robotron

Systemhandbuch

MUTOS – 8000

Teil II Abschnitt 3

Testhilfsprogramm adb

Programmtechnische Beschreibung
fuer Buerocomputer A5120.16

robotron

S Y S T E M H A N D B U C H

MUTOS 8000

Teil II

Abschnitt 3 - Testhilfsprogramm adb

VEB Robotron-Buchungsmaschinenwerk
Karl-Marx-Stadt
Stand: 12/85

Die vorliegende Dokumentation entspricht dem Stand 12/85.

Im Interesse einer staendigen Weiterentwicklung werden alle Leser gebeten, ihre Vorschlaege bzw. Hinweise dem

VEB Robotron-Buchungsmaschinenwerk
9010 Karl-Marx-Stadt
Annabergerstr. 93

mitzuteilen.

Fuer das Betriebssystem MUTOS 8000 wurden folgende Dokumentationen erarbeitet:

- Anwendungsbeschreibung MUTOS 8000 Teil I,
  - Abschnitt 1 Kommandos
  - Abschnitt 2, 3 Systemrufe, Bibliotheksfunktionen
  - Abschnitt 4, 5, 7, 8 Special Files, Fileformate, Makropakete, Systemunterstuetzung

- Systemhandbuch MUTOS 8000 Teil II,
  - Abschnitt 1 Kommandointerpreter Shell
  - Abschnitt 2 Editor
  - Abschnitt 3 Testhilfsprogramm adb

- Systemhandbuch MUTOS 8000 Teil III,
  - Abschnitt 1 Textverarbeitung nroff

- Sprachbeschreibungen
  - Assemblersprachbeschreibungen U8000
  - Sprachbeschreibung C-Sprache

Das Sachwortverzeichnis ist als Anlage in der Anwendungsbeschreibung enthalten.

Zur Programmentwicklung und -Testung fuer den Prozessor U8000 werden zusaetzlich folgende Dokumentationen angeboten:

- Monitorsystem MON8000
- CROSS-Software U8000 zum Betriebssystem UDOS
- Assembler U8000ASM
- Binder ZLINK
- Absolutbinder IMAGER
- Preprozessor INCLUDE

Inhaltsverzeichnis

1. Einleitung

Adb ist das MUTOS 8000 Standardtestprogramm. Durch **adb** wird es moeglich, Core-Files auszuwerten, wie sie bei einem Programmabbruch erzeugt werden, ausfuehrbare oder Core-Files in verschiedenen Formaten zu protokollieren, solche Files zu veraendern oder Programme mit Unterbrechungspunkten abzuarbeiten. Diese Schrift erlaeutert die Verwendung von **adb**. Dabei wird beim Leser die Kenntnis der grundlegenden MUTOS 8000 - Kommandos sowie der Programmiersprache C vorausgesetzt. Der vollstaendige Funktionsumfang des Testhilfsprogramms ist im Systemhandbuch Teil I, Abschnitt 1 beschrieben.

2. Allgemeine Erklaerungen

2.1. Aufruf von adb

Der **adb** - Aufruf hat folgenden Aufbau:

adb objfile corefile

wobei objfile ein ausfuehrbares MUTOS 8000 - File ist. Sehr oft wird der debugger folgendermassen aufgerufen:

adb a.out core

oder einfacher durch

adb

da a.out bzw. core Standard fuer objfile bzw. corefile sind. Der Filename minus (-) bedeutet, dass dieses Argument ausgelassen werden soll, so wie in

adb - core

Adb stellt Kommandos bereit, durch die Speicherplatzinhalte in den angegebenen Files abgefragt werden koennen. Durch das Kommando ? erfolgt normalerweise ein Zugriff auf Speicherplaetze in objfile, und durch / wird im Normalfall auf corefile zugegriffen. Die allgemeine Form dieser Kommandos ist:

address ? format

oder

address / format

2.2. Die aktuelle Adresse

Von **adb** wird eine aktuelle Adresse gefuehrt, vergleichbar der aktuellen Zeile beim MUTOS 8000 - Editor. Diese Adresse kann unter dem Symbol Punkt (.) abgefragt werden. Wird in einem **adb** - Kommando eine Adresse angegeben, so erhaelt Punkt (.) deren Wert. Somit erhaelt durch

```
$56
```

Punkt den Wert hexadezimal 56. Der PLZ/ASM-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 newline erhoeht bzw. durch Eingabe von ^ verringert werden.
Adressen koennen die Form von Ausdruecken haben. Bestandteile von Ausdruecken koennen 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 koennen. (Die gesamte Arithmetik in **adb** 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.

2.3. Formate

Fuer die Arbeit mit **adb** stehen eine Reihe von Buchstaben und Zeichen zur Verfuegung, die zur Beschreibung des Ausgabeformates von Kommandos dienen. Wird ein **adb** - Kommando ohne Spezifikation des Ausgabeformates angegeben, so erfolgt die Ausgabe stets in zuletzt verwendeten Format. Die folgende Uebersicht zeigt die am haeufigsten auftretenden Formatkennzeichen:

b	**ein Byte oktal**
c	**ein Byte als ein ISO-Zeichen**
o	**ein Wort oktal**
d	**ein Wort dezimal**
f	**zwei Worte Gleitkomma**
i	**PLZ/ASM-Befehl**
s	**eine durch 0 begrenzte Zeichenkette**
a	**der Wert von Punkt (.)**
u	**ein Wort als vorzeichenlose ganze Zahl**
n	**Ausgabe eines Zeilenwechsels**
x	**ein Wort hexadezimal**
r	**Ausgabe eines Leerzeichens**
^	**Verringern von Punkt (.) ohne Ausgabe**

2.4. Allgemeines Kommandoformat

Die allgemeine Form eines **adb** - Kommandos ist:

address,count command modifier

wodurch Punkt (.) auf address gesetzt und das Kommando command so oft ausgefuehrt wird, wie durch den Parameter count spezifiziert ist. Die folgende Tabelle enthaelt eine Zusammenstellung wichtiger **adb** - Kommandos und deren Bedeutung:

Kommando	Bedeutung
?	Ausgabe des Inhaltes von objfile
/	Ausgabe des Inhaltes von corefile
=	Ausgabe des Wertes von Punkt (.)
:	Unterbrechungspunktsteuerung
x	Verschiedene Kommandos
;	Kommandoseparator
!	Aufruf von Shell

Adb weist Signale ab, so dass die Arbeit nicht mit dem Quit-Signal (CTRL/C) beendet werden kann. Dies ist durch die Kommandos **xq** und **x%** moeglich.

3. Der Test von C-Programmen

3.1. Auswertung eines Core-Files

Es werden zunaechst einige einfache Kommandos vorgestellt, um ein Core-File auszuwerten, dass beim Abbruch des Programmes a.out (z. B. durch Ueberschreiben des Programmcodes) erzeugt und im aktuellen Directory abgelegt wurde.

Adb wird durch

adb a.out core

aktiviert.

Durch das Kommando

xc

wird zunaechst ein C-Backtrace der bis zum Programmabbruch aufgerufenen Unterprogramme ausgegeben.

Das Kommando

$C

liefert einen C-Backtrace und eine Interpretation aller lokalen Variablen.

Durch das Kommando

$r

erhaelt man die Registerinhalte einschliesslich des Programm-Counters sowie eine Interpretation des Befehls, auf welchen er zeigt.

Anschliessend erfolgt durch

$e

die Ausgabe der Werte aller globalen Variablen.

Fuer jedes von **adb** behandelte File exisiert eine Map (Speicherzuordnung). Die Map fuer objfile kann durch ? und diejenige fuer corefile durch / erreicht werden. Es ist guenstig, beim Programmtest Befehle durch ? und Daten durch / abzufragen. Um Informationen ueber die Maps zu erhalten, existiert das Kommando

$m

Durch dieses Kommando werden beide Maps ausgegeben.

Zur Darstellung der Arbeit mit Variablen nehmen wir an, dass ein Programm getestet werden soll, dessen main-Funktion die formalen Parameter argc und argv uebergeben werden.
Das wurde im C-Programm mit folgendem Funktionsaufruf und Vereinbarungen geloest:

```
main (argc,argv)
 int  argc;
 char **argv;
    .
    .
```

Beim Test wird durch das Kommando

main.argc/d

der Wert von argc in main dem File core entnommen und als Dezimalzahl ausgegeben. Er gibt die Zahl der an die main-Funktion uebergebenen Parameter an.

Mit dem Kommando

***main.argv,2/x**

wird die Hexadezimalausgabe von zwei Worten ab der Adresse, auf die main.argv verweist, veranlasst. Die ausgegebenen Werte bilden den Parametervektor der Funktion main und verweisen auf deren Argumente (Name des ausgefuehrten Files, an das Programm uebergebene Parameter). Sollen nun die Argumente selbst ausgegeben werden, so ist dies folgendermassen moeglich:

$aaaa/s

Beide Argumente sind als mit Null begrenzte Zeichenketten unter den vorher ermittelten Adressen $aaaa gespeichert.

Wenn nach dem Kommandoende

$aaaa/s

sofort

^/s

folgt, bewirkt dieses ebenfalls die Ausgabe der bei $aaaa beginnenden Zeichenkette. Dabei zeigt ^ die Verwendung der zuletzt angegebenen Adresse als Anfang der auszugebenden Zeichenkette an.

Durch

.=x

wird die aktuelle Adresse (nicht ihr Inhalt) hexadezimal ausgegeben. Die aktuelle Adresse kann also dazu benutzt werden, sich die Stelle im Programm in Erinnerung zu bringen, auf die man im Test 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. Arbeit mit Unterbrechungspunkten

Anlage 1 zeigt ein Programm, das Tabulatoren in Leerzeichen umwandelt. Dieses Programm soll unter der Steuerung von **adb** abgearbeitet werden (siehe Anlage 2). Der zu manipulierende Text ist im file data abgelegt (in unserem Beispiel die Zeichenkette 'This is a test of ex1').
Adb wird durch:

```
adb a.out -
```

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

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

Die Kommandos

```
settab+4:b
fopen +4:b
getch +4:b
tabpos+4:b
```

setzen Unterbrechungspunkte auf den Anfang dieser Funktionen. Da der C-Compiler einzelne Anweisungen im generierten Code nicht markiert, sind ohne Kenntnis des generierten Codes Unterbrechungspunkte nur am Funktionsanfang sinnvoll. Die verwendeten Adressen haben die Form: **symbol+4** , so wie sie im C-Backtrace erscheinen, da die erste Anweisung einer jeden Funktion im Aufruf der C-Save-Routine (csv) ist. Ausserdem ist zu beachten, dass Bibliotheksroutinen verwendet werden.
Um eine Uebersicht ueber alle gesetzten Unterbrechungspunkte zu erhalten, gibt es das Kommando:

```
b
```

In der zugehoerigen Ausgabe ist ein count-Feld zu sehen. Ein Unterbrechungspunkt wird count-1 mal uebergangen, ehe es zum Stop kommt. Das command-Feld kann ein **adb** Kommando enthalten, das jedesmal, wenn der Unterbrechungspunkt passiert wird, ausgefuehrt wird. Laesst man sich die ersten Befehle der Funktion settab ausgeben, ist zu sehen, dass der oben festgelegte Unterbrechungspunkt genau auf den Befehl nach dem Aufruf der C-Save-Routine gesetzt ist. Diese Befehle koennen durch

```
settab,5?ia
```

ausgegeben werden, wobei durch 5 die Anzahl der Befehle festgelegt wird. Eine andere Moeglichkeit waere:

```
settab,5?i
```

wodurch bei der Ausgabe nur die Startadresse mit uebermittelt wird. Durch das ? Kommando wird auf Adressen in a.out

zugegriffen. Werden durch ein **adb** - Kommando mehrere Elemente abgefragt, so wird die aktuelle Adresse um die Anzahl der Byte erhoeht, die fuer die Erfuellung des Kommandos noetig sind. In unserem Beispiel werden 5 Befehle ausgegeben und die aktuelle Adresse wird um 10 erhoeht. Soll das Programm unter der Steuerung von **adb** abgearbeitet werden, geschieht das durch

```
:r
```

Das Loeschen eines Unterbrechungspunktes, z.B. des auf den Anfang der Funktion settab gesetzten, geschieht durch:

```
settab+4:d
```

Durch die Eingabe von:

```
:c
```

kann die Arbeit des Programms nach dem Unterbrechungspunkt fortgesetzt werden. Nach dem naechsten Programmstop (in diesem Fall beim Unterbrechungspunkt am Anfang von fopen), koennen durch **adb** - Kommandos Teile des Speicherinhaltes ausgegeben werden, z.B. mit:

```
xC
```

im Stack-Backtrace oder durch

```
tabs,3/8x
```

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

3.3. Weitere Moeglichkeiten bei der Arbeit mit Unterbrechungspunkten

Zunaechst wird die Programmabarbeitung durch

```
:c
```

fortgesetzt. Durch den Unterbrechungspunkt bei getch erfolgt ein Programmstop. Nach

```
:c
```

wird beim erstmaligen Abarbeiten von getch der zugehoerige Puffer ibuf gefuellt. Nach dem naechsten Programmstop am Anfang von getch kann der Anfang des Puffers durch

```
ibuf/20c
```

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

```
,3:c
```

wird zweimal ein Testpunktstop uebergangen 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 Faellen 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, was zum Halt wegen des Unterbrechungspunktes fuehrt. Bei der Umwandlung des Tabulatorzeichens in Leerzeichen wuerde es oft zum Aufruf von tabpos kommen. Da aber gezeigt ist, dass tabpos zum richtigen Zeitpunkt aufgerufen wurde, soll dieser Unterbrechungspunkt geloescht werden. Dies geschieht durch:

```
tabpos+4:d
```

Die verbleibenden Unterbrechungspunkte sollen neu definiert werden, um bei jeder Passage ein Kommando auszufuehren. Im ersten Fall geschieht dies durch:

```
settab+4:b settab,5?ia
```

was bei jeder Passage dieses Punktes die Ausgabe der ersten 5 Befehle von settab zur Folge hat. In einer erneuten Redefinition dieses Unterbrechungspunktes wird dann im Kommandofeld zusaetzlich die Ausgabe des Inhaltes von ptab als Oktalzahl angewiesen. Dies geschieht durch

```
settab+4:b settab,5?ia;ptab/o
```

und zwischen diesen Unterbrechungspunktdefinitionen liegt nach Anlage 2 die Redefinition des Unterbrechungspunktes bei getch+04. Dort wird durch

```
getch+4,3:b main.c?C
```

angewiesen, so dass bei jeder Passage der Inhalt der Variablen c in main als ISO-Schriftzeichen ausgegeben wird. Um eine Uebersicht ueber alle gesetzten Unterbrechungspunkte zu bekommen, wird

```
xb
```

ausgefuehrt.

Wie Anlage 2 zeigt, wird beim Abarbeiten des Programms unter **adb** entsprechend der Unterbrechungspunkte die Arbeit unterbrochen, und es werden die jeweiligen Kommandos ausgefuehrt. An dieser Stelle sei noch darauf hingewiesen, dass beim Kommando

```
:c
```

der Prozess mit dem Signal fortgesetzt wird, durch das er unterbrochen wurde. Durch

```
:c0
```

wird dieses Signal nicht uebergeben.

Ausserdem soll noch darauf hingewiesen werden, dass der Wert von Punkt durch das Abarbeiten eines Programmes unter **adb** nicht veraendert wird (auch nicht durch Kommandos bei Unterbrechungspunkten).

Die Uebergabe von Argumenten zu einem Programm sowie das Umlegen von Standard-Input und -Output ist durch:

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

moeglich.

Der Test im Einzelschritt wird mit dem Kommando

```
:s
```

ausgefuehrt. Dieses Kommando ermoeglicht auch den Programmstart. Es kommt dann zum Stop nach Ausfuehrung des ersten Befehls.

Adb ermoeglicht es, durch

```
address:r
```

ein Programm ab einer beliebigen Adresse abzuarbeiten.

Im count - Feld kann angegeben werden, wieviele Unterbrechungspunkte uebergangen 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, das gerade getestet wird, kann durch

```
:k
```

abgebrochen werden.

4. Maps

MUTOS 8000 unterstuetzt nur das durch die Magic-Nummer 0xE807 gekennzeichnete Format ausfuehrbarer Files. Hier sind Text (Befehle)- und Datensegment kontinuierlich im Speicher abgelegt.
Adb realisiert den Zugriff zu den einzelnen Segmenten von obifile und corfile ueber die Maps. Diese koennen mit

```
*m
```

ausgegeben werden.

Die bereitgestellten Werte haben fuer das Fileformat E807 folgende Bedeutung:

f1: Laenge des Fileheaders
f2: Verschieben Beginn Text- und Datenbereich zu Fileanfang (obifile)
Laenge Header + Laenge Text- und Datensegment (corfile)
b1: Anfangsadresse Text- und Datensegment (obifile)
Anfangsadresse Datensegment (corfile)
b2: entspricht b1 (obifile)
Anfangsadresse Stackbereich (corfile)
e1: Endadresse Text- und Datensegment (obifile)
Endadresse Datensegment (corfile)
e2: entspricht b2 (obifile)
Endadresse Stackbereich (corfile)

Bei Veraenderungen im zweiten corfile-Map-Segment ist der Operator /* notwendig.

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

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

Der Zugiff auf Speicherplaetze kann unter Benutzung einiger durch **adb** zur Verfuegung gestellter Variablen geschehen:

d	**Laenge des Datensegments**
s	**Laenge des Stack-Segments**
m	**Filetyp (E807)**
t	**Laenge des Textsegmentes**

Die Werte dieser Variablen koennen, wenn sie ungleich Null sind, durch das Kommando

```
xv
```

ausgegeben werden. Vom Nutzer kann auf diese Variablen wie folgt zugegriffen werden:

```
<t
```

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 fuer adb

Es ist moeglch, bei **adb** die einzelnen Formate zu kombinieren um Ausgaben uebersichtlich zu gestalten. Im weiteren soll dies an einigen Beispielen erlaeutert werden.

5.1. Formatierte Ausgaben

Die Zeile

```
,-1/4x4^8Cn
```

bewirkt die Ausgabe von 4 Hexadezimalzahlen (2 Byte Laenge) sowie danach die ISO-Interpretation dieser 8 Byte aus corefile. Die Bestandteile dieses Kommandos haben folgende Bedeutung:

,-1	Ausgabe ab aktueller Adresse bis zum Fileende. Ein negativer Wert fuer count bewirkt die Wiederholung eines Kommandos bis zu einem Fehler (hier Fileende).

Das Format **4x4^8Cn** setzt sich aus folgenden Bestandteilen zusammen:

4x	Ausgabe von 4 Worten als Hexadezimalzahlen
4^	Zuruecksetzen der aktuellen Adresse um 4 Worte (auf den urspruenglichen Wert vor Beginn der Ausgabe)
8C	Ausgabe 8 aufeinanderfolgender ISO-Zeichen unter Benutzung der Festlegung, dass Zahlen im Bereich #0 bis #1F als @, gefolgt vom entsprechenden Zeichen im Bereich #40 bis #57 ausgegeben werden; @ wird als @@ ausgegeben.
n	Ausgabe eines Zeilenwechsels

Die Moeglichkeiten der formatierten Ausgabe von **adb** koennen mit der Eingabe vorbereiteter **adb** - Kommandofiles kombiniert werden, um haeufig wiederkehrende Ausgaben zu realisieren. Der **adb** - Aufruf hat dann folgende Gestalt:

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

Ein Beispiel fuer ein solches Kommandofile ist:

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

Adb versucht, Adressen symbolisch auszugeben und zwar in der Form: **symbol + offset.** Das Kommando = ermoeglicht es, Zeichenketten unveraendert 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 unterdrueckt und durch eine Oktalausgabe ersetzt. Das Kommando

```
,-1/8ona
```

gibt das ganze File achtspaltig oktal aus.

5.2. Directory - Ausgabe

Eine andere Moeglichkeit zur Illustration der **adb** - Arbeit ist die Ausgabe von Directory - Inhalten, so wie sie im Directory abgespeichert sind. (Ein Directory - Eintrag besteht aus einer ganzen Zahl, der i-number und 14 ISO - Zeichen, dem Filenamen.)

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

bewirkt die gewuenschte Ausgabe. Durch **u** wird in diesem Beispiel die i-number als vorzeichenlose ganze Zahl ausgegeben. **8t** veranlasst die notwendigen Tabulatorspruenge und **14c** ist das Format fuer den Filename (14 ISO-Zeichen).

5.3 Konvertieren von Zahlen

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

```
072 =  odx
```

folgendes ausgegeben:

```
072  58      #3a
```

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

```
'a' = co
```

erzeugt

```
0141
```

Adb kann auch zur Berechnung von Ausdruecken benutzt werden. Allerdings muss davon abgeraten werden, da alle zweistelligen Operatoren dieselbe Prioritaet besitzen.

6. Aendern von Files

Durch **adb** koennen Files mit Hilfe des **w**- oder **W**-Kommandos geaendert 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** koennen 2 Byte und durch **W** 4 Byte geaendert 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 koennen, muss **adb** folgendermassen aufgerufen werden:

```
adb -w file1 file2
```

So koennte z.B in einem Programm durch

```
adb -w ex4n -
?l 'Th'
?W 'The'
```

das Wort 'This' in 'The' geaendert werden. Durch **?l** wird die suche bei Punkt begonnen und bei Uebereinstimmung der Werte abgebrochen. Haeufiger wird allerdings folgende Form benutzt:

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

wodurch beim ersten Auftreten von 'Th' die ganze Zeichenkette ausgegeben und die Adresse dieser Zeichenkette in Punkt gespeichert wird. Ein anderes Beispiel waere der Test eines C-Programms mit einem internen Flag. Durch **adb** kann dieses Flag gesetzt werden bevor das Programm abgearbeitet wird:

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

Durch **:s** wird der Prozess im Einzelschrittbetrieb fortgesetzt, oder (wenn noch nicht begonnen) gestartet. Durch das angegebene **w** Kommando wird der Wert von flag im Adressraum des Unterprozesses geaendert.

7. Besonderheiten

1. Funktionsaufrufe und Argumente werden durch die C-Save-Routine auf den Stack gebracht, die am Anfang jeder Funktion aufgerufen wird. Werden Unterbrechungspunkte auf den Eintrittspunkt der Funktion gesetzt, erfolgt die Unterbrechung vor Aufruf dieser Routine. Damit ist der Stackinhalt so, als waere die Funktion noch nicht gerufen worden.

2. Bei Ausgaben benutzt **adb** die Text- und Datensymbole von objfile. Dies fuehrt manchmal zu falschen Symbolen fuer 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 1: Ein Programm zum Aufloesen 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 2: Ausgabe fuer den dynamischen Test des C-Programms in Anlage 1

```
adb a.out -

settab+4:b

fopen+4:b

getch+4:b

tabpos+4:b

xb

breakpoints
count     bkpt      command
1     _tabpos+$4
1     _getch+$4
1     _fopen+$4
1     _settab+$4

settab,5?ia

_settab:        call csv
_settab+$4:     jp   _settab+$56
_settab+$8:     clr  -%e(FP)
_settab+$c:     cp   -%e(FP),$%50
_settab+$12:    jp   gt _settab+$52
_settab+$16:

settab,5?i

_settab:  call csv
          jp   _settab+$56
          clr  -%e(FP)
          cp   -%e(FP),$%50
          jp   gt _settab+$52

          :r

a.out: running
breakpoint      _settab+$4:     jp   _settab+$56

          settab+4:d

          :c

a.out: running
breakpoint      _fopen+$4:      call    __findio

          xC

_fopen($f2e,$f38)
```

```
     file:          #f2e
     mode:          #f38
_main(#1,#ffc2)
     col:           0
     ptab:          #266
     flag:          0
     c:             0
     stream:        0
```

```
          tabs,3/8d

_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+#4:     jp   _getch+#50
```

```
          :c

a.out: running
Tbreakpoint    _getch+#4:     jp   _getch+#50
```

```
          ibuf/20c

_ibuf:         This is   a    test of   e
```

```
          ,3:c

a.out: running
hisbreakpoint  _getch+#4:     br   _getch+#50
```

```
          :c

a.out: running
breakpoint     _tabpos+#4:    br   _tabpos+#46
```

```
          tabpos+4:d

          settab+4:b settab,5?ia

          getch+4,5:b main.c?C

          settab+4:b settab,5?ia; ptab/o

          xb

breakpoints
count     bkpt          command
1     _settab+#4        settab,5?ia; ptab/o
5     _getch+#4         main.c?C
1     _fopen+#4

          :r
```

```
a.out: running
_settab:        call csv
_settab+$4:     ei   nvi
_settab+$6:     subb r14,@r5
_settab+$8:     clr  -%e(FP)
_settab+$L:     cp   -%e(FP),$%50
_settab+$12:

$fff0:    06656
breakpoint      _settab+$4:     jp   _settab+$56

          :c

a.out: running
breakpoint      _fopen+$4:      call    __findio

          :c

a.out: running
$ff87:    @`
s$ff87:   @
T$ff87:   T
h$ff87:   h
i$ff87:   i
breakpoint      _getch+$4:      jp   _getch+$50

          xq
```

Kv 458/86 WV/6/1-10 3111a