1 Einleitung

Mit der Dialog-Testhilfe AID V2.2 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.
AID V2.2 kann zum Testen von reinen BS2000-Programmen ab BS2000/OSD-BC V1.0 eingesetzt werden. Das hier beschriebene Testen von POSIX- und Mixed-Mode-Programmen setzt BS2000/OSD-BC V3.0 voraus.

1.1 Konzept des Handbuchs

Diese Ergänzung beschreibt das Testen mit AID unter POSIX. Zusammen mit dem Basishandbuch und dem Handbuch „Testen von C/C++ - Programmen“ enthält diese Ergänzung alle Informationen, die Sie für das Testen von C- oder C++ -Programmen in POSIX-Umgebung benötigen. Zu Beginn jedes Abschnitts werden die Handbücher und die Abschnitte genannt, die von den jeweiligen Änderungen betroffen sind.

Anschließend an die Einleitung enthält das Ergänzungsheft die folgenden Kapitel:

Metasyntax
: Liste der im Handbuch eingesetzten Darstellungsmittel.

Voraussetzungen zum Testen
: Beschreibung der Operanden, die beim Übersetzen, Binden und Laden zur Erzeugung und Weitergabe der LSD an das geladene Programm benötigt werden sowie von Operanden und Optionen, die die Ablauffähigkeit des Programms unter POSIX herstellen.

Erweiterte AID-Kommandos
: Beschreibung der Erweiterungen in den AID-Kommandos %AID, %SHOW und %STOP.

POSIX-Kommando debug
: Beschreibung des Testens von Programmen, die in der POSIX-Shell geladen und gestartet werden.

Besonderheiten beim Testen
: Neben Informationen zur Vererbung des Test-Kontextes und zur Dump-Bearbeitung 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. Das Kapitel enthält ein Anwendungsbeispiel.

Fork-Tasks testen mit AID-FE
: Beschreibung einer komfortablen Möglichkeit, Fork-Tasks zu testen.

Meldungen
: AID-Meldungen, die beim Testen in POSIX-Umgebung auftreten können.

Anhang
: Aufstellung aller aktuell gültigen Handbücher zu AID.

1.2 Konzept der Dokumentation zu AID

Die Dokumentation zu 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 ein Tabellenheft (siehe Seite 49) vor, in dem die Syntax der Kommandos und der Operanden mit kurzen Erläuterungen aufgeführt sind. Außerdem gibt es darin die %SET-Tabellen und eine Gegenüberstellung AID - IDA. Das Handbuch für das Testen auf Maschinencode-Ebene kann statt oder zusätzlich zu einem der sprachspezifischen Handbücher (siehe Seite 47) eingesetzt werden.

AID Basishandbuch

Im Basishandbuch finden Sie einen Überblick über AID und die Beschreibung der Sachverhalte und Operanden, die in allen 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 Meldungen, die in Kommandofolgen unzulässigen BS2000-Kommandos und eine Gegenüberstellung von AID und IDA.

AID Benutzerhandbücher

In den Benutzerhandbüchern finden Sie die Kommandos in alphabetischer Reihenfolge. Alle einfachen Speicherreferenzen sind in diesen Handbüchern beschrieben. Neben dem Handbuch
AID - Testen von C/C++ - Programmen,
auf das sich die vorliegende Ergänzung bezieht, gibt es noch die Benutzerhandbücher

AID - Testen von COBOL-Programmen
AID - Testen von FORTRAN-Programmen
AID - Testen von PL/I-Programmen
AID - Testen von ASSEMBH-Programmen
AID - Testen auf Maschinencode-Ebene

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

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.

1.3 Änderungen zur vorherigen Version

Das AID-Kommando %AID wurde um zwei neue Operanden FORK={OFF | NEXT | ALL} und EXEC={OFF | ON} ergänzt. Ist FORK={NEXT | ALL} gesetzt, dann wird unmittelbar nach einem `fork()`-Aufruf die dadurch entstandene Fork-Task unterbrochen und in den Testmodus versetzt, sodaß Sie wie gewohnt AID-Kommandos zum Testen Ihres Programms eingeben können. Entsprechend wird nach einem `exec()`-Aufruf das neu geladene Programm unterbrochen, wenn Sie %AID EXEC=ON eingeschaltet haben.

In der Ausgabe zu %SHOW %AID wurden die neuen Operanden des Kommandos %AID aufgenommen.

Das AID-Kommando %STOP wurde um zwei neue Operanden T=*tsn* (Task Sequence Number) und PID=*pid* (Process Identification) erweitert. Mit %STOP {T=*tsn* | PID=*pid*} können Sie eine durch `fork()` entstandene Task unterbrechen. AID meldet sich mit der Prozeßnummer (*pid)* der unterbrochenen Task, und Sie können den weiteren Verlauf dieser Task über AID-Kommandos kontrollieren.

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.

halbfett
: Kleinbuchstaben und Sonderzeichen, die unverändert eingegeben werden müssen. Außerdem werden Benutzereingaben in Beispielen halbfett gekennzeichnet.

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.

Im Fließtext eingesetzte Darstellungsmittel

kursiv

Im Fließtext werden Operanden in *kursiven kleinbuchstaben* geschrieben.

Mit diesem Symbol werden die Stellen im Text gekennzeichnet, die Sie besonders beachten sollten.

3 Voraussetzungen zum Testen

Erweiterung im Kapitel „Voraussetzungen zum Testen“ des Handbuchs „AID -Testen von C/C++-Programmen“ (siehe Seite 47).

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 den ersten beiden Abschnitten dieses Kapitels sind die Steueranweisungen für die Erzeugung der LSD durch den C/C++-Compiler auf BS2000- und auf POSIX-Ebene kurz beschrieben. Im AID-Basishandbuch, Kapitel „Voraussetzungen zum Testen mit AID“, finden Sie allgemeine Informationen zu LSD-Sätzen, zum Binden, Laden und Starten.
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.

Wenn das Programm ohne LSD geladen wurde, bietet AID die Möglichkeit, die LSD bei Bedarf nachzuladen. 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 Sie können das gesamte Programm mit LSD aus dem POSIX-Dateisystem mit dem POSIX-Kommando `bs2cp` in eine PLAM-Bibliothek kopieren. Im letzten Abschnitt dieses Kapitels finden Sie das Nachladen von LSD über das AID-Kommando %SYMLIB beschrieben. Dort steht auch eine Kurzfassung der Beschreibung zu `bs2cp`.

3.1 Übersetzen, Binden und Laden in BS2000

Übersetzen

Im START-Kommando für den C/C++-Compiler steuern Sie das Erzeugen der LSD-Informationen mit folgender Option:

`TEST-SUPPORT = {NO | YES}`

NO Bei der Voreinstellung NO erzeugt der Compiler keine LSD-Informationen. Auch ohne diese LSD-Informationen ist eine Rückverfolgung der Aufrufhierarchie (%SDUMP %NEST) möglich, allerdings nur auf Maschinencode-Ebene.

YES Der Compiler erzeugt LSD-Informationen.

Die Erzeugung der LSD ist nur für nicht optimierte Programme möglich. Sollte die Optimierung dennoch eingeschaltet sein (vgl. OPTIMIZATION-Option), setzt der Compiler die Optimierungsstufe auf LOW zurück 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.

Die folgenden beiden Optionen müssen Sie angeben, damit das Programm POSIX-Schnittstellen benutzen kann:

– 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.
`SOURCE-PROPERTIES=PARAMETERS(DEFINE=_OSD_POSIX,...)`

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

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

`RUNTIME-OPTIONS=*PARAMETERS(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 direkt in der Kommandozeile angegeben werden.

Eine vollständige Darstellung der Operanden, die die Übersetzung steuern, finden Sie im C- und C++-Benutzerhandbuch (siehe Seite 50).

Binden, Laden und Starten

Übersetzte Programme binden, laden und starten Sie mit den für alle Sprachen gültigen SDF-Kommandos bzw. TSOSLNK-Anweisungen, die im AID-Basishandbuch, Kapitel „Voraussetzungen zum Testen mit AID“, beschrieben sind. Dort ist auch der jeweilige Parameter beschrieben, mit dem Sie veranlassen, daß die vom Compiler erzeugten LSD-Informationen an den Binder (BINDER oder TSOSLNK) bzw. Dynamischen Bindelader DBL oder an den Statischen Starter (ELDE) weitergereicht werden, damit Sie symbolisch testen können.

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

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

Wenn Sie statisch mit dem BINDER oder mit TSOSLNK binden und die Bibliothek `SYSLNK.CRTE.POSIX` mit einer `INCLUDE-MODULES-` bzw. `INCLUDE-`Anweisung einbinden, ist sichergestellt, daß das Modul aus der „Bindeschalter“-Bibliothek vor den Modulen des Laufzeitsystems gebunden wird.

– BINDER-Anweisung:

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

– TSOSLNK-Anweisung:

  ```
  INCLUDE ,$.SYSLNK.CRTE.POSIX
  ```

Weitergehende Informationen zur gemeinsamen Laufzeitumgebung CRTE finden Sie im Handbuch „CRTE - Common RunTime Environment“ (siehe Seite 52).

3.2 Übersetzen, Binden und Laden unter POSIX

Übersetzen und Binden

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

`cc` Aufruf des Compilers im Kernighan-Ritchie-Sprachmodus

`c89` Aufruf des Compilers im ANSI/ISO-Sprachmodus

`CC` Aufruf des Compilers im C++-Sprachmodus

Der C/C++-Compiler erzeugt LSD-Informationen, wenn Sie die folgende Option angeben:

– in den Kommandos `cc` und `c89`: `-g`

– im Kommando `CC`: `{-g|+g}`
Implizit wird auch die Option `+d` gesetzt, die bewirkt, daß Inline-Funktionen im C++-Quellprogramm nicht expandiert werden.

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

Wenn Sie gleichzeitig mit `{-g|+g}` die Optionen `+e` (Optimierung der Tabellen virtueller C++-Funktionen) oder `-s` (Binden ohne Externadreßbuch und ohne LSD) angeben, werden `+e` und `-s` ignoriert.

Bei eingeschalteter Optimierung mit den Optionen `-O` oder `-F` kann das erzeugte Programm nur eingeschränkt getestet werden:

– Testpunkte können nur an Funktionseingängen gesetzt werden.

– Nur an diesen Testpunkten können Funktionsparameter und globale Daten sicher angezeigt werden.

Im Handbuch „POSIX-Kommandos des C- und des C++-Compilers“ (siehe Seite 51) sind die Kommandos `cc`, `c89` und `CC` ausführlich beschrieben.

Laden und Starten

Um das Programm mit LSD zu laden, verwenden Sie das POSIX-Kommando `debug`. Es ist in Kapitel 5 ausführlich beschrieben. Nach dem Laden gibt AID die Meldung AID0348 (siehe Seite 39) aus, der Sie die Prozeßnummer (pid) des erzeugten Prozesses entnehmen können, und den Prompt des Testmodus. Sie können nun AID-Kommandos zum Testen eingeben und das Programm mit %RESUME 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 ggfs. den Fehler zu beheben und das Programm weiterlaufen zu lassen.

3.3 LSD nachladen

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, diese 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 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 UFS-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 mit `exec()` geladen werden, kann die LSD generell nicht mitgeladen werden. Hier müssen Sie also stets das oben beschriebene Verfahren anwenden, wenn Sie symbolisch testen wollen.

Eine ausführliche Beschreibung des AID-Kommandos %SYMLIB finden Sie in „AID - Testen von C/C++-Programmen“ (siehe Seite 47).

Kurzbeschreibung von 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“ (siehe Seite 51).

Syntax- -

bs2cp[␣**-k**][␣**-h**][␣**-f**]␣[**bs2:**]datei␣[**bs2:**]dateikopie

- -

-k Beim Kopieren wird der Inhalt der Datei konvertiert:

- von ASCII nach EBCDIC, falls *datei* eine Datei des POSIX-Dateisystems ist.
- von EBCDIC nach ASCII, falls *datei* eine BS2000-Datei ist.

Diese Option benötigen Sie nur, wenn Sie Dateien von UNIX-Rechnern kopieren. Programme, die in der POSIX-Shell übersetzt werden, legt der Compiler in EBCDIC ab.

-h Ausgabe der Kommandosyntax mit Erläuterung der Optionen.

-f Falls im BS2000 eine Datei oder ein Bibliothekselement mit dem Namen *dateikopie* bereits existiert, wird ohne Rückfrage überschrieben.

bs2: Die nachfolgende *datei* oder *dateikopie* ist eine BS2000-Datei.

Eine DVS-Datei wird über ihren Namen angesprochen.

Ein Bibliothekselement bezeichnen Sie folgendermaßen:
`'lib(elem[,type[,vers]])'`

- lib Name der PLAM-Bibliothek im BS2000
- elem Name des Elements
- type Typ des Elements. Standardwert ist `S`.
- vers Version des Elements. Standardwert ist `*HIGH`.

datei Name der Datei, die kopiert werden soll.

dateikopie

Dateiname der Kopie

Genau eine der Dateien *datei* oder *dateikopie* muß eine BS2000-Datei sein.

Beispiel

```
$bs2cp prog1 bs2:'test.lib(prog1,1)'
$debug sym_test prog1
%  AID0348 Program stopped due to EXEC event (PID=0000000224)
%0000000224/...
...
%0000000224/%resume
%  AID0348 Program stopped due to EXEC event (PID=0000000224)
%0000000224/%symlib test.lib
...
```

Das Objektmodul `prog1` wird aus dem POSIX-Dateisystem ins BS2000 übertragen und dort in der Bibliothek `TEST.LIB` als LLM mit dem Namen `PROG1` eingetragen. Mit dem POSIX-Kommando `debug` wird das Programm `sym_test` geladen, das einen `exec()`-Aufruf enthält, um Programm `prog1` zu laden. Nach erfolgtem `exec()` wird mit %SYMLIB die Bibliothek angemeldet, die die LSD zu `prog1` enthält. Nun kann auch `prog1` symbolisch getestet werden.

4 Erweiterte AID-Kommandos

4.1 %AID

Erweiterung im Abschnitt „Verwaltungsfunktionen“ des Basishandbuchs und im Kapitel „AID-Kommandos“ der sprachspezifischen Handbücher, des Handbuchs zum Testen auf Maschinencode-Ebene und des Tabellenhefts (siehe Seite 47).

Im %AID-Kommando kann der Operand LOW den zusätzlichen Wert ALL annehmen, der bewirkt, daß auch in der S-Qualifikation Kleinbuchstaben nicht in Großbuchstaben umgesetzt werden. Zwei neue Operanden wurden eingeführt, die das Testen von Fork-Tasks und von Programmen, die durch `exec()` geladen wurden, ermöglichen.

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

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

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

Kommando	Operand
%AID	LOW [= {ON | OFF | ALL}]
	FORK [= {OFF| NEXT| ALL}]
	EXEC [= {OFF| ON}]

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 21) 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 in einem Subkommando stehen.

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 in einer S-Qualifikation beibehalten. 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.

i 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.

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 21) 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 19) 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).

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).

4.2 %SHOW

Erweiterung im Kapitel „AID-Kommandos“ der sprachspezifischen Handbücher, des Handbuchs zum Testen auf Maschinencode-Ebene und des Tabellenhefts (siehe Seite 47).

In der Ausgabe des Kommandos %SHOW %AID wurden die neuen AID-Schalter ergänzt.

Das Kommando ist in Kommandofolgen und in Subkommandos zugelassen.

Beispiel

```
$debug examp
%  AID0348 Program stopped due to EXEC event (PID=0000000891)
%0000000891/%aid low=all
%0000000891/%show %aid
A I D     V02.2A40  OF  1996-01-30
Copyright (C) Siemens Nixdorf Informationssysteme AG 1996
All Rights Reserved

E=VM : %AINT = %MODE31

%AID CHECK      =   NO
%AID REP        =   NO
%AID SYMCHARS   =   STD
%AID OV         =   NO
%AID LOW        =   ALL
%AID DELIM      =   '|'
%AID LANG       =   D
%AID FORK       =   NOT_USED
%AID EXEC       =   ON
```

Nach dem Laden des Programms mit `debug` wurde zunächst die Umsetzung von Klein- in Großbuchstaben auch für Angaben in der S-Qualifikation ausgeschaltet und anschließend mit %SHOW %AID die Auflistung der aktuell gültigen Vereinbarungen mit %AID angefordert. Die Ausgabe des %SHOW-Kommandos zeigt die Voreinstellungen aller Operandenwerte von %AID außer %AID LOW und %AID EXEC. LOW wurde explizit auf ALL gesetzt. EXEC ist in der POSIX-Shell nach dem Laden eines Programms mit `debug` stets eingeschaltet. 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.

4.3 %STOP

Erweiterung im Abschnitt „Verwaltungsfunktionen“ des Basishandbuchs und im Kapitel „AID-Kommandos“ der sprachspezifischen Handbücher, des Handbuchs zum Testen auf Maschinencode-Ebene und des Tabellenhefts (siehe Seite 47).

Das AID-Kommando %STOP wurde um zwei neue Operanden T=*tsn* (Task Sequence Number) und PID=*pid* (Process Identification) erweitert, über die Sie eine durch `fork()` entstandene Task unterbrechen können. AID meldet sich mit der Prozeßnummer (*pid)* der unterbrochenen Task, und Sie können den weiteren Verlauf dieser Task über AID-Kommandos kontrollieren.

Kommando	Operand
%STOP	[{ T=tsn | PID=pid }]

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

Wenn 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 Programm 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.

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.

Beispiel

Programm `exstop` ist ein C-Programm, das einen `fork()`–Aufruf enthält. Nachdem die Fork-Task erzeugt wurde, soll die Vater-Task durch ein %STOP-Kommando angehalten werden:

```
$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.

5 POSIX-Kommando debug

Erweiterung im Handbuch „AID - Testen von C/C++-Programmen“ (siehe Seite 47).

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

```
debug { [␣-e]␣progname[␣argument]... }
      { ␣-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 19), 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 ------------------------------------------------------------- (1)
      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)
%0000000888/%stop pid=889 ------------------------------------------- (2)
%  AID0492 %STOP was sent to fork task (PID=0000000889).
%0000000888/%aid low=all -------------------------------------------- (3)
%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
%0000000889/%aid low=all -------------------------------------------- (4)
%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/...
```

(1) 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.

(2) Auch die Sohn-Task wird unterbrochen. Beide Tasks melden sich in der Folge abwechselnd mit ihren Prompts.

(3) 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.

(4) Dasselbe Verfahren wie unter (3) wird für die Sohn-Task durchgeführt.

6 Besonderheiten beim Testen

Erweiterung im Handbuch „AID - Testen von C/C++-Programmen“ (siehe Seite 47).

6.1 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.

6.2 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. Wesentlich einfacher ist das Testen solcher Programme, wenn Sie AID-FE einsetzen können. Eine Kurzbeschreibung dieser Möglichkeit enthält Kapitel 7.
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 6.3.2 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	OND1		168
F11	930	OND2	110	124, 128
...	...	...	...	...

Tabelle 1: 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/OSD - Benutzer-Kommandos (SDF-Format)“).

6.3 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.

6.3.1 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). 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 (siehe Seite 47) im Abschnitt „Kommandofolgen und Subkommandos“ 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 ------------------------------------------------------------ (1)
%  AID0348 Program stopped due to EXEC event (PID=0000002893)
%0000002893/%on %svc(44) <%trace 1 %instr>
%0000002893/%aid fork=next
%0000002893/%resume ------------------------------------------------------- (2)
ICXSVCTU+D6AA     SVC   44                      2 FCT=POSIX    IR1=01023A40
STOPPED AT V‘EC1OAC‘ = ICXSVCTU + #‘D6AC‘ , END OF TRACE
```

```
%0000002893/%r
ICXSVCTU+71B2     SVC   44                      1 FCT=POSIX    IR1=01023A40
STOPPED AT V‘EBABB4‘ = ICXSVCTU + #‘71B4‘ , END OF TRACE
%0000002893/%r
ICXSVCTU+D2DA     SVC   44                      0 FCT=POSCONIN IR1=01023A00
                                                 PAR=00E4B401 00000000 00000000
STOPPED AT V‘ECOCDC‘ = ICXSVCTU + #‘D2DC‘ , END OF TRACE
%0000002893/%r
ICXSVCTU+9B9A     SVC   44                      1 FCT=POSSPEND IR1=01023BC8
                                                 PAR=00E36301 00000000 00000000
STOPPED AT V‘EBD59C‘ = ICXSVCTU + #‘9B9C‘ , END OF TRACE
%0000002893/%r
ICXSVCTU+93FA     SVC   44                      1 FCT=POSSPMSK IR1=01023BC8
                                                 PAR=00E35F01 00000000 00000000
STOPPED AT V‘EBCDFC‘ = ICXSVCTU + #‘93FC‘ , END OF TRACE
%0000002893/%r
ICXSVCTU+318      SVC   44                      1 FCT=POSFORK  IR1=01023950
                                                 PAR=00E30201 00000000 00000000
STOPPED AT V‘EB3D1A‘ = ICXSVCTU + #‘31A’ , END OF TRACE
%0000002893/ EM  DÜ  ------------------------------------------------- (3)
%  AID0348 Program stopped due to FORK event (PID=0000002897)
%0000002893/ EM  DÜ
%0000002897/%show %base ---------------------------------------------- (4)
%0000002893/ EM  DÜ
%BASE E=VM
TSN: 0J05    TID: 0091017D
%AINT = %MODE31
BS:  V12.0   HW:  CFCS V3
%0000002897/
```

(1) 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.

(2) Die folgenden %RESUME-Kommandos führen das Programm bis zum entscheidenden SVC mit der Nummer 44 aus (`FCT=POSFORK`). Dieser SVC startet die Erzeugung der Fork-Task.

(3) 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.

(4) 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.

6.3.2 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.

6.3.3 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.

6.4 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 „LSD nachladen“ auf Seite 11).
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 ausführlich im Handbuch „Testen von C/C++-Programmen“ 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.

6.5 Anwendungsbeispiel

Das vorliegende Programmbeispiel zeigt einen einfachen Anwendungsfall von `fork()` und `exec()`. Anhand des Ablaufprotokolls können Sie die Vorgehensweise beim Testen unter POSIX an einem konkreten Testverlauf nachvollziehen.
Programm `exfork` erzeugt zunächst eine Fork-Task. Die Vater-Task wartet aufgrund des `wait()`-Aufrufs, bis die Sohn-Task beendet ist. Die Sohn-Task wird durch den `execvp()`-Aufruf in Zeile 21 von einem weiteren Programm (`facul`) überladen. Der Programmname `facul` muß beim Laden des Programms als Parameter an die `main`-Funktion von `exfork` übergeben werden. `facul` berechnet zu einer einzulesenden ganzen Zahl die Fakultät und gibt das Ergebnis am Bildschirm aus.

Zur besseren Lesbarkeit sind Daten-und Funktionsnamen im Fließtext `dicktengleich` dargestellt. In den Ablaufprotokollen sind die Benutzereingaben **fettgedruckt**.

6.5.1 Quelldateien

exfork.c

```
 1  #include <stdio.h>
 2  main (int argc, char* argv[])
 3  {
 4      char* my_name = argv[0];
 5      char* prog = argv[1];
 6      int pid;
 7      if (argc < 2)
 8      {
 9         fprintf (stderr,
10         "usage: %s subprogram [options]\n", my_name);
11         exit(1);
12      }
13      pid = fork();
14      if (pid < 0)
15      {
16         fprintf (stderr, "fork failed\n");
17         exit(1);
18      }
19      if (pid == 0)        /*   child   */
20      {
21         execvp (prog, &argv[1]);
22         fprintf (stderr, "execvp failed\n");
23      }
24      else /*   parent   */
25      {
26         wait ((int*) 0);
27         printf ("\n%s : %s has finished\n",
28                                my_name, prog);
29         exit (0);
30      }
31  }
```

facul.c

```
 1   #include <stdio.h>
 2   int facul(int n)
 3   {
 4      return (n>1 ? n * facul(n-1) : 1);
 5   }
 6
 7   int main(void)
 8   {
 9      unsigned n;
10      printf("\nn? : ");
11      scanf("%d",&n);
12      if (n>16) return 0;
13      printf("%d! : %d\n",n,facul(n));
14      return 0;
15   }
```

6.5.2 Testablauf

1. Schritt

Die Programme `exfork.c` und `facul.c` werden mit dem C-Compiler in der POSIX-Shell übersetzt. Die Option `-g` veranlaßt, daß der Compiler beim Übersetzen LSD erzeugt. Da `facul` über `execvp()`-Aufruf geladen werden soll und dabei prinzipiell keine LSD mitgeladen werden kann, wird das Programm mit `bs2cp` ins BS2000 übertragen. Dies ist notwendig, damit im späteren Testverlauf die LSD über %SYMLIB zugewiesen werden kann.

```
$ c89 -g -o exfork exfork.c
%  NMH1102 MESSAGE OUTPUT FILE ':1OST:$TSOS.SYSMES.BINDER.013', ACCESS=
ISAM, ACTION=ADD
$ c89 -g -o facul facul.c
%  NMH1102 MESSAGE OUTPUT FILE ':1OST:$TSOS.SYSMES.BINDER.013', ACCESS=
ISAM, ACTION=ADD
$ bs2cp facul bs2:'test.lib(facul,1)'
%  NMH1102 MESSAGE OUTPUT FILE ':1OST:$TSOS.SYSMES.LMS.031', ACCESS=ISAM,
ACTION=ADD
```

2. Schritt

Das Programm wird mit dem POSIX-Kommando `debug` geladen und in den Testmodus versetzt. Als Parameter wird an `exfork` der Name des Programms übergeben, das die spätere Fork-Task überlagern soll. Die Vater-Task erhält die Prozeßnummer 5241. Damit AID die Fork-Task unmittelbar nach ihrer Erzeugung anhält, wird mit dem Kommando %AID der entsprechende Schalter gesetzt. Auch die Vater-Task soll vor Programmende anhalten, dazu dient der %INSERT S'28'. Es empfiehlt sich, diesen Testpunkt jetzt zu setzen, da nach dem `fork()` das Programm in der Fork-Task weiterläuft, während die Vater-Task aufgrund des `wait()`–Aufrufs auf die Beendigung der Fork-Task wartet. Sie könnten auch über einen %STOP PID=*pid* die Kontrolle über die Vater-Task zurückerhalten. Die Vater-Task würde sich dann unmittelbar nach der Beendigung der Sohn-Task mit ihrem Prompt melden. Mit dem abschließenden %RESUME-Kommando wird das Programm gestartet. Die Vater-Task wird nun bis zum `wait()`–Aufruf ausgeführt, die Sohn-Task wird unmittelbar nach ihrer Erzeugung angehalten. AID gibt eine entsprechende Meldung und die Prozeßnummer der Sohn-Task aus.

```
$ debug exfork facul
%  AID0348 Program stopped due to EXEC event (PID=0000005241)
%0000005241/%aid fork=next
%0000005241/%insert s'28'
%0000005241/%resume
%  AID0348 Program stopped due to FORK event (PID=0000005242)
```

3. Schritt

In der Sohn-Task werden über %SHOW %AID die aktuell gültigen Einstellungen ausgegeben. Alle Operandenwerte sind zurückgesetzt; da in der Vater-Task %AID FORK=NEXT gesetzt war, ist der Schalter FORK in der Sohn-Task auf NOT_USED gesetzt. Um in dem mit `execvp()` geladenen Programm `facul` testen zu können, wird %AID EXEC=ON eingegeben. %AID LOW=ALL wird benötigt, um in der S-Qualifikation Kleinbuchstaben verwenden zu können. Dann wird versucht, die Variable `pid` auszugeben. `pid` kann aber nicht ohne Qualifikation angesprochen werden, da die Unterbrechungsstelle unmittelbar nach `fork()` im Laufzeitsystem CRTE liegt, was mit %DISPLAY %LOC(*adresse*) verifiziert wird. Mit dem anschließenden %TRACE-Kommando wird das Programm bis zur nächsten Anweisung der Sohn-Task ausgeführt. Hier kann nun `pid` ohne Qualifikation angesprochen werden. Der Inhalt von `pid` ist 0 in der Sohn-Task.
Mit %RESUME wird das Programm wieder gestartet. Der `execvp()`–Aufruf wird ausgeführt, und das damit geladene Programm `facul` geht in den Testmodus und wird angehalten.

```
%0000005242/%show %aid
A I D     V02.2A40  OF  1996-01-30
Copyright (C) Siemens Nixdorf Informationssysteme AG 1996
All Rights Reserved

E=VM : %AINT = %MODE31
%AID CHECK      =   NO
%AID REP        =   NO
%AID SYMCHARS   =   STD
%AID OV         =   NO
%AID LOW        =   OFF
%AID DELIM      =  '|'
%AID LANG       =   D
%AID FORK       =   NOT_USED
%AID EXEC       =   OFF
%0000005242/%aid exec=on
%0000005242/%aid low=all
%0000005242/%display pid
*** TID: 006F01EE *** TSN: 1A4I ******************************************
ABSOLUT: VT00EB0D1AT SOURCE: ICXSVCTU  PROC: ICXSVCTU  ******************
%  AID0378 Symbolic information missing
%0000005242/%display %loc(%pc->)
CURRENT PC: 00EB0D1A    CSECT: ICXSVCTU  ***********************************
V'00EB0D1A' = CONTEXT:$CRTEC@@@02@0@@@
              LMOD :  ICORTSXS
              OMOD :  IC@RTSXS
              CSECT : ICXSVCTU (00EB0A00) + 0000031A
%0000005242/%trace 1 in s=n'exfork.c'
14                     IF
STOPPED AT SRC_REF: 14, SOURCE: exfork.c , PROC: main , END OF TRACE
%0000005242/%display pid
SRC_REF: 14 SOURCE: exfork.c  PROC: main ************************************
pid             =           0
%0000005242/%resume
%  AID0348 Program stopped due to EXEC event (PID=0000005242)
```

4. Schritt

Da das Programm `facul` mit dem `execvp()`–Aufruf geladen wurde, muß die LSD explizit über %SYMLIB zugewiesen werden. Danach wird mit %TRACE 1 bis zum Programmanfang von `facul` vorgerückt. Mit dem Subkommando des %INSERT S'4' soll jeweils der aktuelle Wert von `n` und die Aufrufhierarchie der zugehörigen Rekursionsstufe ausgegeben werden. Mit %RESUME wird das Programm gestartet. `facul` läuft bis Programmende durch; der `printf()`–Aufruf in Zeile 13 von `facul` gibt das richtige Ergebnis aus: der Wert von 4! ist 24.
In der Vater-Task war ganz zu Anfang ein Testpunkt auf S'28' gesetzt worden, der nun aktiv wird. Um die Variable `pid` der Vater-Task ausgeben zu können, muß auch für die Vater-Task die Beachtung der Klein-/Großschreibung eingeschaltet werden. `pid` enthält in der Vater-Task die Prozeßnummer der Sohn-Task, nämlich 5242. Das abschließende %RESUME-Kommando bewirkt, daß die letzten beiden Anweisungen der Vater-Task ausgeführt werden.

```
%0000005242/%symlib test.lib
%0000005242/%trace 1 in s=n'facul.c'
10                      EXT.PROC START    , BLOCK START, CALL
STOPPED AT SRC_REF: 10, SOURCE: facul.c , PROC: main , END OF TRACE
%0000005242/%insert sU4U <%display n;%sdump %nest>
%0000005242/%resume
n? : 4
*** TID: 006F01EE *** TSN: 1A4I ***************************************
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
n               =           4
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 13 SOURCE: facul.c  PROC: main  ***********************************
ABSOLUT: VTE60432T    SOURCE: ICS$MAIO  PROC: _main  ************************
ABSOLUT: V'1002234'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  **********************

n               =           3
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 13 SOURCE: facul.c  PROC: main  ***********************************
ABSOLUT: VTE60432T    SOURCE: ICS$MAIO  PROC: _main  ************************
ABSOLUT: V'1002234'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  **********************

n               =           2
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 13 SOURCE: facul.c  PROC: main  ***********************************
ABSOLUT: VTE60432T    SOURCE: ICS$MAIO  PROC: _main  ************************
ABSOLUT: V'1002234'   SOURCE: IC@MAIN@  PROC: IC@MAIN@  **********************
```

Fortsetzung ...

Fortsetzung ...

```
n               =           1
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 4 SOURCE: facul.c  PROC: facul  ***********************************
SRC_REF: 13 SOURCE: facul.c  PROC: main  ***********************************
ABSOLUT: VTE60432T    SOURCE: ICS$MAIO  PROC: _main  ************************
ABSOLUT: V‘1002234‘   SOURCE: IC@MAIN@  PROC: IC@MAIN@  ***********************

4! : 24

STOPPED AT SRC_REF: 28, SOURCE: exfork.c , BLK: 25
%0000005241/%aid low
%0000005241/%display pid
*** TID: 00EB169E *** TSN: 1A4G ****************************************************
SRC_REF: 28 SOURCE: exfork.c  BLK : 25  ************************************
pid             =        5242
%0000005241/%resume
exfork: facul has finished
```

7 Fork-Tasks testen mit AID-FE

Wenn Sie eine UNIX-Anlage und XVision (Windows) auf PC oder ein UNIX-Terminal zur Verfügung haben, bietet AID-FE eine andere, wesentlich komfortablere Möglichkeit, Fork-Tasks zu testen. AID-FE wird zusammen mit AID ausgeliefert.

AID-FE wird in XVision am PC bzw. an einem X-Terminal unter UNIX aufgerufen. Dieses gibt nach Öffnen des Fensters ein AID-Kommando aus, das Sie im BS2000 eingeben müssen, um die Verbindung aufzubauen. Wenn Sie dieses Kommando in einer Fork-Task eingeben, so kommuniziert das AID dieser Task mit AID-FE, und es wird kein Prompt des Testmodus mehr ausgegeben. Jede Fork-Task kommuniziert mit einem eigenen AID-FE, das nur den Quellcode-Ausschnitt zeigt, in dem sich die aktuelle Position dieser Fork-Task befindet. Sie können jede Fork-Task unabhängig von weiteren, parallel ablaufenden Fork-Tasks testen. Ein-/Ausgaben sind über das Primär- oder das Protokollfenster des jeweiligen AID-FE eindeutig zuzuordnen. Die meisten AID-Kommandos wie %TRACE, %DISPLAY oder %INSERT können Sie über einen Kommandoknopf per Mausklick aufrufen. Ein komplexeres Kommando geben Sie in der gewohnten Form im Kommandoeingabefeld ein, oder Sie hinterlegen es in einem selbstdefinierten Kommandoknopf.

Die vollständige Beschreibung des Testens mit AID-FE finden Sie im Handbuch „AID-FE; Grafische Testoberfläche auf Basis von AID“ (siehe Seite 50).

Beispiel

```
/START-POSIX-SHELL
$debug myprog
%0000002893/%AID UI,PARTNER=D255S258,APPL=AIDG001
```

Nach dem Starten der POSIX-Shell und dem Laden von Programm `myprog` mit dem POSIX-Kommando `debug` wird mit dem Kommando %AID die Verbindung zu AID-FE hergestellt.

8 Meldungen

Erweiterung im Kapitel „Meldungen“ des Basishandbuchs (siehe Seite 47).

```
AID0348    Program stopped due to (&00) event (&01)
AID0348    Programm angehalten wegen (&00)-Ereignis (&01)
```

Bedeutung
Ein Ereignis (&00), dessen Ueberwachung durch einen AID-Schalter (FORK oder EXEC) aktiviert wurde, ist eingetreten, oder es wurde ein %STOP von einer anderen Task der Familie abgesetzt.
In einer Fork-Task wird mit (&01) die PID ausgegeben.

```
AID0373    Stop not allowed for given (&00). (Reason (&01))
AID0373    Stop Kommando fuer angegebene (&00) nicht erlaubt. (Grund (&01))
```

Bedeutung
Die Anweisung ist nicht zugelassen fuer (&00).

Grund -10 : eigene Task
-14 : keine Fork Task
-18 : kein Mitglied der Task-Familie

```
AID0391    Task with (&00) is unknown.
AID0391    Task mit (&00) ist unbekannt.
```

Bedeutung
Die Task mit der angegebenen TSN/PID (&00) wurde nicht gefunden.

```
AID0492    %STOP was sent to fork task ((&00)).
AID0492    %STOP wurde an die Fork-Task mit (&00) geschickt.
```

Bedeutung
Es wurde eine Contingency-Routine fuer die angegebene Task eingehaengt, die bei der naechsten Aktivierung der Task den Debug-Modus veranlasst.

9 Anhang

Liste der aktuell gültigen Dokumentation zu AID V2.2

Titel	Version	Ausgabestand
Testen unter POSIX	AID V2.2	Dezember 1996
AID-FE	AID V2.1	August 1995
Testen auf Maschinencode-Ebene	AID V2.1	April 1995
Testen von C/C++-Programmen	AID V2.1	März 1995
Basishandbuch	AID V2.1	Juli 1994
Testen von COBOL-Programmen	AID V2.1	Juli 1994
Tabellenheft	AID V2.0	Mai 1993
Ergänzungsheft	AID V2.0	April 1993
Testen von PL/I-Programmen	AID V2.0	März 1992
Testen von ASSEMBH-Programmen	AID V2.0	Dezember 1991
Testen von FORTRAN-Programmen	AID V2.0	September 1991

Tabelle 2: AID-Handbücher im Überblick

Zusätzlich gibt es eine Readme-Datei zur Version 2.2 unter dem Namen SYSRME.AID.022.D vom März 1996, in der Sie die Besonderheiten beschrieben finden, die beim Testen von Fortran90-Programmen zu beachten sind. Außerdem enthält die Readme-Datei die Änderungen von Version 2.1A auf Version 2.1B.

Fachwörter

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).

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.

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.

Kommandomodus

genauer: BS2000-Kommandomodus.
Bezeichnet in den AID-Handbüchern den EXPERT-Modus der SDF-Kommandosprache. Im EXPERT-Modus fordert Sie das System mit „/“ zur Kommandoeingabe auf.

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.

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.

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.

Vater-Task

Erste Task in der Hierarchie einer Task-Familie.

Sohn-Task

Durch einen `fork()`–Aufruf erzeugte Task.

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.

Task-Familie

Alle Tasks, die durch `fork()` in beliebigen Generationen aus einer Task hervorgegangen sind.

UFS-Datei

Datei des UNIX-File-Systems.
Ebenso wie bei UNIX 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.

Literatur

AID

AID (BS2000)
Advanced Interactive Debugger
Basishandbuch
Benutzerhandbuch

Zielgruppe
Programmierer im BS2000
Inhalt
– Überblick über AID
– Beschreibung der Sachverhalte und Operanden, die für alle Programmiersprachen gleich sind
– Meldungen
– Gegenüberstellung von AID-IDA
Einsatz
Testen von Programmen im Dialog- und Stapelbetrieb

AID V2.1A (BS2000/OSD)
Advanced Interactive Debugger
Testen von C/C++ - Programmen
Benutzerhandbuch

Zielgruppe
Das Handbuch richtet sich an C/C++-Programmierer.
Inhalt
Das Handbuch enthält die Beschreibung der AID-Kommandos und der C/C++-spezifischen Adreßoperanden zum symbolischen Testen von C/C++-Programmen sowie Anwendungsbeispiele.
Einsatz
Testen von C/C++-Programmen im Dialog- und Stapelbetrieb

AID (BS2000)
Advanced Interactive Debugger
Testen von COBOL-Programmen
Benutzerhandbuch

Zielgruppe
COBOL-Programmierer
Inhalt
– Beschreibung der AID-Kommandos für das symbolische Testen von COBOL-Programmen
– Anwendungsbeispiel
Einsatz
Testen von COBOL-Programmen im Dialog- und Stapelbetrieb

AID (BS2000)
Advanced Interactive Debugger
Testen von FORTRAN-Programmen
Benutzerhandbuch

Zielgruppe
FORTRAN-Programmierer
Inhalt
– Beschreibung der AID-Kommandos für das symbolische Testen von FORTRAN-Programmen
– Anwendungsbeispiel
Einsatz
Testen von FORTRAN-Programmen im Dialog- und Stapelbetrieb

AID (BS2000)
Advanced Interactive Debugger
Testen von PL/I-Programmen
Benutzerhandbuch

Zielgruppe
PL/I-Programmierer
Inhalt
– Beschreibung der AID-Kommandos für das symbolische Testen von PL/I-Programmen
– Anwendungsbeispiel
Einsatz
Testen von PL/I-Programmen im Dialog- und Stapelbetrieb

AID (BS2000/OSD)
Advanced Interactive Debugger
Testen von ASSEMBH-Programmen
Benutzerhandbuch

Zielgruppe
Assembler-Programmierer
Inhalt
– Beschreibung der AID-Kommandos für das symbolische Testen von ASSEMBH-Programmen
– Anwendungsbeispiel
Einsatz
Testen von ASSEMBH-Programmen im Dialog- und Stapelbetrieb

AID (BS2000/OSD)
Advanced Interactive Debugger
Testen auf Maschinencode-Ebene
Benutzerhandbuch

Zielgruppe
Programmierer und Tester
Inhalt
– Beschreibung der AID-Kommandos für das Testen auf Maschinencode-Ebene
– Anwendungsbeispiel
Neu aufgenommen wurden die Kommandos %SHOW und %SDUMP %NEST,Kontext-COMMON-Qualifikation sowie auf ESA-Anlagen für Daten-Räume die ALET/SPID-Qualifikationen. Es gibt zusätzliche Schlüsselwörter.

AID (BS2000)
Advanced Interactive Debugger
Tabellenheft
Benutzerhandbuch

Zielgruppe
Programmierer im BS2000
Inhalt
– Testen von Programmen der Programmiersprachen ASSEMBH, C/C++, COBOL, FORTRAN, PL/I und auf Maschinencode-Ebene
– Kurzfassung der AID-Kommandos und Operanden
– %SET-Tabellen
– Gegenüberstellung von AID und IDA
Einsatz
Testen von Programmen im Dialog- und Stapelbetrieb

AID-FE (SINIX)
Grafische Testoberfläche auf Basis von AID V2.1B (BS2000)
Benutzerhandbuch

Zielgruppe
C-, C++- und COBOL85-Programmierer in einer BS2000-Umgebung mit SINIX-Vernetzung und grafikfähigem Terminal
Inhalt
Beschreibung der Systemvoraussetzungen, der Funktionalität, der Konfigurierung und der Netzschnittstellen von AID-FE, der grafischen Testoberfläche auf Basis von AID. Die Maßnahmen, die zum Vorbereiten einer Testsitzung notwendig sind, werden ausführlich beschrieben. Das Fachwortverzeichnis unterstützt das Einarbeiten in den Umgang mit grafischen Oberflächen in SINIX/Windows-Umgebungen.

C/C++

C (BS2000)
C-Compiler
Benutzerhandbuch

Zielgruppe
C-Anwender im BS2000
Inhalt
- Beschreibung aller Tätigkeiten zum Erzeugen eines ablauffähigen C-Programms: Übersetzen, Binden, Laden, Testen;
- Programmierhinweise und weitergehende Informationen zu: Optimierung, Programmablaufsteuerung, Funktions- und Sprachverknüpfung, Sprachumfang des C-Compilers.

C++ V2.2A (BS2000/OSD)
C++-Compiler
Benutzerhandbuch

Zielgruppe
C- und C++-Anwender im BS2000.
Inhalt
- Beschreibung aller Tätigkeiten zum Erzeugen von ablauffähigen C- und C++-Programmen: Übersetzen, Binden, Laden, Testen;
- Programmierhinweise und weitergehende Informationen zu: Optimierung, Programmablaufsteuerung, Funktions- und Sprachverknüpfung, C- und C++-Sprachumfang des Compilers.

POSIX

POSIX V1.0A (BS2000/OSD)
Grundlagen für Anwender und Systemverwalter
Benutzerhandbuch

Zielgruppe
BS2000-Systemverwalter, POSIX-Verwalter, BS2000-Benutzer, Benutzer von UNIX-/ SINIX-Workstations.
Inhalt
Einführung und Arbeiten mit POSIX; BS2000-Softwareprodukte im Umfeld von POSIX; POSIX installieren und steuern; Dateisysteme verwalten, POSIX-Benutzer verwalten, BS2000-Kommandos für POSIX.

POSIX V1.1A (BS2000/OSD)
Kommandos
Benutzerhandbuch

Zielgruppe
Das Handbuch wendet sich an alle Benutzer der POSIX-Shell.
Inhalt
Dieses Handbuch ist ein Nachschlagewerk. Es beschreibt das Arbeiten mit der POSIX-Shell sowie die Kommandos der POSIX-Shell in alphabetischer Reihenfolge.

C/C++ V2.2A (BS2000/OSD)
POSIX-Kommandos des C- und des C++-Compilers
Benutzerhandbuch

Zielgruppe
C- und C++-Anwender im BS2000/OSD.
Inhalt
- Einführung in die C-/C++-Programmentwicklung in POSIX-Shell-Umgebung.
- Übersetzen und Binden von C- und C++-Programmen mit den POSIX-Kommandos cc, c89 und CC.
- Steuern des globalen C- und C++-Listengenerators mit dem POSIX-Kommando ccxref.

BS2000

CRTE (BS2000)
Common RunTime Environment
Benutzerhandbuch
Zielgruppe
Programmierer und Systemverwalter im BS2000
Inhalt
Beschreibung der gemeinsamen Laufzeitumgebung für COBOL85-, C-, und C++-Objekte sowie für "Fremdsprachenmix":
- Komponenten des CRTE
- Programmkommunikationsschnittstlle ILCS
- Bindebeispiele

BS2000/OSD-BC
Kommandos Band 1, A-L
Benutzerhandbuch

Zielgruppe
Das Handbuch wendet sich sowohl an den nichtprivilegierten Anwender als auch an die Systembetreuung.
Inhalt
Es enthält die Kommandos ADD-... bis LOGOFF (BS2000/OSD-Grundausbau und ausgewählte Produkte) mit der Funktionalität für alle Privilegien. Die Einleitung gibt Hinweise zur Kommandoeingabe.

BS2000/OSD-BC
Kommandos Band 2, M-SG
Benutzerhandbuch
Zielgruppe
Das Handbuch wendet sich sowohl an den nichtprivilegierten Anwender als auch an die Systembetreuung.
Inhalt
Es enthält die Kommandos MODIFY... bis SET... (BS2000/OSD-Grundausbau und ausgewählte Produkte) mit der Funktionalität für alle Privilegien.

Bestellen von Handbüchern

Die aufgeführten Handbücher finden Sie mit ihren Bestellnummern im *Druckschriftenverzeichnis* der Siemens Nixdorf Informationssysteme AG. Neu erschienene Titel finden Sie in den *Druckschriften-Neuerscheinungen*.

Beide Veröffentlichungen erhalten Sie regelmäßig, wenn Sie in den entsprechenden Verteiler aufgenommen sind. Wenden Sie sich bitte hierfür an Ihre zuständige Geschäftsstelle. Dort können Sie auch die Handbücher bestellen.

Stichwörter

Inhalt

AID V2.2 (BS2000/OSD)

Testen unter POSIX

Zielgruppe
Das Handbuch richtet sich an C/C++ - Programmierer.
Inhalt
– Beschreibung des Testens von C/C++-Programmen, die POSIX-Schnittstellen benutzen, bzw. die vollständig in der POSIX-Shell ablaufen.
– Anwendungsbeispiel
– Meldungen

Ausgabe: Dezember 1996

Datei: AID_POS.PDF

Information on this document

On April 1, 2009, Fujitsu became the sole owner of Fujitsu Siemens Computers. This new subsidiary of Fujitsu has been renamed Fujitsu Technology Solutions.

This document from the document archive refers to a product version which was released a considerable time ago or which is no longer marketed.

Please note that all company references and copyrights in this document have been legally transferred to Fujitsu Technology Solutions.

Contact and support addresses will now be offered by Fujitsu Technology Solutions and have the format *...@ts.fujitsu.com*.

The Internet pages of Fujitsu Technology Solutions are available at *http://ts.fujitsu.com/*...
and the user documentation at *http://manuals.ts.fujitsu.com*.

Hinweise zum vorliegenden Dokument

Zum 1. April 2009 ist Fujitsu Siemens Computers in den alleinigen Besitz von Fujitsu übergegangen. Diese neue Tochtergesellschaft von Fujitsu trägt seitdem den Namen Fujitsu Technology Solutions.

Das vorliegende Dokument aus dem Dokumentenarchiv bezieht sich auf eine bereits vor längerer Zeit freigegebene oder nicht mehr im Vertrieb befindliche Produktversion.

Bitte beachten Sie, dass alle Firmenbezüge und Copyrights im vorliegenden Dokument rechtlich auf Fujitsu Technology Solutions übergegangen sind.

Kontakt- und Supportadressen werden nun von Fujitsu Technology Solutions angeboten und haben die Form *...@ts.fujitsu.com*.

Die Internetseiten von Fujitsu Technology Solutions finden Sie unter *http://de.ts.fujitsu.com/*..., und unter *http://manuals.ts.fujitsu.com* finden Sie die Benutzerdokumentation.