| ANWENDER-DOKUMENTATION | Anleitung für den Programmierer | MOS |
|---|---|---|
| 11/87 | Debugger adb | MUTOS 1700 |

Programmtechnische Beschreibung Teil 2

# Anleitung für den Programmierer

# Debugger adb

AC A 7100/7150

VEB Robotron-Projekt Dresden

C 8014-0001-1 M3030

Ausgabe: 11/87

Die Ausarbeitung dieser Dokumentation erfolgte durch ein Kollektiv der Technischen Hochschule Ilmenau im Auftrage des VEB Robotron-Elektronik Dresden Stammbetrieb des VEB Kombinat Robotron.



Im Interesse einer ständigen Weiterentwicklung werden alle Leser gebeten, Hinweise zur Verbesserung der Dokumentation dem Herausgeber mitzuteilen.

Herausgeber:

VEB Robotron-Projekt Dresden
Leningrader Str. 9
Dresden 8010



Kurzreferat

Testwerkzeuge liefern im allgemeinen eine Menge von Informationen über die innere Arbeit von Programmen. Der Debugger adb im Besonderen ist ein sehr komplexes und eigenständiges MUTOS 1700 - Werkzeug, das eine Vielzahl von Anforderungen befriedigen kann. Dieses Dokument soll eine Einführung in die Arbeit mit adb geben. An Beispielen werden verschieden formatierte Ausgaben, Testtechniken für C-Programme und die Auswertung der bei Programmabbrüchen enstehenden Core-Files erläutert. Ausserdem werden weitere Anwendungsmöglichkeiten von adb, wie z.B. das Auslisten von Filesysteminformationen, erläutert.

Inhaltsverzeichnis Seite

# 1. Einleitung

Der Mutos 1700 - Debugger adb ermöglicht es Core-Files auszuwerten, wie sie bei einem Programmabbruch erzeugt werden, ausführbare oder Core-Files in verschiedenen Formaten zu protokollieren, solche Files zu verändern oder Programme mit Unterbrechungspunkten abzuarbeiten. Diese Schrift erläutert an Beispielen die Verwendung von adb. Dabei werden beim Leser die Kenntnis der grundlegenden MUTOS 1700 -Kommandos sowie der Programmiersprache C vorausgesetzt. Während in der adb-Kurzbeschreibung, adb(1), alle Kommandos angeführt sind, soll hier mit einer Untermenge dieser Kommandos die Arbeit mit adb erläutert werden.

# 2. Kurzer überblick

## 2.1. Aufruf von adb

Der adb-Aufruf hat folgenden Aufbau:

```
adb objfile corefile
```

wobei objfile ein ausführbares MUTOS 1700-File und corefile ein MUTOS 1700-Core-File ist. Sehr oft wird der Debugger folgendermaßen aufgerufen:

```
adb a.out core
```

oder einfacher durch

```
adb
```

da a.out bzw. core Standard für objfile bzw. corefile sind. Der File-Name minus (-) bedeutet, daß dieses Argument ausgelassen werden soll, so wie in

```
adb --- cooore
```

Durch adb werden Kommandos bereitgestellt, durch die Speicherplatzinhalte in den angegebenen Files abgefragt werden können. Durch das Kommando ? erfolgt normalerweise ein Zugriff auf Speicherplätze in objfile und durch / wird im Normalfall auf corefile zugegriffen. Die allgemeine Form dieser Kommandos ist:

```
adress ? format

oder

adress / format
```

## 2.2. Aktuelle Adresse

Von adb wird eine aktuelle Adresse geführt, die der aktuellen Zeile beim MUTOS 1700-Editor vergleichbar ist. Diese Adresse kann unter dem Symbol Punkt (.) abgefragt werden. Wird in einem adb-Kommando eine Adresse angegeben, so erhält Punkt (.) deren Wert. Somit erhält durch

```
0126?i
```

Punkt den Wert oktal 126 und der A 7100-Befehl, der auf dieser Adresse beginnt, wird ausgegeben.

```
.,10d
```

bewirkt die Ausgabe von 10 Dezimalzahlen beginnend bei der aktuellen Adresse. Punkt zeigt anschließend auf die Adresse, deren Inhalt zuletzt ausgegeben wurde. In Verbindung mit dem ? oder / Kommando kann die aktuelle Adresse durch Eingabe von Neuer Zeile (newline) erhöht und durch Eingabe von ^ verringert werden.

Adressen können die Form von Ausdrücken haben. Bestandteile von Ausdrücken können ganze Dezimal-, Oktal- und Hexadezimalzahlen oder Symbole des zu testenden C-Programmes sein, die mit den Operatoren +,-,*,% ( Integerdivision), & (bitweises "und"), | (bitweises "inklusiv-oder") und ~ (Negation) kombiniert werden können. (Die gesamte adb-Arithmetik ist eine 32-Bit Arithmetik). Bei der Benutzung symbolischer Adressen eines C-Programms kann sowohl name als auch _name verwendet werden. Beide Formen werden von adb akzeptiert.

# 3. Test von C-Programmen

Für die meisten in den weiteren Abschnitten angeführten Beispiele sind in den Anlagen sowohl die C-Programme, auf die sich diese beziehen, als auch die Reaktionen von adb protokolliert. Daher ist es sicher nützlich, beim weiteren Lesen oft mit den Anlagen zu vergleichen, auch wenn dazu nicht explizit aufgefordert wird.

## 3.1. Auswertung eines Core-Files

Das Programm in Anlage A hat einen Fehler (die Abbruchbedingung bei der Laufanweisung fehlt), der zu einem Programmabbruch durch Speicherschutzverletzung führt. Dabei wird ein Speicherabzug des Programms (Core-File) zum Zeitpunkt des Programmabbruchs erzeugt und in der aktuellen Directory unter dem Namen core gespeichert. Durch die Architektur des A 7100-Prozessors kann der Speicherinhalt (speziell der Stack-Inhalt) leicht überschrieben werden. Dadurch enthält der Speicherabzug keine vernünftigen Werte und eine Auswertung des Core-Files mit dem adb-Kommando ist nicht möglich (siehe dazu Anlage B).
Solche fehlerhaften Programme sollten mit Hilfe des Debuggers dynamisch getestet werden, um die Ursache für Speicherschutzverletzungen zu finden. Dieser Test ist in Anlage C dargestellt. Der Debugger wird durch

```
adb an1
```

aktiviert.
Durch das Kommando

```
main+#39:b
```

wird ein Unterbrechungspunkt auf das Inkrementierungsstatement der 'for'-Anweisung gesetzt. Danach wird das Programm durch den Befehl

```
:r
```

gestartet. Die Abarbeitung wird bei dem Unterbrechungspunkt angehalten und man kann sich mit dem Kommando

```
$e
```

alle globalen Variablen und deren Werte ausgeben. Nun kann man mittels

```
,40:c
```

den Unterbrechungspunkt 40-mal übergehen und sich danach nochmals die Werte von i, sum und vector ansehen. Man stellt fest, daß die Werte 0105757 für i und 0105740 für sum nicht mehr vernünftig sind. Hieraus folgt, daß in der 'for'-Anweisung ein Fehler existiert, der zu unkontrolliertem Anwachsen der Laufvariablen i führt.

Für jedes von adb behandelte File existiert eine Map (Speicherzuordnung). Die

Map für objfile wird durch ? und diejenige für corefile durch / benutzt. Da ein Core-File das Textsegment eines Programmes nicht enthält, müssen Zugriffe auf den Befehlsteil über die Map von objfile, also mit ?, erfolgen. Die aktuellen Daten erhält man aus corefile bei der Verwendung des Kommandos /. Eine Ausnahme bilden ausführbare Files, die kein Textsegment besitzen. Dort befinden sich alle Befehle im Datensegment und sind somit auch im Core-File enthalten (siehe unter 4. Maps)
Um Informationen über die Maps zu erhalten, existiert das Kommando

```
$m
```

Durch dieses Kommando werden beide Maps ausgegeben.
Anhand des Beispiels in Anlage A sollen noch einige Möglichkeiten von adb erläutert werden. Man kann auf lokale Variablen der zuletzt aufgerufenen Funktion durch das Kommando

```
name.lokaleVariable/format
```

zugreifen. Sollte dies nicht möglich sein, so kann man Funktionsparameter und lokale Variablen über das Register bp (den Stackframeheader) erreichen. Der erste Parameter ist durch bp+4, der zweite durch bp+6 u.s.w. zu adressieren. Die lokalen Variablen einer Funktion können, beginnend mit bp-6, absteigend ausgewählt werden. In dem Beispiel in Anlage C entsprechen die Werte von bp+4 und bp+6 den Parameterwerten von argc und argv. Durch

```
*(<bp+6)/o
```

kann man sich die Adresse der ersten Zeichenkette des Parametervektors der Funktion main ausgeben. Diese Zeichenkette kann man nun durch

```
0177712p8
```

erreichen.
Durch

```
.=o
```

wird die aktuelle Adresse (nicht ihr Inhalt) oktal ausgegeben. Die aktuelle Adresse kann also dazu benutzt werden, um sich die Stelle im Programm in Erinnerung zu bringen, auf die man zuletzt Bezug genommen hat.
Es ist auch gestattet, Adressen in Kommandos relativ zu Punkt (.) auszugeben. So wird zum Beispiel durch:

```
.-10/d
```

der Inhalt der Adresse mit dem Wert "aktuelle Adresse - 10" ausgegeben.

### 3.2. Geschachtelte Funktionsaufrufe

Zur Veranschaulichung dieser Problematik ist das Programm in Anlage D gut geeignet, welches als ausführbares File in a.out vorliegen soll. Es werden nach dem Aufruf von main nacheinander immer wieder die Funktionen f, g und_h aufgerufen, bis es durch diese endlosen Funktionsaufrufe zu einem Stack-überlauf kommt, der zum Programmabbruch und dem Abspeichern eines Core-Files führt. Zur Auswertung dieses Core-Files wird durch

```
adb
```

der Debugger aktiviert, der die Namen a.out und core standardmäßig für das ausführbare File und das Core-File benutzt.
Das Kommando

```
$c
```

liefert eine Menge von Backtrace-Eintragungen von f, g und h. In Anlage E ist eine solche Liste zu sehen, deren Ausgabe nach einiger Zeit abgebrochen wurde (durch CTRL/C wird das adb-Kommando abgebrochen und adb wartet auf neue Kommandos).
Für jede der Funktionen f, g und h gibt es einen Zähler, der bei jedem Aufruf der Funktion um 1 erhöht wird.
Durch

```
fcnt/d
gcnt/d
hcnt/d
```

kann man den Stand dieser Zähler für die einzelnen Funktionen erfahren. Den Wert einer automatic-Variablen, zum Beispiel den von x im letzten Aufruf von h, kann man durch

```
h.x/d
```

abfragen. Diese Abfragemöglichkeit existiert in dieser Form nur für die zuletzt gerufene Funktion. Alle anderen Werte können durch

```
$C
```

ausgegeben werden. Durch address$c kann das Stack-Backtrace an einer beliebigen Stelle begonnen werden.
Wie an den Beispielen zu ersehen ist, können die Werte globaler Variablen direkt abgefragt werden, während die Werte lokaler Variablen, außer bei der zuletzt gerufenen Funktion, über das C-Backtrace ermittelt werden können.

## 3.3. Arbeit mit Unterbrechungspunkten

Anlage F zeigt ein Programm, das Tabulatoren in Leerzeichen umwandelt. Dieses Programm soll unter der Steuerung von adb abgearbeitet werden (siehe Anlage G).
Adb wird durch:

```
adb a.out -
```

aufgerufen. Durch folgendes Kommando ist es möglich, Unterbrechungspunkte zu setzen:

```
address:b [ count ] [command ]
```

Die Kommandos

```
settab+#8:b
fopen +#8:b
getch +#8:b
tabpos+#8:b
```

setzen Unterbrechungspunkte auf den Anfang dieser Funktionen. Da der C-Compiler einzelne C-Quelltext-Anweisungen im generierten Code nicht markiert, sind ohne Kenntnis des generierten Codes Unterbrechungspunkte nur am Funktionsanfang sinnvoll. Die verwendeten Adressen haben die Form: symbol+#8. Der Offset 8 ist notwendig, weil die ersten Anweisungen einer jeden Funktion abgearbeitet werden müssen, damit beim C-Backtrace, das den Stack auswertet, der letzte Funktionsaufruf berücksichtigt werden kann. Dazu dient auch der Aufruf der Funktion chkstk, die immer dann aufgerufen wird, wenn in der Funktion lokale Variablen (außer Registervariablen) deklariert sind. Wurde das C-Quellprogramm mit Assemblercodeoptimierung (Option -O) übersetzt, steht die chkstk-Routine am Anfang der Funktion, sonst an deren Ende.
Um eine übersicht über alle gesetzten Unterbrechungspunkte zu erhalten, gibt es das Kommando:

```
$b
```

In der zugehörigen Ausgabe ist ein count -Feld zu sehen. Ein Unterbrechungspunkt wird count -1 mal übergangen, ehe es zum Stop kommt. Das command -Feld kann ein adb-Kommando enthalten, das jedesmal, wenn der Unterbrechungspunkt passiert wird, zur Ausführung kommt.
Wenn man sich die ersten Befehle der Funktion settab ausgeben läßt, ist zu sehen, daß der oben festgelegte Unterbrechungspunkt genau auf den Befehl gesetzt ist, der einem Aufruf der chkstk-Routine folgt. Dieser sowie einige folgende Befehle können durch

```
settab,10?ia
```

ausgegeben werden, wobei durch 10 die Anzahl der Befehle festgelegt wird. Eine andere Möglichkeit wäre:

```
settab,10?i
```

wodurch bei der Ausgabe nur die Startadresse mit ausgegeben wird.
Durch das Kommando ? wird auf Adressen in a.out zugegriffen. Wenn durch ein adb-Kommando ausgegeben wird, so wird die aktuelle Adresse um die Anzahl der Byte erhöht, die für die Erfüllung des Kommandos nötig sind. In unserem Beispiel werden 10 Befehle ausgegeben und die aktuelle Adresse wird um 25 erhöht (Gesamtlänge dieser 10 Befehle).

Soll das Programm unter der Steuerung von adb abgearbeitet werden, geschieht das durch

```
:r
```

Soll z.B. nach dem ersten Programmstop ein Unterbrechungspunkt gelöscht werden, hier der auf den Anfang der Funktion settab gesetzte, kann dies durch

```
settab+#8:d
```

geschehen. Durch die Eingabe von

```
:c
```

kann die Programmabarbeitung nach dem Unterbrechungspunkt fortgesetzt werden.
Nach dem nächsten Programmstop (in diesem Fall beim Unterbrechungspunkt am Anfang von fopen), können durch adb-Kommandos z.B. Teile des Speicherinhaltes ausgegeben werden, etwa mit

```
$c
```

beim Stack-Backtrace oder durch

```
tabs,3/8o
```

drei Zeilen mit je 8 Elementen des Feldes tabs. Zu diesem Zeitpunkt (unmittelbar nach dem Aufruf von fopen) ist im C-Programm die Funktion settab bereits abgearbeitet worden, wobei jedem achten Element von tabs der Wert 1 zugewiesen wurde.

## 3.4. Weitere Möglichkeiten bei der Arbeit mit Unterbrechungspunkten

Zunächst wird die Programmabarbeitung durch

```
:c
```

fortgesetzt (siehe Anlage G). Durch den Unterbrechungspunkt bei getch erfolgt ein Programmstop. Nach

```
:c
```

wird bei der erstmaligen Abarbeitung von getch der zugehörige Puffer ibuf gefüllt. Nach dem nächsten Programmstop am Anfang von getch kann der Anfang des Puffers durch

```
ibuf/20c
```

in Schriftform ausgegeben werden.
Durch Angabe eines Wiederholungsfaktors für das Continue-Kommando (:c) wie in

```
,3:c
```

wird zweimal ein Testpunktstop übergangen und erst beim dritten zu passierenden Testpunkt (wieder bei getch ) erfolgt ein Programmstop. Wird danach durch

```
:c
```

der Programmtest normal fortgesetzt, wird dies wie in allen vorangegangenen Fällen durch

```
a.out:running
```

quittiert.
Nachdem durch die Aufrufe von getch das Wort "This" eingelesen wurde, folgt in der Eingabedatei ein Tabulatorzeichen und somit der Aufruf von tabpos, wo es zum Halt wegen des Unterbrechungspunktes kommt. Bei der Umwandlung des Tabulatorzeichens in Leerzeichen würde es oft zum Aufruf von tabpos kommen. Da aber gezeigt ist, daß tabpos zum richtigen Zeitpunkt aufgerufen wurde, soll dieser Unterbrechungspunkt gelöscht werden. Dies geschieht durch:

```
tabpos+#8:d
```

Die verbleibenden Unterbrechungspunkte sollen neu definiert werden, um bei jeder Passage ein Kommando zur Ausführung zu bringen. Im ersten Fall geschieht dies

durch:

```
settab+#8:b settab,10?ia
```

was bei jeder Passage dieses Punktes die Ausgabe der ersten 10 Befehle von settab zur Folge hat. In einer erneuten Redefinition dieses Unterbrechungspunktes wird dann im Kommandofeld zusätzlich die Ausgabe des Inhaltes von ptab als Oktalzahl angewiesen. Dies geschieht durch

```
settab+#8:b settab,10?ia;<bp+4/o
```

und zwischen diesen Unterbrechungspunktdefinitionen liegt nach Anlage G die Redefinition des Unterbrechungspunktes bei getch +#8. Um eine übersicht über alle gesetzten Unterbrechungspunkte zu bekommen, wird

```
$b
```

ausgeführt.

Wie Anlage G zeigt, wird bei Abarbeitung des Programms unter adb entsprechend der Unterbrechungspunkte die Abarbeitung unterbrochen, und es werden die jeweiligen Kommandos ausgeführt.

An dieser Stelle sei noch darauf hingewiesen, daß beim Kommando

```
:c
```

der Prozeß mit dem Signal fortgesetzt wird, durch das er vorher unterbrochen wurde. Dies geschieht, zum Beispiel, um eine eventuell angeschlossene Trap-Routine abarbeiten zu können. Durch

```
:c0
```

wird dieses Signal nicht übergeben.

Außerdem soll noch darauf hingewiesen werden, daß der Wert von Punkt durch die Abarbeitung eines Programmes unter adb nicht verändert wird (auch nicht durch Kommandos bei Unterbrechungspunkten).

Wenn man das Programm in Anlage D unter der Ausnutzung eben erläuterter adb-Möglichkeiten abarbeitet, läßt sich ohne Zwischenhalt die Abarbeitung der Funktionen f,g und h verfolgen.
Wie in Anlage H zu sehen, wird zunächst

```
adb ex2 -
```

ausgeführt (ex2 soll hier das ausführbare File zum C-Programm in Anlage D enthalten).

Zunächst werden folgende Unterbrechungspunkte gesetzt und der dynamische Programmtest gestartet:

```
h+#8:b hcnt/d; h.hr/
g+#8:b gcnt/d; g.gr/
f+#8:b fcnt/d; f.fr/
:r
```

Jede der angegebenen Variablen soll dezimal protokolliert werden. Dabei muß d

in jeder Kommandofolge nur einmal angegeben werden und bleibt dann eingestellt. Bei der zugehörigen Ausgabe nach Anlage H wird zweierlei sichtbar. Zum einen wird die Richtigkeit der Kommandozeile erst beim dynamischen Programmtest überprüft. Das heißt, alle Fehler werden erst zur Laufzeit festgestellt, wobei es dann zum Programmstop beim entsprechenden Speicherplatz kommt. Zum anderen etwas zur Behandlung von Registervariablen in adb: Adb benutzt die Symboltabelle, um Adressen von Variablen zu bestimmen. Registervariable, wie f.fr, sind aber nicht wie andere Variable im Stack abgespeichert und können deshalb auf diese Art und Weise nicht abgefragt werden. Daher die Meldung "automatic variable not found ".

Eine andere Möglichkeit, sich die Daten in diesem Beispiel ausgeben zu lassen ist es, die Variablen in den Prozeduraufrufen zu verwenden.
Dies geschieht durch:

```
f+#8:b fcnt/d; f.a/; f.b/; f.fi/
g+#8:b gcnt/d; g.p/; g.q/;       g.gi/
h+#8:b hcnt/d; h.x/; h.y/;     h.hi/
```

Danach kann durch

```
:c
```

die Programmabarbeitung fortgesetzt werden.
Der Operator / wird anstelle von ? verwendet, was ein Lesen von Core-File (hier der Testprozeß unter adb) bewirkt.
Wie in Anlage H zu sehen ist, hat die Ausgabe stets dasselbe Format, z.B. wird für die Funktion f stets der Wert und der Name der externen Variablen fcnt ausgegeben, außerdem die Adresse im Stack und die Werte von a, b und fi.

Wenn der Programmtest nicht unterbrochen würde, käme es zum Ausdruck vieler Seiten, ehe die Abarbeitung wegen Stack-überlaufs abgebrochen würde.
Durch die Anweisung:

```
f+#8:b fcnt/d; <bp+4/"="d; <bp+6/"b="d; <bp-6/"fi="d
```

kann die Ausgabe angenehmer gestaltet werden. Wie dies für alle Funktionen möglich ist und welche Gestalt dann die Ausgabe hat, zeigt Anlage H.

## 3.5. Andere Möglichkeiten bei der Arbeit mit Unterbrechungspunkten

- Die übergabe von Argumenten zu einem Programm sowie die Umlegung von Standard-Input und -Output ist durch

  ```
  :r arg1 arg2     ... <infile >outfile
  ```

  möglich.

- Der Test im Einzelschritt ist durch

  ```
  :s
  ```

  möglich. Dieses Kommando ermöglicht auch den Programmstart. Es kommt dann zum Stop nach Ausführung des ersten Befehls.

- Adb ermöglicht es, durch

  address:r

  ein Programm ab einer beliebigen Adresse abzuarbeiten.

- Im count - Feld kann angegeben werden, wieviele Unterbrechungspunkte übergangen werden sollen, bevor es zum ersten Programmstop kommt:

  ,n:r

  oder bei Programmfortsetzung

  ,n:c

- Ein Programm kann durch

  address:c

  an einer beliebigen Adresse fortgesetzt werden.

- Das Programm, daß gerade getestet wird, kann durch

  :k

  abgebrochen werden.

# 4. Maps

MUTOS 1700 unterstützt für ausführbare Files verschiedene Formate. Diese werden benötigt, um dem Lader anzuzeigen wie ein File zu laden ist. Der am häufigsten vorkommende File-Typ ist dabei der der 407-er Files, die zum Beispiel durch cc pgm.c erzeugt werden. 411-er Files werden durch die übersetzung mit dem Kommando cc -i pgm.c erzeugt. Adb arbeitet mit diesen verschiedenen Formaten für objfile und realisiert den Zugriff zu den einzelnen Segmenten über die Maps (Anlagen I und J). Diese Maps können mit

```
$m
```

ausgegeben werden.

Beim 407-er Format sind Befehls- und Datensegment vermischt. Dies macht es für adb unmöglich, Befehle von Daten zu unterscheiden und manche der ausgegebenen symbolischen Adressen können falsch sein; z.B. werden Datenadressen als eine Funktionsadresse+Verschiebung ausgegeben.

Bei 411-er Files (getrennter Befehls- und Datenraum) sind Befehle und Daten getrennt. In diesem Fall werden die Daten durch das Datensegmentregister (ds) auf den Speicher abgebildet. Somit ist die Basisadresse des Datensegments ebenfalls relativ zu Null definiert. Da sich in diesen Fall die Adressen in objfile überlappen ist es hier unumgänglich, für den Zugriff den Operator ?* zu verwenden. Durch diesen Operator wird der Zugriff zum Datensegment von objfile über das zweite Tripel der Map realisiert. (überlappen sich die Segmente nicht, wird auch bei ? über das zweite Map-Segment zugegriffen, wenn die Berechnung der File-Adresse in objfile aus der Kommandoadresse über des erste Map-Segment scheiterte.)

Bei 411-er Files enthält das zugehörige Core-File kein Textsegment. Der erste Teil der Map für corefile verweist immer auf das Datensegment (bei 407-er Files existiert praktisch nur das Datensegment, welches auch die Befehle enthält, weshalb bei diesem File-Typ auch der Textteil enthalten ist) und das zweite Mapsegment verweist immer auf den Stack-Bereich. Da sich diese beiden Teile nie überlappen können, genügt im Normalfall der Zugriff durch /, um Adressen im Kommando auf Core-File-Adressen abzubilden. Nur wenn in der Map selbst Veränderungen ausgeführt werden sollen, ist für den Bezug auf das zweite Core-Map-Segment der Operator /* nötig.

Anlage J zeigt die Ausgabe für die Maps, wenn die 2 Arten von ausführbaren Files für das Programm nach Anlage A erzeugt werden. Dabei werden die Felder b, e und f für die Berechnung der File-Adresse aus der Kommandoadresse wie folgt genutzt:

Der Wert von f1 gibt die Länge des File-Headers am File-Anfang an (bei objfile normalerweise 020 Byte und bei corefile normalerweise 02000 Byte). Der Wert von f2 ist bei objfile der Wert der Verschiebung vom File-Anfang bis zum Beginn des Datenbereichs. Bei 407-er Files mit gemischtem Befehls- und Datensegment ist dieser Wert auch immer gleich der Header-Länge. Bei 411-er Files ist dieser Wert die Summe aus Header-Länge und der Länge des Textsegments. Bei corefile ist f2 stets die Summe aus Header-Länge und Länge des Datensegment (bei 407-er Files

ist im Datensegment auch der Textteil enthalten).

Die Felder b und e geben stets die Anfangs- bzw. Endadresse eines Segmentes an.

Ist in einem Kommando die Adresse A angegeben, so wird die File-Adresse daraus wie folgt berechnet:

```
b1<=A<e1 => file address = (A-b1)+f1
b2<=A<e2 => file address = (A-b2)+f2
```

Der Zugiff auf Speicherplätze kann unter Benutzung einiger durch adb zur Verfügung gestellter Variablen geschehen:

| | |
|---|---|
| b | Basisadresse des Datensegments |
| d | Länge des Datensegments |
| s | Länge des Stack-Segments |
| t | Länge des Textsegments |
| m | Filetyp (407,411) |

In Anlage J sind jeweils die Werte aller Variablen, die ungleich Null sind, mit angegeben (Ausgabe durch $v). Vom Nutzer kann auf diese Variablen wie folgt zugegriffen werden:

```
<b
```

im Adreßfeld bewirkt die Verwendung des Wertes von b als Adresse. Eine Änderung des Inhaltes der Variablen ist z.B. durch

```
04000>b
```

möglich. Hier wurde b der Wert 04000 zugewiesen. Diese Variablen sind besonders dann sehr nützlich, wenn das untersuchte File ein ausführbares oder ein Core-File ist.

Zum Setzen dieser Variablen liest adb den Header von corefile und entnimmt diesem die entsprechenden Werte. Wenn das zweite File im adb-Aufruf kein Core-File ist, werden die Werte dem Header von objfile entnommen.

# 5. Weitere Anwendungen für adb

Es ist möglch bei adb die einzelnen Formate zu kombinieren, um Ausgaben übersichtlich zu gestalten. Im weiteren soll dies an einigen Beispielen erläutert werden.

## 5.1. Formatierte Ausgaben

Die Zeile

```
<b,-1/4o4^8Cn
```

bewirkt die Ausgabe von 4 Oktalzahlen (2 Byte Länge) sowie danach die ASCII-Interpretation dieser 8 Byte aus corefile. Die einzelnen Bestandteile dieses Kommandos haben folgende Bedeutung:

`<b` Die Basisadresse des Datensegments

`<b,-1` Ausgabe von der Basisadresse an bis zum File-Ende. Ein negativer Wert für count bewirkt die Wiederholung eines Kommandos bis zu einem Fehler (hier File-Ende).

Das Format 4o4^8Cn setzt sich aus folgenden Bestandteilen zusammen:

`4o` Ausgabe von 4 Worten als Oktalzahlen

`4^` Zurücksetzen der aktuellen Adresse um 4 Worte (auf den ursprünglichen Wert vor Beginn der Ausgabe)

`8C` Ausgabe von 8 aufeinanderfolgenden ASCII-Zeichen unter Benutzung der Festlegung, daß Zahlen im Bereich 0 bis 037 als @ gefolgt vom entsprechenden Zeichen im Bereich 0100 bis 0137 ausgegeben werden; @ wird als @@ ausgegeben.

`n` Ausgabe eines Zeilenwechsels

Durch das Kommando

```
<b,<d/4o4^8Cn
```

wird dieselbe Ausgabe wie oben realisiert, nur daß die Ausgabe am Ende des Datensegments abgebrochen wird (<d liefert die Länge des Datensegments in Byte).

Die Möglichkeiten der formatierten Ausgabe von adb können nun mit der Eingabe vorbereiteter adb-Kommando-Files kombiniert werden, um häufig wiederkehrende Ausgaben zu realisieren. Der adb-Aufruf hat dann folgende Gestalt:

```
adb a.out core <comfile
```

Ein Beispiel für ein solches Kommando-File ist:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,-1/8ona
```

Das Kommando 120$w setzt die Breite der Ausgabe auf 120 Zeichen pro Zeile (Standard ist 80). Adb versucht Adressen symbolisch auszugeben und zwar in der Form: symbol + offset. Das Kommando 4095$s stellt den Maximalwert für offset als Verschiebung zur nächstgelegenen symbolischen Adresse auf 4095 (Standard ist 255). Das Kommando = ermöglicht es, Zeichenketten unverändert auszugeben. Mit $v wird die Ausgabe aller adb - Variablen mit Werten ungleich Null veranlaßt. Das Kommando 0$s setzt die maximale Verschiebung bei der Verwendung von Symbolen auf 0. Somit wird die symbolische Ausgabe unterdrückt und durch eine Oktalausgabe ersetzt. Das Kommando

```
<b,-1/8ona
```

gibt das ganze File achtspaltig oktal aus. Anlage L zeigt das Ergebnis einiger formatierter Ausgaben für das C - Programm in Anlage K.

## 5.2. Directory - Ausgabe

Eine andere Möglichkeit zur Illustration der adb-Arbeit ist die Ausgabe von Directory-Inhalten, so wie sie in der Directory abgespeichert sind. (Ein Directory-Eintrag besteht aus einer ganzen Zahl, der i-number und 14 ASCII-Zeichen, dem File-Namen.

```
adb test -
="I-num"8t8t"Name"
0,-1?u8t14cn
```

bewirkt die gewünschte Ausgabe. Durch u wird in diesem Beispiel die i-number als vorzeichenlose ganze Zahl ausgegeben. 8t veranlaßt die notwendigen Tabulatorsprünge und 14c ist das Format für den File-Name (14 ASCII-Zeichen).

## 5.3. Konvertieren von Zahlen

Adb kann auch zur Zahlenkonvertierung benutzt werden. Zum Beispiel wird durch

```
072 = odx
```

folgendes ausgegeben:

```
072 58 #3a
```

Es wird also der oktale, dezimale und hexadezimale Wert von 072 ausgegeben. Das Format wird gespeichert, so daß bei der Eingabe weiterer Zahlen ebenfalls die zugehörigen Oktal-, Dezimal- und Hexadezimalwerte ausgegeben werden. Werte von Zeichen können ebenso konvertiert werden:

```
'a' = co
```

erzeugt

```
a 0141
```

Adb kann auch zur Berechnung von Ausdrücken benutzt werden. Allerdings muß man dabei beachten, daß alle zweistelligen Operatoren dieselbe Priorität besitzen.

## 6. Ändern von Files

Durch adb können Files mit Hilfe des w- oder W-Kommandos geändert werden. (Dieses Kommando hat nichts mit dem Write-Kommando des Editors gemein). Oft wird dieses Kommando zusammen mit l oder L verwendet. Beide Kommandos haben folgende Syntax:

```
?l value

bzw.

?w value
```

Durch w können 2 Byte und durch W 4 Byte geändert werden und mit l kann ein Wort im Speicher mit dem Wert value gesucht werden. Bei L wird ein Doppelwort mit diesem Wert gesucht. Um ein File modifizieren zu können, muß adb folgendermaßen aufgerufen werden:

```
adb -w file1file2
```

So kann z.B im Programm in Anlage K durch

```
adb -w an10 -
?l 'Th'
?W 'The'
```

das Wort 'This' in 'The' geändert werden. Durch ?l wird die Suche bei Punkt begonnen und bei Übereinstimmung der Werte abgebrochen.

Häufiger wird allerdings folgende Form benutzt:

```
?l 'Th'; ?s
```

wodurch beim ersten Auftreten von 'Th' die ganze Zeichenkette ausgegeben wird und die Adresse dieser Zeichenkette in Punkt gespeichert ist.

Ein anderes Beispiel wäre der Test eines C-Programms mit einem internen Flag. Durch adb kann dieses Flag gesetzt werden bevor das Programm abgearbeitet wird:

```
adb a.ot -
:s arg1 arg2
flag/w 1
:c
```

Durch :s wird der Prozeß im Einzelschrittbetrieb fortgesetzt oder, wenn noch nicht vorhanden, gestartet. Durch das angegebene w-Kommando wird der Wert von flag im Adreßraum des Unterprozesses geändert.

# 7. Besonderheiten

1. Durch Befehle, die zu Beginn jeder Funktion stehen, werden die Registerinhalte gerettet und die Funktionsargumente auf den Stack gebracht. Setzt man einen Unterbrechungspukt auf den Eintrittspunkt einer Funktion, erfolgt die Unterbrechung vor Abarbeitung dieser Befehle und damit ist der Stackinhalt so, als wäre die Funktion noch nicht gerufen worden.

2. Bei Ausgaben benutzt adb die Text- und Datensymbole von objfile. Dies führt manchmal zu falschen Symbolen für Daten. Deshalb sollte bei Ausgaben von Daten immer / und bei Befehlen ? verwendet werden.

3. Adb kann in der zuletzt gerufenen Funktion keine Registervariablen behandeln.

## Anlage A  Ein fehlerhaftes C-Programm

```
#include   <stdio.h>

int  vector[10]     = {0,1,2,3,4,5,6,7,8,9};
int  i;
int  sum            = 0;

main(argc,argv)
int  argc;
char **argv;
{

     for(i=0; "end condition is missing" ; i++){
                    sum += vector[i];
                    vector[i] = sum;
     }
}
```

## Anlage B  Adb - Ausgabe für das Programm in Anlage A

```
               adb a.out core

               $c

     ()
     c routine not found

               $r

     fl      0                       ip      0start+#0
     cs      0                       ds      0
     ss      0                       es      0
     sp      0                       bp      0
     si      0                       di      0
     ax      0                       bx      0
     cx      0                       dx      0

               $q
```

## Anlage C  Adb - Ausgabe für den dynamischen Test des C-Programms in Anlage A

```
               adb an1 -

               main+#39:b

               :r

     an1: running

     breakpoint      main+#39:      inc     i

               ,40:c

     an1: running

     breakpoint      main+#39:      inc     i

               $e

     environ:    0105707
     Isgadr:         04
     Iscadr:         02
     vector:     0
     i:             0105757
     sum:           0105740
     _stkmax:      0
     errno:          0

               $m

     ? map            `an1'
     b1 = 0               e1 = 0560           f1 = 020
     b2 = 0               e2 = 0560           f1 = 020
     / map            `-'
     b1 = 0               e1 = 0100000000     f1 = 0
     b2 = 0               e2 = 0              f2 = 0

               <bp+4/d

     0177662:  1

               *(<bp+666),2/o

     0177674:  0177706  0

               0177706/s

     0177706:  an1

               .=o

               0177706

               .-10/d
```

```
0177674:  -58
          $q
```

## Anlage D  Geschachtelte Funktionsaufrufe zur C-Backtrace-Illustration

```
int  fcnt,gcnt,hcnt;
h(x,y)
{
     int hi; register int hr;
     hi = x+1;
     hr = x-y+1;
     hcnt++ ;
     hj:
     f(hr,hi);
}

g(p,q)
{
     int gi; register int gr;
     gi = q-p;
     gr = q-p+1;
     gcnt++ ;
     gj:
     h(gr,gi);
}

f(a,b)
{
     int fi; register int fr;
     fi = a+2*b;
     fr = a+b;
     fcnt++ ;
     fj:
     g(fr,fi);
}

main()
{
     f(1,1);
}
```

## Anlage E  Adb - Ausgabe für das Programm in Anlage D

```
               adb

               $c

     _g(04507,011214)
     _f(02,04505)
     _h(04504,04503)
     _g(04505,011210)
     _f(02,04503)
     _h(04502,04501)
     _g(04503,011204)
     _f(02,04501)
     _h(04500,04477)
     _g(04501,011200)

               CTRL/C

     adb

               ,5$C

     _g(04507,011214)
          p:            04507
          q:            011214
          gi:           ?
     _f(02,04505)
          a:            02
          b:            04505
          fi:           011214
          fr:           04507
     _h(04504,04503)
          x:            04504
          y:            04503
          hi:           04505
          hr:           02
     _g(04505,011210)
          p:            04505
          q:            011210
          gi:           04503
          gr:           04504
     _f(02,04503)
          a:            02
          b:            04503
          fi:           011210
          fr:           04505

          fcnt/d

     _fcnt:        1187

          gcnt/d
```

```
_gcnt:        1186
     hcnt/d
_hcnt:        1186
     h.x/d
020552:       2372
     $q
```

## Anlage F  Ein Programm zum Auflösen von Tabs in Leerzeichen

```
#include  <stdio.h>

#define MAXLINE         80
#define YES             1
#define NO              0
#define TABSP           8
#define FIRST           1
#define LATER           0

char input[] = "data";
char ibuf[BUFSIZ];
int  tabs[MAXLINE];

main()
{
     int col, *ptab, flag;
     char c;
     FILE *stream;

     ptab = tabs;
     settab(ptab);  /*Set initial tab stops */
     col = 0;
     if((stream = fopen(input,"r")) == 0) {
          printf("%s : not found\n",input);
          exit(8);
     }
     setbuf(stream,ibuf);
     while((c = getch(stream)) != EOF) {
          switch(c) {
               case '\t':          /* TAB */
                    flag = FIRST;
                    while(tabpos(col,flag) != YES) {
                         flag = LATER;
                         putchar(' ');  /* put BLANK */
                         col++ ;
                    }
                    break;
               case '\n':               /*NEWLINE */
                    putchar('\n');
                    col = 0;
                    break;
               default:
                    putchar(c);
                    col++ ;
          }
     }
}

/* Tabpos return YES if col is a tab stop */
tabpos(col,flag)
int col;
int flag;
{
     if(col > MAXLINE)
```

```
          return(YES);
     else
          if(flag == FIRST) return (NO);
          else return(tabs[col]);
}

/* Settab - Set initial tab stops */
settab(tabp)
int *tabp;
{
     int i;

     for(i = 0; i<= MAXLINE; i++)
          (i%TABSP) ? (tabs[i] = NO) : (tabs[i] = YES);
}

getch(stream)
FILE *stream;
{
     return(getc(stream));
}
```

## Anlage G  Adb - Ausgabe für den dynamischen Test des C-Programms in Anlage F

```
               adb a.out -

               settab+#8:b

               fopen+#8:b

               getch+#8:b

               tabpos+#8:b

               $b

     breakpoints
     count     bkpt      command
     1    _tabpos+#8
     1    _getch+#8
     1    _fopen+#8
     1    _settab+#8

               settab,111000?ia

     _settab:  push bp
     _settab+#1:    mov  bp,sp
     _settab+#3:    push di
     _settab+#4:    push si
     _settab+#5:    jmp  _settab+#48
     _settab+#8:    mov  *-6.(bp),#/0
     _settab+#d:    cmp  *-6.(bp),*/50
     _settab+#11:   jl   _settab+#16
     _settab+#13:   jmp  _settab+#45
     _settab+#16:   mov  ax,*-6.(bp)
     _settab+#19:

               settab,111000?i

     _settab:  push bp
               mov  bp,sp
               push di
               push si
               jmp  _settab+#48
               mov  *-6.(bp),#/0
               cmp  *-6.(bp),*/50
               jl   _settab+#16
               jmp  _settab+#45
               mov  ax,*-6.(bp)

               :r

     a.out: running
     breakpoint     _settab+#8:    mov *-6.(bp),#/0

               settab+#8:d
```

```
          :c

a.out: running
breakpoint     _fopen+#8:     pushax

          $c

_fopen(011300,011306)
_main(01,0177674,0177700)

               tabs,3/8o

_tabs:         01   0    0    0    0    0    0    0
               01   0    0    0    0    0    0    0
               01   0    0    0    0    0    0    0

          :c

a.out: running
breakpoint     _getch+#8:     mov  di,*4.(bp)

          :c

a.out: running
Tbreakpoint    _getch+#8:     mov  di,*4.(bp)

          ibuf/20c

_ibuf:         This           is   atest oft

          ,3:c

a.out: running
hisbreakpoint  _getch+#8:     mov  di,*4.(bp)

          :c

a.out: running
breakpoint     _tabpos+#8:    cmp  *4.(bp),*/50

          tabpos+#8:d

          settab+#8:b settab,111000?ia

          getch+#8,5:b

          settab+#8:b settab,111000?ia; <bp+4/o

          $b

breakpoints
count     bkpt      command
1    _settab+#8     settab,10?ia; <bp+4ö
5    _getch+#8
1    _fopen+#8

          :r
```

```
     a.out: running
     _settab:       push bp
     _settab+#1:    mov  bp,sp
     _settab+#3:    push di
     _settab+#4:    push si
     _settab+#5:    jmp  _settab+#48
     _settab+#8:    int
     _settab+#9:    inc  si
     _settab+#a:    cli
     _settab+#b:    addb (bp)+(si),al
     _settab+#d:    cmp  *-6.(bp),*/50
     _settab+#11
     0177634:  012614
     breakpoint     _settab+#8:    mov  *-6.(bp),#/0

               :c

     a.out: running
     breakpoint     _fopen+#8:     push ax

               :c

     Thisbreakpoint                _getch+#8:movdi,*4.(bp)

               $q
```

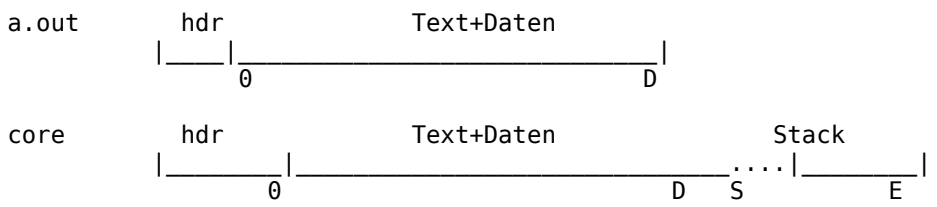
## Anlage H  Adb - Ausgabe für die Arbeit mit Unterbrechungspunkten mit dem Programm nach Anlage D

```
               adb ex2 -

               h+#8:b hcnt/d; h.hi/; h.hr/

               g+#8:b gcnt/d; g.gi/; g.gr/

               f+#8:b fcnt/d; f.fi/; f.fr/

               :r

     ex2: running
     _fcnt:    0
     0177640:  333
     automatic variable not found

               f+#8:b fcnt/d; f.a/; f.b/; f.fi/

               g+#8:b gcnt/d; g.p/; g.q/; g.gi/

               h+#8:b hcnt/d; h.x/; h.y/; h.hi/

               ,6:c

     ex2: running
     _gcnt:    0
     0177632:  2
     0177634:  3
     0177620:  280
     _hcnt:    0
     0177612:  2
     0177614:  1
     0177600:  228
     _fcnt:    1
     0177572:  2
     0177574:  3
     0177560:  333
     _gcnt:    1
     0177552:  5
     0177554:  8
     0177540:  280
     _hcnt:    1
     0177532:  4
     0177534:  3
     0177520:  228
     breakpoint     _h+#8:              movax,*4.(bp)

               CTRL/C

     adb

               f+#8:b fcnt/d; <bp+4/"a="d; <bp+6/"b="d; <bp-6/"fi="d
```

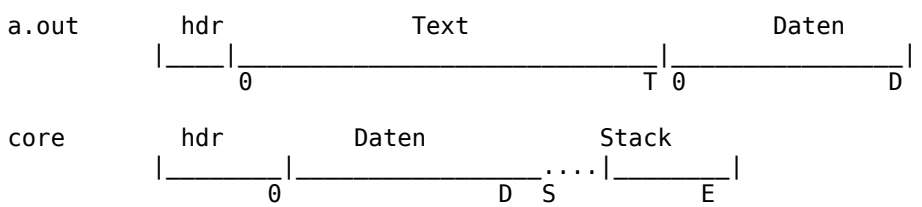
```
          g+#8:b gcnt/d; <bp+4/"p="d; <bp+6/"q="d; <bp-6/"gi="d

          h+#8:b hcnt/d; <bp+4/"x="d; <bp+6/"y="d; <bp-6/"hi="d

          ,7:r
ex2: running
_fcnt:    0
0177652:  a=1
0177654:  b=1
0177640:  fi=333
_gcnt:    0
0177632:  p=2
0177634:  q=3
0177620:  gi=280
_hcnt:    0
0177612:  x=2
0177614:  y=1
0177600:  hi=228
_fcnt:    1
0177572:  a=2
0177574:  b=3
0177560:  fi=333
_gcnt:    1
0177552:  p=5
0177554:  q=8
0177540:  gi=280
_hcnt:    1
0177532:  x=4
0177534:  y=3
0177520:  hi=228
_fcnt:    1
0177512:  a=2
0177514:  b=5
0177500:  fi=333
breakpoint     _f+#8:              movax,*4.(bp)

          CTRL/C

          adb

          $q
```

Anlage I  ADB - Adress - Maps
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

407 Files




411 Files (getrennter Befehls- und Datenraum )




Die folgenden _a_d_b-Variablen werden gesetzt.

| | | 407 | 411 |
|---|---|---|---|
| b | Anfang des Datensegments | 0 | 0 |
| d | Länge des Datensegments | D | D |
| s | Länge des Stack-Segments | S | S |
| t | Länge des Textsegments | 0 | T |
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
               adb ex1-407 ex1core-407

               $m

     ? map       `ex1-407'
     b1 = 0                e1 = 0560           f1 = 020
     b2 = 0                e2 = 0560           f2 = 020
     / map       `ex1core-407'
     b1 = 0                e1 = 04000          f1 = 04000
     b2 = 0174000          e2 = 02000000       f2 = 010000

               $v

     variables
     d = 04000
     m = 0407
     s = 04000

               $q




               adb ex1-411 ex1core-411

               $m

     ? map       `ex1-411'
     b1 = 0                e1 = 0500           f1 = 020
     b2 = 0                e2 = 072            f2 = 0520
     / map       `ex1core-411'
     b1 = 0                e1 = 04000          f1 = 04000
     b2 = 0174000          e2 = 02000000       f2 = 0100000

               $v

     variables
     d = 04000
     m = 0411
     s = 04000
     t = 04000

               $q
```

## Anlage K  Programm zur Illustration von formatierten Ausgaben und ändern von Files

```
char    str1[]  "This is a character string";
int     one     1;
int     number  456;
long    lnum    123456;
float   fpt     1.25;
char    str2[]  "This is the second character string";
main()
{
     while(1)
     one = 2;
}
```

```
               adb ex4n ex4ncore

               <b+200,20/4o4^8Cn

     0310:033070470020166751G@f8@a@b@`im
          0164777020016375103777@i@p@`ig@@`
          037500134000010575lh@g@`8@a@`i@k
          00303063215057374@`@`C@`@mf|^
          05653703030105525053354_]C@`U@klV
          07321301054100311605721З@kv@h@kN@f@k^
          01774040140360105401057323@d@@~@x@a@kS^
          04163066243013400101777775@h#l@a8@@
          01502130141535017735301773536@kP]Ck~k~
          0177353004042400k~@`@`@d@`@`E

     Iscadr:020424000641240715516@b@`@`EThis
          06444002016302014106414З is a ch
          07114106154106256402016Zaracter
          07216306456206355608string@`@`

     _one:020710010161100@b@`H@a@a@`@@b

     _fpt:00100440064124071551@`@` @aThis
          06444002016306416402014З is the
          06256306754306215606144Osecond c
          06055006056207214307114Saracter
          07144007116406715101З7 string@`

               Iscadr+4,3/4o4^8t8cna

     _str1:          064124  071551  064440  020163          This is
     _str1+#8:       020141  064143  071141  061541          a charac
     _str1+#10:      062564  020162  072163  064562          ter stri
     _str1+#18:

               str1,10/2b8t^2cn

     _str1:          0124    0150            Th
                     0151    0163            is
                     040     0151            i
                     0163    040             s
                     0141    040             a
                     0143    0150            ch
                     0141    0162            ar
                     0141    0143            ac
                     0164    0145            te
                     0162    040             r

               fpt/f^4dn

     _fpt:+1.25-32480267082954526912

               $Q
```

## Anlage M  Directory-Ausgabe

```
               adb test -

               =n"I-num"8t8t"Name"  ;  0,-1?u8t14cn

                               I-num           Name
               0:              187             .
                               141             ..
                               189             IMI
                               185             ATA
                               184             OMO
                               183             file

               $Q
```