an, die zur eindeutigen Adressierung genügen.
Die ::-Qualifikation kann hier nicht verwendet werden. Darum können Sie Daten und Funktionen, die zur Ansprache die ::-Qualifikation benötigen, nur mit %DISPLAY, nicht aber mit %SDUMP ausgeben lassen.
Endet *dump-bereich* mit einer Qualifikation, so werden alle Daten und Funktionen des durch die Qualifikation bezeichneten Programmteils aufgelistet.

E ={VM | Dn}
Endet die Adressierung mit der Basisqualifikation, erhalten Sie alle Namespaces, Klassen, Objekte von Klassen, Daten und Funktionen der entsprechenden Aufrufhierarchie.
Ansonsten geben Sie eine Basisqualifikation nur an, wenn für *dump-bereich* die aktuelle Basisqualifikation nicht gelten soll.
Vor dem Schlüsselwort %NEST ist nur eine Basisqualifikation möglich.

S=srcname

Endet die Adressierung mit der S-Qualifikation, erhalten Sie alle Namespaces, Klassen, Objekte von Klassen, Daten und Funktionen der entsprechenden Übersetzungseinheit. Die Übersetzungseinheit muss in der aktuellen Aufrufhierarchie enthalten sein.
Ansonsten geben Sie eine S-Qualifikation nur an, um einen Namespace, ein Objekt einer Klasse, einen Datennamen, eine Funktion oder einen Block aus einer anderen Übersetzungseinheit anzusprechen.

PROC=funktion

geben Sie an, wenn *dump-bereich* nur für die angegebene Funktion gelten soll. *funktion* muss in der Aufrufhierarchie liegen.

Endet die Adressierung mit der PROC-Qualifikation, erhalten Sie alle Klassen, Objekte von Klassen, Daten und Funktionen, die dem Namensraum der entsprechenden Funktion zugeordnet sind.
Ansonsten geben Sie eine PROC-Qualifikation an, um eine Klasse, ein Objekt einer Klasse, einen Datennamen oder eine Funktion eindeutig anzusprechen.

Bei Funktionen aus C-Programmen ist *funktion* der im Quellprogramm vergebene Name der Funktion ohne Klammern und ohne Signatur.

Funktionen aus C++-Programmen müssen Sie, je nachdem, um welchen Typ es sich handelt, in der Schreibweise `n'...'` oder `t'...'` angeben. Ist die Funktion in einem Namespace oder in einer Klasse definiert, so wird dem Funktionsnamen die Namespace- oder Klassen-Qualifikation vorangestellt. Die Signatur `void` darf nicht mehr geschrieben werden. Wie auch in C++ möglich geben Sie in diesem Fall nur die beiden Klammern nach dem Funktionsnamen an. Es ergibt sich die folgende Syntax (*f_template* und *signatur* sind in der Syntax aus Platzgründen abgekürzt):

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

                                                 ⎧n'funktion([sign])'          ⎫
PROC=[namespace::[...]][klasse::[...]]           ⎨                             ⎬
                                                 ⎩t'f_templ<arg[,...]>([sign])'⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Abweichend davon werden die Funktionen `main` und `_ _STI_ _` sowie alle Funktionen mit C-Linkage auch beim Testen in C++-Programmen nur mit dem Funktionsnamen bezeichnet (siehe Seite 60).

Ist die Funktion in einer lokalen Klasse definiert, so müssen Sie vor der PROC-Qualifikation noch eine weitere PROC-Qualifikation für die übergeordnete Funktion angeben, die die Definition der lokalen Klasse enthält. Zwischen die beiden PROC-Qualifikationen schreiben Sie bei Funktionen, die in einem inne-

ren Block der übergeordneten Funktion definiert sind, eine oder ggfs. mehrere BLK-Qualifikationen (siehe Seite 62).
Das Ansprechen von Funktionen, die in lokalen Klassen innerer Blöcke definiert sind, wird nur bei Programmen unterstützt, die mit einem C/C++-Compiler ab V3.0B übersetzt wurden.

BLK='[f-]n[:b]'

Endet die Adressierung mit der BLK-Qualifikation, erhalten Sie alle Objekte von Klassen und Daten des entsprechenden Blocks. Der Block muss in der aktuellen Aufrufhierarchie enthalten sein.
Ansonsten geben Sie eine BLK-Qualifikation an, um eine Klasse, ein Objekt einer Klasse oder einen Datennamen eindeutig anzusprechen.

Wenn Sie mit einer PROC-Qualifikation eine Funktion aus einer lokalen Klasse bezeichnen und die Definition dieser lokalen Klasse steht in einem inneren Block, dann müssen Sie vor die PROC-Qualifikation die entsprechende BLK-Qualifikation schreiben (siehe oben PROC=*funktion*).

NESTLEV= level-nummer

level-nummer Nummer einer Ebene in der aktuellen Aufrufhierarchie

Auf *level-nummer* kann nur *datenname* folgen.
Das %SDUMP-Kommando gibt entweder einen symbolischen Dump für alle Daten aus, die auf der angegebenen Ebene definiert wurden, oder den Datennamen *datenname*, der auf der angebenen Ebene der Aufrufhierarchie definiert wurde.

namespace

ist der im Quellprogramm deklarierte Name eines Namespaces.

Endet der *dump-bereich*-Operand mit dem Namen eines Namespaces, so listet AID alle darin definierten Daten und Funktionen auf. Die Funktionen werden in C++-Standard-Schreibweise aufgeführt; die Anfangsadresse des zugehörigen Prologs wird ausgegeben. Bei geschachtelten Namespaces wird der Inhalt der inneren Schachtelungen mit ausgegeben. Enthält ein Namespace eine `using`-Direktive auf einen weiteren Namespace, so wird dieser nur mit seinem Namen aufgeführt.

Im Adressierungspfad zu Klassen, Daten oder Funktionen, die im Namespace definiert sind, geben Sie die Namespace-Qualifikation nur an, wenn die gewünschte Komponente des Namespaces an der Unterbrechungsstelle nicht sichtbar ist.

Vor der Namespace-Qualifikation sind nur die E- oder S-Qualifikation möglich. Weitere Informationen zu den Namespaces finden Sie im Abschnitt „Namespaces“ auf Seite 89.

{ klasse | this-> | objekt }
: ist der im Quellprogramm deklarierte Name einer Klasse, der `this`-Zeiger oder der Name eines Objekts einer Klasse.
Klassennamen, den `this`-Zeiger mit nachfolgendem Pointer-Operator sowie die Namen von Objekten von Klassen geben Sie an, um den Adressierungspfad zu Daten-Membern zu beschreiben (siehe Abschnitt „Klassen“ auf Seite 65).

Liegt die aktuelle Unterbrechungsstelle in einer dynamischen Member-Funktion, können Sie die Daten der Klasse gemäß den aus C++ bekannten Scoperegeln ansprechen.

Unabhängig von der Unterbrechungsstelle erreichen Sie die dynamischen Daten eines Objekts über den Objektnamen mit nachfolgendem Punkt, falls das Objekt in der aktuellen Aufrufhierarchie liegt.

Statische Daten-Member erreichen Sie von jeder Stelle des Programms über den zugehörigen Klassennamen mit den beiden nachfolgenden Doppelpunkten. Bei geschachtelten Klassen enthält der Pfad zum Datum alle Klassennamen von außen nach innen, jeweils durch zwei Doppelpunkte getrennt. Der äußerste Klassennamen benötigt die dem Gültigkeitsbereich entsprechende Qualifikation. Ist das Programm in einer Member-Funktion der Klasse unterbrochen, so gelten für das Ansprechen von statischen Daten-Member die Regeln des Klassenscopes, d.h. wenn das Daten-Member nicht durch eine gleichnamige Defintion verdeckt ist, so kann es unqualifiziert angesprochen werden.

Besteht der *dump-bereich*-Operand aus einem oder mehreren Klassennamen, werden die statischen Daten-Member und alle Member-Funktionen außer den virtuellen aufgelistet. Zu den Daten wird der aktuelle Inhalt ausgegeben. Die Funktionen werden in C++-Standard-Schreibweise aufgeführt; die Anfangsadresse des zugehörigen Prologs wird ausgegeben. Bei abgeleiteten Klassen gibt AID auch die Basisklassen aus. Bei geschachtelten Klassen wird der Inhalt der inneren Schachtelungen mit ausgegeben.
Endet der *dump-bereich*-Operand mit einem Objektnamen, gibt AID zusätzlich die dynamischen Daten aus. Dem Namen einer virtuellen Funktion wird die aktuell gültige Funktion zugeordnet. Die gleiche Ausgabe erhalten Sie mit `%SDUMP *this`, falls das Programm in einer dynamischen Member-Funktion der Klasse unterbrochen ist. Um in einer abgeleiteten Klasse eine Basisklasse zu bezeichnen, geben Sie ausgehend vom Objektnamen bzw. von `this->` im Pfad den Namen der gewünschten Basisklasse an.
objekt benötigt die seinem Gültigkeitsbereich entsprechende Qualifikation. Vor `this` ist nur eine Basisqualifikation sinnvoll.

datenname
: ist der im Quellprogramm definierte Name eines Datums. *datenname* wird wie im Quellprogramm angegeben.

Daten können Sie bis auf folgende Ausnahmen wie in C/C++ ansprechen:
Mit einem Vektornamen ohne Index sprechen Sie alle Vektorelemente an.
Einzelne Vektorelemente können Sie nur über Index ansprechen, nicht über Zeiger.
Es können auch Indexbereiche ausgegeben werden.
Wenn `%AID C=YES` gesetzt ist (siehe Seite 119), fasst AID die Vektorelemente eines `char`-Vektors, die über den am weitesten rechts stehenden Index adressiert werden, zu C-Strings zusammen und gibt den Vektorinhalt in Form von C-String-Literalen aus; zum Arbeiten mit Vektoren siehe auch die Abschnitte „Indexschreibweise“ auf Seite 31 und „C-Strings“ auf Seite 37.
Bei Variablen vom Typ `long double` wertet AID nur die ersten 8 Bytes aus.
Variablen vom Typ `char` gibt AID im Ausgabetyp %C aus. Mit Hilfe einer Typmodifikation (%A oder %F) können Sie sich den entsprechenden numerischen Wert ausgeben lassen (siehe auch Seite 30). Die Datentypen `unsigned char` und `signed char` behandelt AID dagegen wie kleine Integer-Variablen.

Bezeichnet *datenname* einen Pointer-to-Member, so erhalten Sie in der Ausgabe den Namen desjenigen Klassen-Members, auf das der Pointer-to-Member aktuell verweist. Einen dereferenzierten Pointer-to-Member können Sie bei %SDUMP nicht angeben. In diesem Fall müssen Sie das gewünschte Datum direkt über das zugehörige Objekt ansprechen.

datenname kann folgendermaßen angegeben werden. Die Formate können auch kombiniert werden (siehe Abschnitt „Datennamen“ auf Seite 30).

Indexschreibweise:	*datenname* `[`*index*`]` `{...}`
Zeigerschreibweise:	*datenname1* `->` *datenname2*
Strukturqualifizierung:	*übergeordneter datenname*`.` `{...}` *datenname*
Dereferenzierung:	`[(]*{...}` *datenname*`[)]`

funktion

ist der Name einer Funktion, der im Quellprogramm vergeben wurde bzw. der Name einer Bibliotheksfunktion. Die Adresse des ersten Befehl des vom Compiler erzeugten Prologs einer Funktion wird ausgegeben (siehe PROC=*funktion* auf Seite 248 und Kapitel „C++-spezifische Adressierung“ auf Seite 59).

Virtuelle Funktionen sprechen Sie mit der folgenden Syntax an:
`p->n'funktion([signatur])'`
p ist eine Zeigervariable, die auf das Objekt einer Klasse verweist, das die gewünschte Member-Funktion enthält. Wenn *p* von der aktuellen Unterbrechungsstelle aus nicht zu erreichen ist, muss dem Gültigkeitsbereich entsprechend qualifiziert werden. Liegt die Unterbrechungsstelle in der virtuellen Funktion selbst, dann können Sie die Prologadresse der aktuellen Funktion ansprechen, indem Sie statt *p* den `this`-Zeiger einsetzen (siehe Beschreibung von `this` auf Seite 66 und Abschnitt „Virtuelle Funktionen“ auf Seite 76).

%NEST

ist ein AID-Schlüsselwort, das die Ausgabe der aktuellen Aufrufhierarchie veranlasst.
Für die unterste Hierarchiestufe gibt AID die Source-Referenz der Unterbrechungsstelle und die Nummer des Blocks bzw. den Namen der Funktion aus. Für höhere Hierarchiestufen gibt AID für Blöcke die Source-Referenz der ersten ausführbaren Anweisung nach Verlassen des Blocks und den Namen der Funktion, die den Block enthält, oder die Blocknummer des übergeordneten Blocks aus. Für Funktionen wird die Source-Referenz des Funktionsaufrufs und der Name der aufrufenden Funktion ausgegeben.

Auch für Programme, zu denen keine LSD-Sätze vorhanden sind, gibt AID die aktuelle Aufrufhierarchie aus. Ist das Programm jedoch im Prolog einer Bibliotheksfunktion unterbrochen (etwa nach `%INSERT` *bibliotheksfunktion*`->;%R`), dann fehlt u.U. in der Aufrufhierarchie der unmittelbare Aufrufer der Bibliotheksfunktion. AID kann die Aufrufhierarchie erst dann vollständig ermitteln, wenn der Prolog durchlaufen und die erste ausführbare Anweisung der Funktion erreicht ist. Falls LSD zu der Funktion zur Verfügung steht, können Sie mit `%TRACE 1 %STMT` auf die erste ausführbare Anweisung der Funktion positionieren und sich dann mit `%SDUMP %NEST` die vollständige Aufrufhierarchie ausgeben lassen.

Hinweis für Tester auf Maschinencode-Ebene:
Für C- und C++-Programme ohne LSD-Sätze gibt %SDUMP %NEST die Adressen der Unterbrechungstelle, die CSECT-Namen und die vom Compiler generierten Entry-Namen der an der Aufrufhierarchie beteiligten Funktionen aus (siehe Beispiel 3 auf Seite 258). Adressen, CSECT- und Entry-Namen finden Sie im Object-Listing Ihres Programms.

Vor %NEST können Sie nur eine Basisqualifikation angeben.

medium-u-menge

legt fest, über welches oder über welche Medien die Ausgabe erfolgen soll und ob AID Zusatzinformationen ausgeben soll. Ohne diesen Operanden und ohne eine Vereinbarung mit dem %OUT-Kommando arbeitet AID mit dem Standardwert T = MAX.

```
medium-u-menge-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - -

{ I  }     { MIN  }
{ H  }   = { MAX  }
{ Fn }     { XMAX }
{ P  }     { XFLAT}

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

medium-u-menge ist ausführlich im AID-Basishandbuch, Kapitel „Operand Medium-und-Menge“ [1] beschrieben.

T Terminal-Ausgabe
H Hardcopy-Ausgabe (schließt die Terminal-Ausgabe mit ein und kann nicht gemeinsam mit *T* angegeben werden)
Fn Datei-Ausgabe
P Ausgabe nach SYSLST

MAX Ausgabe mit Zusatzinformationen.

MIN Ausgabe ohne Zusatzinformationen.

XMAX Ausgabe wie bei MAX, jedoch erweitert um Typ-Informationen:
Zusätzlich geht jedem Datenelement ein Typ-Tag voraus, das Typ, Größe und Ausgabe-Format dieses Datenelements definiert. Syntax des Typ-Tags:
`<data-type(memory-size-in-bytes),output-format>`

XFLAT Ausgabe wie bei XMAX, jedoch mit folgenden Einschränkungen:
Für strukturierte Datentypen wird nur die jeweils oberste Strukturebene ausgegeben. Bei langen Daten (z.B. langen Strings oder Arrays) werden nur die ersten Elemente ausgegeben.

Datentypen

Wenn Sie den Operandenwert XMAX oder XFLAT angegeben haben, generiert AID die Ausgabe wie bei MAX erweitert um die folgenden Typ-Tags:

```
<INT(size),D>
int-name = int-value
```

size Speicherlänge in Bytes.
int-name bezeichnet ein Element vom Typ Integer.
int-value Dezimalzahl (D); Wert von *int-name*.

```
<POINTER(size),X>
pointer-name = pointer-value
```

size Speicherlänge in Bytes.
pointer-name bezeichnet ein Elementvom Typ Pointer.
pointer-value Hexadezimalzahl (X); Wert von *pointer-name*.

```
<FLOAT(size),E>
float-name = float-value
```

size Speicherlänge in Bytes.
float-name bezeichnet ein Element vom Typ Floating Point Number.

float-value	Gleitkommazahl dargestellt als Dezimalbruch mit Exponent (E); Wert von *float-name.*

```
<CHARS(1),C>
char-name = |character|
```

char-name	bezeichnet ein Element vom Typ Character.
character	das Zeichen als abdruckbares Zeichen (C); Wert von *char-name*. Ein nicht abdruckbares Zeichen wird als `|.|` dargestellt.

```
<CHARS(size),C>
chars-name = "string"
```

size	Speicherlänge in Bytes.
chars-name	bezeichnet ein Element vom Typ String, also Array vom Typ Character.
string	Folge von abdruckbaren Zeichen (C); Wert von *chars-name;* Nicht abdruckbare Zeichen werden als hexadezimaler Wert dargestellt. Ist *string* länger als 80 Zeichen, dann werden bei XFLAT nur die ersten 72 Zeichen ausgegeben, gefolgt von drei Punkten ... , um die Unvollständigkeit der Ausgabe anzuzeigen. Siehe auch Hinweis 1 am Ende der Liste.

```
<UNSIGN(size),D>
unsign-name = unsign-value
```

size	Speicherlänge in Bytes.
unsign-name	bezeichnet ein Element vom Typ Integer ohne Vorzeichen (unsigned).
unsign-value	Dezimalzahl (D): Wert von *unsign-name*.

```
<ADDR(size),X>
addr-name = addr-value
```

size	Speicherlänge in Bytes.
addr-name	bezeichnet ein Element einer relativen oder absoluten Speicheradresse.
addr-value	Hexadezimalzahl (X), Wert von *addr-name*.

```
<CLASS(size),S>
class-name = class-value
```

size	Speicherlänge in Bytes.
class-name	bezeichnet ein Element vom Typ enum.
class-value	symbolische Konstante (S), Wert von *class-name*.

```
<ARRAY(size),type | STRUCT>
array-name (dimension)
(a1) value1 (a2) value2 (a3) value3 ...
```

size	Primärspeicher-Länge in Bytes.
type	Datentyp (CHARS, INT, FLOAT,...), wenn das Array aus einem bestimmten Datentyp besteht.
STRUCT	das Array besitzt eine komplexe, aus unterschiedlichen Datentypen zusammengesetzte Struktur.
array-name	bezeichnet ein Element vom Typ Array.
dimension	die Dimensionen des Arrays.
(*a1*) *value1* (*a2*) *value2* (*a3*) *value3* ...	*a1*, *a2*, *a3*, ... bezeichnet die Unterelemente des Arrays, *value1*, *value2*, *value3*, ... deren Werte. Die Darstellung der Werte hängt vom jeweiligen Datentyp ab. Bei XMAX werden alle Unterelemente ausgegeben. Bei XFLAT werden keine Unterelemente ausgegeben, siehe auch Hinweis 1. Details zu Array-Bereichen siehe Hinweis 2.

```
<STRUCT(size)>
level struct-name
sub-elements
```

size	Speicherlänge in Bytes.
level	Schachtelungstiefe der Struktur oder eines Struktur-Elementes (01, 02, 03, ...). 01 steht für die oberste Ebene.
struct-name	bezeichnet ein Element vom Typ Struktur.
sub-elements	weitere Elemente, die in der Struktur enthalten sind. Bei XMAX werden alle Elemente ausgegeben. Bei XFLAT wird nur ein Teil der Elemente, siehe Abschnitt „Strukturen mit XFLAT“. Siehe auch Hinweis 1 am Ende der Liste.

```
<SET(size)>
01 set-name
<UNSIGN(size),D>
 02 class-name1 = class-value1
<UNSIGN(size),D>
 02 class-name2 = class-value2
...
```

size	jeweilige Speicherlänge in Bytes.
set-name	bezeichnet ein Element vom Typ Set.
class-name1/2... *class-value1/2...*	Namen der symbolischen Konstanten. Werte der symbolischen Konstanten. Bei XMAX und XFLAT mit *set-name* werden alle Elemente ausgegeben, bei XFLAT ohne *set-name* wird nur die Ebene 01 ausgegeben.

Hinweise

1. Um den gesamten Inhalt eines Strings, einer Struktur oder eines Arrays aufgeteilt auf mehrere Zeilen abzufragen, verwenden Sie folgende Syntax:

   ```
   %SDUMP name {T | H | Fn | P} = {XMAX | MAX}
   ```

2. Um den Inhalt der Array-Elemente innerhalb des bestimmten Bereichs abzufragen, verwenden Sie folgende Syntax:

   ```
   %SDUMP name [from:to] {T | H | Fn | P} = {XMAX | XFLAT | MAX}
   ```

Strukturen mit XFLAT

Für Strukturen generiert AID verschiedene XFLAT-Datenausgaben abhängig davon, ob das %SDUMP-Kommando Datenoperanden enthält oder nicht.

- %SDUMP ohne Datenoperand

  ```
  %SDUMP {T | H | Fn | P} = XFLAT
  ```

  Nur der Typ-Tag und der Name werden ausgegeben (Ebene 01). Die Ausgabe der Struktur-Elemente entfällt.

- %SDUMP mit einer Struktur als Operand

  ```
  %SDUMP structure-name {T | H | Fn | P} = XFLAT
  ```

  Der Struktur-Name und die Struktur-Elemente werden ausgegeben (Ebene 02). Elemente mit elementaren Typen werden normal ausgegeben, Elemente mit Array-Typ mit Namen und Dimension, Elemente mit Struktur-Typ nur mit ihrem Namen. Dabei geht jedem Element ein Typ-Tag voraus. Der Name wird durch eine Zahl, die Schachtelungstiefe, erweitert.

- %SDUMP mit einer Unterstruktur als Operand

  ```
  %SDUMP structure-name.substruct-name {T | H | Fn |P} = XFLAT
  ```

  Gibt zusätzlich die Struktur-Elemente der Unterstruktur aus (Ebene 03)

  Es können auch weitere Schachtelungstiefen angegeben werden, indem die weiteren Unterstruktur-Namen durch einen Punkt verkettet werden:

  ```
  structure-name.substruct1-name.substruct2-name.substruct3-name. ....
  ```

Um den gesamten Inhalt einer Struktur und ihrer Unterstrukturen abzufragen, verwenden Sie XMAX statt XFLAT.

Beispiele

1. `%aid c=yes`
   `%sdump`

   Mit diesem Kommando wird ein symbolischer Dump aller Daten der aktuellen Aufrufhierarchie angefordert. Der Wert für *medium-u-menge* ist T=MAX. Die Übersetzungsliste zu dieser SDUMP-Ausgabe finden Sie im Beispiel 10 der %DISPLAY-Beschreibung auf Seite 172.

```
SRC_REF: 46  SOURCE: OUTPUT.C  PROC: main  ******************************
int1            =      -32768

int2            =         234

int3            =        -567

un1             =     65535

un2             =       78900

un3             =       90123

sll             =   -9223372036854775808

ull             =   18446744073709551615

fl1             = +.1234559 E+003

fl2             = +.5678899999999999 E+003

fl3             = +.3334439999999999 E+003
```

   Die %SDUMP-Ausgabe beginnt mit einer Kopfzeile. Sie enthält die Source-Referenz der Anweisung, bei der das Programm unterbrochen wurde und den Namen der aktuellen Übersetzungseinheit. Es folgen die Variablen vom Typ `signed` und `unsigned int` sowie `float` mit Namen und Inhalt.

```
char1           =  |A|

char2           =     -63

chstr           = 01001188

chvek           = "Character-Vektor"

c_out           = 01001248
```

   Die char-Variablen `char1` und `char2` gibt AID im Character-Format aus, d.h. das dem Inhalt entsprechende Zeichen wird ausgegeben. Falls Sie mit dem Operanden DELIM des Kommandos %AID keinen anderen Begrenzer vereinbart haben, wird die Ausgabe durch senkrechte Striche (bei Variablen vom Typ `char`) bzw. Anführungszeichen (bei

Strings) begrenzt.
Zu den Zeiger-Variablen `chstr` und `c_out` wird die Adresse ausgegeben, auf die sie verweisen.
Da `%AID C=YES` gesetzt wurde, gibt AID den `char`-Vektor `chvek` als C-String-Literal aus.

```
01          ::std
 02         printf          = 01001E5C

::main             = 01000000
```

Zu den Funktionen `printf` und `main` werden die Prologadressen ausgegeben.

2. `%sd %nest`

   Die aktuelle Aufrufhierarchie zu Beispiel 1 soll ausgegeben werden.

```
SRC_REF: 55  SOURCE: OUTPUT.C  PROC: main  ******************************
ABSOLUT: V'113CF88'   SOURCE: IC@RT20A  PROC: IC@RT20A  ******************
ABSOLUT: V'10015E8'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  ******************
```

   Das Programm wurde vor der Ausführung der Anweisung in Zeile 55 unterbrochen. Die Unterbrechungsstelle liegt in der `main`-Funktion, die Übersetzungseinheit heißt `OUTPUT.C`.

3. Gegenüberstellung der Aufrufhierarchie beim symbolischen Testen und beim Testen auf Maschinencode-Ebene:

   Als Beispiel dient Programm `STRING.C` aus Abschnitt „C++-Anwendungsbeispiel in BS2000“ auf Seite 322. Das Programm wurde an der Source-Referenz S’50’ unterbrochen.

   – Aufrufhierarchie mit LSD

```
/%symlib mylib
/%sdump %nest
SRC_REF: 50  SOURCE: STRING.C
                PROC: string::operator const char *() const  ********
SRC_REF: 30  SOURCE: STRING.C
                PROC: string::string(const string &)  ****************
SRC_REF: 72  SOURCE: STRING.C  PROC: main  ************************
ABSOLUT: V'113CF88'     SOURCE: IC@RT20A  PROC: IC@RT20A  ************
ABSOLUT: V'1002A70'     SOURCE: IC@MAIN@  PROC: IC@MAIN@  ************
```

– Aufrufhierarchie ohne LSD

```
/%symlib
/%sdump %nest
ABSOLUT: V'10002FE'       SOURCE: STRING$O&@
                PROC: operator const char *  ***********************************
ABSOLUT: V'1000816'       SOURCE: STRING$O&@   PROC: string  *********
ABSOLUT: V'1000B1C'       SOURCE: STRING$O&@   PROC: main  ***********
ABSOLUT: V'113CF88'       SOURCE: IC@RT2OA  PROC: IC@RT2OA  **********
ABSOLUT: V'1002A70'       SOURCE: IC@MAIN@  PROC: IC@MAIN@  **********
```

In der Aufrufhierarchie auf Maschinencode-Ebene wird als Source statt des Namens der Übersetzungseinheit der CSECT-Name eingesetzt. Unter PROC gibt AID statt der Funktionsnamen aus dem Quellprogramm die vom Compiler erzeugten Entry-Namen der Funktion aus.

4. Das folgende Programmbeispiel enthält die rekursive Funktion `facul`.

   Mit `%in s'16:3' <%sd; %sd %nest>` veranlassen Sie AID, vor dem endgültigen Verlassen von facul alle Daten und die Funktionen der aktuellen Aufrufhierarchie auszugeben. Zunächst das Source-Error-Listing:

```
*** SOURCE - ERROR - LISTING ** BS2000 C/C++ COMPILER 03.2E21 DATE:2015-02-27 PAGE:    1
            SOURCENAME:*LIB-ELEM(MYLIB,EXAMP.C(*HIGHEST-EXISTING),S)
_____________________________________________________________________________

   EXP   INC    FILE   SRC BLOCK
   LIN   LEV     NO    LIN  LEV

     1     0      0      1     0  #include <stdio.h>
  1746     0      0      2     0  int facul(int n);
  1747     0      0      3     0  int main()
  1748     0      0      4     0  {
  1749     0      0      5     1     unsigned n;
  1750     0      0      6     1     printf("n? : ");
  1751     0      0      7     1     scanf("%d",&n);
  1752     0      0      8     1     if (n > 16) return 0;
  1753     0      0      9     1     printf("%d! : %d\n", n, facul(n));
  1754     0      0     10     1     return 0;
  1755     0      0     11     0  }
  1756     0      0     12     0
  1757     0      0     13     0  int facul(int n)
  1758     0      0     14     0  {
  1759     0      0     15     1     if (n < 0) return (-1);
  1760     0      0     16     1     if (n == 0 || n == 1) return (1);
  1761     0      0     17     1     else return (n * facul(n-1));
  1762     0      0     18     1  }
```

Im folgenden nun die Kommandoeingabe (fettgedruckt) und die entsprechende Ausgabe des %SDUMP:

```
/LOAD-PROG *M(MYLIB,EXAMP,RUN-MODE=ADVANCED,PROG-MODE=ANY),TEST-OPT=AID
%  BLS0523 ELEMENT 'EXAMP', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB'
IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$EXAMP$', VERSION ' ' OF '1999-01-11
12:47:37' LOADED
/%aid low
/%in s'16:3' <%sd; %sd %nest>
/%r
 n? : 4
*** TID: 000401F9 *** TSN: 88G5 ****************************************
SRC_REF: 16:3  SOURCE: EXAMP.C       PROC: facul  ***********************
n               =           1

SRC_REF: 17:2  SOURCE: EXAMP.C       PROC: facul  ***********************
n               =           2

SRC_REF: 17:2  SOURCE: EXAMP.C       PROC: facul  ***********************
n               =           3

SRC_REF: 17:2  SOURCE: EXAMP.C       PROC: facul  ***********************
n               =           4

SRC_REF: 9  SOURCE: EXAMP.C       PROC: main  ****************************
n               =           4

::printf        = 01001B14

::scanf         = 01001B4C

::main          = 01000000
::facul         = 01000180
SRC_REF: 16:3  SOURCE: EXAMP.C       PROC: facul  ***********************
SRC_REF: 17:2  SOURCE: EXAMP.C       PROC: facul  ***********************
SRC_REF: 17:2  SOURCE: EXAMP.C       PROC: facul  ***********************
SRC_REF: 17:2  SOURCE: EXAMP.C       PROC: facul  ***********************
SRC_REF: 9  SOURCE: EXAMP.C       PROC: main  ****************************
ABSOLUT: V'113CF88'   SOURCE: IC@RT2OA  PROC: IC@RT2OA  ******************
ABSOLUT: V'10012A8'   SOURCE: IC@MAIN@  PROC: IC@MAIN   ******************
4! : 24
%  CCM0998  CPU TIME USED: 0.0269 SECONDS
```

Zunächst wird zu jedem Aufruf von `facul` die Source-Referenz und der zugehörige Inhalt von `n` ausgegeben. Zum letzten Aufruf von `facul`, der hier der obersten Hierarchiestufe entspricht, gibt AID zusätzlich zur Source-Referenz und zu `n` die Adressen der Funktionen `scanf`, `printf` und `main` aus.
In der anschließenden Ausgabe der aktuellen Aufrufhierarchie sehen Sie ebenfalls zu jedem Selbstaufruf der Funktion `facul` eine Zeile mit derselben Source-Referenz bis zum letzten Aufruf von `facul`, der übergeordneten main-Funktion und der Routinen `IC@RT2OA` und `IC@MAIN@` des Laufzeitsystems.

5. Beispiel für XFLAT und XMAX

   Folgendes C-Programm soll getestet werden:

```
#include <stdio.h>
#include <string.h>
struct Universe
{
  int id;
  struct Galaxy
  {
    int id;
    struct Planet
    {
      int id;
      struct Continent
      {
        int id;
        struct Country
        {
          int id;
          struct Region
          {
            int id;
            struct Dense
            {
              int id;
              struct Forest
              {
                int id;
                struct Anthill
                {
                  int id;
                  struct Ant
                  { float x,y;
                  } ant;
                } anthill;
              } forest;
            } dense;
          } region;
        } country;
      } continent;
    } planet;
  } galaxy;
} universe;

void main(void)
{
```

```
  struct {
     struct {int x; char y;} inner[5][5];
     struct {float u, v;} *pointer;
     union {
            char c100[100];
            char               chx12345678901234567890123456789;
            char unsigned      uch12345678901234567890123456789;
            short              isx12345678901234567890123456789;
            short unsigned     isu12345678901234567890123456789;
            int                ixx12345678901234567890123456789;
            int unsigned       iux12345678901234567890123456789;
            long               ilx12345678901234567890123456789;
            long unsigned      ilu12345678901234567890123456789;
            float              flx12345678901234567890123456789;
            double             dbx12345678901234567890123456789;
           } databox;
     } outer [2];

strcpy(outer[0].databox.c100,
"12345678901234567890123456789012345678901234567890123456789012345678901234567890");

STOP: ;
}
```

Nach dem Laden des C-Programms werden folgende AID-Kommandos eingegeben:

```
%AID C=YES
%INSERT STOP
%RESUME
```

Die folgenden Varianten zeigen die Wirkung verschiedener Angaben bei XFLAT und XMAX:

- XFLAT ohne Daten-Operand
- XFLAT für das Array-Element outer
- XFLAT für die Struktur databox des Array-Elements outer
- XMAX für ein Unter-Array-Element
- XMAX für ein tief geschachteltes Element

XFLAT ohne Daten-Operand

Wenn Sie XFLAT ohne Operand angeben, werden von den Strukturen nur die oberste Ebene 01 ausgegeben.

```
/%SD T=XFLAT
SRC_REF: 59  SOURCE: X-C2  PROC: main  *******
<ADDR(4),X>
STOP          = 010000B6

<ARRAY(624),STRUCT>
outer( 0: 1)

<ADDR(4),X>
::strcpy      = 7DB2BEC8

<ADDR(4),X>
::main        = 01000000

<STRUCT(44)>
01         ::universe
```

XFLAT für das Array-Element outer

```
/%SD outer[0] T=XFLAT
SRC_REF: 59  SOURCE: X-C2  PROC: main  ******
<STRUCT(312)>
01          outer( 0)
<ARRAY(200),STRUCT>
 02          inner( 0: 4, 0: 4)
<POINTER(4),X>
 02          pointer  = 00000000
<STRUCT(104)>
 02          databox
```

XFLAT für die Struktur databox des Array-Elements outer

Lange Strings werden verkürzt dargestellt.

```
/%SD outer[0].databox T=XFLAT
SRC_REF: 59  SOURCE: X-C2  PROC: main
*************************************
<STRUCT(104)>
02          outer.databox( 0)
<CHARS(71),C>
 03         c100  =
"12345678901234567890123456789012345678901234567890123456789012345678901234567890" ...
<CHARS(1),C>
 03         chx12345678901234567890123456789 = |1|
```

```
<UNSIGN(1),D>
 03         uch12345678901234567890123456789 = 241
<INT(2),D>
 03         isx12345678901234567890123456789 =  -3598
<UNSIGN(2),D>
 03         isu12345678901234567890123456789 =  61938
<INT(4),D>
 03         ixx12345678901234567890123456789 =  -235736076
<UNSIGN(4),D>
 03         iux12345678901234567890123456789 = 4059231220
<INT(4),D>
 03         ilx12345678901234567890123456789 =  -235736076
<UNSIGN(4),D>
 03         ilu12345678901234567890123456789 = 4059231220
<FLOAT(4),E>
 03         flx12345678901234567890123456789 = -.9531502 E+059
<FLOAT(8),E>
 03         dbx12345678901234567890123456789 = -.9531502657561182 E+059
```

XMAX für ein Unter-Array-Element

Das Element *inner* ist selbst wieder ein Teil eines Array-Elements.

```
/%SD outer[0].inner[2][2] T=XMAX
SRC_REF: 59  SOURCE: X-C2  PROC: main  ******
<STRUCT(8)>
02         outer.inner( 0, 2, 2)
<INT(4),D>
 03         x         =  0
<CHARS(1),C>
 03         y         = |.|
```

XMAX für ein tief geschachteltes Element

```
/%SD universe.galaxy.planet.continent.country.region.dense.forest.anthill.ant T=XMAX
SRC_REF: 59  SOURCE: X-C2  PROC: main  ****************************************
<STRUCT(8)>
02  ::universe.galaxy.planet.continent.country.region.dense.forest.anth ill.ant
<FLOAT(4),E>
 03  x         = +.0000000 E+000
<FLOAT(4),E>
 03  y         = +.0000000 E+000
```

%SET

Mit %SET übertragen Sie Speicherinhalte oder AID-Literale auf Speicherstellen im geladenen Programm. Vor der Übertragung werden die Speichertypen von *sender* und *empfänger* auf Verträglichkeit geprüft. Der Inhalt von *sender* wird in den Speichertyp von *empfänger* konvertiert. AID überträgt nur byteweise.

– Mit *sender* bezeichnen Sie eine Variable, ein Objekt einer Klasse oder eine Komponente daraus, eine Struktur oder die Komponente einer Struktur, ein Vektorelement, eine Konstante, eine Länge, eine Adresse, einen Durchlaufzähler, ein Register oder ein AID-Literal. *sender* kann im virtuellen Speicher des geladenen Programms oder in einer Dump-Datei liegen.

– Mit *empfänger* bezeichnen Sie eine Variable, ein Objekt einer Klasse oder eine Komponente daraus, eine Struktur oder die Komponente einer Struktur, ein Vektorelement, einen Durchlaufzähler oder ein Register, das überschrieben werden soll. *empfänger* kann nur im virtuellen Speicher des geladenen Programms liegen.

Kommando	Operand
%S[ET]	sender INTO empfänger

Im Gegensatz zum %MOVE überprüft AID beim %SET vor der Übertragung, ob der Speichertyp von *empfänger* mit dem von *sender* verträglich ist und ob der Inhalt von *sender* zu seinem Speichertyp passt. Andernfalls lehnt AID die Übertragung ab und gibt eine Fehlermeldung aus.
Ist *sender* länger als *empfänger*, wird er entsprechend seinem Speichertyp links oder rechts abgeschnitten, und AID gibt eine Warnung aus. *sender* und *empfänger* können sich überlappen. Bei der numerischen Übertragung wird *sender* bei Bedarf in den Speichertyp von *empfänger* konvertiert, und der Inhalt von *sender* wird werterhaltend in *empfänger* abgelegt. Passt der Wert nicht vollständig in *empfänger*, wird eine Warnung ausgegeben.

Sind *sender* und *empfänger* Zeiger auf Objekte von Klassen und gilt dabei Folgendes:

– *sender* zeigt auf ein Objekt einer abgeleiteten Klasse,

– *empfänger* zeigt auf ein Objekt einer Basisklasse dieser abgeleiteten Klasse,

dann werden nur die Einträge der Basisklasse übertragen. Alle weiteren Einträge der abgeleiteten Klasse werden ignoriert. AID prüft die Zulässigkeit der Übertragung wie C++: ist die Basisklasse nicht eindeutig von der abgeleiteten Klasse aus zu erreichen, wenn z.B. die über *empfänger* referenzierte Klasse zugleich direkte und indirekte Basisklasse der mit *sender* angesprochenen abgeleiteten Klasse ist, dann lehnt AID die Übertragung ab und gibt eine Fehlermeldung aus.

Für ältere Objekte, die mit einer C/C++-Compilerversion bis 2.2C übersetzt wurden, fehlt die Information zu Basisklassen und abgeleiteten Klassen in der LSD. Daher kennt AID bei diesen Objekten den Bezug zwischen Basis- und abgeleiteten Klassen nicht. Die oben beschriebene Übertragung von abgeleiteten Klassen in Basisklassen via Zeiger ist dann nicht möglich.

Eine Übersicht darüber, welche Speichertypen miteinander verträglich sind und wie übertragen wird, finden Sie am Ende der %SET-Beschreibung.

Unmittelbar nach dem Laden können Sie nur globale und static vereinbarte Daten ansprechen. Zum Zugriff benötigt AID die entsprechende Qualifikation.

Neben den hier beschriebenen Operandenwerten können Sie auch die im Handbuch für das Testen auf Maschinencode-Ebene [2] beschriebenen Operandenwerte einsetzen.

Mit `%AID CHECK=ALL` können Sie zur Kontrolle einen Änderungsdialog einschalten, der Ihnen vor Durchführung der Übertragung den alten und neuen Inhalt von *empfänger* zeigt und Ihnen die Möglichkeit zum Abbruch des %SET gibt.

%SET verändert den Programmzustand nicht.

sender INTO empfänger

Für *sender* oder *empfänger* können Sie eine Variable, ein Objekt einer Klasse oder eine Komponente daraus, eine Struktur, eine Strukturkomponente, ein Vektorelement, einen C-String (falls `%AID C=YES` gesetzt ist), einen Durchlaufzähler, ein Register oder eine komplexe Speicherreferenz angeben. Adressen, Längen, Konstanten und Literale können Sie nur als *sender* einsetzen. *sender* kann sowohl im virtuellen Speicherbereich des geladenen Programms als auch in einer Dump-Datei liegen. *empfänger* dagegen kann nur im virtuellen Speicherbereich des geladenen Programms liegen. Übertragen bzw. überschreiben Sie Programmbereiche mit Befehlscode, kann das zu unerwünschten Ergebnissen führen, wenn Adressen betroffen sind, die zu einem *control*- oder *trace-bereich* gehören oder auf die mit %INSERT ein *testpunkt* gesetzt wurde (siehe „AID-Basishandbuch“, Abschnitt „Wechselwirkungen“ [1]).

sender-OPERAND

empfänger-OPERAND

- Steht der Punkt an führender Stelle, ist er das Kennzeichen für eine *vorqualifikation*. Sie muss mit einem vorhergehenden %QUALIFY definiert worden sein. Aufeinanderfolgende Qualifikationen werden durch einen Punkt getrennt. Außerdem muss zwischen der letzten Qualifikation und dem anschließenden Operandenteil ein Punkt stehen.

qua Eine oder mehrere Qualifikationen geben Sie an, wenn *sender* oder *empfänger* von der aktuellen Unterbrechungsstelle aus anders nicht zu erreichen sind oder um einen Datennamen anzusprechen, der an der Unterbrechungsstelle durch eine gleichnamige Definition lokal verdeckt ist. Sie geben nur die Qualifikationen an, die zur eindeutigen Ansprache des Speicherobjekts genügen.

{E={VM | Dn} für *sender* | E=VM für *empfänger*}
: geben Sie nur an, wenn für einen Datennamen, einen Namespace, eine Klasse, ein Objekt einer Klasse, einen Anweisungsnamen, eine Source-Referenz oder ein Schlüsselwort die aktuelle Basisqualifikation nicht gelten soll (siehe %BASE). *sender* kann sowohl im virtuellen Speicher als auch in einer Dump-Datei liegen. *empfänger* kann dagegen nur im virtuellen Speicher liegen.

S=srcname
: geben Sie nur an, wenn Sie einen Datennamen, einen Namespace, eine Klasse, ein Objekt einer Klasse, einen Anweisungsnamen oder eine Source-Referenz ansprechen, die nicht in der aktuellen Übersetzungseinheit liegen (siehe Abschnitt „Qualifikationen“ auf Seite 21).

:: Die beiden vorangestellten Doppelpunkte verwenden Sie, um ein globales Datum anzusprechen, das durch eine gleichnamige Definition an der Unterbrechungsstelle lokal verdeckt ist. Außerdem müssen Sie die beiden Doppelpunkte vor den Namen eines globalen Datums oder einer Funktion setzen, weil entweder das Datum oder die Funktion nicht in der Aufrufhierarchie liegen oder weil deren Definition erst nach der Unterbrechungsstelle steht.
Im Gegensatz zu den übrigen Qualifikationen wird zwischen den beiden Doppelpunkten und dem anschließenden Operanden kein Punkt geschrieben.

PROC=funktion
: geben Sie nur an, wenn Sie den Namen eines Datums oder eines Objekts einer Klasse ansprechen wollen, das zwar in der aktuellen Funktion definiert ist, aber von einer gleichnamigen Definition an der Unterbrechungsstelle verdeckt wird. Außerdem geben Sie eine PROC-Qualifikation an, wenn Sie eine Marke oder einen static vereinbarten Datennamen ansprechen wollen, der einer Funktion außerhalb der aktuellen Aufrufhierarchie zugeordnet ist (siehe Kapitel „Adressierung in C- und C++-Programmen“ auf Seite 21). Wenn Sie eine Source-Referenz angeben, die in einer Instanz eines Funktionstemplates liegt oder die einer Funktion zugeordnet ist, die in einer Instanz eines Klassentemplates definiert ist (siehe Abschnitt „Templates“ auf Seite 98), müssen Sie bei Mehrdeutig-

keit ebenfalls die entsprechende PROC-Qualifikation davorschreiben.
Bei Funktionen aus C-Programmen ist *funktion* der im Quellprogramm vergebene Name der Funktion ohne Klammern und ohne Signatur.
Funktionen aus C++ - Programmen müssen Sie, je nachdem, um welchen Typ es sich handelt, in der Schreibweise `n'...'` oder `t'...'` angeben. Ist die Funktion in einem Namespace oder in einer Klasse definiert, so wird dem Funktionsnamen die Namespace- oder Klassen-Qualifikation vorangestellt. Die Signatur `void` darf nicht mehr geschrieben werden. Wie auch in C++ möglich geben Sie in diesem Fall nur die beiden Klammern nach dem Funktionsnamen an. Es ergibt sich die folgende Syntax (*f_template* und *signatur* sind in der Syntax aus Platzgründen abgekürzt):

```
PROC=[namespace::[...]][klasse::[...]]{n'funktion([sign])'          }
                                      {t'f_templ<arg[,...]>([sign])'}
```

Abweichend davon werden die Funktionen `main`, `_ _STI_ _` und alle Funktionen mit C-Linkage auch beim Testen in C++-Programmen nur mit dem Funktionsnamen bezeichnet (siehe Seite 60).

Ist die Funktion in einer lokalen Klasse definiert, so müssen Sie vor der PROC-Qualifikation noch eine weitere PROC-Qualifikation für die übergeordnete Funktion angeben, die die Definition der lokalen Klasse enthält. Zwischen die beiden PROC-Qualifikationen schreiben Sie bei Funktionen, die in einem inneren Block der übergeordneten Funktion definiert sind, eine oder ggfs. mehrere BLK-Qualifikationen (siehe Seite 62).
Das Ansprechen von Funktionen, die in lokalen Klassen innerer Blöcke definiert sind, wird nur bei Programmen unterstützt, die mit einem C/C++-Compiler ab V3.0B übersetzt wurden.

BLK='[f-]n[:b]'
geben Sie an, wenn Sie einen Datennamen ansprechen wollen, der einem Block innerhalb der aktuellen Aufrufhierarchie zugeordnet ist und von einer gleichnamigen Definition an der Unterbrechungsstelle verdeckt wird oder wenn Sie einen static vereinbarten Datennamen ansprechen wollen, der einem Block außerhalb der aktuellen Aufrufhierarchie zugeordnet ist (siehe Kapitel „Adressierung in C- und C++-Programmen“ auf Seite 21).

Des weiteren geben Sie eine BLK-Qualifikation an, wenn Sie mit einer nachfolgenden PROC-Qualifikation eine Funktion aus einer lokalen Klasse bezeichnen

und die Definition dieser lokalen Klasse steht im angegebenen Block (siehe oben PROC=*funktion*).

Der Name für einen Block wird aus Zeilennummer (*n*) und gegebenenfalls FILE-Nummer (*f*) und relativer Blocknummer (*b*) gebildet

NESTLEV= level-nummer

level-nummer Nummer einer Ebene in der aktuellen Aufrufhierarchie
Auf *level-nummer* muss *datenname* folgen.
NESTLEV= *level-nummer* geben sie an, wenn Sie einen Datennamen in einer bestimmten Ebene der aktuellen Aufrufhierachie ansprechen wollen. Diese Qualifikation kann nur mit E= kombiniert werden, nicht mit anderen Qualifikationen.

namespace
ist der im Quellprogramm deklarierte Name eines Namespaces.

Den Namen eines Namespaces geben Sie an, um den Adressierungspfad zu Klassen, Daten oder Funktionen zu beschreiben, die in dem Namespace definiert sind (siehe Abschnitt „Namespaces“ auf Seite 89), wenn die gewünschte Komponente des Namespaces an der Unterbrechungsstelle nicht sichtbar ist.

Vor der Namespace-Qualifikation sind nur die E- oder S-Qualifikation oder die beiden Doppelpunkte (`::`) für den globalen Namespace möglich.

{ klasse | this-> | objekt | zeiger->objekt}
ist der im Quellprogramm deklarierte Name einer Klasse, der `this`-Zeiger, der Name eines Objekts einer Klasse oder ein Zeiger auf ein Objekt einer Klasse. Klassennamen, den `this`-Zeiger mit nachfolgendem Pointer-Operator oder die Namen von Objekten von Klassen geben Sie an, um den Adressierungspfad zu Daten-Membern zu beschreiben (siehe Abschnitt „Klassen“ auf Seite 65).
Handelt es sich bei der Klasse um eine Instanz eines Klassentemplates, müssen Sie die folgende Schreibweise verwenden: `t'`*k_template*`<`*arg*`[,...]>'`. Existiert nur eine einzige Instanz des Templates, genügt die Angabe: `t'`*k_template*`'`.

Die Namen von Objekten oder von Zeigern auf Objekte von Klassen verwenden Sie, um Objekte von Klassen als Ganzes zu übertragen. Dies ist nur unter den folgenden Voraussetzungen möglich:

1. *sender* und *empfänger* referenzieren Objekte derselben Klasse. Ist das Programm in einer dynamischen Member-Funktion der Klasse unterbrochen, dann können Sie das gesamte Objekt der Klasse auch mit `*this` ansprechen.

2. *sender* ist ein Zeiger auf ein Objekt einer abgeleiteten Klasse und *empfänger* zeigt auf ein Objekt einer zugehörigen Basisklasse. Diese Übertragung entspricht somit der C++-Zuweisungsanweisung:
   `pointer_a = pointer_b;`
   wenn `pointer_a` ein Zeiger auf ein Objekt einer Basisklasse und `pointer_b` ein Zeiger auf ein Objekt einer abgeleiteten Klasse dieser Basisklasse ist.

AID überträgt den dynamischen Datenteil des Objekts der Klasse und, falls vorhanden, vom Compiler generierte Hilfsvariablen und die Adresse der Tabelle der virtuellen Funktionen. Jede Komponente wird entsprechend ihrem Speichertyp übertragen. Falls *objekt* oder *zeiger* an der Unterbrechungsstelle nicht sichtbar sind, müssen Sie entsprechend qualifizieren. Vor `this` ist nur eine Basisqualifikation sinnvoll.
Bei abgeleiteten Klassen umfasst *sender* oder *empfänger* auch die Basisklassen, bei geschachtelten Klassen auch die inneren Schachtelungen. Wenn *empfänger* auf ein Objekt einer Basisklasse von *sender* verweist, werden nur die Daten der Basisklasse übertragen.

Statische Daten-Member können Sie nur einzeln ansprechen. Sie erreichen sie von jeder Stelle des Programms aus über den zugehörigen Klassennamen mit den beiden nachfolgenden Doppelpunkten. Bei geschachtelten Klassen enthält der Pfad zum Datum alle Klassennamen von außen nach innen, jeweils durch zwei Doppelpunkte getrennt. Der äußerste Klassennamen benötigt die dem Gültigkeitsbereich entsprechende Qualifikation. Ist das Programm in einer Member-Funktion der Klasse unterbrochen, so gelten für das Ansprechen von statischen Daten-Member die Regeln des Klassenscopes, d.h. wenn das Daten-Member nicht durch eine gleichnamige Defintion verdeckt ist, kann es unqualifiziert angesprochen werden.

Liegt die aktuelle Unterbrechungsstelle in einer dynamischen Member-Funktion, können Sie die Daten-Member genauso ansprechen wie in C++, d.h. wenn das Datum an der Unterbrechungsstelle sichtbar ist, können Sie es mit AID direkt, also ohne Qualifikation, ansprechen. Lokal verdeckte Daten benötigen wie in C++ die entsprechende Klassenqualifikation. Außerdem können Sie dynamische Daten-Member über den `this`-Zeiger mit anschließendem Pointer-Operator ansprechen. Dies ist gleichbedeutend mit der Verwendung des Objektnamens und einem nachfolgenden Punkt.

Unabhängig von der Unterbrechungsstelle erreichen Sie die dynamischen Daten eines Objekts über den Objektnamen mit nachfolgendem Punkt, falls das Objekt in der aktuellen Aufrufhierarchie liegt.

Endet *sender* oder *empfänger* auf `this`, dann überträgt bzw. überschreibt der %SET die in `this` eingetragene Anfangsadresse des Objekts, auf das `this` verweist. Mit anschließendem Pointer-Operator, also mit `this->`, bezeichnen Sie 4 Bytes ab der Anfangsadresse des aktuellen Objekts im Speichertyp %X.

datenname

ist der im Quellprogramm definierte Name eines Datums. *datenname* wird wie im Quellprogramm angegeben.

Daten können Sie bis auf folgende Ausnahmen wie in C/C++ ansprechen:
Vektorelemente können Sie nur über Index ansprechen, nicht über Zeiger.
Eine Index-Bereichsangabe ist nicht erlaubt.
C-Strings erkennt AID nur, wenn `%AID C=YES` gesetzt ist (siehe Seite 119). Dann können Sie C-String-Vektoren modifizieren, wie Sie es von der Spache C/C++ her gewohnt sind (siehe auch „C-Strings“ auf Seite 37).
Ist `%AID C=NO` gesetzt oder handelt es sich nicht um C-Strings, sondern um Vektoren anderer Datentypen, dann können Vektoren als Ganzes weder übertragen noch überschrieben werden. Wenn Sie einen Vektornamen ohne Index angeben, lehnt AID die Übertragung ab.
Eine Ausnahme bilden die Namen von Vektoren als Übergabeparameter; jedoch sprechen Sie mit dem Namen des Übergabeparameters nicht den Vektor, sondern nur seine Adresse an.
Zum Arbeiten mit Vektoren siehe auch Abschnitt „Indexschreibweise“ auf Seite 31.

Bei Variablen vom Typ `long double` wertet AID nur die ersten 8 Bytes aus.
Variablen vom Typ `char` betrachtet AID stets als Character. Den entsprechenden numerischen Wert können Sie erst nach einer Typmodifikation mit %F (`signed char`) oder %A (`unsigned char`) verwenden. Die Datentypen `unsigned char` und `signed char` behandelt AID dagegen wie Integer-Variablen.

Geben Sie als *empfänger* einen Datennamen vom Typ Pointer-to-Member an, so kann *sender* entweder ebenfalls ein Pointer-to-Member sein, oder Sie geben für *sender* die Adresse eines Daten-Members oder einer Member-Funktion einer Klasse an. Bei der Modifizierung von Pointer-to-Member muss Folgendes gelten:

Die dem Sender zugeordnete Klasse muss mit der Klasse übereinstimmen, auf die sich der Empfänger bezieht oder sie muss eindeutige Basisklasse in der Klasse des Empfängers sein (siehe Abschnitt „Zeiger auf Klassen-Member (Pointer-to-Member)“ auf Seite 77).

datenname kann folgendermaßen angegeben werden. Die Formate können auch kombiniert werden (siehe Abschnitt „Datennamen“ auf Seite 30).

Indexschreibweise:	*datenname* `[`*index*`]` `{...}`
Zeigerschreibweise:	*datenname1* `->` *datenname2*
Strukturqualifizierung:	*übergeordneter datenname*`.` `{...}` *datenname*
Dereferenzierung:	`[(]*{...}` *datenname*`[)]`
Pointer-to-Member-Dereferenzierung:	*datenname1*`.*`*datennname2* oder *datenname1*`->*`*datenname2*

Strukturen können mit %SET nur übertragen werden, wenn sowohl *sender* als auch *empfänger* als Strukturen definiert sind und die Definitionen der Komponenten übereinstimmen.

Zeiger können mit `%SET 0 INTO` *zeiger* auf binär Null gesetzt werden.

```
{funktion    }
{L'label'    }
{S'[f-]n[:a]'}
```

Anweisungsnamen und Source-Referenzen sind Adresskonstanten. Sie können nur als *sender* angegeben werden. Es wird die in der Adresskonstanten hinterlegte Adresse übertragen.
Mit nachfolgendem Pointer-Operator (->) bezeichnen Sie 4 Bytes des an der entsprechenden Adresse stehenden Maschinencodes. Die Maschinenbefehle können Sie mit %DISASSEMBLE ausgeben lassen, um eventuell eine Längenmodifikation vorzunehmen.
funktion`[%al4]->`, `L'`*label*`'->` und `S'[`*f*`-]`*n*`[:`*a*`]'->` können Sie als *sender* und als *empfänger* verwenden (siehe Beispiel 9 im Anschluss an die %SET-Beschreibung, Seite 283).

funktion
: ist der Name einer Funktion, der im Quellprogramm vergeben wurde bzw. der Name einer Bibliotheksfunktion. Sie bezeichnen damit die Anfangsadresse des vom Compiler erzeugten Prologs einer Funktion (siehe PROC=*funktion* auf Seite 268 und Kapitel „C++-spezifische Adressierung“ auf Seite 59).

  Virtuelle Funktionen sprechen Sie mit der folgenden Syntax an:
  `p->n'funktion([signatur])'`

  p ist eine Zeigervariable, die auf das Objekt einer Klasse verweist, das die gewünschte Member-Funktion enthält. Wenn *p* von der aktuellen Unterbrechungsstelle aus nicht zu erreichen ist, muss dem Gültigkeitsbereich entsprechend qualifiziert werden. Liegt die Unterbrechungsstelle in der virtuellen Funktion selbst, dann können Sie die Prologadresse der aktuellen Funktion ansprechen, indem Sie statt *p* den this-Zeiger einsetze*n* (siehe Beschreibung von *this* auf Seite 66 und Abschnitt „Virtuelle Funktionen“ auf Seite 76).

  Wenn Sie eine Funktion ansprechen wollen, die über Pointer-to-Member adressiert wird, stehen Ihnen die folgenden beiden Möglichkeiten zur Verfügung:

  Das Objekt der Klasse, das die gewünschte Funktion enthält, wird mit seinem Namen bezeichnet. Als Dereferenzierungsoperator schreiben Sie `.*`

  ```
  [qua.] objekt.*[objekt.][klasse::][...]pointer-to-function-member
  ```

Das Objekt der Klasse wird über einen Zeiger angesprochen. Als Dereferenzierungsoperator schreiben Sie **->***

```
[qua•] zeiger->*[objekt•][klasse::][...]pointer-to-function-member
```

Das Objekt der Klasse bezeichnen Sie mit dem Operanden, der auf der linken Seite der Dereferenzierungsoperatoren `•*` bzw. **->*** steht:
mit *objekt* bezeichnen Sie das Objekt der Klasse mit seinem Namen,
mit *zeiger* adressieren Sie das Objekt über einen Zeiger.
Rechts der Deferenzierungsoperatoren schreiben Sie den Namen des Pointer-to-Function-Member. Evtl. müssen Sie das Objekt, das die Definition des Pointer-to-Function-Member enthält, und innerhalb des Objekts die zur eindeutigen Ansprache nötige Klassenqualifikation davorschreiben, falls der Pointer-to-Member von der Unterbrechungsstelle aus anders nicht zu erreichen ist.

Wenn Sie mit %SET den Befehlscode einer über Pointer-to-Member angesprochenen Funktion übertragen oder überschreiben wollen, so müssen Sie beachten, dass Sie nicht unmittelbar an eine der obigen Syntaxen den Pointer-Operator anfügen können. Sie müssen vielmehr zunächst durch eine Typmodifikation, nämlich %al4, den Übergang auf die Maschinencode-Ebene auslösen. Es ergibt sich die folgende Syntax:

dereferenzierter-pointer-to-function-member `%al4->`

Sie bezeichnen damit die ersten 4 Bytes des Befehlscodes, der an der Prologadresse steht.

Näheres zum Arbeiten mit Pointer-to-Function-Member finden Sie auf Seite 83.

L'label'
: bezeichnet die Adresse der ersten ausführbaren Anweisung nach einer Marke. *label* ist der Name einer Marke, der im Quellprogramm vergeben wurde.

S'[f-]n[:a]'
: ist eine Source-Referenz und bezeichnet die Adresse einer ausführbaren Anweisung. Die Source-Referenz wird aus der Zeilennummer (n) und gegebenenfalls aus FILE-Nummer (f) und relativer Anweisungsnummer (a) gebildet. Liegt die Source-Referenz in einer Funktion, die durch Instanziierung aus einem Funktionstemplate hervorgegangen ist oder ist die Funktion, die die Source-Referenz enthält, in einer Instanz eines Klassentemplates definiert, so müssen Sie bei Mehrdeutigkeit vor die Source-Referenz die entsprechende PROC-Qualifikation schreiben.

schlüsselwort

ist ein Durchlaufzähler, der Befehlszähler oder ein Register. Vor *schlüsselwort* können Sie nur eine Basisqualifikation angeben.
Im AID-Basishandbuch [1], sind die impliziten Speichertypen der Schlüsselwörter angegeben.

```
%•[subkdoname]     Durchlaufzähler
%•                 Durchlaufzähler des gerade aktiven Subkommandos
%PC                Befehlszähler (Program Counter)
%n                 Mehrzweckregister, 0 ≤ n ≤ 15
%nD|E              Gleitpunktregister, n = 0,2,4,6
%nQ                Gleitpunktregister, n = 0,4
%nG                AID-Mehrzweckregister, 0 ≤ n ≤ 15
%nGD               AID-Gleitpunktregister, n = 0,2,4,6
```

Der Befehlszähler enthält die Adresse, an der das Programm mit %CONTINUE, %RESUME oder %TRACE fortgesetzt wird. Eine andere Fortsetzungsadresse können Sie vereinbaren, wenn Sie den Befehlszähler (%PC) überschreiben. Sie müssen allerdings selbst dafür sorgen, dass Registerstände, Dateistatus, Inhalte von Indizes und ähnliches zur neuen Fortsetzungsadresse passen, damit das Programm fehlerfrei fortgesetzt werden kann.

kompl-speicherref

Folgende Operationen können darin vorkommen (siehe AID-Basishandbuch, Abschnitt „Komplexe Speicherreferenzen“ [1]):

- Adressversatz (•)
- indirekte Adressierung (->)
- Typmodifikation (%T(datenname), %X, %C, %E, %D, %F, %A, %S, %SX)
- Längenmodifikation (%L(...), %L=(ausdruck), %Ln)
- Adressselektion (%@(...))

Mit einer expliziten Typ- oder Längenmodifikation können Sie die Speichertypen von *sender* und *empfänger* einander anpassen. Passt der Speichertyp jedoch nicht zum Speicherinhalt, dann wird dies von AID auch bei der Typmodifikation abgelehnt.

Beginnt eine *kompl-speicherref* mit einer Adresskonstanten (z.B. eine Source-Referenz, eine Marke), muss anschließend der Pointer-Operator geschrieben werden. In diesem Fall muss eine Marke stets mit `L'...'` angegeben werden. Ohne den Pointer-Operator können Adresskonstanten innerhalb der *kompl-speicherref* überall da stehen, wo auch Sedezimalzahlen geschrieben werden können.
Nach Adressversatz (•) oder Pointer-Operation (->) gehen impliziter Speichertyp und implizite Länge der Ausgangsadresse verloren. An der errechneten Adresse gilt der Speichertyp %XL4, falls Sie nicht Typ und Länge explizit angeben.
Für keinen Operanden in einer komplexen Speicherreferenz darf der zugeordnete

Speicherbereich durch einen Adressversatz oder eine Längenmodifikation überschritten werden, sonst führt AID das Kommando nicht aus und schreibt eine Fehlermeldung. Durch die Verbindung von Adressselektion (%@) mit Pointer-Operator (->) verlassen Sie die symbolische Ebene. Nun können Sie die Adresse eines Datums verwenden, ohne auf dessen Bereichsgrenzen achten zu müssen.

& ist der Adressoperator. Damit können Sie die Anfangsadresse eines Datums, eines Objekts einer Klasse oder einer Funktion als *sender* verwenden. Eine Adresse darf nur in einen *empfänger* vom Typ Zeiger übertragen werden. Die *sender* und *empfänger* zugeordneten Datentypen müssen allerdings nicht übereinstimmen.

Wenn jedoch die Adresse eines Objekts einer Klasse in einen Zeiger auf eine Klasse übertragen werden soll, führt AID folgende Prüfung durch:

– Die Klasse, auf die der Empfänger zeigt, muss mit der Klasse, deren Adresse übertragen werden soll, übereinstimmen,

  oder

– sie muss Basisklasse der dem Sender zugeordneten Klasse sein. Diese Basisklasse muss in der Klasse des Senders ein eindeutiges Subobjekt besitzen.

Außerdem besteht die Möglichkeit, mit dem Adressoperator & die relative Adresse eines dynamischen Daten-Members einer Klasse zu ermitteln. Dabei ist Folgendes zu beachten:

– Wenn die Unterbrechungsstelle außerhalb der Klasse liegt, die das Daten-Member enthält, schreiben Sie nach dem Adressoperator die entsprechende Klassenqualifikation und dann den Namen des Datums.

– Liegt die Unterbrechungsstelle dagegen in einer dynamischen Member-Funktion der Klasse, müssen Sie vor den Adressoperator eine Basis- oder Bereichsqualifikation (S-, PROC- oder ::-Qualifikation) setzen, damit AID gewissermaßen von außen auf die Klasse zugreift.

Auch die relative Adresse, also die Distanz des Daten-Members zum Klassenanfang in Bytes, können Sie nur in einen Zeiger übertragen.

Der Adressoperator ist im Gegensatz zum Adressselektor %@(...) (siehe Seite 277) eine reine High-Level-Funktion; daher kann er auf komplexe Speicherreferenzen nicht angewandt werden.

Weitergehende Informationen zum Adressoperator finden Sie im Abschnitt „Adressoperator & und Adressselektor %@(...)“ auf Seite 44.

sizeof()

ist der Längenoperator. Die Länge eines Datums oder einer Klasse wird übertragen.
Um die Länge einer Klasse zu ermitteln, können Sie sowohl den Namen der Klasse selbst als auch den Namen eines Objekts der Klasse als Operanden angeben. Sie erhalten die Anzahl Bytes, die die dynamischen Daten-Member der Klasse und eventuell vom Compiler generierte Hilfsvariablen belegen.
Den Namen eines Namespaces können Sie hier nur im Pfad zu einer Komponente des Namespaces angegeben.

Bitfelder und Registervariablen können Sie nicht angeben.

Ausführlich beschrieben ist der Längenoperator im Abschnitt „Längenoperator sizeof() und Längenselektor %L(...)“ auf Seite 49.

%@(...)

Mit dem Adressselektor (siehe AID-Basishandbuch, Abschnitt „Adress-, Typ- und Längenselektor“ [1]) können Sie die Anfangsadresse eines Datums, des Objekts einer Klasse oder einer komplexen Speicherreferenz als *sender* verwenden. Einen Klassennamen können Sie nur im Pfad zur Basisklasse eines Objekts einer abgeleiteten Klasse angeben, um die Anfangsadresse der dynamischen Daten der Basisklasse zu bezeichnen.
Den Namen eines Namespaces können Sie hier nur im Pfad zu einer Komponente des Namespaces angegeben.

Der Adressselektor lässt sich nicht auf Konstanten anwenden, dazu gehören auch die Marken, die Source-Referenzen und die Funktionen.

%L(...)

Mit dem Längenselektor (siehe AID-Basishandbuch, Abschnitt „Komplexe Speicherreferenzen“ [1]) können Sie die Länge eines Datums oder einer Klasse als *sender* verwenden. Wenn Sie den Längenselektor auf eine Klasse oder ein Objekt einer Klasse anwenden, entspricht das Ergebnis dem von sizeof() in C++; Sie erhalten also die Länge der dynamischen Daten und eventuell vom Compiler generierter Hilfsvariablen.
Den Namen eines Namespaces können Sie hier nur im Pfad zu einer Komponente des Namespaces angegeben.

AID gibt immer die Länge in Bytes aus. Für Bitfelder gibt AID die Anzahl der Bytes aus, über die sich das Bitfeld erstreckt.
Beispiel: `%set %l(var1) into %3g`
Die Länge von `var1` wird übertragen.

%L=(ausdruck)

Mit der Längenfunktion können Sie sich einen Wert errechnen und in *empfänger* abspeichern lassen.
In *ausdruck* können Sie den Inhalt von Speicherreferenzen, Konstanten und Ganzzahlen mit den arithmetischen Operatoren (+,-,*,/) verknüpfen. Speicherreferenzen müssen vom Typ %F oder %A sein (ganzzahlig).
Die Längenfunktion liefert als Ergebnis eine Ganzzahl (siehe AID-Basishandbuch, Abschnitt „Adress-, Typ- und Längenselektor“ [1]).

Falls Sie in Ihrem Programm mit überladenen Operatoren arbeiten, müssen Sie beachten, dass AID das Überladen der Operatoren nicht nachvollzieht, sondern stets die Standard-Operatoren verwendet.

Beispiel: `%set %l=(var1) into %3g`
Der Inhalt von `var1` wird übertragen, wenn er ganzzahlig ist (Datentyp int). Sonst gibt AID eine Fehlermeldung aus.

literal

Alle AID-Literale, die im AID-Basishandbuch, Kapitel „AID-Literale“ [1] beschrieben sind, können Sie beim %SET verwenden. Bitte beachten Sie die dort beschriebenen Konvertierungen der AID-Literale in den jeweiligen Empfänger-Typ:

```
{C'x...x' | 'x...x'C | 'x...x'}   Character-Literal
{X'f...f' | 'f...f'X}             Sedezimal-Literal
{B'b...b' | 'b...b'B}             Binär-Literal
[{±}]n                            Ganzzahl
#'f...f'                          Sedezimalzahl
[{±}]n.m                          Dezimalpunktzahl
[{±}]mantisseE[{±}]exponent       Gleitpunktzahl
```

Ist `%AID C=YES` gesetzt, können Sie auch ein C-String-Literal ("*x...x*") in einen `char`-Vektor übertragen (siehe Beispiel 10 im Anschluss an die %SET-Beschreibung, Seite 283).

%SET-Tabelle

Die folgende Tabelle gibt eine Übersicht über die zulässigen Kombinationen von Sender- und Empfänger-Typen.

Sender	Empfänger				
	int, float %F %A %D	char %C	%X	Zeiger	C-Strings (%AID C=YES)
int, float %F %A %D {±}n	num	–	bin	–	–
#'f...f'	num	–	bin	bin	–
{±}n.m {±}mantE{±}exp	num	–	–	–	–
char %C C'x...x'	num(1)	char	bin	–	–
%X X'f...f' B'b...b'	bin	bin	bin	bin	bin
Zeiger, Adresse	–	–	bin	bin	–
C-Strings C-String- Literale (%AID C=YES)	–	–	–	–	char(1)

Tabelle 6: zulässige Kombinationen von Sender- und Empfängertypen

bin Binäre Übertragung; linksbündig;
sender < *empfänger*: rechts wird mit binären Nullen aufgefüllt.
sender > *empfänger*: rechts wird abgeschnitten.
Ein numerisches Literal (nur Ganzzahl erlaubt) entspricht bei der Übertragung in den Speichertyp %X einem Integer-Wert mit Vorzeichen in der Länge 4 Bytes (%FL4), die binär übertragen werden.

char Character-Übertragung; linksbündig;
sender < *empfänger*: rechts wird mit Leerzeichen (X'40') aufgefüllt.
sender > *empfänger*: rechts wird abgeschnitten.

char(1) Character-Übertragung; linksbündig;
sender < *empfänger*: rechts wird mit binär Null aufgefüllt.
sender > *empfänger*: rechts wird abgeschnitten.

num Numerische Übertragung; werterhaltend;
sender wird bei Bedarf in den Speichertyp von *empfänger* konvertiert.

num(1) Wenn als Sender ein Character-Literal angegeben wird, das nur Ziffern enthält und höchstens 18 Stellen lang ist, und wenn der Empfänger vom Typ numerisch ist, führt AID eine numerische Übertragung durch. Alle übrigen Sender vom Typ Character können nicht in numerische Empfänger übertragen werden.

– keine Übertragung
AID meldet die Unverträglichkeit der Speichertypen.

Beispiele

In einem C-Programm sind die folgenden Variablen, Vektoren und Strukturen definiert:

```
C-Programm
==========================================
         unsigned short  count;
         float           x_arr[16];
         struct tele {
            char         name[10];
            unsigned     number;
            }
        struct tele      person_1;
        struct tele      person_2;
        char             c_arr[10];
        int              i;
==========================================
```

Für die folgenden Beispiele wurde mit `%aid check=all` der Änderungsdialog eingeschaltet. So sehen Sie den Inhalt des Empfangsfelds vor und nach der Ausführung des %SET:

1. `%set #'61' into count`

```
OLD CONTENT:
          1
NEW CONTENT:
         97
%  AIDO274 Change desired? Reply (Y=Yes; N=No)?y
```

Zum selben Ergebnis führt folgendes Kommando:

```
%set 97 into count
```

2. `%qualify proc=main`
   `%set .count into .x_arr[15]`

```
OLD CONTENT:
+.1234499 E+003
NEW CONTENT:
+.9700000 E+002
%  AID0274 Change desired? Reply (Y=Yes; N=No)?y
```

3. `%s 'A' into person_1.name[0]`

```
OLD CONTENT:
|T|
NEW CONTENT:
|A|
%  AID0274 Change desired? Reply (Y=Yes; N=No)?y
```

4. `%s 'ABCDEF' into %@(person-1.name)->%cl6`

```
OLD CONTENT:
|uvwxyz|
NEW CONTENT:
|ABCDEF|
%  AID0274 Change desired? Reply (Y=Yes; N=No)?y
```

5. `%set person_1 into person_2`

```
OLD CONTENT:
01          person_2
 02         name( 0: 9)
            ( 0) |H|  ( 1) |u|  ( 2) |b|  ( 3) |e|  ( 4) |r|  ( 5) |.|
            ( 6) |.|  ( 7) |.|  ( 8) |.|  ( 9) |.|
 02         number          =       4444
NEW CONTENT:
01          person_2
 02         name( 0: 9)
            ( 0) |M|  ( 1) |a|  ( 2) |i|  ( 3) |e|  ( 4) |r|  ( 5) |.|
            ( 6) |.|  ( 7) |.|  ( 8) |.|  ( 9) |.|
 02         number          =      12345
%  AID0274 Change desired? Reply (Y=Yes; N=No)?y
```

6. `%set 123.45 into count`

```
I39O WARNING: SOURCE TRUNCATED
OLD CONTENT:
    9876
NEW CONTENT:
     123
%  AIDO274 Change desired? Reply (Y=Yes; N=No)?y
```

7. Interpretation eines `char`-Zeichens als `unsigned` und als `signed integer`

```
%s x'ff' into c_arr[O]
%s c_arr[O]%a into i
%s c_arr[O]%f into i
```

– erstes %SET-Kommando

```
OLD CONTENT:
OO   .
NEW CONTENT:
FF   ~
%  AIDO274 Change desired? Reply (Y=Yes; N=No)?y
```

– zweites %SET-Kommando

```
OLD CONTENT:
          0
NEW CONTENT:
        255
%  AIDO274 Change desired? Reply (Y=Yes; N=No)?y
```

– drittes %SET-Kommando

```
OLD CONTENT:
        255
NEW CONTENT:
         -1
%  AIDO274 Change desired? Reply (Y=Yes; N=No)?y
```

8. `%set Y::n'f()' into %2g`

Die Prologadresse der Member-Funktion `Y::f()` wird in das AID-Register %2G geschrieben.

9.
```
%da 5 from s'12'->
%set s'12'->%l2 into %3g
```

   Mit dem %DISASSEMBLE lassen Sie sich zwei Maschinenbefehle ab der Source-Referenz S’12’ rückübersetzen. Der erste Befehl ist ein 2-Byte-Befehl.
   Dieser erste Befehl wird mit %SET in das AID-Register %3G übertragen.

10.
```
%aid c=yes
%set "HALLO" into c_arr
```

   Mit dem Kommando `%AID C=YES` schalten Sie die Behandlung von `char`-Vektoren als C-Strings ein. Anschließend können Sie den `char`-Vektor `c_arr` mit dem C-String-Literal `"HALLO"` überschreiben. Die restlichen Bytes von `c_arr` werden mit binären Nullen aufgefüllt.

%SHOW

Mit %SHOW können Sie sich über die aktuellen Vereinbarungen zu einzelnen AID-Kommandos informieren, und Sie können sich zeigen lassen, wie die letzte Eingabe eines Kommandos aussah und welches Kommando zuletzt eingegeben wurde. Außerdem können Sie über den Subkommandonamen das Kommando anfordern, in dem es definiert wurde oder sich eine Liste aller eingetragenen Subkommandonamen mit dem zugehörigen Kommandotyp ausgeben lassen. Entsprechend der Vereinbarung zur Groß-/Kleinschreibung im %AID-Kommando wird die Orginaleingabe des Kommandos wiedergegeben, oder der Eingabestring ist in Großbuchstaben umgesetzt.

– Mit *show-ziel* geben Sie ein Kommando, einen Subkommandonamen oder ein AID-Schlüsselwort für alle aktuellen Subkommandos an.

Kommando	Operand
%SH[OW]	[show-ziel]

%SHOW ohne Operand gibt Ihnen das unmittelbar vorher eingegebene AID-Kommando aus. Wurde für die Task noch kein AID-Kommando eingegeben, erhalten Sie eine Fehlermeldung. Ein %SHOW für eins der nicht vorgesehenen Kommandos führt zum Syntaxfehler. Das Kommando ist in Kommando- und Subkommandofolgen zugelassen.

%SHOW verändert den Programmzustand nicht.

show-ziel

bezeichnet ein AID-Kommando, ein bestimmtes Subkommando oder alle eingetragenen Subkommandos. In *show-ziel* können Sie die für dieses Kommando zugelassenen Kommandos auch in ihrer Kurzform angeben.

Kommando oder Subkommando	Information
%AID	Die geltenden aktuellen Einstellungen der Kommandos %AID, %AINT, %BASE und die Version des geladenen AID.
%ALIAS	Liste aller vereinbarten Aliasnamen mit den zugehörigen Originalnamen.
%BASE	Die aktuellen Einstellungen für %BASE, %AINT und %SYMLIB, die TSN, TID sowie die Version des Betriebssystems und der Typ des Rechners.

Fortsetzung ...

Kommando oder Subkommando	Information
%C[ONTROL]	Der Eingabestring zu jedem angemeldeten %CONTROLn.
%D[IS]A[SSEMBLE]	Die aktuelle anzahl und start-Adresse (V' ...').
%F[IND]	Das eingegebene Kommando und gegebenenfalls die virtuelle Adresse des letzten Treffers.
%IN[SERT] [testpunkt]	Ohne die Angabe testpunkt werden alle aktiven Testpunkte ausgegeben. Sonst zeigt AID das eingegebene Kommando, in dem testpunkt vereinbart wurde. Es wird darauf hingewiesen, dass AID i.a. die Prologadresse nicht der entsprechenden Funktion zuordnen kann. Wenn Sie daher bei Funktionen aus C++-Programmen der Einfachheit halber im %INSERT die Funktionsadressen verwenden, die AID zu %DISPLAY {namespace|objekt| klasse} bzw. in der %SDUMP- Ausgabe auflistet, um sich die Eingabe der in der Regel sehr langen Namen zu ersparen, so gibt AID mit %SHOW %INSERT u.U. den Namen einer davor liegenden Funktion aus.
%ON	Der Eingabestring zu jedem aktiven %ON-Kommando
%OUT	Die geltenden medium-u-menge-Werte für die über %OUT steuerbaren Kommandos.
%OUTFILE	Alle implizit oder explizit angemeldeten Ausgabedateien ihren Linknamen.
%QUALIFY	Das zuletzt eingegebene %QUALIFY-Kommando.
%SYMLIB	Die angemeldeten Bibliotheken mit der zu gehörigen Basisqualifikation und der TSN
%TRACE	Die %TRACE-Parameter bei Anmeldung des %TRACE werden ausgegeben. Für Operanden, die nicht explizit angegeben waren, ergänzt AID die Standardwerte. Die Übersetzungseinheit wird angegeben, in der der %TRACE abläuft. Es wird berücksichtigt, ob der letzte %TRACE symbolisch oder maschinennah war. In Folgezeilen gibt AID aus, wieviel Befehle oder Anweisungen schon mit dem aktuellen %TRACE bearbeitet wurden und wie der Eingabestring des letzten %TRACE-Kommandos aussah.
%.*	Die Namen aller aktiven Subkommandos mit dem Typ des AID-Kommandos, in dem sie definiert wurden.
%.subkdoname	Das Kommando, in dem subkdoname definiert wurde.

Tabelle 7: Operandenwerte des Kommandos %SHOW und die dazu angezeigten Informationen

Beispiel

```
$debug examp
%  AID0348 Program stopped due to EXEC event (PID=0000000891)
%0000000891/%aid c=yes
%0000000891/%show %aid
A I D   V03.4B11 OF  2016-03-17
Copyright (C) Fujitsu Technology Solutions 2016
All Rights Reserved

E=VM : %AINT = %MODE31

%AID CHECK      =   NO
%AID REP        =   NO
%AID SYMCHARS   =   NOSTD
%AID OV         =   NO
%AID LOW        =   ALL
%AID DELIM      =   '|'
%AID LANG       =   D
%AID FORK       =   NOT_USED
%AID EXEC       =   ON
%AID C          =   YES
%AID EBCDIC     =   EDF03IRV
%AID CCS        =   EDF03IRV
```

Nach dem Laden des Programms unter POSIX mit `debug` wurde zunächst mit dem Kommando %AID die Option `C=YES` eingeschaltet, die bewirkt, dass AID C-String-Literale in "" akzeptiert und `char`-Vektoren als C-Strings interpretiert. die Umsetzung von Klein- in Großbuchstaben auch für Angaben in der S-Qualifikation ausgeschaltet und Anschließend wurde mit `%SHOW %AID` die Auflistung der aktuell gültigen Vereinbarungen mit %AID angefordert. Die Ausgabe des %SHOW-Kommandos zeigt die Voreinstellungen der Operandenwerte von %AID bis auf folgende Ausnahmen:

– SYMCHARS wurde implizit durch das Setzen von `C=YES` auf NOSTD geschaltet, d.h. der Bindestrich (-) wird stets als Minuszeichen interpretiert, da Bindestriche in Namen in C/C++-Programmen ohnehin nicht erlaubt sind.

– LOW wurde ebenfalls implizit durch `%AID C=YES` auf ALL gesetzt, d.h. die Umsetzung von Klein- in Großbuchstaben ist auch für Angaben in der S-Qualifikation ausgeschaltet.

– `EXEC` ist in der POSIX-Shell nach dem Laden eines Programms mit `debug` stets eingeschaltet.

– `C=YES` wurde explizit gesetzt.

Bei `%AID FORK` entspricht die Angabe `NOT_USED` der Einstellung `OFF`; `NOT_USED` weist lediglich darauf hin, dass der Schalter `FORK` in dieser Task noch nicht gesetzt war.

%STOP

Mit %STOP veranlassen Sie AID, das Programm anzuhalten, in den Kommandomodus zu gehen und eine STOP-Meldung auszugeben. Dieser Meldung können Sie entnehmen, an welcher Anweisung, in welcher Übersetzungseinheit und in welcher Funktion bzw. in welchem Block das Programm unterbrochen wurde.

Wird das Kommando am Terminal oder aus einer Prozedurdatei eingegeben, so wird der Programmzustand nicht verändert, da das Programm ja bereits steht. In diesen Fällen können Sie das Kommando anwenden, um mit der STOP-Meldung Lokalisierungsinformation über die Programmunterbrechungsstelle zu erhalten.

Unter POSIX können Sie mit %STOP eine durch `fork()` entstandene Task unterbrechen, um den weiteren Verlauf dieser Task über AID-Kommandos zu kontrollieren. Mit den Operanden T=*tsn* (Task Sequence Number) und PID=*pid* (Process Identification) kann eine durch `fork()` entstandene Task unterbrochen werden. AID meldet sich mit der Prozessnummer (*pid)* der unterbrochenen Task, und Sie können den weiteren Verlauf dieser Task über AID-Kommandos kontrollieren.

– Mit *T=tsn* bezeichnen Sie die Task Sequence Number (TSN) der Task, die AID unterbrechen soll.

– Mit *PID=pid* bezeichnen Sie die Process Identification (pid) der Task, die AID unterbrechen soll.

Kommando	Operand
%STOP	[{ T=tsn | PID=pid }]

Steht %STOP in einer Kommandofolge oder in einem Subkommando, werden nachfolgende Kommandos nicht mehr ausgeführt.

Wenn Sie als Basisqualifikation mit %BASE eine Dump-Datei eingestellt haben und dann ein %STOP-Kommando eingeben, gibt AID eine STOP-Meldung aus. Diese Meldung enthält die Lokalisierungsinformation zu der Adresse, an der das Programm unterbrochen war als der Dump geschrieben wurde.

Wenn ein Programm durch Drücken der K2-Taste oder eine Fork-Task durch ein %STOP-Kommando unterbrochen wurde, liegt die Unterbrechungsstelle u.U. nicht im Benutzerprogramm, sondern in den Routinen des Laufzeitsystems. Um Funktionen und Variablen des Programms ansprechen zu können, ohne jedesmal die vollständige Qualifikation anzugeben, empfiehlt es sich, das Programm zunächst mit `%TRACE 1 IN S=`*srcname* bis zur nächsten ausführbaren Anweisung weiterlaufen zu lassen.

%STOP verändert den Programmzustand.

Ein %STOP in einem Subkommando bezieht sich stets auf das geladene Programm.

T

tsn Die Fork-Task, die in den Testmodus versetzt werden soll, wird über ihre TSN (Task Sequence Number) angesprochen.

PID

pid Die Fork-Task, die in den Testmodus versetzt werden soll, wird über ihre Prozessnummer (Process Identification) angesprochen.

Beispiele

1.

```
/%in s'10' <%display abc_arr; %stop>
/%resume

abc_arr( 0: 26)
(  0) |A|  (  1) |B|  (  2) |C|  (  3) |D|  (  4) |E|  (  5) |F|  (  6) |G|
(  7) |H|  (  8) |I|  (  9) |J|  ( 10) |K|  ( 11) |L|  ( 12) |M|  ( 13) |N|
( 14) |O|  ( 15) |P|  ( 16) |Q|  ( 17) |R|  ( 18) |S|  ( 19) |T|  ( 20) |U|
( 21) |V|  ( 22) |W|  ( 23) |X|  ( 24) |Y|  ( 25) |Z|  ( 26) |.|
STOPPED AT SRC_REF: 10 ,  SOURCE: EXAMP.C ,  PROC: main
```

Mit %INSERT wird ein Testpunkt auf die erste Anweisung in Zeile 10 gesetzt. Das Subkommando enthält die Kommandos %DISPLAY und %STOP. Nach der Ausgabe von `abc_arr` hält AID das Programm an und schreibt eine STOP-Meldung mit Source-Referenz, Übersetzungseinheit und Funktion der aktuellen Unterbrechungsstelle.

2.

```
$debug exstop
%  AID0348 Program stopped due to EXEC event (PID=0000000876)
%0000000876/%aid fork=next
%0000000876/%aid low=all
%0000000876/...
%0000000876/%resume
%  AID0348 Program stopped due to FORK event (PID=0000000877)
%0000000877/...
%0000000877/%stop pid=876
%  AID0492 %STOP was sent to fork task (PID=0000000876)
%0000000877/<EM><DÜ>
%  AID0348 Program stopped due to STOP event (PID=0000000876)
%0000000876/%trace 1 in s=n'exstop.c'
%0000000877/<EM><DÜ>
45                      BLOCK END, LOOP END
STOPPED AT SRC_REF: 45, SOURCE: exstop.c , BLK: 39 , END OF TRACE
%0000000876/%display count
%0000000877/<EM><DÜ>
*** TID: 003400D1 *** TSN: OEUV ****************************************
SRC_REF: 45 SOURCE: exstop.c  BLK : 39  ********************************
count           =         933
```

Nach dem Laden des Programms mit dem POSIX-Kommando `debug` wird mit `%AID FORK=NEXT` festgelegt, dass auch die von `exstop` erzeugte Fork-Task im Testmodus ablaufen soll. Außerdem wird auch `%AID LOW=ALL` gesetzt, weil sonst der Name der Quelldatei `exstop.c` in der S-Qualifikation in Großbuchstaben umgesetzt würde. Die Vater-Task läuft unter der `pid` 876, die Sohn-Task erhält die `pid` 877. Mit dem Kommando `%STOP PID=876` wird die Vater-Task unterbrochen. AID meldet sich mit dem Prompt `%0000000876/`. Durch das nachfolgende %TRACE-Kommando erreichen Sie, dass die Vater-Task vor der nächsten ausführbaren Anweisung anhält. Jetzt können Sie Variablen der Vater-Task ohne Qualifikation ansprechen. Da jetzt beide Tasks um das Terminal konkurrieren, müssen Sie den Prompt der nicht erwünschten Task mit EM DÜ beantworten, damit die Task, die Sie testen wollen, Gelegenheit hat, sich am Terminal zu melden.

%SYMLIB

Mit %SYMLIB veranlassen Sie AID, PLAM-Bibliotheken zu öffnen oder zu schließen. Auf geöffnete PLAM-Bibliotheken greift AID zu, wenn Sie in einem Kommando symbolische Speicherreferenzen ansprechen, die in einer Übersetzungseinheit liegen, zu der AID keine LSD-Sätze geladen hat.

– Mit *qualifikation-u-lib* melden Sie eine oder mehrere Bibliotheken an oder ab, in denen Bindemodule mit den zugehörigen LSD-Sätzen abgespeichert sind. Sie können jede Bibliothek dem aktuellen Programm oder einer Dump-Datei zum Nachladen der LSD-Sätze zuordnen, indem Sie die entsprechende Basisqualifikation dazu angeben.

Kommando	Operand
%SYMLIB	[qualifikation-u-lib][,...]

Bei Ausführung dieses Kommandos stellt AID nur fest, ob die angegebene Bibliothek geöffnet werden kann; es überprüft nicht, ob der Inhalt einer Bibliothek zu dem Programm passt, das gerade bearbeitet wird. Vorbereitend können Sie also die Bibliotheken anmelden, die Sie während eines Testlaufs benötigen. Erst beim Zugriff auf die nachgeladenen LSD-Sätze überprüft AID, ob der Bindemodul (OM) bzw. Bindelademodul (LLM) des angesprochenen Programms mit dem der PLAM-Bibliothek übereinstimmt.
Sind zu einer Basisqualifikation mehrere Bibliotheken angemeldet, so durchsucht AID sie in der Reihenfolge, in der sie im %SYMLIB-Kommando angegeben wurden.
Verläuft die Suche von AID nicht erfolgreich oder ist keine Bibliothek angemeldet, so können Sie nach der entsprechenden Meldung mit einem neuen %SYMLIB-Kommando die richtige Bibliothek zuweisen und dann das Kommando wiederholen, zu dessen Ausführung die LSD-Sätze fehlten.

Eine Bibliothek bleibt solange angemeldet, bis sie abgemeldet wird durch:
– ein neues %SYMLIB-Kommando zur selben Basisqualifikation,
– ein %SYMLIB-Kommando ohne Operanden,
– ein %DUMPFILE-Kommando, mit dem die *Dn* zugewiesene Dump-Datei geschlossen wird,
– /LOGOFF bzw. /EXIT-JOB.

Außerdem können Sie in einer durch einen `fork()`-Aufruf entstandenen Task nicht auf eine Bibliothek zugreifen, die in der Vater-Task angemeldet wurde.

Enthält ein neues Kommando neue Dateinamen, dann werden diese Bibliotheken angemeldet und geöffnet.

%SYMLIB verändert den Programmzustand nicht.

qualifikation-u-lib

ist eine Basisqualifikation und/oder der Dateiname einer PLAM-Bibliothek.

– Geben Sie eine Basisqualifikation und einen Dateinamen an, meldet AID die angegebene Bibliothek zu dieser Basisqualifikation an und öffnet sie. Bisher angemeldete Bibliotheken zu derselben Basisqualifikation werden abgemeldet.
– Geben Sie nur einen Dateinamen an, meldet AID die Bibliothek zur gerade eingestellten Basisqualifikation an (siehe Beschreibung des Kommandos %BASE) und öffnet sie. Alle zur aktuellen Basisqualifikation angemeldeten Bibliotheken werden abgemeldet.
– Geben Sie nur eine Basisqualifikation an, werden alle dazu angemeldeten Bibliotheken abgemeldet.

AID kann maximal 15 Bibliotheks-Anmeldungen verwalten. Dabei zählt eine Bibliothek, die gleichzeitig mit verschiedenen Basisqualifikationen angemeldet ist, so oft, wie sie angegeben wird.

```
qualifikation-u-lib-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - -

                  ⎧VM ⎫
[•][E=⎨   ⎬•][dateiname]
                  ⎩Dn ⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

- Steht der Punkt an führender Stelle, ist er das Kennzeichen für eine *vorqualifikation*. Sie muss mit einem vorhergehenden %QUALIFY definiert worden sein und kann nur für eine Basisqualifikation stehen.

E=VM
: %SYMLIB gilt für das geladene Programm (siehe Beschreibung des Kommandos %BASE).

E=Dn
: %SYMLIB gilt für einen Speicherabzug in einer Dump-Datei mit dem Linknamen *Dn* (siehe Beschreibung der Kommandos %BASE und %DUMPFILE).

dateiname
: ist der BS2000-Katalogname einer PLAM-Bibliothek. Sie wird zu der explizit oder mit *vorqualifikation* angegebenen Basisqualifikation angemeldet. Ohne die Angabe einer Qualifikation wird sie zur gerade eingestellten Basisqualifikation angemeldet.

Beispiel

```
%symlib           e=d5.mylib,out.cpp
```

Wenn AID für die Bearbeitung eines Speicherabzugs in der Dump-Datei mit dem Linknamen `D5` LSD-Sätze benötigt, versucht es, diese aus der Bibliothek `MYLIB` zu laden. Die Bibliothek `OUT.CPP` wird zur aktuell eingestellten Basisqualifikation angemeldet. Wurde bisher kein %BASE eingegeben, verwendet AID diese Bibliothek zum Nachladen von LSD-Sätzen zum geladenen Programm.

%TITLE

Mit %TITLE definieren Sie einen eigenen Seitenkopf-Text. Diesen verwendet AID, wenn die Kommandos %DISASSEMBLE, %DISPLAY, %HELP, %SDUMP und %TRACE in die Systemdatei SYSLST schreiben.

– Mit *seitenkopf* geben Sie den Text der Kopfzeile an, veranlassen AID, den Seitenzähler auf 1 zu setzen und vor der nächsten Druckzeile SYSLST auf Seitenanfang zu positionieren.

Kommando	Operand
%TITLE	[seitenkopf]

Mit einem %TITLE ohne *seitenkopf*-Operanden wechseln Sie wieder zur AID-Standard-Überschrift. AID setzt den Seitenzähler wieder auf 1 und positioniert SYSLST vor der nächsten Druckzeile auf Seitenanfang.

Eine mit %TITLE vereinbarte Seitenüberschrift gilt bis zu einem neuen %TITLE oder bis Programmende.

%TITLE verändert den Programmzustand nicht.

seitenkopf

gibt den variablen Teil der Seitenüberschrift an. Er wird von AID mit der Uhrzeit, dem Datum und dem Seitenzähler ergänzt.

seitenkopf
: ist ein Character-Literal in der Form {C'x...x' | 'x...x'C | 'x...x'} und kann maximal 80 Zeichen lang sein. Ein längeres Literal wird mit einer Fehlermeldung abgewiesen, in der aber nur die ersten 52 Stellen des Literals protokolliert werden.
Auf eine Druckseite werden außer der Seitenüberschrift bis zu 58 Zeilen gedruckt.

%TRACE

Mit %TRACE schalten Sie die AID-Ablaufverfolgung ein und starten das Programm oder setzen es an der Unterbrechungsstelle fort.

- Mit *anzahl* legen Sie fest, wie viele Anweisungen maximal verfolgt, d.h. vor ihrer Ausführung protokolliert werden sollen.
- Mit *fortsetzung* steuern Sie, ob das Programm nach Beendigung des %TRACE anhält (Standardwert) oder ohne Protokollierung weiterläuft.
- Mit *kriterium* wählen Sie verschiedene Typen von Programmanweisungen aus, die AID protokollieren soll. Die Protokollierung erfolgt vor der Ausführung der ausgewählten Anweisungen.
- Mit *trace-bereich* legen Sie den Programmbereich fest, in dem *kriterium* berücksichtigt werden soll.

Kommando	Operand
%T[RACE]	[anzahl] [fortsetzung] [kriterium][,...] [IN trace-bereich]

Ein %TRACE kann nicht gleichzeitig mit einem *write-ereignis* des %ON angemeldet sein.

Wird der Programmablauf während eines %TRACE unterbrochen, so kann der %TRACE mit %CONTINUE fortgesetzt werden. Dies trifft auf folgende Fälle zu:

- Ein Subkommando, das ein %STOP-Kommando enthält, wurde ausgeführt.
- Die K2-Taste wurde gedrückt (siehe Seite 19).

Beendet wird der %TRACE dagegen durch folgende Ereignisse:

- Die maximale Anzahl der zu überwachenden Anweisungen wurde erreicht.
- Ein Subkommando wurde ausgeführt, das ein %RESUME- oder %TRACE-Kommando enthält.
- Nach einer der oben beschriebenen Programmunterbrechungen wird mit %RESUME fortgefahren.
- Ein `fork()`- oder `exec()`-Aufruf wurde ausgeführt.
- Das Programmende wurde erreicht.

Die Operandenwerte eines %TRACE gelten so lange, bis sie durch Angaben aus einem späteren %TRACE überschrieben werden, bis ein `fork()`- oder `exec()`-Aufruf ausgeführt wird oder bis Programmende. In einem neuen %TRACE setzt AID also für einen nicht angegebenen Operanden den Wert aus dem vorhergehenden %TRACE ein. Beim *trace-bereich*-Operanden ist dies nur der Fall, wenn die aktuelle Unterbrechungsstelle in dem zu übernehmenden *trace-bereich* liegt. Gibt es keine zu übernehmenden Werte, setzt AID die Standardwerte ein:

– für *anzahl* ist dies 10,
– für *fortsetzung* wird `S` eingesetzt,
– für *kriterium* wird `%STMT` ergänzt,
– für *trace-bereich* wird die Übersetzungseinheit eingesetzt, in der die aktuelle Unterbrechungsstelle liegt.

Mit %OUT können Sie steuern, welche Informationen eine Protokollzeile enthält und auf welches Ausgabemedium das Protokoll ausgegeben werden soll.

Steht %TRACE in einer Kommandofolge oder in einem Subkommando, werden nachfolgende Kommandos nicht mehr ausgeführt.

Wenn Sie unmittelbar nach dem Laden im Prolog bzw. nach Beendigung von `main` im Epilog eines C++-Programms die Ablaufverfolgung einschalten, können Sie die Source-Referenzen und die zugehörigen Anweisungstypen des %TRACE-Protokolls nur bedingt den Quellprogrammanweisungen im Source-Error-Listing zuordnen.

trace-bereich kann nur im geladenen Programm liegen, deshalb muss die Basisqualifikation E=VM eingestellt sein (siehe Beschreibung des Kommandos %BASE) oder explizit angegeben werden.

%TRACE verändert den Programmzustand.

anzahl

gibt an, wie viele Programmanweisungen vom Typ *kriterium* maximal protokolliert und ausgeführt werden sollen.

anzahl ist eine Ganzzahl mit $1 \leq anzahl \leq 2^{31}-1$. Standardwert ist 10. Er wird von AID in ein %TRACE-Kommando ohne *anzahl*-Operanden eingesetzt, wenn es keinen Wert aus einem vorhergehenden %TRACE gibt.

Nachdem die vorgegebene *anzahl* von Anweisungen überwacht wurde, wird das Programm, abhängig vom Operandenwert von *fortsetzung*, angehalten oder ohne Protokollierung fortgesetzt. Ist für *fortsetzung* S eingestellt, so gibt AID über SYSOUT eine Meldung aus, der Sie entnehmen können, an welcher Anweisung, in welcher Übersetzungseinheit und in welcher Funktion bzw. in welchem Block das Programm angehalten wurde.

fortsetzung

gibt an, ob AID das Programm nach Beendigung des %TRACE anhalten oder fortsetzen soll. Der Operand gilt solange, bis in einem neuen %TRACE ein anderer Operandenwert angegeben wird oder bis Programmende.

```
fortsetzung=OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - -

{S | R}

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

S Das Programm wird angehalten. AID gibt eine STOP-Meldung aus, die Lokalisierungsinformationen über die Unterbrechungsstelle enthält. S ist Standardwert.

R Das Programm wird ohne Ausgabe einer Meldung fortgesetzt.

kriterium

ist ein Schlüsselwort, das den Typ der Anweisungen festlegt, die beim Ablauf überwacht werden sollen. Sie können mehrere Schlüsselwörter gleichzeitig angeben, die dann gemeinsam wirken. Zwischen zwei Schlüsselwörtern muss ein Komma stehen.
Wird kein *kriterium* vereinbart, arbeitet AID mit dem Standardwert %STMT, wenn nicht noch aus einem vorhergehenden %TRACE eine *kriterium*-Vereinbarung gültig ist.

kriterium	**Ausgabe der Protokollierung vor**
%STMT	jeder Anweisung, die durchlaufen wird
%ASSGN	jeder Zuweisungs-Anweisung
%CALL	jedem Funktionsaufruf
%COND	jeder if- und switch-Anweisung, jedem else-Zweig der if-Anweisung und jedem Bedingungsteil der do-, while- oder for-Anweisung
%EH %EXCEPTION	jeder catch- und throw-Anweisung
%GOTO	jeder goto-, break- und continue-Anweisung
%LAB	jeder Anweisung mit einer Marke, gilt jedoch nicht für case- und default-Marken
%PROC	der ersten und der letzten Anweisung einer Funktion

Tabelle 8: Werte des Operanden *kriterium* und ihre Bedeutung

trace-bereich

legt den Programmbereich fest, in dem die Ablaufverfolgung stattfinden soll. Nur innerhalb dieses Bereiches werden die mit *kriterium* ausgewählten Anweisungen überwacht und protokolliert. Außerhalb dieses Bereiches ist der %TRACE inaktiv und wird erst bei Rückkehr in den Bereich wieder aktiv. *trace-bereich* kann nur im geladenen Progamm liegen, und eine angegebene Übersetzungseinheit muss zum Zeitpunkt der Eingabe des %TRACE bzw. bei Abarbeitung des Subkommandos, in dem der %TRACE enthalten ist, geladen sein.

Eine *trace-bereich*-Definition ist wirksam bis zu einem neuen %TRACE mit eigenem *trace-bereich*-Operanden, einem %TRACE, der außerhalb dieses Bereiches eingegeben wird, bis ein `fork()`- oder `exec()`-Aufruf ausgeführt wird oder bis Programmende. Wird *trace-bereich* nicht angegeben, wird die Bereichsdefinition aus einem vorhergehenden %TRACE übernommen, wenn die aktuelle Unterbrechungsstelle in diesem Bereich liegt. Sonst setzt AID den Standardwert ein, das ist die Übersetzungseinheit, in der die aktuelle Unterbrechungsstelle liegt.

Die Fortsetzungsadresse für den Programmablauf kann mit %TRACE nicht beeinflusst werden. Eine andere Fortsetzungsadresse können Sie nur festlegen, indem Sie mit %SET den Befehlszähler (%PC) ändern (siehe Beschreibung des Kommandos %SET, *schlüsselwort auf* Seite 275).

```
trace-bereich-OPERAND - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

                  ⎧S=srcname                                                ⎫
                  ⎪                                                         ⎪
IN  [•][E=VM•] ⎨              ⎧[qua•][PROC=]funktion                     ⎫⎬
                  ⎪[S=srcname•] ⎨BLK='[f-]n[:b]'                          ⎬⎪
                  ⎪              ⎩([PROC=funktion•]src-ref:src-ref)       ⎭⎪
                  ⎩                                                         ⎭

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

- Steht der Punkt an führender Stelle, so ist er das Kennzeichen für eine *vorqualifikation*. Sie muss mit einem vorhergehenden %QUALIFY-Kommando definiert worden sein.
  Aufeinanderfolgende Qualifikationen werden durch einen Punkt getrennt. Außerdem muss zwischen der letzten Qualifikation und dem anschließenden Operandenteil ein Punkt stehen.

E=VM

Da *trace-bereich* nur im virtuellen Speicher des geladenen Programms liegen kann, geben Sie E=VM nur an, wenn als aktuelle Basisqualifikation eine Dump-Datei vereinbart ist (siehe Beschreibung des Kommandos %BASE).

S=srcname

geben Sie an, wenn *trace-bereich* nicht in der aktuellen Übersetzungseinheit liegen soll oder wenn eine vereinbarte Bereichseinschränkung nicht mehr gelten soll. Endet *trace-bereich* mit einer S-Qualifikation, so umfasst er die gesamte angegebene Übersetzungseinheit.

[qua•][PROC=]funktion

trace-bereich wird durch eine PROC-Qualifikation festgelegt und umfasst die gesamte angegebene Funktion.

Bei Funktionen aus C-Programmen ist *funktion* der im Quellprogramm vergebene Name der Funktion ohne Klammern und ohne Signatur.
Funktionen aus C++-Programmen müssen Sie, je nachdem, um welchen Typ es sich handelt, in der Schreibweise `n'...'` oder `t'...'` angeben. Ist die Funktion in einem Namespace oder in einer Klasse definiert, so wird dem Funktionsnamen die Namespace- oder Klassen-Qualifikation vorangestellt.
Die Signatur `void` darf nicht mehr geschrieben werden. Wie auch in C++ möglich geben Sie in diesem Fall nur die beiden Klammern nach dem Funktionsnamen an.
Für *funktion* ergibt sich die folgende Syntax:

```
                                    ⎧n'funktion([signatur])'              ⎫
[namespace::[...]][klasse::[...]]⎨                                        ⎬
                                    ⎩t'f_template<arg[,...]>([signatur])'⎭
```

Abweichend davon werden die Funktionen `main` und `__STI__` sowie alle Funktionen mit C-Linkage auch beim Testen in C++-Programmen nur mit dem Funktionsnamen bezeichnet (siehe Seite 60).

Virtuelle Funktionen sprechen Sie mit der folgenden Syntax an:
`p->n'funktion([signatur])'`

p ist eine Zeigervariable, die auf das Objekt einer Klasse verweist, das die gewünschte Member-Funktion enthält. Wenn *p* von der aktuellen Unterbrechungsstelle aus nicht zu erreichen ist, muss dem Gültigkeitsbereich entsprechend qualifiziert werden. Liegt die Unterbrechungsstelle in der virtuellen Funktion selbst, dann können Sie die Prologadresse der aktuellen Funktion ansprechen, indem Sie statt *p* den this-Zeiger einsetze*n* (siehe Beschreibung von *this* auf Seite 66 und Abschnitt „Virtuelle Funktionen“ auf Seite 76).

Um als *trace-bereich* eine Funktion über Pointer-to-Member anzugeben, stehen Ihnen die folgenden beiden Möglichkeiten zur Verfügung:

Das Objekt der Klasse, das die gewünschte Funktion enthält, wird mit seinem Namen bezeichnet. Als Dereferenzierungsoperator schreiben Sie •*

```
[qua•]objekt•*[objekt•][klasse::][...]pointer-to-function-member
```

Das Objekt der Klasse wird über einen Zeiger angesprochen. Als Dereferenzierungsoperator schreiben Sie **->***

```
[qua•]zeiger->*[objekt•][klasse::][...]pointer-to-function-member
```

Das Objekt der Klasse bezeichnen Sie mit dem Operanden, der auf der linken Seite der Dereferenzierungsoperatoren •* bzw. **->*** steht:
mit *objekt* bezeichnen Sie das Objekt der Klasse mit seinem Namen,
mit *zeiger* adressieren Sie das Objekt über einen Zeiger.
Rechts der Deferenzierungsoperatoren schreiben Sie den Namen des Pointer-to-Function-Member. Evtl. müssen Sie das Objekt, das die Definition des Pointer-to-Function-Member enthält, und innerhalb des Objekts die zur eindeutigen Ansprache nötige Klassenqualifikation davorschreiben, falls der Pointer-to-Member von der Unterbrechungsstelle aus anders nicht zu erreichen ist. Näheres zum Arbeiten mit Pointer-to-Function-Member finden Sie auf Seite 83.

qua

Ist die Funktion in einer lokalen Klasse definiert, so müssen Sie vor dem Namen der gewünschten Funktion eine PROC-Qualifikation für die übergeordnete Funktion angeben, die die Definition der lokalen Klasse enthält. Für Funktionen, die in einem inneren Block der übergeordneten Funktion definiert sind, schließen Sie an die PROC-Qualifikation für die übergeordnete Funktion eine oder ggfs. mehrere BLK-Qualifikationen, jeweils durch einen Punkt getrennt, an, um den Pfad zu der lokalen Klasse zu beschreiben (siehe Seite 62).

Syntax für *qua*:

```
PROC=übergeordnete_fkt.[BLK='[f-]n[:b]'•[...]]
```

Das Ansprechen von Funktionen, die in lokalen Klassen innerer Blöcke definiert sind, wird nur bei Programmen unterstützt, die mit einem C/C++-Compiler ab V3.0B übersetzt wurden.

BLK='[f-]n[:b]'

trace-bereich wird durch eine BLK-Qualifikation festgelegt und umfasst den gesamten angegebenen Block. Der Name für einen Block wird aus Zeilennummer (*n*) und gegebenenfalls FILE-Nummer (*f*) und relativer Blocknummer (*b*) gebildet

Die BLK-Qualifikation kann nicht zusammen mit dem *kriterium* %PROC verwendet werden.

([PROC=funktion•]src-ref : src-ref)

Mit Source-Referenzen können Sie *trace-bereich* durch Angabe einer Anfangs- und Endadresse festlegen. Beide müssen innerhalb derselben Übersetzungseinheit liegen, und es gilt:
Anfangsadresse ≤ Endadresse.
Es ist zu beachten, dass aufsteigenden Source-Referenzen nur innerhalb eines Funktionsblocks aufsteigende Adressen zugeordnet sind. Falls die Bedingung Anfangsadresse ≤ Endadresse nicht erfüllt ist, weist AID das Kommando mit einer entsprechenden Fehlermeldung ab.

In C++-Programmen gilt außerdem, dass zu impliziten Konstruktor- und Destruktor-Aufrufen sowie bei Konversionsoperationen zusätzliche Source-Referenzen generiert werden, die zwar nicht im Source-Error-Listing erscheinen, die jedoch vom %TRACE protokolliert werden.

Soll *trace-bereich* nur eine Anweisung umfassen, müssen Anfangs- und Endadresse gleich sein.

PROC=funktion•

Die PROC-Qualifikation müssen Sie nur dann schreiben, wenn die angegebenen Source-Referenzen in der Übersetzungseinheit nicht eindeutig sind. Dies ist dann der Fall, wenn die Source-Referenzen in einer Funktion liegen, die durch Instanziierung aus einem Funktionstemplate hervorgegangen ist, oder wenn die Funktion, die die Source-Referenzen enthält, in einem Klassentemplate definiert ist, und wenn zu dem Template mindestens zwei Instanzen existieren (siehe oben sowie Seite 112).

src-ref

wird mit `S'[`*f*`-]`*n*`[:`*a*`]'` angegeben und bezeichnet die Adresse einer ausführbaren Anweisung. Dabei ist *n* die Zeilennummer, *f* die FILE-Nummer, falls >0 und *a* die relative Anweisungsnummer innerhalb der Zeile, falls >1.

Ausgabe des %TRACE-Protokolls

Das %TRACE-Protokoll wird standardmäßig in ausführlicher Form (%OUT-Operandenwert T=MAX) über SYSOUT ausgegeben. Mit %OUT können Sie die Ausgabemedien und den Informationsumfang für die Ausgabe festlegen (siehe AID-Basishandbuch, Kapitel „Operand Medium-und-Menge“ [1]).

Ein %TRACE-Protokoll mit Zusatzinformationen (T=MAX) enthält die Nummer und den Typ der Anweisung, die ausgeführt wurde.

In einem %TRACE-Protokoll mit verkürzter Information (T=MIN) wird der Typ der Anweisung nicht ausgegeben.

AID berücksichtigt die Modi XMAX und XFLAT für die Ausgabe des %TRACE-Protokolls nicht. Statt dessen generiert es die Standardausgabe (T=MAX).

Beispiele

1.
```
/%out %trace    t=max
/%t 3
15                          EXT.PROC START    , BLOCK START, IF
16                          IF
17:2                          THEN/ELSE, END
STOPPED AT SRC_REF: 17:2 ,  SOURCE: EXAMP.C ,  PROC: facul ,
END OF TRACE
```

Mit %OUT wurde die Ausgabe auf Terminal zurückgeschaltet und festgelegt, dass der maximale Informationsumfang ausgegeben werden soll.
Das %TRACE-Kommando soll drei C-Anweisungen verfolgen. Nach der dritten Anweisung kommt die Abschlussmeldung für diesen %TRACE: Das Programm steht vor der zweiten Anweisung in Zeile 17, die zur Funktion `facul` in der Übersetzungseinheit `EXAMP.C` gehört.

2.
```
/%out %t t=min
/%t 3
15
16
17:2
STOPPED AT SRC_REF: 17:2 ,  SOURCE: EXAMP.C ,  PROC: facul ,
END OF TRACE
```

Mit dem %OUT-Kommando wird der Informationsumfang für das Kommando %TRACE reduziert. Der danach eingebene %TRACE gibt das Protokoll mit verkürztem Informationsumfang aus.

3. `%trace 5 r %instr`

   5 Anweisungen des Programms werden ausgeführt und protokolliert. Danach wird das Programm ohne Protokollierung fortgesetzt.

4. `%c1 %call in s=testprog <%trace 1 r>`

   Alle Unterprogrammaufrufe der Programmeinheit `TESTPROG` werden protokolliert. Das Programm wird jeweils nach Ausführung und Protokollierung der CALL-Anweisung fortgesetzt.

7 POSIX-Kommando debug

Das Kommando `debug` ermöglicht das Testen von POSIX-Programmen, die in der POSIX-Shell gestartet werden. Mit `debug` können Sie in der POSIX-Shell ein Programm mit LSD laden oder einen laufenden Prozess unterbrechen und in den Testmodus versetzen.

In POSIX-Sitzungen, die über `rlogin` eröffnet werden, ist `debug` aus Gründen der Systemsicherheit nicht zugelassen.

Syntax

```
      ⎧ [␣-e]␣progname[␣argument]...  ⎫
debug ⎨                               ⎬
      ⎩ ␣-p␣pid                       ⎭
```

debug [␣**-e**] ␣progname [␣argument]...

Programm *progname* wird in einer von der Shell durch `fork()` erzeugten Task geladen und in den Testmodus versetzt; AID meldet sich mit einem Prompt, der aus der Prozessnummer (pid) der Task gebildet wird, und Sie können AID-Kommandos zum Testen eingeben. Über die Option `-e` können Sie steuern, ob die LSD für das symbolische Testen mitgeladen werden soll (ohne `-e`) oder nicht (mit `-e`). Das Kommando `debug` *progname* in der POSIX-Shell entspricht somit dem BS2000-Kommando `LOAD-PROGRAM` *progname* mit dem Operanden `TEST-OPTIONS=YES` in BS2000-Umgebung.

-e *progname* wird ohne LSD geladen.

progname
: Name des Programms, das getestet werden soll.

argument
: Argument von *progname*.

debug ␣**-p**␣pid

Der Prozess mit der angegebenen *pid* wird von AID übernommen und unterbrochen, falls der mit *pid* bezeichnete Prozess der eigenen Task-Familie angehört. Dabei ist die POSIX-Shell Vater-Task für alle in der Shell gestarteten Prozesse.
`debug -p` *pid* in der POSIX-Shell entspricht dem AID-Kommando `%STOP PID=`*pid* (siehe Seite 287), das Sie im BS2000-Kommandomodus oder im Testmodus einer Task eingeben können.

-p Das Programm wird über die zugehörige *pid* übernommen.

pid Prozessnummer der Task, die von AID übernommen und unterbrochen werden soll.

Beispiel

Das Beispiel zeigt die Übernahme eines bereits laufenden Programms durch AID:

```
$ ps -ef
      UID   PID  PPID  C    STIME TTY       TIME CMD
   D89239   890   824  0 10:22:38 term/003  0:01 [ps]
   D89239   888   824  0 10:22:27 term/003  0:00 [pexec]
   D89239   889   888  0 10:22:28 term/003  0:00 [pexec]
   D89239   830     1  0 09:35:13 term/004  0:04 [sh]
   D89239   824     1  0 09:31:22 term/003  0:06 [sh]
$ debug -p 888
%  AID0492 %STOP was sent to fork task (PID=0000000888).
%  AID0348 Program stopped due to STOP event (PID=0000000888)
```

Zunächst wird mit dem POSIX-Kommando `ps -ef` eine Liste aller laufenden Prozesse angefordert. Dieser Liste können Sie die PID des Prozesses entnehmen, der mit AID untersucht werden soll (888). Dieser Prozess ist Vater-Task für die Fork-Task mit der PID 889. Mit `debug -p 888` wird die Vater-Task unterbrochen und in den Testmodus versetzt.

```
%0000000888/%stop pid=889
%  AID0492 %STOP was sent to fork task (PID=0000000889).
```

Auch die Sohn-Task wird unterbrochen. Beide Tasks melden sich in der Folge abwechselnd mit ihren Prompts.

```
%0000000888/%aid low=all
%0000000888/%symlib test.lib
%  AID0348 Program stopped due to STOP event (PID=0000000889)
%0000000889/<EM><DÜ>
%0000000888/%trace 1 in s=n'pexec.c'
%0000000889/<EM><DÜ>
%0000000889/<EM><DÜ>
38                        BLOCK END, LOOP END
STOPPED AT SRC_REF: 38, SOURCE: pexec.c , PROC: main , END OF TRACE
```

Im nächsten Schritt soll die Vater-Task bis zur nächsten Anweisung nach der Unterbrechungsstelle ausgeführt werden. Damit AID das Kommando `%TRACE 1 IN S=`*srcname* bearbeiten kann, ist es notwendig, mit `%AID LOW=ALL` die Unterscheidung von Groß-/Kleinschreibung für die S-Qualifikation einzuschalten und mit %SYMLIB die PLAM-Bibliothek anzumelden, die die LSD zum Programm `pexec` enthält.
Da Vater- und Sohn-Task parallel laufen, empfiehlt es sich, der Übersichtlichkeit wegen den Prompt der jeweils anderen Task mit EM DÜ zu beantworten, bis die Ausgabe des %TRACE-Kommandos vollständig ist.

```
%0000000889/%aid low=all
%0000000888/<EM><DÜ>
%0000000889/%symlib test.lib
%0000000888/<EM><DÜ>
%0000000889/%trace 1 in s=n'pexec.c'
%0000000888/<EM><DÜ>
27                        BLOCK END, LOOP END
STOPPED AT SRC_REF: 27, SOURCE: pexec.c , BLK: 17 , END OF TRACE
%0000000888/...
```

Dasselbe Verfahren wie im vorherigen Schritt wird für die Sohn-Task durchgeführt.

8 Besonderheiten beim Testen unter POSIX

Neben Informationen zur Vererbung des Test-Kontextes bei `fork()`- oder `exec()`-Aufruf und zur Dump-Bearbeitung beim Testen unter POSIX finden Sie in diesem Kapitel Hinweise dazu, welche Strategien beim Testen von Fork-Tasks und von Programmen, die mit `exec()`-Aufruf geladen wurden, zum Erfolg führen.

8.1 Vererbung des Test-Kontextes

In einer durch `fork()` erzeugten Task gilt als einzige Einstellung `%AID FORK=ALL` weiter, falls dies in der Vater-Task gesetzt war. Alle übrigen Vereinbarungen wie:

– mit %AID festgelegte Einstellungen,
– gesetzte Testpunkte,
– mit %ON überwachte Ereignisse,
– mit %SYMLIB angemeldete PLAM-Bibliotheken, usw.

sind in der Fork-Task zurückgesetzt.

In einem Programm, das mit `exec()` geladen wurde, bleiben dagegen Einstellungen mit %AID und Vereinbarungen mit %SYMLIB erhalten. Alle übrigen Vereinbarungen sind aber ebenso wie in einer Fork-Task zurückgesetzt.

8.2 Teststrategien

Wenn Sie zum Testen von Fork-Tasks nur ein BS2000-Terminal oder eine entsprechende Emulation zur Verfügung haben, kann sich das Testen mehrerer paralleler Fork-Tasks, da diese um das Terminal konkurrieren, problematisch gestalten.

In diesem Abschnitt erhalten Sie Hinweise auf eine zweckmäßige Vorgehensweise, um beim Testen von Fork-Tasks und von Programmen, die über einen `exec()`–Aufruf geladen werden, möglichst rasch zum Erfolg zu kommen.

Eine geeignete Strategie besteht darin, jeden Programmteil, also Vater-Task, durch `fork()` erzeugte Tasks und durch `exec()` geladene Programme, zunächst unabhängig voneinander vollständig auszutesten. Für die Programme, die später über `exec()`–Aufrufe geladen werden sollen, bringt dies einen weiteren Vorteil mit sich. Wird das Programm über `exec()`–Aufruf geladen, kann die LSD nicht mitgeladen werden und muss explizit über %SYMLIB zugewiesen werden. Wenn Sie das Programm jedoch direkt mit dem POSIX-Kommando `debug` laden, können Sie die LSD mitladen lassen.
Den Aufruf-Kontext sollten Sie separat testen. Erst wenn alle Programmteile ohne Fehler sind und auch der Aufruf-Kontext fehlerfrei abläuft, sollten Sie darangehen, das gesamte Programmgefüge zu testen. Dazu empfiehlt es sich, sukzessive `fork()`– und `exec()`–Aufrufe dazuzunehmen, während die jeweils übergeordnete Task ruht, was Sie durch den vorübergehenden Einbau einer ausreichend langen Schleife oder durch einen geeigneten `wait()`–Aufruf erreichen.

In der Testphase sollte jeder Programmteil seine Ausgaben kennzeichnen, damit diese richtig zugeordnet werden können. Auf die Zuordnung der Ein-/Ausgaben beim gleichzeitigen Test mehrerer Fork-Tasks wird im Abschnitt „Zuordnung“ auf Seite 312 noch ausführlicher eingegangen.

Sie sollten das Programm zunächst in der POSIX-Shell testen. Hier laufen die verschiedenen Fork-Tasks gleichberechtigt ab, d.h. jede Fork-Task erhält gleichermaßen die Möglichkeit, sich am Terminal zu melden, um Eingaben anzufordern oder Ausgaben zu machen. Wird das Programm dagegen in der LOGON-Task gestartet, dann hat der BS2000-Kommandomodus höhere Priorität als der Testmodus der Fork-Tasks. Dies hat zur Folge, dass die Vater-Task u.U. das Terminal blockiert und die Fork-Tasks keine Gelegenheit zu Ein-/Ausgaben am Terminal erhalten.
Hilfreich ist beim gleichzeitigen Testen mehrerer Fork-Tasks eine Tabelle der folgenden

Form, in der Sie sich zu der Nummer der jeweiligen Fork-Task die zugehörige Prozessnummer und TSN sowie die Source-Referenzen des `fork()`-Aufrufs und der aktuellen Unterbrechungsstelle notieren können:

Fork-Nummer	**pid**	**TSN**	**Source-Referenz**	
			Start Source-code des Fork	**aktuelle Unterbrechung**
F1	929	0ND1		168
F11	930	0ND2	110	124, 128
...	...	...	...	...

Tabelle 9: Übersicht über aktive Fork-Tasks

Des weiteren wird darauf hingewiesen, dass das Programm unmittelbar nach einem `fork()` oder `exec()` im Laufzeitsystem unterbrochen wird. Von dieser Unterbrechungsstelle aus können Sie Daten, Funktionen und Source-Referenzen nur mit der vollen Qualifikation ansprechen. Um sich Schreibarbeit zu sparen, empfiehlt es sich, zunächst mit `%TRACE 1 IN S=`*srcname* bis zur nächsten ausführbaren Anweisung des Benutzerprogramms vorzurücken.

Unter POSIX können Sie die K2-Taste nicht verwenden. Um einen POSIX-Prozess abzubrechen, müssen Sie die Zeichenfolge „@@c“ eingeben. Die POSIX-Shell meldet sich danach mit ihrem Prompt, in der Regel „$“. Eine Task im Testmodus können Sie mit dem BS2000-Kommando EXIT-JOB bzw. LOGOFF abbrechen, oder Sie geben von einer weiteren Task aus das Kommando CANCEL-JOB mit der TSN der abzubrechenden Task ein (siehe „BS000-Benutzer-Kommandos (SDF-Format)“).

8.3 Ein-/Ausgaben

Beim Testen von Fork-Tasks konkurrieren die einzelnen Tasks um das Terminal. Ausgaben der verschiedenen Fork-Tasks werden zunächst in eine Warteschlange eingehängt und dann der Reihe nach abgearbeitet. Es gibt daher für Ein-/Ausgaben im Testmodus bestimmte Regeln zu beachten, die von denen für „normales“ Testen im BS2000-Kommandomodus abweichen können. In den folgenden Abschnitten werden mögliche Eingaben im Testmodus, Probleme bei der Zuordnung von Ein-/Ausgaben zu den verschiedenen Tasks sowie mögliches Fehlerverhalten behandelt.

8.3.1 Mögliche Eingaben

Im Testmodus können Sie alle AID-Kommandos und die meisten BS2000-Kommandos eingeben. Es sind alle die BS2000-Kommandos zugelassen, die Sie auch in einer Kommandofolge und in Subkommandos angeben können (siehe AID-Basishandbuch [1]). Ein geführter SDF-Dialog ist nicht möglich.

Kommandofolgen, bestehend aus AID- und BS2000-Kommandos, die durch Semikolon (;) getrennt sind, können ebenfalls eingegeben werden. Auch hier gelten die Einschränkungen, die im AID-Basishandbuch im Abschnitt „Kommandofolgen und Subkommandos“ [1] beschrieben sind. Diese Einschränkungen gelten ebenfalls, wenn im Testmodus nur BS2000-Kommandos eingegeben werden, also auch einzelne.
Wie im BS2000-Kommandomodus können Sie auch im Testmodus nur EM DÜ eingeben. Diese „leere“ Eingabe ist im Testmodus nötig, um der gewünschten Task von mehreren einer Fork-Familie die Möglichkeit zu geben, sich am Terminal mit dem Prompt zu melden. Eventuell müssen Sie mehrmals EM DÜ eingeben, da sich die Tasks in der Reihenfolge am Terminal melden, in der die zugehörigen Ein-/Ausgabeanforderungen in der Warteschlange eingetragen sind.

Beispiel

```
$ debug ex1fork
%  AID0348 Program stopped due to EXEC event (PID=0000002893)
%0000002893/%on %svc(44) <%trace 1 %instr>
%0000002893/%aid fork=next
```

Zunächst wird das Programm mit dem POSIX-Kommando `debug` geladen. Programm `ex1fork` enthält einen `fork()`-Aufruf.
Mit dem AID-Kommando %ON wird die Überwachung des SVC mit der Nummer 44 eingeschaltet. Das zugehörige Subkommando sorgt dafür, dass der SVC ausgeführt und dass das Programm unmittelbar danach angehalten wird.
Mit `%AID FORK=NEXT` schalten Sie den Testmodus für Fork-Tasks der ersten Generation ein.

```
%0000002893/%resume
ICXSVCTU+7C46     SVC   44                      1 FCT=POSPWENT IR1=010BCA40
                                                PAR=00E4B601 00000000 00000000
STOPPED AT V'10646D0' = ICXSVCTU + #'7C48' , END OF TRACE
%0000002893/%r
ICXSVCTU+7C46     SVC   44                      1 FCT=POSLDENV IR1=010BCA40
                                                PAR=00E34101 00000000 00000000
STOPPED AT V'10646D0' = ICXSVCTU + #'7C48' , END OF TRACE
%0000002893/%r
...
%0000002893/%r
ICXSVCTU+7C46     SVC   44                      1 FCT=POSFORK  IR1=010BCEB0
                                                PAR=00E30201 00000000 00000000
STOPPED AT V'10646D0' = ICXSVCTU + #'7C48' , END OF TRACE
```

Die folgenden %RESUME-Kommandos führen das Programm bis zum entscheidenden SVC 44 aus (FCT=POSFORK). Dieser SVC startet die Erzeugung der Fork-Task.

```
%0000002893/<EM><DÜ>
%  AID0348 Program stopped due to FORK event (PID=0000002897)
%0000002893/<EM><DÜ>
```

Den Prompt der Vater-Task beantworten Sie nun mit einer leeren Eingabe (EM DÜ). AID gibt die Meldung AID0348 aus, die bestätigt, dass die Fork-Task erzeugt wurde. Den nachfolgenden Prompt der Vater-Task schicken Sie wieder mit EM DÜ ab (erzwungener Task-Wechsel). Nun fordert Sie AID mit dem Prompt der Fork-Task zur Kommandoeingabe auf.

```
%0000002897/%show %base
%0000002893/<EM><DÜ>
%BASE E=VM
TSN: 0J05    TID: 0091017D
%AINT = %MODE31
BS:  V13.0   HW:  CFCS V3   ESA
%0000002897/
```

Damit die Informationen des Kommandos %SHOW %BASE am Terminal ausgegeben werden können, ist es wiederum notwendig, den Prompt der Vater-Task mit einer leeren Eingabe zu beantworten.

8.3.2 Zuordnung

Wie schon mehrfach erwähnt, ist es generell von Vorteil, jeweils nur eine Task zu testen und weitere vom Programm erzeugte Tasks solange ruhen zu lassen. Wenn es dennoch einmal vorkommt, dass mehrere Tasks parallel laufen und aufs Terminal ausgeben, so gilt Folgendes:

– Eindeutig zuordnen lassen sich nur die Eingaben, und zwar geht jede Eingabe stets an die Task, mit deren Prompt sie abgeschickt wurde.

– Ausgaben können i.a. nicht zugeordnet werden. Eine Ausnahme bilden Protokolle des %TRACE, die sich anhand der Source-Referenzen zuordnen lassen.
Wenn mehrere Tasks versuchen, gleichzeitig am Terminal auszugeben, ist die Reihenfolge, in der ausgegeben wird, mehr oder weniger zufällig. Solange Sie auf die Ausgabe einer Task warten, sollten Sie unbedingt den Prompt einer weiteren Task mit leerer Eingabe abschicken, bis die erwartete Ausgabe vollständig ist (siehe Beispiel oben).

– Ausgaben der Programme sollten Sie während der Testphase entweder in eine Datei umleiten oder vorübergehend mit einem vorangestellten Programmkürzel kennzeichnen, damit die Zuordnung gewährleistet ist.

8.3.3 Fehlerverhalten

Eine Fork-Task kann in den folgenden Fällen keine Ein-/Ausgabe am Terminal durchführen:

– In der LOGON-Task ist kein Programm geladen.

– In der LOGON-Task ist ein anderes Programm geladen als das, aus dem die Fork-Task (direkt oder indirekt) erzeugt wurde.

– Die LOGON-Task wurde beendet.

Dies gilt analog auch für Programme, die in der POSIX-Shell gestartet wurden.

In allen diesen Fällen wird die Fork-Task bei dem Versuch, vom Terminal einzulesen oder auf das Terminal auszugeben, ohne weitere Fehlermeldung abgebrochen. Dies gilt auch, wenn die Ein-/Ausgabe-Anforderung bereits in der Warteschlange eingetragen ist.

Die Fork-Task wird nicht abgebrochen, solange die Ein-/Ausgabe in Dateien umgeleitet ist.

8.4 Dump bearbeiten

Dumps (Speicherabzüge) von Fork-Tasks und von Programmen, die über einen `exec()`-Aufruf geladen wurden, können Sie wie gewohnt bearbeiten. Generell werden Dumps im BS2000 abgelegt, auch wenn das Programm, das den Dump erzeugt hat, in der POSIX-Shell gestartet wurde. Falls AID zum Dump eines POSIX-Programms LSD über das AID-Kommando %SYMLIB nachladen soll, müssen Sie beachten, dass %SYMLIB nicht auf POSIX-Dateien zugreifen kann. Die entsprechende Datei muss zunächst mit dem POSIX-Kommando `bs2cp` als L-Element in eine PLAM-Bibliothek im BS2000 kopiert werden und kann dann mit %SYMLIB zugewiesen werden (siehe auch Abschnitt „Nachladen der LSD“ auf Seite 18).
Kommando %DUMPFILE, das AID veranlasst, die Dump-Datei zu öffnen, und Kommando %BASE, mit dem Sie AID angeben, dass in der Folge der Speicherabzug, der in dieser Dump-Datei steht, untersucht werden soll, sind im Kapitel „AID-Kommandos“ auf Seite 117 beschrieben.

In der POSIX-Shell wird zu Programmen, die wegen eines Fehlers abgebrochen werden, stets ein User-Dump geschrieben. Die Abfrage „IDA0N45 Dump desired?“, die Sie vom BS2000 her kennen, unterbleibt. Das Programm wird entladen.
Zum Testen empfiehlt es sich daher, Programmfehler mit %ON %ANY abzufangen. Dann meldet AID im Fehlerfall die Adresse der Unterbrechungsstelle, an der der Fehler aufgetreten ist und das Ereignis, das den Fehler verursacht hat. Das Programm bleibt geladen. Sie können sofort den Fehlerkontext untersuchen. Falls sich der Fehler mit AID-Kommandos beheben lässt, können Sie den Programmablauf mit %RESUME fortsetzen. Wenn eine Fortsetzung des Programms jedoch nicht möglich ist, können Sie die Task mit EXIT-JOB bzw. LOGOFF beenden, um danach mit weiteren Tests den Programmfehler zu analysieren.

9 Anwendungsbeispiele

In diesem Kapitel finden Sie drei AID-Testsitzungen mit kleinen C- bzw. C++-Programmen. Anhand dieser Testsitzungen können Sie die Anwendung und Wirkung einiger AID-Kommandos nachvollziehen; die Vorgehensweise ist bewusst einfach gehalten.

Die Beispiele in diesem Kapitel wurden auf einer bestimmten BS2000-Anlage durchgeführt. Der Testverlauf auf einer anderen Maschine kann von dem hier abgebildeten geringfügig abweichen, abhängig vom installierten Betriebssystem sowie den übrigen Systemkomponenten.

9.1 C-Anwendungsbeispiel in BS2000

Das Programm soll maximal fünf Namen und Telefonnummern einlesen, die Namen sollen sortiert werden und als Liste mit den Telefonnummern ausgegeben werden.
Zunächst ist das Source-Error-Listing des C-Programms abgebildet; der Testablauf ist anschließend dargestellt.
Zur besseren Lesbarkeit sind Benutzereingaben in den Ablaufprotokollen **fettgedruckt** .
Daten- und Funktionsnamen sind im Fließtext in `Schreibmaschinenschrift` dargestellt.

9.1.1 Source-Error-Listing

```
*** SOURCE - ERROR - LISTING ** BS2000 C/C++ COMPILER 03.2E21 DATE:2015-02-27  PAGE:    1
                     SOURCENAME:*LIB-ELEM(MYLIB,NLIST.C(*HIGHEST-EXISTING),S)
_____________________________________________________________________________________

  EXP     SRC  BLOCK
  LIN ... LIN   LEV

    1       1     0  #include <stdio.h>
 1662       2     0  #include <stdlib.h>
 2128       3     0  #include <string.h>
 2414       4     0  #define MAX 5
 2415       5     0
 2416       6     0  struct tele {                 /* Structure tele */
 2417       7     0     char           name[15];
 2418       8     0     unsigned long  number;
 2419       9     0     };
 2420      10     0
 2421      11     0  void nlist(struct tele array[], int count);  /* Function nlist */
 2422      12     0  int nread(struct tele array[]);              /* Function nread */
 2423      13     0
 2424      14     0  int main(void)
 2425      15     0  {
 2426      16     1     struct tele arrp[MAX];     /* Phone list */
 2427      17     1     int nentry;                /* Number of entries */
 2428      18     1
 2429      19     1     nentry = nread(arrp);      /* Read in */
 2430      20     1     qsort (arrp, nentry, sizeof(struct tele),
 2431      21     1        (int(*)(const void*, const void*))strcmp);  /* Sort list */
 2432      22     1     nlist (arrp, nentry);      /* Output list */
 2433      23     1     return 0;
 2434      24     1  }
 2435      25     0
 2436      26     0  int nread(struct tele f[])        /* Read in names and numbers */
 2437      27     0  {
 2438      28     1     int i;
 2439      29     1
 2440      30     1     for (i=0; i<=MAX; i++)
 2441      31     1     {   printf ("Enter a name (end = 9): ");
 2442      32     2         if ((scanf ("%s", f[i].name) != 1) ||
 2443      33     2               (f[i].name[0] == '9')) break;
 2444      34     2     do {
 2445      35     3         fflush(stdin);
 2446      36     3         printf ("Now enter the phone number: ");
 2447      37     3        } while (scanf ("%lu", f[i].number) != 1);
 2448      38     2     }
 2449      39     1     return 0;
 2450      40     1  }
 2451      41     0
 2452      42     0  void nlist (struct tele f[], int n)             /* Output list */
 2453      43     0  {
 2454      44     1     int i;
 2455      45     1
 2456      46     1     printf ("%-25s | %10s\n", "Name", "Number");      /* Header */
 2457      47     1     printf ("----------------------------------------\n");
 2458      48     1     for (i=0; i<n; i++)
 2459      49     1        printf ("%-25s | %10lu\n", f[i].name, f[i].number);
 2460      50     1     printf ("\n\n");
 2461      51     1     return;
 2462      52     1  }
```

9.1.2 Testablauf

1. Schritt

Das C-Quellprogramm in der Datei `NLIST.C` wird mit dem C-Compiler übersetzt. Wegen der Angabe der Anweisung `MODIFY-TEST-PROPERTIES TEST-SUPPORT=*YES` erzeugt C die LSD-Sätze, die das symbolische Testen ermöglichen. Die Optimierung wird bei der Übersetzung durch die Anweisung `MODIFY-OPTIMIZATION-PROPERTIES LEVEL=*LOW` unterdrückt (siehe Abschnitt „Übersetzen in BS2000“ auf Seite 14). Die Übersetzung ergibt keine Fehler.

```
/START-CPLUS-COMPILER
%  BLS0523 ELEMENT 'SDFCC', VERSION '03.2E21', TYPE 'L' FROM LIBRARY
         ':2OSH:$TSOS.SYSLNK.CPP.032' IN PROCESS
%  BLS0524 LLM 'SDFCC', VERSION '03.2E21' OF '2015-02-24 07:17:56' LOADED
%  BLS0551 COPYRIGHT (C) 2015 Fujitsu Technology Solutions GmbH. ALL RIGHTS RESERVED
%  CDR9992 : BEGIN C/C++ VERSION 03.2E21
//MODIFY-SOURCE-PROPERTIES       LANGUAGE=*C(MODE=*ANSI)",DEFINE=..."
//MODIFY-TEST-PROPERTIES         TEST-SUPPORT=*YES
//MODIFY-OPTIMIZATION-PROPERTIES LEVEL=*LOW
...
%  CDR9907 : NOTES: 0 WARNINGS: 0 ERRORS: 0   FATALS: 0
%  CDR9997 : MODULES GENERATED
//MODIFY-BIND-PROPERTIES  INCLUDE = *LIB-ELEM(L=MYLIB,E=NLIST.O)
//BIND                    OUTPUT  = *LIB-ELEM(L=MYLIB,E=NLIST)
...
//END
%  CDR9908 : END   C TIME USED = 3.5300 SEC
%  CCM0998 CPU TIME USED: 3.6159 SECONDS
```

2. Schritt

Das Programm wird geladen. Mit dem Kommando `%AID C=YES` geben Sie an, dass AID C-String-Literale akzeptiert und `char`-Vektoren als C-Strings verarbeiten kann. Implizit wird durch `C=YES` die Unterscheidung von Klein- und Großbuchstaben und die Interpretation des Bindestrichs als Minuszeichen eingeschaltet.
Das Kommando `%ON %ANY` sorgt dafür, dass im Falle eines Fehlers das Programm nicht entladen wird und AID eine Fehlermeldung ausgibt, die die Adresse der Unterbrechungsstelle und das Ereignis enthält, das die Unterbrechung verursacht hat.

Nach dem Starten des Programms mit %RESUME läuft es zunächst richtig ab, Namen und Nummern werden angefordert, aber die Funktion `nread` bricht das Einlesen nach der 5. Nummer nicht ab, sondern fordert einen weiteren Namen an. Das Einlesen des 6. Namens führt jedoch zu einem Seitenfehler. An der Unterbrechungsstelle kann nun untersucht werden, was den Fehler verursachte.
Mit den folgenden beiden %DISPLAYs wird der Index und das zugehörige Vektorelement `f[i].name` ausgegeben. AID meldet eine ungültige Adresse für `f[i].name`: der Vektor hat nur 5 Elemente, also ist 4 der höchste zugelassene Wert für `i`. Wenn Sie sich die Startanweisung der Schleife in Zeile 30 ansehen, stellen Sie fest, dass die Abfrage auf Schleifenende falsch gestellt ist: statt `i<=MAX` muss auf `i<MAX` abgeprüft werden.

Um trotz des aufgetretenen Fehlers im Programm fortzufahren, setzen Sie die Adresse der `return`-Anweisung, die zur Source-Referenz S'39' hinterlegt ist, in den Befehlszähler und starten das Programm mit %RESUME an dieser Adresse. Es ist allerdings Vorsicht geboten, wenn durch Überschreiben des Befehlszählers größere Sprünge im Programm durchgeführt werden sollen. Nicht immer ist am Sprungziel gewährleistet, dass die Registerstände, Dateistatus, Inhalte von Indizes und ähnliches stimmen, um das Programm fehlerfrei fortzusetzen.
Das Programm läuft jetzt bis zum Ende durch; die Funktion `nlist` gibt jedoch nur die Überschrift aus.

```
/LOAD-PROG *M(MYLIB,NLIST,RUN-MODE=ADV,PROGRAM-MODE=ANY), TEST-OPT=AID
%  BLS0523 ELEMENT 'NLIST', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB' IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$NLIST', VERSION ' ' OF '2015-01-14 12:24:58'  LOADED
/%aid c=yes
/%on %any
/%resume
Enter a name (end = 9): Heinz
Now enter the phone number: 123
Enter a name (end = 9): Kraft
Now enter the phone number: 456
Enter a name (end = 9): Bauer
Now enter the phone number: 789
Enter a name (end = 9): Peters
Now enter the phone number: 101112
Enter a name (end = 9): Burgenbaur
Now enter the phone number: 131415
Enter a name (end = 9): Harbiger
STOPPED AT SRC_REF: 32, SOURCE: NLIST.C , BLK: 31 , EVENT: PAGING ERROR
/%d i
*** TID: 00420333 *** TSN: 8MEH ******************************************************
SRC_REF: 32 SOURCE: NLIST.C   BLK: 31  ***************************************************
i               =          5
/%d f[i].name
*.name( 0: 14)
%  AID0396 Invalid address for name
/%s s'39' into %pc
/%r
Name                       |     Number
-----------------------------------


%  CCM0998 CPU TIME USED: 0.0540 SECONDS
STOPPED AT V'1018846' = ITOTRM@@ + #'2E' , EVENT: TERM (NORMAL,PROGRAM,NODUMP)
```

3. Schritt

Das Programm wird neu geladen. Um den Programmfehler mit Hilfe von AID zu umgehen, wird über das Subkommando im %INSERT S'31' das Einlesen von Namen und Nummern nach 5 Durchgängen abgebrochen. Das nachfolgende Kommando %CONTROL1 bewirkt, dass zu Beginn und vor dem Verlassen von Funktion `nlist` angehalten wird. So kann untersucht werden, wieso `nlist` nur die Überschrift der Telefonliste ausgibt.

Nach dem Starten mit %RESUME werden jetzt ganz richtig genau 5 Namen und Nummern eingelesen. Wegen des %CONTROL hält das Programm an der ersten ausführbaren Anweisung von `nlist` an. Mit dem %DISPLAY-Kommando lassen Sie sich den Inhalt des

Übergabeparameters `n` ausgeben: `n` hat den Wert 0. Die Variable `nentry`, die das Resultat von `nread` aufnimmt und an `nlist` weiterreicht, wird offenbar nicht richtig gesetzt, was das folgende Kommando `%DISPLAY proc=main.nentry` bestätigt. Ein Blick ins Quellprogramm zeigt, dass Funktion `nread` anstatt `i` den Wert 0 übergibt: die `return`-Anweisung in Zeile 39 muss durch `return i;` ersetzt werden.

Mit dem nächsten %SET-Kommando setzen Sie `n` auf 5, woraufhin `nlist` die Liste der Namen und Telefonnummern ausgibt. Die Namen sind jedoch nicht sortiert, was damit zusammenhängt, dass beim Aufruf der Bibliotheksfunktion `qsort` die Variable `nentry` noch den Wert 0 hatte:
Außerdem stimmen die Nummern nicht. Die Vermutung liegt nahe, dass beim Funktionsaufruf von `scanf` ein Adressoperator vergessen wurde. Das würde bedeuten, dass `scanf` den Inhalt von `f[0].number` als Adresse verwendet und an dieser Adresse die Nummer eingetragen hat. Im folgenden %DISPLAY wird die falsche Adressierung nachvollzogen. Tatsächlich steht dort die 1. Telefonnummer „123“.
Der `scanf`-Aufruf in Zeile 37 müsste richtig lauten:
`scanf ("%lu", &f[i].number).`

```
/LOAD-PROG *M(MYLIB,NLIST,RUN-MODE=ADV,PROGRAM-MODE=ANY), TEST-OPT=AID
%  BLS0523 ELEMENT 'NLIST', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB' IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$NLIST', VERSION ' ' OF '2015-01-14 12:24:58'  LOADED
/%on %any
/%in s'31' <s31:(i eq 5): %s s'39' into %pc>
/%c1 %proc in proc=nlist
/%r
Enter a name (end = 9): Heinz
Now enter the phone number: 123
Enter a name (end = 9): Kraft
Now enter the phone number: 456
Enter a name (end = 9): Bauer
Now enter the phone number: 789
Enter a name (end = 9): Peters
Now enter the phone number: 101112
Enter a name (end = 9): Burgenbaur
Now enter the phone number: 131415
STOPPED AT SRC_REF: 46, SOURCE: NLIST.C , PROC: nlist
/%d n
*** TID: 00420333 *** TSN: 8MEH ***********************************************************
SRC_REF: 46 SOURCE: NLIST.C  PROC: nlist  ************************************************
n               =           0
/%d proc=main.nentry
nentry          =           0
/%s 5 into n
/%r
Name                        |     Number
--------------------------------------
Heinz                       |   17988680
Kraft                       |   17989292
Bauer                       |   17989136
Peters                      |          0
Burgenbaur                  |          0


STOPPED AT SRC_REF: 51, SOURCE: NLIST.C , PROC: nlist
/%d f[0].number->%f
CURRENT PC: 010004E2    CSECT: NLIST$0&@  ********************************************
V'01127C48' = IC@RT20A + #'00027C48'
01127C48 (00027C48)        +123
```

4. Schritt

Das Programm wird erneut geladen. Die gefundenen Fehler werden mit den %INSERT-Kommandos behoben:

– Der erste %INSERT ersetzt den fehlenden Adressoperator vor `f[i].number` beim Aufruf von `scanf`.

– Der zweite %INSERT dient der Kontrolle der eingegebenen Namen und Nummern, zusätzlich wird der jeweilige Index ausgegeben.

– Der dritte %INSERT bricht wie zuvor die Schleife in `nread` ab.

– Der vierte INSERT korrigiert schließlich den Wert von `nentry`.

Mit %RESUME starten Sie das Programm.

Namen und Nummern werden jetzt richtig eingelesen, die Telefonliste wird sortiert ausgegeben.
Aufgrund des Kommandos `%ON %ANY` wird auch bei ordnungsgemäßem Ablauf vor dem endgültigen Entladen des Programms angehalten und eine entsprechende STOP-Meldung ausgegeben.

```
/LOAD-PROG *M(MYLIB,NLIST,RUN-MODE=ADV,PROGRAM-MODE=ANY), TEST-OPT=AID
%  BLS0523 ELEMENT 'NLIST', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB' IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$NLIST', VERSION ' ' OF '2015-01-14 12:24:58'  LOADED
/%on %any
/%in s'32' <s32: %set %@(f[i].number) into f[i].number>
/%in s'38' <s38: %d i, f[i].number, f[i].name>
/%in s'31' <s31: (i eq 5): %s s'39' into %pc>
/%in s'20' <s20: %set 5 into nentry>
/%r
Enter a name (end = 9): Heinz
Now enter the phone number: 123
*** TID: 00420333 *** TSN: 8MEH ********************************************************
SRC_REF: 38 SOURCE: NLIST.C  PROC: nread  ***********************************************
i               =           0
*.number        =         123
*.name          = "Heinz"
Enter a name (end = 9): Kraft
Now enter the phone number: 456
i               =           1
*.number        =         456
*.name          = "Kraft"
Enter a name (end = 9): Bauer
Now enter the phone number: 789
i               =           2
*.number        =         789
*.name          = "Bauer"
Enter a name (end = 9): Peters
Now enter the phone number: 101112
i               =           3
*.number        =      101112
*.name          = "Peters"
Enter a name (end = 9): Burgenbaur
Now enter the phone number: 131415
i               =           4
*.number        =      131415
*.name          = "Burgenbaur"
```

Fortsetzung ...

Fortsetzung ...

```
Name                      |     Number
--------------------------------------
Bauer                     |        789
Burgenbaur                |     131415
Heinz                     |        123
Kraft                     |        456
Peters                    |     101112


%  CCM0998 CPU TIME USED: 0.1435 SECONDS
STOPPED AT V'1018846' = ITOTRM@@ + #'2E' , EVENT: TERM (NORMAL,PROGRAM,NODUMP)
```

9.2 C++-Anwendungsbeispiel in BS2000

Das vorliegende Testprogramm soll zwei Zeilen Text ausgeben und benutzt dazu die Klasse `string`.
Zur besseren Lesbarkeit sind Daten- und Funktionsnamen im Fließtext in `Schreibmaschinenschrift` gesetzt. In den Ablaufprotokollen sind die Eingabekommandos **fettgedruckt**.

9.2.1 Source-Error-Listing

```
*** SOURCE - ERROR - LISTING ** BS2000 C/C++ COMPILER 03.2E21 DATE:2015-02-27  PAGE:    1
                                SOURCENAME: *LIB-ELEM(MYLIB,STRING.C(*HIGHEST-EXISTING),S)
_____________________________________________________________________________________________

EXP   INC    FILE   SRC
LIN   LEV     NO    LIN

  1    0      0      1
  2    0      0      2  extern "C" void* malloc(unsigned);
  3    0      0      3  extern "C" void  free(void*);
  4    0      0      4
  5    0      0      5  extern "C" int   strlen( const char* );
  6    0      0      6  extern "C" char* strcpy( char*, const char* );
  7    0      0      7  extern "C" char* strcat( char*, const char* );
  8    0      0      8
  9    0      0      9  extern "C" int   printf( const char*, ... );
 10    0      0     10
 11    0      0     11  class string
 12    0      0     12  {
 13    0      0     13          int length;
 14    0      0     14          char* start;
 15    0      0     15  public:
 16    0      0     16          /*
 17    0      0     17           * constructors
 18    0      0     18           */
 19    0      0     19          string() : length(0), start(0) {};
 20    0      0     20          string( const char *s ) {
 21    0      0     21                length = strlen(s) + 1;
 22    0      0     22                start = new char[length];
 23    0      0     23                strcpy( start, s );
 24    0      0     24          };
 25    0      0     25          string( const string &s ) {
 26    0      0     26                length = s.length;
 27    0      0     27                start  = new char[length];
 28    0      0     28                strcpy( start, s );
 29    0      0     29          };
 30    0      0     30          const string& operator=( const string& s )
 31    0      0     31          {
 32    0      0     32                length = s.length;
 33    0      0     33                start  = new char[length];
 34    0      0     34                strcpy( start, s );
 35    0      0     35                return *this;
 36    0      0     36          };
 37    0      0     37          /*
 38    0      0     38           * destructor
 39    0      0     39           */
 40    0      0     40          ~string() {
 41    0      0     41                delete start;
 42    0      0     42          };
```

```
43    0       0     43          /*
44    0       0     44           * conversion
45    0       0     45           */
46    0       0     46          operator char*() const {
47    0       0     47                return start;
48    0       0     48          };
49    0       0     49  };
50    0       0     50  /*
51    0       0     51   * string concatenation
52    0       0     52   */
53    0       0     53  string& operator + ( const string& p, const string& q )
54    0       0     54  {
55    0       0     55        static string s = p;      // copy first string
56    0       0     56        s = strcat(s,q);          // cat second string
57    0       0     57        return s;
58    0       0     58  }
59    0       0     59
60    0       0     60  string s = "Hello";
61    0       0     61
62    0       0     62  int main(void)
63    0       0     63  {
64    0       0     64        string p(s);              // p is "Hello"
65    0       0     65
66    0       0     66        string q("World\n");      // q is "World\n"
67    0       0     67
68    0       0     68        printf(p + " C++ " + q); // should print "Hello C++ World\n"
69    0       0     69
70    0       0     70        p = "Goodbye";            // p is now "Goodbye"
71    0       0     71
72    0       0     72        q = "C " + q;             // q is now "C World\n"
73    0       0     73
74    0       0     74        printf(p + q);            // should print "Goodbye C World\n"
75    0       0     75
76    0       0     76        return 0;
77    0       0     77  }
```

9.2.2 Testablauf

1. Schritt

Das C++-Quellprogramm in der Datei `STRING.C` wird mit dem C++-Compiler übersetzt. Wegen der Angabe der Anweisung `MODIFY-TEST-PROPERTIES TEST-SUPPORT=*YES` erzeugt C++ die LSD-Sätze, die das symbolische Testen ermöglichen. Die Optimierung wird bei der Übersetzung durch die Anweisung `MODIFY-OPTIMIZATION-PROPERTIES LEVEL=*LOW`, das Expandieren von Inline-Funktionen wird durch `BUILTIN-FUNCTIONS=*NONE` unterdrückt (siehe Abschnitt „Übersetzen in BS2000“ auf Seite 14). Die Übersetzung ergibt keine Fehler. Anschließend soll das Programm mit der BIND-Anweisung des C/C++-Compilers gebunden werden. Damit die LSD-Informationen in das gebundene Modul miteingeschrieben werden, müssen Sie zunächst in der Anweisung `MODIFY-BIND-PROPERTIES` die Option `TEST-SUPPORT=*YES` setzen.

```
/START-CPLUS-COMPILER
%  BLS0523 ELEMENT 'SDFCC', VERSION '03.2E21', TYPE 'L' FROM LIBRARY
        ':2OSH:$TSOS.SYSLNK.CPP.032' IN PROCESS
%  BLS0524 LLM 'SDFCC', VERSION '03.2E21' OF '2015-02-24 07:17:56' LOADED
%  BLS0551 COPYRIGHT (C) 2015 Fujitsu Technology Solutions GmbH. ALL RIGHTS RESERVED
%  CDR9992 : BEGIN C/C++ VERSION 03.2E21
//MODIFY-SOURCE-PROPERTIES       LANGUAGE=*CPLUSPLUS()
//MODIFY-TEST-PROPERTIES         TEST-SUPPORT=*YES
//MODIFY-OPTIMIZATION-PROPERTIES LEVEL=*LOW, BUILTIN-FUNCTIONS=*NONE
//COMPILE SOURCE=*LIB(MYLIB,STRING.C),MODULE-OUTPUT=*LIB(MYLIB,STRING.O)
%  CDR9907 : NOTES: 0 WARNINGS: 0 ERRORS: 0   FATALS: 0
%  CDR9997 : MODULES GENERATED
//MODIFY-BIND-PROPERTIES START-LLM-CREATION = *YES,         -
//INCLUDE = *LIB-ELEM(L=MYLIB,E=STRING.O),                 -
//STDLIB=*STATIC,                                          -
//RUNTIME-LANGUAGE =*CPLUSPLUS(MODE=ANSI), TEST-SUPPORT = *YES
//BIND OUTPUT = *LIB-ELEM(LIB=MYLIB,ELEM=STRING)
%  BND1501 LLM FORMAT: '1'
%  BND1101 BINDER NORMALLY TERMINATED.
//END
%  CDR9908 : END   C TIME USED = 4.7600 SEC
%  CCM0998 CPU TIME USED: 4.8956
```

2. Schritt

Das Programm wird geladen. Mit dem Kommando %AID LOW wird die Unterscheidung von Klein- und Großbuchstaben eingeschaltet. Das Kommando %ON %ANY sorgt dafür, dass im Falle eines Fehlers das Programm nicht entladen wird und AID eine Fehlermeldung ausgibt, die die Adresse der Unterbrechungsstelle und das Ereignis enthält, das die Unterbrechung verursacht hat. Nach dem Starten mit %RESUME läuft das Programm ohne Fehler bis zum Ende durch, das Ergebnis ist jedoch falsch.

```
/LOAD-PROG *MOD(LIB=MYLIB,ELEM=STRING, -
/RUN-MOD=ADVANCED,PROGRAM-MODE=*ANY),TEST-OPT=*AID
%  BLS0523 ELEMENT 'STRING', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB'
IN PROCESSING
%  BLS0524 LLM '$LIB-ELEM$MYLIB$STRING$', VERSION ' ' OF '2015-03-06 13:26:24' LOADED
/%aid c=yes
/%on %any
/%resume
Hello C+World
Hello C+World
World
Hell
%  CCM0998 CPU TIME USED: 0.0047 SECONDS
STOPPED AT V'101B846' = ITOTRM@@ + #'2E' , EVENT: TERM (NORMAL,PROGRAM,NODUMP)
```

3. Schritt

Die Ausgaben deuten darauf hin, dass die Inhalte der `string`-Objekte überschrieben wurden. Um herauszufinden, welche Anweisung im Programm diesen Fehler verursacht, wird nach jeder dynamischen Speicheranforderung und der zugehörigen Initialisierung die Schreibüberwachung für den zugewiesenen Speicherbereich eingeschaltet. Mit %RESUME wird das Programm gestartet und läuft bis zur ersten durch die eingeschaltete Schreibüberwachung hervorgerufene Unterbrechung. Mit `%SDUMP %NEST` lassen Sie sich die Aufrufhierarchie anzeigen, um zu sehen, welche Anweisung des Programms den Inhalt von `start` überschrieben hat.

```
/LOAD-PROG *MOD(LIB=MYLIB,ELEM=STRING, -
/RUN-MOD=ADVANCED,PROGRAM-MODE=*ANY),TEST-OPT=*AID
%  BLS0523 ELEMENT 'STRING', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB'
IN PROCESSING
%  BLS0524 LLM '$LIB-ELEM$MYLIB$STRING$', VERSION ' ' OF '2015-03-06 13:26:24' LOADED
/%in s'24' <%on %write(start-> %l=(length))>
/%in s'29' <%on %write(start-> %l=(length))>
/%in s'35' <%on %write(start-> %l=(length))>
/%r
%  AID0496 Warning: previously defined event %WRITE is replaced
%  AID0496 Warning: previously defined event %WRITE is replaced
%  AID0496 Warning: previously defined event %WRITE is replaced
STOPPED AT V'1001E98' = IC@STRG@ + #'258' , EVENT: WRITE
/%sd %nest
*** TID: 010802A9 *** TSN: 6EJP ***************************************************
ABSOLUT: V'1001E98'   SOURCE: IC@STRG@  PROC: STRCAT  ********************************
SRC_REF: 56  SOURCE: STRING.C  PROC: operator+(const string &, const string &)  ********
SRC_REF: 68  SOURCE: STRING.C  PROC: main  ***************************************
ABSOLUT: V'10237B8'   SOURCE: ICS$MAI@  PROC: ICS$MAI@  ********************************
ABSOLUT: V'10019C8'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  ********************************
```

4. Schritt

In S’56’ wird die Funktion `strcat()` aufgerufen, die für das fehlerhafte Überschreiben verantwortlich ist. Im folgenden wird dies nun genauer analysiert. Da in der Anweisung in Zeile 56 das `string`-Objekt `s` verwendet wird, liegt der Verdacht nahe, dass dessen Inhalt modifiziert wurde, was durch die beiden %DISPLAYs bestätigt wird. Zunächst wird jedoch mit %QUALIFY eine Vorqualifikation festgelegt, damit in den %DISPLAYs nicht jedesmal die gesamte Qualifikationsangabe geschrieben werden muss.

```
/%q s=n'STRING.C'.proc=n'operator+(const string &, const string &)'
/%d .s.start->%l20
V'010E75C8' = ABSOLUT  + #'010E75C8'
010E75C8 (010E75C8) C8859393 9640C34E 4E400000 00000000    Hello C++ ......
010E75D8 (010E75D8) 00000000                               ....
/%d .s.length
ABSOLUT: V'01001E98' SOURCE: IC@STRG@  PROC: STRCAT  ***********************************
s.length        =           6
```

5. Schritt

Der Fehler wird also durch die Anweisung S'56', `s = strcat(s,q);` hervorgerufen. Die Funktion `strcat` modifiziert und verlängert ihr erstes Argument, was schon daran zu erkennen ist, dass in der Deklaration von `strcat` das erste Argument mit Typ `char*` und nicht mit Typ `const char*` angegeben ist.
Die Funktion `operator+(const string &, const string &)` wird so geändert, dass sie einen String in der Gesamtlänge der Strings aus `p` und `q` anlegt. Dazu muss sie auf die Komponenten `start` und `length` der Klasse `string` zugreifen und deshalb als Friend-Funktion der Klasse deklariert werden. In der Folge müssen noch einige Anpassungen vorgenommen werden, so dass das folgende Programm entsteht:

```
*** SOURCE - ERROR - LISTING ** BS2000 C/C++ COMPILER 03.2E21 DATE:2015-02-27  PAGE:    1
                                SOURCENAME: *LIB-ELEM(MYLIB,STRING.C(*HIGHEST-EXISTING),S)
_____________________________________________________________________________________

EXP   INC    FILE   SRC
LIN   LEV     NO    LIN

  1    0      0      1
  2    0      0      2  extern "C" void* malloc(unsigned);
  3    0      0      3  extern "C" void  free(void*);
  4    0      0      4
  5    0      0      5  extern "C" int   strlen( const char* );
  6    0      0      6  extern "C" char* strcpy( char*, const char* );
  7    0      0      7  extern "C" char* strcat( char*, const char* );
  8    0      0      8
  9    0      0      9  extern "C" int   printf( const char*, ... );
 10    0      0     10
 11    0      0     11  class string
 12    0      0     12  {
 13    0      0     13        int length;
 14    0      0     14        char* start;
 15    0      0     15  public:
 16    0      0     16        /*
 17    0      0     17         * constructors
 18    0      0     18         */
 19    0      0     19        string(int n=0) : length(n){
 20    0      0     20              start = new char[length];
 21    0      0     21        };
 22    0      0     22        string( const char *s ) {
 23    0      0     23              length = strlen(s) + 1;
 24    0      0     24              start = new char[length];
 25    0      0     25              strcpy( start, s );
 26    0      0     26        };
 27    0      0     27        string( const string &s ) {
 28    0      0     28              length = s.length;
 29    0      0     29              start  = new char[length];
 30    0      0     30              strcpy( start, s );
 31    0      0     31        };
 32    0      0     32        const string& operator=( const string& s )
 33    0      0     33        {
 34    0      0     34              delete start;
 35    0      0     35              length = s.length;
 36    0      0     36              start  = new char[length];
 37    0      0     37              strcpy( start, s );
 38    0      0     38              return *this;
 39    0      0     39        };
 40    0      0     40        /*
 41    0      0     41         * destructor
 42    0      0     42         */
 43    0      0     43        ~string() {
 44    0      0     44              delete start;
 45    0      0     45        };
```

```
46    0       0     46        /*
47    0       0     47         * conversion
48    0       0     48         */
49    0       0     49        operator const char*() const {
50    0       0     50              return start;
51    0       0     51        };
52    0       0     52  friend string& operator+(const string&,const string&);
53    0       0     53  };
54    0       0     54
55    0       0     55  /*
56    0       0     56   * string concatenation
57    0       0     57   */
58    0       0     58  string& operator + ( const string& p, const string& q )
59    0       0     59  {
60    0       0     60        static string s;
61    0       0     61        s = p.length + q.length -1;
62    0       0     62                                        //allocate right length
63    0       0     63        s.start = strcpy(s.start,p);    //copy first string
64    0       0     64        s.start = strcat(s.start,q);    //copy second string
65    0       0     65        return s;
66    0       0     66  }
67    0       0     67
68    0       0     68  string s = "Hello";
69    0       0     69
70    0       0     70  int main(void)
71    0       0     71  {
72    0       0     72        string p(s);              // p is "Hello"
73    0       0     73
74    0       0     74        string q("World\n");      // q is "World\n"
75    0       0     75
76    0       0     76        printf(p + " C++ " + q); // should print "Hello C++ World\n"
77    0       0     77
78    0       0     78        p = "Goodbye";            // p is now "Goodbye"
79    0       0     79
80    0       0     80        q = " C " + q;            // q is now "C World\n"
81    0       0     81
82    0       0     82        printf(p + q);            // should print "Goodbye C World\n"
83    0       0     83
84    0       0     84        return 0;
85    0       0     85  }
```

6. Schritt

Das geänderte Programm wird übersetzt, geladen und gestartet. Es läuft bis zum Ende durch, bringt jedoch wieder ein falsches Ergebnis.

```
/START-CPLUS-COMPILER
%  BLS0523 ELEMENT 'SDFCC', VERSION '03.2E21', TYPE 'L' FROM LIBRARY
        ':2OSH:$TSOS.SYSLNK.CPP.032' IN PROCESS
%  BLS0524 LLM 'SDFCC', VERSION '03.2E21' OF '2015-02-24 07:17:56' LOADED
%  BLS0551 COPYRIGHT (C) 2015 Fujitsu Technology Solutions GmbH. ALL RIGHTS RESERVED
%  CDR9992 : BEGIN C/C++ VERSION 03.2E21
//MODIFY-SOURCE-PROPERTIES       LANGUAGE=*CPLUSPLUS()
//MODIFY-TEST-PROPERTIES         TEST-SUPPORT=*YES
//MODIFY-OPTIMIZATION-PROPERTIES LEVEL=*LOW, BUILTIN-FUNCTIONS=*NONE
//COMPILE SOURCE=*LIB(MYLIB,STRING.C),MODULE-OUTPUT=*LIB(MYLIB,STRING.O)
%  CDR9907 : NOTES: 0 WARNINGS: 0 ERRORS: 0   FATALS: 0
%  CDR9997 : MODULES GENERATED
//MODIFY-BIND-PROPERTIES START-LLM-CREATION = *YES,         -
//INCLUDE = *LIB-ELEM(L=MYLIB,E=STRING.O),                 -
//STDLIB=*STATIC,                                          -
//RUNTIME-LANGUAGE =*CPLUSPLUS(MODE=ANSI), TEST-SUPPORT = *YES
//BIND OUTPUT = *LIB-ELEM(LIB=MYLIB,ELEM=STRING)
%  BND1501 LLM FORMAT: '1'
%  BND1101 BINDER NORMALLY TERMINATED.
%//END
%  CDR9908 : END   C TIME USED = 5.6300 SEC
%  CCM0998 CPU TIME USED: 5.7149
/LOAD-PROG *MOD(LIB=MYLIB,ELEM=STRING, -
/RUN-MOD=ADVANCED,PROGRAM-MODE=*ANY),TEST-OPT=*AID
%  BLS0523 ELEMENT 'STRING', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB' IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$STRING', VERSION ' ' OF '2015-01-25 10:57:18' LOADED
/%on %any
/%r
World
Goodbye C World
%  CCM0998 CPU TIME USED: 0.0027 SECONDS
STOPPED AT V'101C846' = ITOTRM@@ + #'2E' , EVENT: TERM (NORMAL,PROGRAM,NODUMP)
```

7. Schritt

Die erste Textzeile ("Hello C++ World") ist nicht richtig zusammengesetzt worden. Der Fehler liegt vermutlich in der Funktion `operator+(const string &, const string &)`. Deshalb werden nach dem erneuten Laden des Programms mehrere %CONTROL-Kommandos eingegeben, die den Inhalt von `s`, `p` und `q` jeweils vor Ausführung der Anweisungen S'61' bis S'63' ausgeben. Der nachfolgende %INSERT S'78' bewirkt, dass die Funktion `operator+(const string &, const string &)` nur bis zur Ausgabe der ersten Textzeile überwacht wird.
Die Stop-Meldung am Schluss des Protokolls wird aufgrund des Kommandos `%ON %ANY` ausgegeben, das auch bei ordnungsgemäßem Ablauf das Programm vor dem endgültigen Entladen anhält und den aktuellen Befehlszählerstand und den Namen der Laufzeitroutine ausgibt, die die Endebehandlung durchführt.

```
/LOAD-PROG *MOD(LIB=MYLIB,ELEM=STRING, -
/RUN-MOD=ADVANCED,PROGRAM-MODE=*ANY),TEST-OPT=*AID
%  BLS0523 ELEMENT 'STRING', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB' IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$STRING', VERSION ' ' OF '2015-01-25 10:57:18' LOADED
/%on %any
/%c1 %stmt in (s'61':s'63') <(s.length eq 0): %d s.start>
/%c2 %stmt in (s'61':s'63') <(s.length ne 0): %d ' ',s.length, -
/ 's.start->:',s.start->%l=(s.length)>
/%c3 %stmt in (s'61':s'63') <%d p.length,'p.start->:' -
/ ,p.start->%l=(p.length)>
/%c4 %stmt in (s'61':s'63') <%d q.length,'q.start->:' -
/ ,q.start->%l=(q.length)>
/%in s'78' <%rem %c>
/%r
*** TID: 010802A9 *** TSN: 6EJP ******************************************************
SRC_REF: 61 SOURCE: STRING.C    PROC: operator+(const string &, const string &)
s.start         = 010E85C8
p.length        =          6
p.start->:
CURRENT PC: 01000588    CSECT: STRING$O&@  *******************************************
V'010E8598' = ABSOLUT  + #'010E8598'
010E8598 (010E8598) C8859393 9600                          Hello.
SRC_REF: 61 SOURCE: STRING.C    PROC: operator+(const string &, const string &)
q.length        =          6
q.start->:
CURRENT PC: 01000588    CSECT: STRING$O&@  *******************************************
V'010E85B8' = ABSOLUT  + #'010E85B8'
010E85B8 (010E85B8) 40C34E4E 4000                           C++ .

SRC_REF: 63 SOURCE: STRING.C    PROC: operator+(const string &, const string &)  *******
s.length        =         11
s.start->:
CURRENT PC: 01000988    CSECT: STRING$O&@  *******************************************
'010E85F8' = ABSOLUT  + #'010E85F8'
010E85F8 (010E85F8) 00000000 00000000 000000               ..........
SRC_REF: 63 SOURCE: STRING.C    PROC: operator+(const string &, const string &)  *******
p.length        =          6
p.start->:
CURRENT PC: 010005EC    CSECT: STRING$O&@  *******************************************
V'010E8598' = ABSOLUT  + #'010E8598'
010E8598 (010E8598) C8859393 9600                          Hello.
```

Fortsetzung...

Fortsetzung ...

```
SRC_REF: 63 SOURCE: STRING.C   PROC: operator+(const string &, const string &)  *******
q.length        =           6
q.start->:
CURRENT PC: 010005EC    CSECT: STRING$0&@  ******************************************
V'010E6598' = ABSOLUT  + #'010E6598'
0010E6598 (010E6598) 40C34E4E 4000                            C++ .

SRC_REF: 61 SOURCE: STRING.C   PROC: operator+(const string &, const string &)  *******
s.length        =          11
s.start->:
CURRENT PC: 01000924    CSECT: STRING$0&@  ******************************************
V'010E65D8' = ABSOLUT  + #'010E65D8'
010E85F8 (010E85F8) C8859393 9640C34E 4E4000              Hello C++ .
SRC_REF: 61 SOURCE: STRING.C   PROC: operator+(const string &, const string &)  *******
p.length        =          11
p.start->:
CURRENT PC: 01000588    CSECT: STRING$0&@  ******************************************
V'010E85F8' = ABSOLUT  + #'010E85F8'
010E85F8 (010E85F8) C8859393 9640C34E 4E4000              Hello C++ .
SRC_REF: 61 SOURCE: STRING.C   PROC: operator+(const string &, const string &)  *******
q.length        =           7
q.start->:
CURRENT PC: 01000588    CSECT: STRING$0&@  ******************************************
V'010E85A8' = ABSOLUT  + #'010E85A8'
010E85A8 (010E85A8) E6969993 841500                           World..

SRC_REF: 63 SOURCE: STRING.C   PROC: operator+(const string &, const string &) ********
s.length        =          17
s.start->:
CURRENT PC: 010005EC    CSECT: STRING$0&@  ******************************************
V'010E85F8' = ABSOLUT  + #'010E85F8'
010E85F8 (010E85F8) 00859393 9640C34E 4E400000 00000000    .ello C++ ......
010E8608 (010E8608) 00                                      .
SRC_REF: 63 SOURCE: STRING.C   PROC: operator+(const string &, const string &) ********
p.length        =          17
p.start->:
CURRENT PC: 010005EC    CSECT: STRING$0&@  ******************************************
V'010E85F8' = ABSOLUT  + #'010E85F8'
010E85F8 (010E85F8) 00859393 9640C34E 4E400000 00000000    .ello C++ ......
010E8608 (010E8608) 00                                      .
SRC_REF: 63 SOURCE: STRING.C   PROC: operator+(const string &, const string &) ********
q.length        =           7
q.start->:
CURRENT PC:  010005EC    CSECT: STRING$0&@  ******************************************
V'010E85A8' = ABSOLUT  + #'010E85A8'
010E85A8 (010E85A8) E6969993 841500                           World..
World
Goodbye C World
%  CCM0998 CPU TIME USED: 0.1770 SECONDS
STOPPED AT V'101C846' = ITOTRM@@ + #'2E' , EVENT: TERM (NORMAL,PROGRAM,NODUMP)
```

8. Schritt

Wie schon vermutet, wird beim zweiten Aufruf von `operator+(const string &, const string &)` der String "World\n" nicht an den alten Text angehängt, sondern wieder von Anfang an in `s.start->` eingetragen. Um mit Hilfe von AID aufzuklären, was im einzelnen in der Anweisung S'61', `s = p.length + q.length -1;` geschieht, die ja implizite Aufrufe von Konstruktor `string::string(int)` und Funktion `operator=(const string &)` beinhaltet, wird nach erneutem Laden des Programms mit zwei %INSERT-Kommandos der Inhalt der Komponenten `length` und `start` an verschiedenen Testpunkten in `string::string(int)` und `operator=(const string &)` ausgegeben. Das letzte %INSERT-Kommando sorgt wieder dafür, dass das Programm nach der Ausgabe der ersten Textzeile ohne Überwachung bis zum Ende durchläuft.

```
/LOAD-PROG *MOD(LIB=MYLIB,ELEM=STRING, -
/RUN-MOD=ADVANCED,PROGRAM-MODE=*ANY),TEST-OPT=*AID
%  BLS0523 ELEMENT 'STRING', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB' IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$$STRING', VERSION ' ' OF '2015-01-25 10:57:18' LOADED
/%in s'61' <(%. eq 2): %in s'20' <%d length,'start->:',start->%1=(length)>>
/%in s'61' <(%. eq 2): %c1 %proc in proc=string::n'operator=(const string &)'-
/<%d ' ',length,s.length;%d 'start->:',start->%120; %d 's.start->:',s.start->%120>>
/%in s'78' <%rem %c; %rem %in>
/%r
*** TID: 010802A9 *** TSN: 6EJP ************************************************************
SRC_REF: 20 SOURCE: STRING.C  PROC: string::string(int)  ******************************
string.length   =          17
start->:
CURRENT PC: 01000236    CSECT: STRING$0&@  ********************************************
V'010E85D8' = ABSOLUT  + #'010E85D8'
010E85D8 (010E85D8) 00000000 00000000 00000000 00000000    ................
010E85E8 (010E85E8) 00                                     .

SRC_REF: 34 SOURCE: STRING.C  PROC: string::operator=(const string &)  *****************
string.length   =          11
s.length        =          17
start->:
CURRENT PC: 010003CA    CSECT: STRING$0&@  ********************************************
V'010E85F8' = ABSOLUT  + #'010E85F8'
010E85F8 (010E85F8) C8859393 9640C34E 4E400000 00000000    Hello C++ ......
010E8608 (010E8608) 00000000                               ....
s.start->:
V'010E85D8' = ABSOLUT  + #'010E85D8'
010E85D8 (010E85D8) 00000000 00000000 00000000 00000000    ................
010E85E8 (010E85E8) 00000000                               ....

SRC_REF: 38 SOURCE: STRING.C  PROC: string::operator=(const string &)  *****************
string.length   =          17
s.length        =          17
start->:
CURRENT PC: 0100046C    CSECT: STRING$0&@  ********************************************
V'010E85F8' = ABSOLUT  + #'010E85F8'
010E85F8 (010E85F8) 00859393 9640C34E 4E400000 00000000    .ello C++ ......
010E8608 (010E8608) 00000000                               ....
s.start->:
V'010E85D8' = ABSOLUT  + #'010E85D8'
010E85D8 (010E85D8) 00000000 00000000 00000000 00000000    ................
010E85E8 (010E85E8) 00000000                               ....
World
Goodbye C World
%  CCM0998       CPU TIME USED: 0.0920 seconds
```

9. Schritt

Aus den von AID gelieferten Informationen lässt sich der folgende Sachverhalt ableiten: beim zweiten Aufruf von *operator+(const string &, const string &)*, wenn der 3. Teil des Textes ("World\n") an das Ergebnis des ersten Durchgangs angehängt werden soll, ist der Übergabeparameter *p* gleich dem statischen Objekt *s*, in dem das Ergebnis des ersten Durchgangs von *operator+(const string &, const string &)* eingetragen wurde. Daher enthalten nach dem zweiten Aufruf der Funktion *s.start* und *p.start* dieselbe Adresse.
Die Anweisung S'61' ruft den Konstruktor *string::string(int)* auf, der für die berechnete Länge des Gesamtstrings *(p.length + q.length -1)* einen neuen Bereich anlegt, der die Ergebniszeile aufnehmen soll. Dieser Bereich enthält hier im Beispiel binäre Nullen.
Die Funktion *operator=(const string &)* kopiert den neu angeforderten Bereich nach *s*. Dadurch wird das erste Byte von *s.start->* mit X'00' überschrieben und, da *s* und *p* beim zweiten Durchgang identisch sind, gleichzeitig auch *p.start->* zerstört. Funktion *strcat()* in Anweisung S'63' kettet also *q* mit dem nunmehr leeren String *s.start->*.
Um den Fehler zu beheben, muss in *operator+(const string &, const string &)* der jeweils neue Text in ein dynamisches Objekt von *string* zwischengespeichert werden. Mit dieser Ergänzung sieht der endgültige Code für diese Funktion folgendermaßen aus:

```
string& operator + ( const string& p, const string& q )
{
   static string s;
   string s1 = p.length + q.length -1;         //allocate right length
   s1.start = strcpy(s1.start,p);              //copy first string
   s1.start = strcat(s1.start,q);              //copy second string
   s = s1;
   return s;
}
```

10. Schritt

Das Programm wird neu übersetzt, geladen und gestartet. Der Text wird nun richtig ausgegeben.

```
/START-CPLUS-COMPILER
%  BLS0523 ELEMENT 'SDFCC', VERSION '03.2E21', TYPE 'L' FROM LIBRARY
        ':2OSH:$TSOS.SYSLNK.CPP.032' IN PROCESS
%  BLS0524 LLM 'SDFCC', VERSION '03.2E21' OF '2015-02-24 07:17:56' LOADED
%  BLS0551 COPYRIGHT (C) 2015 Fujitsu Technology Solutions GmbH. ALL RIGHTS RESERVED
%  CDR9992 : BEGIN C/C++ VERSION 03.2E21
//MODIFY-SOURCE-PROPERTIES       LANGUAGE=*CPLUSPLUS()
//MODIFY-TEST-PROPERTIES         TEST-SUPPORT=*YES
//MODIFY-OPTIMIZATION-PROPERTIES LEVEL=*LOW, BUILTIN-FUNCTIONS=*NONE
//COMPILE SOURCE=*LIB(MYLIB,STRING.C),MODULE-OUTPUT=*LIB(MYLIB,STRING.O)
%  CDR9907 : NOTES: 0 WARNINGS: 0 ERRORS: 0   FATALS: 0
%  CDR9997 : MODULES GENERATED
//MODIFY-BIND-PROPERTIES START-LLM-CREATION = *YES,         -
//INCLUDE = *LIB-ELEM(L=MYLIB,E=STRING.O),                 -
//STDLIB=*STATIC,                                          -
//RUNTIME-LANGUAGE =*CPLUSPLUS(MODE=ANSI), TEST-SUPPORT = *YES
//BIND OUTPUT = *LIB-ELEM(LIB=MYLIB,ELEM=STRING)
%  BND1501 LLM FORMAT: '1'
%  BND1101 BINDER NORMALLY TERMINATED.
%//END
%  CDR9908 : END   C TIME USED = 5.6300 SEC
%  CCM0998 CPU TIME USED: 5.7149
/LOAD-PROG *MOD(LIB=MYLIB,ELEM=STRING, -
/RUN-MOD=ADVANCED,PROGRAM-MODE=*ANY),TEST-OPT=*AID
%  BLS0523 ELEMENT 'STRING', VERSION '@' FROM LIBRARY ':2OS2:$TEST.MYLIB' IN PROCESS
%  BLS0524 LLM '$LIB-ELEM$MYLIB$STRING$', VERSION ' ' OF '2015-01-26 10:46:33' LOADED
/%r
Hello C++ World
Goodbye C World
%  CCM0998 CPU TIME USED: 0.0026 SECONDS
```

9.3 C-Anwendungsbeispiel unter POSIX

Ein C-Anwendungsbeispiel unter Posix finden Sie im Handbuch „POSIX-Kommandos“ [11].

10 Anhang

10.1 Gegenüberstellung: Testen älterer Objekte / Objekte von C++ ab V3.0

Für C++-Programme, die mit einer älteren Version des C/C++-Compilers bis 2.2C übersetzt wurden, gelten beim Testen mit AID ab V3.4B aufgrund der unterschiedlichen LSD-Struktur weiterhin die im Handbuch zu AID V2.1A „Testen von C/C++-Programmen“ beschriebenen Regeln. Abweichungen ergeben sich beim Ansprechen von Daten und Funktionen aus Klassen sowie beim Übertragen von abgeleiteten Klassen in Basisklassen.

mit C/C++ bis V2.2C übersetzte Objekte	**mit C/C++ ab V3.0 übersetzte Objekte**
Dynamische Daten einer Klasse können aus einer dynamischen Member-Funktion heraus nur über den `this`-Zeiger angesprochen werden.	Dynamische Daten einer Klasse können aus einer dynamischen Member-Funktion heraus nach denselben Scoperegeln angesprochen werden, die auch in C++ gelten.
Die Klassen-Qualifikation einer Member-Funktion geht in die Funktionsbezeichnung mit ein: `n'`*klasse*`::[...]`*funktion*`(`*signatur*`)'`	Die Klassen-Qualifikation einer Member-Funktion steht vor der Funktionsbezeichnung: *klasse*`::[...]n'`*funktion*`([`*signatur*`])'`
Die Signatur in einer Funktionsbezeichnung muss stets angegeben werden, auch wenn sie `void` ist.	Ist die Signatur in einer Funktionsbezeichnung `void`, so darf sie nicht angegeben werden. Wie in C++ müssen jedoch die beiden Klammern geschrieben werden.
Die Zuweisung „Zeiger auf Basisklasse = Zeiger auf abgeleitete Klasse“ kann mit dem %SET-Kommando nicht ausgeführt werden. Es müssen vielmehr alle dynamischen Daten einzeln übertragen werden.	Die Zuweisung „Zeiger auf Basisklasse = Zeiger auf abgeleitete Klasse“ kann mit dem %SET-Kommando nachvollzogen werden.

Tabelle 10: Unterschiede beim Testen von älteren Objekten zu Objekten des C++-Compilers ab V3.0

Fachwörter

/390

Prozessorbezeichnung im System 7.500 (CISC)

Ablaufüberwachung

%CONTROL*n*, %INSERT und %ON sind Kommandos zur Ablaufüberwachung. Kommt der Programmablauf an eine Anweisung der gewählten Gruppe (%CONTROL*n*) oder an die vereinbarte Programmadresse (%INSERT) oder tritt das ausgewählte Ereignis ein (%ON), wird der Programmablauf unterbrochen, und AID bearbeitet das vereinbarte Subkommando.

Ablaufverfolgung

%TRACE ist das Kommando zur Ablaufverfolgung. Mit ihm vereinbaren Sie, welche und wieviele Anweisungen (symbolisches Testen) oder Befehle (Maschinencode-Ebene) protokolliert werden sollen. Den Programmablauf können Sie am Bildschirm mitverfolgen, wenn die Ausgabe nicht mit %OUT %TRACE auf ein anderes Ausgabemedium umgeleitet wurde.

Adressierungsmodus

bestimmt, wie Adressen für die Ausführung der Maschinenbefehle umgesetzt werden sollen. AID übernimmt standardmäßig den Adressierungsmodus des Testobjekts. Das trifft sowohl für die Adressbreite (24- oder 31-Bit) von Programmen (%AMODE), als auch für die Adressierung von Datenräumen (%ASC) zu.
Mit dem Schlüsselwort %AMODE sprechen Sie die Systeminformation Adressbreite an. Sie kann mit %DISPLAY abgefragt und mit dem folgenden Kommando geändert werden: `%MOVE %MODE{24|31} INTO %AMODE.`
Mit dem Schlüsselwort %ASC (access space control mode) sprechen Sie die Systeminformation AR-Modus (access register mode) an. Sie gibt Auskunft darüber, ob Zugriffsregister für die Adressierung von Datenräumen in die Adressumsetzung einbezogen werden. Sie kann mit %DISPLAY abgefragt werden.

Adressoperand

ist ein Operand, mit dem Sie eine Speicherstelle oder einen Speicherbereich adressieren. Sie können virtuelle Adressen, Datennamen, Anweisungsnamen, Source-Referenzen, Schlüsselwörter, komplexe Speicherreferenzen oder eine S-, PROC- oder BLK-Qualifikation angeben. Die Speicherstelle bzw. der Spei-

cherbereich liegen entweder im geladenen Programm oder in einem Speicherabzug in einer Dump-Datei.
Wenn ein Name in Ihrem Programm mehrfach vergeben wurde und somit kein eindeutiger Bezug auf eine Adresse gewährleistet ist, können Sie ihn mit Bereichsqualifikationen oder über eine Strukturqualifikation eindeutig der gewünschten Adresse zuordnen.

Adressversatz

oder Byte-Offset ist in AID eine Operation zur Adressberechnung. Damit können Sie von einer Adresse aus in Byte-Schritten vorwärts- oder zurückgehen.

Änderungsdialog

Mit dem Kommando `%AID CHECK=ALL` schalten Sie den Änderungsdialog ein. Er wird bei der Ausführung von %MOVE oder %SET wirksam. AID fragt im Dialog nach, ob die Änderung des Speicherinhalts durchgeführt werden soll. Wird als Antwort ein `N` eingegeben, unterbleibt die Änderung; wird ein `Y` eingegeben, führt AID die Übertragung aus.

AID-Arbeitsbereich

ist der Adressraum, in dem Sie Speicherreferenzen ohne Angabe einer Basisqualifikation ansprechen können.
Er umfasst den nicht-privilegierten Teil des virtuellen Speichers in Ihrer Task, der vom Programm samt allen konnektierten Subsystemen belegt ist, oder den entsprechenden Bereich in einem Speicherabzug.
In einem Kommando können Sie vom AID-Arbeitsbereich abweichen, indem Sie im Adressoperanden eine Basisqualifikation angeben. Mit dem Kommando %BASE können Sie den AID-Arbeitsbereich vom geladenen Programm in einen Speicherabzug verlegen oder umgekehrt.

AID-Ausgabedateien

sind die Dateien, in die Sie sich die Ausgaben der Kommandos %DISASSEMBLE, %DISPLAY, %HELP, %SDUMP und %TRACE schreiben lassen können. Die Dateien werden in den Ausgabekommandos über ihre Linknamen F0 bis F7 angesprochen (siehe %OUT und %OUTFILE).
In die Datei, die dem Linknamen F6 zugewiesen wurde, werden die REP-Sätze geschrieben (siehe %AID REP=YES und %MOVE).
Es gibt drei Wege, eine Ausgabedatei anzulegen bzw. eine Ausgabedatei zuzuweisen:

1. %OUTFILE-Kommando mit dem Link- und Dateinamen
2. ADD-FILE-LINK-Kommando mit dem Link- und Dateinamen

3. Für einen Linknamen, dem noch kein Dateiname zugewiesen ist, setzt AID einen FILE-Makro mit dem Dateinamen AID.OUTFILE.Fn ab.

Eine AID-Ausgabedatei hat stets das Format FCBTYPE=SAM, RECFORM=V und wird mit OPEN=EXTEND geöffnet.

AID-Eingabedateien

sind Dateien, die AID zur Ausführung von AID-Funktionen benötigt, im Unterschied zu Eingabedateien, die das Programm benutzt. AID verarbeitet nur Platten-Dateien. AID-Eingabedateien sind:

– Dump-Dateien, in denen sich Speicherabzüge befinden (%DUMPFILE)

– PLAM-Bibliotheken, in denen sich Bindemodule (OMs) oder Bindelademodule (LLMs) befinden. Wird die Bibliothek mit %SYMLIB zugewiesen, kann AID die LSD-Sätze nachladen.

AID-Literale

AID stellt Ihnen Zeichen-Literale und numerische Literale zur Verfügung (siehe AID-Basishandbuch, Kapitel „AID-Literale“ [1]):

```
{C'x...x' | 'x...x'C | 'x...x'}   Character-Literal
{X'f...f' | 'f...f'X}             Sedezimal-Literal
{B'b...b' | 'b...b'B}             Binär-Literal
[{±}]n                            Ganzzahl
#'f...f'                          Sedezimalzahl
[{±}]n.m                          Dezimalpunktzahl
[{±}]mantisseE[{±}]exponent       Gleitpunktzahl
```

AID-Standard-Adressinterpretation

Indirekte Adressen, d.h. Adressen vor einem Pointer-Operator, werden im Standardfall entsprechend dem gerade gültigen Adressierungsmodus des Testobjekts interpretiert. Mit %AINT können Sie von der Standard-Adressinterpretation abweichen und festlegen, ob AID bei indirekter Adressierung mit 24-Bit- oder 31-Bit-Adressen arbeiten soll.

AID-Standard-Arbeitsbereich

Das ist der nicht-privilegierte Teil des virtuellen Speichers in Ihrer Task, der vom Programm samt allen konnektierten Subsystemen belegt ist.
Ohne Vereinbarung mit %BASE und ohne Angabe einer Basisqualifikation gilt der AID-Standard-Arbeitsbereich.

Aktuelle Aufrufhierarchie

ist der Stand der Block- bzw. Funktionsverschachtelung an der Unterbrechungsstelle. Sie reicht von dem Block bzw. der Funktion, in der das Programm unterbrochen wurde, über die mittleren Hierarchiestufen (das sind die überge-

ordneten Blöcke oder die Funktionen, in denen sich der entsprechende Funktionsaufruf befindet,) bis zur `main`-Funktion. Die aktuelle Aufrufhierarchie wird mit `%SDUMP %NEST` ausgegeben.
Bei rekursiven Funktionen wird auch jeder Selbstaufruf ausgegeben.

Aktuelles Programm
ist das Programm, das in der Task geladen ist, in der Sie AID-Kommandos eingeben.

Aktuelle Übersetzungseinheit
ist die Übersetzungseinheit, in der das Programm unterbrochen wurde. Die STOP-Meldung gibt ihren Namen aus.

Anweisungsname
bezeichnet einen im Quellprogramm vergebenen Namen, mit dem in AID eine ausführbare Anweisung angesprochen werden kann. Das sind beim Testen von C/C++-Programmen Marken und Funktionsnamen. Hierzu ist in den LSD-Sätzen eine Adresskonstante hinterlegt. Bei Marken enthält sie die Adresse der ersten Anweisung nach der Marke.
Mit dem Funktionsnamen bezeichnen Sie in den Kommandos %DISASSEMBLE und %INSERT die erste ausführbare Anweisung der Funktion; in allen anderen Kommandos sprechen Sie damit die Prologadresse der Funktion an.

Attribute
Jedes Speicherobjekt hat bis zu sechs Attribute:
Adresse, Name (opt), Inhalt, Länge, Speichertyp, Ausgabetyp
Mit Selektoren können Sie auf Adresse, Länge und Speichertyp zugreifen. Über den Namen findet AID in den LSD-Sätzen alle zugehörigen Attribute, um damit zu arbeiten.
Adresskonstanten und Konstanten aus dem Quellprogramm haben nur bis zu fünf Attribute:
Name (opt), Wert, Länge, Speichertyp, Ausgabetyp
Sie haben keine Adresse. Beim Ansprechen einer Konstanten greift AID nicht auf ein Speicherobjekt zu, sondern setzt nur den dafür vorgemerkten Wert ein.

Ausgabetyp
Attribut eines Speicherobjekts, das bestimmt, wie der Speicherinhalt von AID ausgegeben wird. Jedem Speichertyp ist ein Ausgabetyp zugeordnet. Im AID-Basishandbuch, Abschnitt „Allgemeine Speichertypen“ sind die AID-spezifischen Speichertypen mit den zugehörigen Ausgabetypen aufgelistet. Für die in C/C++ verwendeten Datentypen gilt eine entsprechende Zuordnung. Eine Typmodifikation bei %DISPLAY und %SDUMP bewirkt eine Änderung des Ausgabetyps.

Basisqualifikation

ist die Qualifikation, mit der Sie das geladene Programm oder einen Speicherabzug in einer Dump-Datei bezeichnen. Sie wird mit E={VM| D*n*} angegeben. Die Basisqualifikation können Sie global mit %BASE vereinbaren oder im Adressoperanden für eine einzelne Speicherreferenz angeben.

Bereichsgrenzen

Jedem Speicherobjekt ist ein bestimmter Bereich zugeordnet, der bei Datennamen und Schlüsselwörtern durch die Attribute Adresse und Länge festgelegt ist. Bei virtuellen Adressen liegen die Bereichsgrenzen zwischen V'0' und der letzten Adresse des virtuellen Speichers (V'7FFFFFFF').
Für eine CSECT bzw. einen COMMON als Speicherobjekt ergeben sich die Bereichsgrenzen aus Anfangs- und Endadresse der CSECT/des COMMON (siehe AID-Basishandbuch, Abschnitt „maschinennahe Speicherreferenzen“ [1]).

Bereichsqualifikationen

Die S-, PROC-, BLK- und die ::-Qualifikation werden als Bereichsqualifikationen bezeichnet. Die S-Qualifikation bezeichnet eine Übersetzungseinheit und wird verwendet, um den Pfad zu einem Speicherobjekt zu beschreiben, das nicht in der aktuellen Übersetzungseinheit liegt. Die PROC-Qualifikation bezeichnet eine Funktion, die BLK-Qualifikation bezeichnet einen Block. Die ::-Qualifikation für den Superblock geben Sie an, um globale Daten anzusprechen oder um Funktionen zu bezeichnen, die von der aktuellen Unterbrechungsstelle aus nicht sichtbar sind, da ihre Definition erst danach steht.
Die Bereichsqualifikationen werden verwendet, um den Pfad zu einem Speicherobjekt zu beschreiben, das nicht im Gültigkeitsbereich der Unterbrechungsstelle liegt oder das an der Unterbrechungsstelle von einer gleichnamigen Definition lokal verdeckt ist.

Bereichsüberprüfung

AID überprüft bei Adressversatz, Längenmodifikation und bei *empfänger* in einem %MOVE, ob die Bereichsgrenzen der angesprochenen Speicherobjekte überschritten werden und gibt im Fehlerfall eine entsprechende Meldung aus.

Benutzerbereich

ist der Bereich des virtuellen Speichers, der vom geladenen Programm samt allen konnektierten Subsystemen belegt ist. Er entspricht dem Bereich, der durch das Schlüsselwort %CLASS6 bzw. %CLASS6ABOVE und %CLASS6BELOW repräsentiert wird.

Byte-Offset

siehe Adressversatz

Character-Format

Ausgabetyp bei %DISPLAY (siehe AID-Basishandbuch, Kapitel „Allgemeine Speichertypen“) . Wird ein Speicherbereich mit der abschließenden Typmodifikation %C aufbereitet, gibt AID dessen Inhalt in Character-Darstellung aus.

CSECT-Informationen

stehen in der Objekt-Strukturliste.

Datenname

steht für alle Namen, die im Quellprogramm für Daten vergeben wurden. Mit einem Datennamen sprechen Sie Daten beim symbolischen Testen an.
Den Datennamen können Sie bis auf folgende Ausnahmen wie in C/C++ angeben:
Vektorelemente können Sie nur über Index ansprechen, nicht über Zeiger.
Bei Variablen vom Typ `long double` wertet AID nur die ersten 8 Bytes aus.
Variablen vom Typ `char` betrachtet AID im Gegensatz zu C/C++ nicht als arithmetischen Typ (siehe Datentyp). Rechnen können Sie mit ihnen erst nach einer Typmodifikation mit %A (`unsigned char`) oder %F (`signed char`).
Für Strukturen können Sie die Zeigerschreibweise und die Strukturqualifizierung verwenden.
Für Vektoren können Sie nur die Indexschreibweise verwenden.
Für Zeiger können Sie die Indexschreibweise, die Zeigerschreibweise und die Dereferenzierung verwenden.

Datentyp

Gemäß dem im Quellprogramm deklarierten Datentyp ordnet AID allen Daten einen AID-Speichertyp zu:

- Binärstring (≙ %X)
- Character (≙ %C)
- numerisch (≙ %D, %F, %A)

Der Datentyp `char`, der in C/C++ numerisch behandelt wird, ist für AID nicht numerisch. Rechnen können Sie mit Variablen vom Typ `char` erst nach einer Typmodifikation mit %A (`unsigned char`) oder %F (`signed char`). Dagegen werden Daten vom Typ `signed char` und `unsigned char` auch von AID wie Integerwerte mit bzw. ohne Vorzeichen behandelt.
Der zugeordnete Speichertyp bestimmt, wie das Datum von %DISPLAY und %SDUMP ausgegeben, von %SET übertragen bzw. überschrieben und wie es in der Bedingung eines Subkommandos verglichen wird.

Dump-Format

Ausgabetyp bei %DISPLAY, korrespondiert mit dem Speichertyp %X (siehe AID-Basishandbuch, Kapitel „Allgemeine Speichertypen“). Wird ein Speicherbereich mit der abschließenden Typmodifikation %X aufbereitet, gibt AID dessen Inhalt sowohl sedezimal als auch in Character-Darstellung aus.

DVS-Datei

Datei des BS2000-Datenverwaltungssystems.
Es können dies einzelne Dateien sein oder Module, die in PLAM-Bibliotheken abgelegt sind. Zwischen POSIX und BS2000 können mit dem POSIX-Kommando `bs2cp` Dateien kopiert werden (siehe auch UFS-Datei).

Eingabepuffer

AID hat einen internen Eingabepuffer. Reicht er für die Aufnahme der Eingabe eines Kommandos nicht aus, wird das Kommando mit einer Fehlermeldung als zu lang abgewiesen. Sie müssen dann das Kommando oder die Kommandofolge kürzen oder die Funktionen auf mehrere Kommandos aufteilen.

ESD/ESV

External **S**ymbol **D**ictionary bei OMs / **E**xternal **S**ymbols **V**ector bei LLMs ist das Verzeichnis der Externbezüge eines Moduls. Es wird vom Compiler erstellt. Hierin sind unter anderem Informationen über CSECTs, DSECTs und COMMONs enthalten. Der Binder greift auf dieses Verzeichnis zu, wenn er die Objekt-Strukturliste/Externadressbuch erzeugt.

exec()

Bezeichnet eine Funktionsgruppe, zu der die folgenden Funktionen zählen: `execl()`, `execv()`, `execle()`, `execve()`, `execlp()`, `execvp()`.
Ein `exec()`–Aufruf bewirkt, dass das im Aufruf angegebene Programm das aufrufende Programm überlädt.

Externadressbuch

Auf Basis des ESV (External Symbols Vector) erstellt der BINDER das Externadressbuch, wenn seine Erzeugung nicht unterdrückt wird.
Wird das Externadressbuch später mitgeladen, ohne dass auch die LSD mitgeladen wird, so können Sie sich dennoch mit `%SDUMP %NEST` die aktuelle Aufrufhierarchie ausgeben lassen. Statt der Source-Referenzen und der Namen der Übersetzungseinheiten und Funktionen gibt AID absolute Adressen, CSECT-Namen und die vom Compiler generierten Entry-Namen der Funktionen aus (siehe auch Objekt-Strukturliste).

fork()

Systemaufruf, der eine Kopie des Prozesses erzeugt, der den `fork()`–Aufruf enthält. Nach dem `fork()` existiert ein zusätzlicher identischer Prozess im System.

Fork-Task

Durch einen `fork()`–Aufruf erzeugte Task.

globale Einstellungen

AID stellt Ihnen Kommandos zur Verfügung, mit denen Sie das Verhalten von AID Ihren Testerfordernissen anpassen können, die Ihnen die Adressierung erleichtern oder Schreibarbeit ersparen. Die Voreinstellungen gelten während der gesamten Testsitzung, wenn sie nicht explizit geändert werden (siehe %AID, %AINT, %BASE, %OUT und %QUALIFY).

Gültigkeitsbereich

Für lokale Daten beginnt der Gültigkeitsbereich an der Stelle ihrer Definition bis zum Ende des Blocks, der die Definition enthält und umfasst alle darin verschachtelten Blöcke. Steht die Definition am Anfang der Funktion oder handelt es sich um Übergabeparameter, so umfasst der Gültigkeitsbereich die gesamte Funktion. Bei externen Variablen und Funktionen reicht er von der Vereinbarung an bis zum Ende der Übersetzungseinheit. Der Gültigkeitsbereich einer Marke ist die ganze Funktion, in der sie definiert ist.
AID kann Daten, Funktionsnamen oder Marken genau dann ohne Qualifikationen ansprechen, wenn die aktuelle Unterbrechungsstelle im Gültigkeitsbereich des betreffenden Namens liegt. Bei einem Datennamen gilt dies nur, wenn er nicht durch eine gleichnamige Definition lokal verdeckt ist.

Index

Über den Index werden die Elemente eines Vektors angesprochen. Wie in C/C++ können Sie auch bei AID die Indexschreibweise nicht nur für Vektoren, sondern auch für Zeiger verwenden. Der Index kann wie in C/C++ in eckigen Klammern angegeben werden.

Kommandofolge

Mehrere Kommandos werden mit Semikolon (;) zu einer Folge verbunden, die von links nach rechts abgearbeitet wird. Wie im Subkommando darf eine Kommandofolge AID- und BS2000-Kommandos enthalten. In Kommandofolgen nicht zugelassen sind die AID-Kommandos %AID, %ALIAS, %BASE, %DUMPFILE, %HELP, %OUT, %QUALIFY und die im Anhang des AID-Basishandbuchs aufgelisteten BS2000-Kommandos.
Enthält eine Kommandofolge eines der Kommandos zur Ablaufsteuerung, wird die Kommandofolge an der Stelle abgebrochen und das Programm gestartet (%CONTINUE, %RESUME, %TRACE) oder angehalten (%STOP). Nachfolgende Kommandos aus der Kommandofolge werden nicht mehr ausgeführt.

Kommandomodus

Mit Kommandomodus wird in den AID-Handbüchern der EXPERT-Modus der SDF-Kommandosprache bezeichnet. Falls Sie gerade in einem anderen Modus
(`{GUIDANCE={MAXIMUM | MEDIUM | MINIMUM | NO}`) arbeiten, sollten Sie

mit Kommando /MODIFY-SDF-OPTIONS GUIDANCE=EXPERT in den EXPERT-Modus umschalten, wenn Sie AID-Kommandos eingeben wollen. AID-Kommandos verfügen nicht über eine SDF-Syntax:

– Operanden werden nicht über Menüs abgefragt.
– Im Fehlerfall gibt AID eine Fehlermeldung aus, führt aber keinen Korrekturdialog.

Im EXPERT-Modus fordert Sie das System mit „/“ zur Kommandoeingabe auf.

Konstante

Eine Konstante repräsentiert einen Wert, der nicht über eine Adresse im Programmspeicher hinterlegt ist.
Zu den Konstanten gehören die Ergebnisse von Längenselektion, Längenfunktion und Adressselektion sowie die Anweisungsnamen und die Source-Referenzen.
Die Länge einer Speicherreferenz können Sie entweder über den AID-Längenselektor %L(...) oder den Längenoperator sizeof(...), der dem sizeof-Operator der Sprache C/C++ entspricht, ermitteln. sizeof müssen Sie wie in C/C++ mit Kleinbuchstaben eingeben (%AID C=YES oder %AID LOW={ON|ALL} einschalten!).
Entsprechend stehen Ihnen zwei Operatoren zur Verfügung, mit der Sie die Adresse einer Speicherreferenz angeben können: den AID-Adressselektor %@(...) und den Adressoperator &, der Ihnen von C/C++ her vertraut ist.
Eine Adresskonstante repräsentiert eine Adresse. Adresskonstanten sind Anweisungsnamen, Source-Referenzen und das Ergebnis einer Adressselektion mit dem Adressselektor %@(...) oder mit dem Adressoperator &.
Über eine Adresskonstante in Verbindung mit einem Pointer-Operator (->) können Sie auf die entsprechende Speicherstelle zugreifen.

LIFO

Last **I**n **F**irst **O**ut; Treffen an einem Testpunkt (%INSERT) oder bei Auftreten eines Ereignisses (%ON) Anweisungen aus verschiedenen Eingaben zusammen, so werden die zuletzt eingegebenen zuerst abgearbeitet (siehe AID-Basishandbuch, Abschnitt „Ketten“).

LOGON-Task

Task, die mit dem SDF-Kommando /SET-LOGON-PARAMETERS gestartet wird. Der Kommandomodus der LOGON-Task hat höhere Priorität als der Testmodus einer durch fork() erzeugten Task, was zu Problemen beim simultanen Testen von Vater- und Sohn-Task führen kann.

Lokalisierungsinformation

Die **statische** Programmverschachtelung zu der angegebenen Speicherstelle gibt Ihnen AID aus, mit
%DISPLAY %HLLOC(*speicherref*) für die symbolische Ebene

%DISPLAY %LOC(*speicherref*) für die Maschinencode-Ebene.
Im Gegensatz dazu erhalten Sie die **dynamische** Programmverschachtelung, die sogenannte Aufrufhierarchie zur aktuellen Programmunterbrechungsstelle mit %SDUMP %NEST.

LSD

List for **S**ymbolic **D**ebugging ist ein Verzeichnis der im Modul definierten Daten- und Anweisungsnamen. Ebenso sind dort die vom Compiler erzeugten Source-Referenzen hinterlegt. Die LSD-Sätze werden vom Compiler erzeugt. AID holt sich hieraus die Informationen zur symbolischen Adressierung.

Namensraum

umfasst alle Namen, die in den LSD-Sätzen einer Übersetzungseinheit, einer Funktion oder einem Block zugeordnet sind. Er entspricht dem Gültigkeitsbereich in C/C++. Der Namensraum wird mit %SDUMP bei Angabe der entsprechenden Qualifikation angesprochen.

numerische Ausgabetypen

sind %F, %A und %D (siehe AID-Basishandbuch, Kapitel „Allgemeine Speichertypen“ [1]). Wird ein Speicherbereich mit %DISPLAY ausgegeben und mit einem der numerischen Ausgabetypen aufbereitet, so gelten folgende Zuordnungen:

- %F Ganzzahl mit Vorzeichen (entspricht `int` in C/C++)
- %A Ganzzahl ohne Vorzeichen (entspricht `unsigned int` in C/C++); %A wird in AID außerdem zur Interpretation eines Speicherinhalts als Adresse vor einem Pointer-Operator verwendet.
- %D Gleitpunktzahl (entspricht `float` in C/C++)

Objekt-Strukturliste

Wird die Objekt-Strukturliste später mitgeladen, ohne dass auch die LSD mitgeladen wird, so können Sie sich dennoch mit `%SDUMP %NEST` die aktuelle Aufrufhierarchie ausgeben lassen. Statt der Source-Referenzen und der Namen der Übersetzungseinheiten und Funktionen gibt AID absolute Adressen, CSECT-Namen und die vom Compiler generierten Entry-Namen der Funktionen aus (siehe auch Externadressbuch).

Pointer-Operator

heißt die Zeichenfolge ->, die Sie in einem Adressoperanden schreiben, wenn der Inhalt eines Speicherobjekts oder der Wert einer Konstanten zur indirekten Adressierung herangezogen wird (siehe AID-Basdishandbuch, Abschnitt „Indirekte Adressierung“ [1]). Bei indirekter Adressierung wird der Adressierungsmodus berücksichtigt.

POSIX-Shell

Ein portiertes Unix-Systemprogramm, das die Kommunikation zwischen dem Benutzer und dem System übernimmt. Die POSIX-Shell ist ein Kommando-Interpreter. Sie übersetzt die eingegebenen POSIX-Kommandos in eine Sprache, die das System verarbeiten kann.

Programmzustand

AID unterscheidet drei Programmzustände, in denen sich das zu testende Programm befinden kann:

- Das Programm steht.
  %STOP oder K2-Taste unterbrechen ein laufendes Programm. Außerdem wird das Programm unterbrochen, wenn ein %TRACE abgearbeitet ist und der Operand *fortsetzung* auf `S` gesetzt ist. Die Task befindet sich im Kommandomodus. Sie können Kommandos eingeben.
- Das Programm läuft **ohne** Ablaufverfolgung.
  Das Programm wurde mit START-EXECUTABLE-PROGRAM geladen und gestartet oder mit %RESUME gestartet oder fortgesetzt. Ist kein %TRACE vereinbart, kann dazu auch %CONTINUE verwendet werden.
- Das Programm läuft **mit** Ablaufverfolgung.
  %TRACE startet ein Programm oder setzt es fort. Der Programmablauf wird entsprechend der Vereinbarungen im %TRACE protokolliert. %CONTINUE bewirkt dasselbe, wenn noch ein %TRACE aktiv ist.

Prozess

Begriff aus der Unix-Welt, der auch unter POSIX verwendet wird. Ein Prozess entspricht einer Task auf BS2000-Ebene. Mit Prozess wird der Adressraum und das darin ausgeführte Programm sowie die dafür benötigten Betriebsmittel des Systems bezeichnet.
Ein Prozess wird von einem anderen Prozess durch den Aufruf der Funktion `fork()` erzeugt. Der Prozess, der `fork()` aufruft, heißt Vaterprozess (in BS2000 Vater-Task); der neue, durch `fork()` erzeugte Prozess, heißt Sohnprozess (in BS2000 Sohn-Task).

Prozessnummer (pid)

Eine Nummer, die das System vergibt, um einen Prozess eindeutig zu kennzeichnen. AID bildet aus der Prozessnummer (Process Identification/pid) den Prompt, den eine Fork-Task zur Eingabeaufforderung ausgibt.

Qualifikation

Mit einer Qualifikation können Sie ein Speicherobjekt ansprechen, das nicht im AID-Arbeitsbereich liegt oder darin keinen eindeutigen Namen hat.
Die **Basisqualifikation** gibt an, ob das Speicherobjekt im geladenen Pro-

gramm oder in einem Speicherabzug liegt.
Die **S-Qualifikation** gibt an, in welcher Übersetzungseinheit das Speicherobjekt liegt.
Die **PROC-** oder **BLK-Qualifikation** gibt an, in welcher Funktion bzw. in welchem Block das Speicherobjekt liegt. Diese beiden Qualifikationen werden verwendet, um static definierte Datennamen anzusprechen, die in einer Funktion bzw. einem Block außerhalb der aktuellen Aufrufhierarchie liegen oder um Datennamen anzusprechen, die in einer Funktion bzw. einem Block innerhalb der aktuellen Aufrufhierarchie liegen und von einer gleichnamigen Definition an der Unterbrechungsstelle verdeckt werden.
Die **::-Qualifikation** bezeichnet ein globales Datum oder eine Funktion. Globale Daten und Funktionen können mit den vorangestellten beiden Doppelpunkten von jeder Unterbrechungsstelle aus angesprochen werden, auch wenn deren Definition erst nach der Unterbrechungsstelle steht.

Signatur

bezeichnet die in runde Klammern eingeschlossenen Typangaben der Übergabeparameter einer Funktion. Bei Funktionen aus C++-Programmen geht die Signatur in den Funktionsnamen ein. Wegen der Sonderzeichen (Klammern und evtl. Kommas) muss der Name in `n'...'` angegeben werden. Zusätzliche Leerzeichen dürfen nicht eingefügt werden. Ist die Signatur `void`, so werden nur die Klammern geschrieben. Den exakten Funktionsnamen in C++-Standard-Schreibweise können Sie der %SDUMP-Ausgabe entnehmen. Ist die Funktion in einer Klasse definiert, so gibt AID den exakten Funktionsnamen in der %DISPLAY-Ausgabe zu dieser Klasse aus.

Sohn-Task

Durch einen `fork()`–Aufruf erzeugte Task.

Source-Referenz

bezeichnet einen vom Compiler erzeugten Namen, mit dem in AID jede ausführbare Anweisung angesprochen werden kann. Der Name besteht aus der Zeilennummer, der wahlweise eine `FILE`-Nummer vorangehen oder eine zeilenrelative Anweisungsnummer folgen kann: `S'[`*f*`-]`*n*`[:`*a*`]'`. Hierzu ist in den LSD-Sätzen eine Adresskonstante hinterlegt, die die Adresse der Anweisung enthält. Genauer ist es die Adresse des ersten Befehls, der zu der Anweisung erzeugt wurde.

Speicherobjekt

ist eine bestimmte Anzahl von zusammenhängenden Bytes im Speicher. Auf Programmebene sind das die Daten des Programms, sofern ihnen ein Speicherbereich zugewiesen ist, und der Befehlscode. Außerdem gehören alle Register, der Befehlszähler sowie alle anderen Bereiche, die nur über Schlüsselwörter angesprochen werden können, ebenfalls zu den Speicherobjekten.

Keine Speicherobjekte hingegen sind die Anweisungsnamen, Source-Referenzen, die Ergebnisse von Adressselektion, Längenselektion und Längenfunktion und die AID-Literale. Sie repräsentieren einen Wert, der nicht verändert werden kann.

Speicherreferenz

Mit einer Speicherreferenz sprechen Sie ein Speicherobjekt an. Es gibt einfache und komplexe Speicherreferenzen.
Einfache Speicherreferenzen sind virtuelle Adressen, eine abschließende C-/COM-Qualifikation, Schlüsselwörter und Namen, zu denen AID sich die Adresse aus den LSD-Informationen holen kann. Anweisungsnamen und Source-Referenzen sind in den AID-Kommandos %CONTROLn, %DISASSEMBLE, %INSERT und %REMOVE und %TRACE als Speicherreferenz erlaubt, obwohl sie nur Adresskonstanten sind.
Mit **komplexen** Speicherreferenzen geben Sie AID eine Vorschrift an, wie die gewünschte Adresse errechnet werden soll und welcher Typ und welche Länge gelten sollen. Folgende Operationen können in einer komplexen Speicherreferenz vorkommen:

– Adressversatz
– indirekte Adressierung
– Typmodifikation
– Längenmodifikation
– Adressselektion

Speichertyp

ist entweder der Datentyp, der im Quellprogramm festgelegt wurde oder der durch Typmodifikation gewählte. AID kennt die allgemeinen Speichertypen %X, %C, %E, %P, %D, %F und %A und die Speichertyypen zur Interpretation von Maschinenbefehlen %SX und %S (siehe %SET und AID-Basishandbuch, Kapitel „Adressierung in AID“ und „Schlüsselwörter“).

Subkommando

ist ein Operand der Überwachungskommandos %CONTROL*n*, %INSERT oder %ON. Ein Subkommando kann einen Namen, eine Bedingung und einen Kommandoteil enthalten. Der Kommandoteil kann aus einem einzelnen Kommando oder aus einer Kommandofolge bestehen. Er kann AID- und BS2000-Kommandos enthalten. Jedes Subkommando hat einen Durchlaufzähler. Wie eine Ausführungsbedingung formuliert wird, wie Name und Durchlaufzähler vergeben und angesprochen werden, und welche Kommandos innerhalb von Subkommandos nicht erlaubt sind, ist im AID-Basishandbuch, Kapitel „Subkommando“ [1] beschrieben.
Der Kommandoteil des Subkommandos wird dann ausgeführt, wenn die Überwachungsbedingung des entsprechenden Kommandos (*kriterium*, *testpunkt*, *ereignis*) zutrifft und die eventuell definierte Ausführungsbedingung erfüllt ist.

Superblock
bezeichnet den äußersten Block in einer Übersetzungseinheit. In C++ entspricht dies dem globalen Namespace.
Dem Superblock sind alle globalen Daten und alle Funktionen zugeordnet. Namespaces können ausschließlich im Superblock definiert werden.

Task-Familie
Alle Tasks, die durch `fork()` in beliebigen Generationen aus einer Task hervorgegangen sind.

Testmodus
bezeichnet den Zustand einer Task, in dem Sie AID-Kommandos zum Testen eingeben können. In der LOGON-Task ist der Testmodus identisch mit dem BS2000-Kommandomodus. Bei Fork-Tasks übernimmt AID die Abwicklung des Dialogs zwischen Anwender und Task. AID fordert Sie mit der Ausgabe eines Prompts, der aus der Prozessnummer der Fork-Task gebildet wird, auf, Kommandos einzugeben.
Der Testmodus hat gegenüber dem Kommandomodus der LOGON-Task niedrigere Priorität, d.h. die Fork-Task kann das Terminal nicht gleichberechtigt mit der LOGON-Task benutzen.

Übersetzungseinheit
ist der Teil eines C/C++-Programms, der als eine Einheit übersetzt wurde. Sie wird mit der S-Qualifikation angesprochen.

UFS-Datei
Datei des Unix-File-Systems.
Ebenso wie bei Unix-Systemen sind auch unter POSIX die Dateien in hierarchisch organisierten Dateiverzeichnissen abgelegt. Der C/C++-Compiler kann sowohl UFS- als auch DVS-Dateien (siehe dort) verarbeiten. Das AID-Kommando %SYMLIB können Sie dagegen nur auf PLAM-Bibliotheken im BS2000 anwenden.

Unterbrechungsstelle
Die Adresse, an der ein Programm unterbrochen wurde, wird Unterbrechungsstelle genannt. Aus der STOP-Meldung können Sie entnehmen, an welcher Adresse und in welchem Programmteil die Unterbrechungsstelle liegt. Dort wird das Programm fortgesetzt.

Vater-Task
Erste Task in der Hierarchie einer Task-Familie.

Literatur

Die Handbücher finden Sie im Internet unter *http://manuals.ts.fujitsu.com*. Handbücher, die mit einer Bestellnummer angezeigt werden, können Sie auch in gedruckter Form bestellen.

AID

[1] **AID** (BS2000)
Advanced Interactive Debugger
Basishandbuch
Benutzerhandbuch

[2] **AID** (BS2000)
Testen auf Maschinencode-Ebene
Benutzerhandbuch

[3] **AID** (BS2000)
Advanced Interactive Debugger
Testen von COBOL-Programmen
Benutzerhandbuch

[4] **AID** (BS2000)
Advanced Interactive Debugger
Testen von FORTRAN-Programmen
Benutzerhandbuch

[5] **AID** (BS2000)
Advanced Interactive Debugger
Testen unter Posix
Benutzerhandbuch

[6] **AID** (BS2000)
Advanced Interactive Debugger
Testen von ASSEMBH-Programmen
Benutzerhandbuch

[7] **AID** (BS2000)
Advanced Interactive Debugger
Tabellenheft
Benutzerhandbuch

C/C++

[8] **C/C++** (BS2000)
C/C++-Compiler
Benutzerhandbuch

[9] **C/C++** (BS2000)
POSIX-Kommandos des C/C++-Compilers
Benutzerhandbuch

POSIX

[10] **POSIX** (BS2000)
Grundlagen für Anwender und Systemverwalter
Benutzerhandbuch

[11] **POSIX** (BS2000)
Kommandos
Benutzerhandbuch

BS2000

[12] **CRTE** (BS2000)
Common RunTime Environment
Benutzerhandbuch

[13] **BINDER** (BS2000)
Benutzerhandbuch

[14] **BS2000 OSD/BC**
Bindelader-Starter
Benutzerhandbuch

[15] **BS2000 OSD/BC**
Makroaufrufe an den Ablaufteil
Benutzerhandbuch

[16] **BS2000 OSD/BC**
Kommandos
Benutzerhandbuch

[17] **XHCS**
8-bit-Code- und Unicode-Unterstützung im BS2000
Benutzerhandbuch

Stichwörter

T