

Starten und Bedienen des GNU Debuggers (GDB)

P.Gleichmann und D. Tillinger

GDB 5.3

Stand: 21.05.2003

Inhalt

1. Legende	1
2. Starten des GDB aus der Kommandozeile	1
3.1. Beenden des GDB	2
3.2. Nachträgliches Bestimmen des Ziels	2
3.3. Source-Listing.....	2
3.4. Programmfluß steuern	3
3.5. Breakpoints (BP).....	3
3.6. Watchpoints	4
3.7. Hardwareebene.....	4
3.8. Konstante Informationen	4
3.9. Informationen zur Laufzeit.....	5
3.10. Werte verändern	5
3.11. Automatische Anzeige.....	5
3.12. Benutzerbefehle.....	5
4. Sonstiges Erwähnenswertes.....	6
5. Glossar.....	6

1. Legende

[x]	= x ist optional
x y	= x oder y
bold	= Argument für den Befehl
<i>italic</i>	= Verweis auf einen anderen Befehl
<u>underlined</u>	= Kurzform des jeweiligen Befehles
#x	= Erläuterung bzw. Hinweise im Glossar

2. Starten des GDB aus der Kommandozeile

`gdb [executable]` Starten des GDB. Die Angabe der zu debuggenden **executable**^{#1} ist optional.

`gdb --args executable` Zusätzlich Angabe der Argumente^{#2} für die **executable**^{#1}.
`↳--arg1[,arg2,...,argN]`

Die folgenden Einträge sind alles Befehle, die im GDB selbst eingegeben werden können.

3.1. Beenden des GDB

<code>quit</code>	Beenden des GDB.
-------------------	------------------

3.2. Nachträgliches Bestimmen des Ziels

<code>file executable</code>	Einlesen der executable ^{#1} und laden der Symbole ^{#3} .
<code>core-file coredump</code>	Einspielen des Zustandes vor dem Crash, wobei coredump meistens “~/core” seien sollte. Ermöglicht die Analyse der Registerinhalte, etc.
<code>attach prozessID</code>	Debuggen eines laufenden Prozesses. Falls sich die Executable ^{#1} von prozessID im akt. Arbeitsverzeichnis befindet, so wird sie automatisch nachgeladen, andernfalls muß sie bereits vorher eingeladen worden sein.
<code>target remote host:port</code>	Verbinden mit dem bereits laufenden GDBServer. Die Executable ^{#1} muß bereits geladen sein.

3.3. Source-Listing

Hinweis: Außer speziell erwähnt, werden immer 10 Zeilen ausgegeben.

<code>list</code>	Ausgabe der aktuellen (d.h. Programmstart bzw. Programmcounter) Zeilen des Source codes. War das vorherige Kommando bereits <i>list</i> , dann die folgenden Zeilen.
<code>list -</code>	Die 10 vorherige Zeilen.
<code>list [datei:]zeile [,ende]</code>	Ausgabe ab zeile bis ende . Falls ende fehlt, so werden 10 Zeilen ausgegeben. Mit datei kann der Name einer alternativen Sourcedatei angegeben werden
<code>list [datei:]funktion</code>	Ausgabe des Anfangs einer funktion aus. Wie bei “ <i>list zeile</i> ” kann man sich auf eine andere datei beziehen.
<code>list *adresse</code>	Bestimmen der zur adresse ^{#4} gehörige Zeilennummer und Ausgabe der folgenden Zeilen.
<code>search regexp</code>	Suchen des regulären Ausdrucks regexp nach der letzten ausgegebenen Zeile.

3.4. Programmfluß steuern

set args [arg1 ,..., argN]	Setzen der Programmargumente ^{#2} .
run [arg1 ,..., argN]	Starten der Ausführung. Falls argumente ^{#2} angegeben, so wird implizit vor Ausführung “set args” aufgerufen.
continue	Fortsetzen der angehaltenen Ausführung.
next [anzahl]	Ausführen einer oder anzahl Zeilen. Ist die Zeile ein Funktionsaufruf, so wird der Aufruf ausgewertet, d.h. nicht in die Funktion reingesprungen.
step [anzahl]	Wie <i>next</i> , jedoch ist bei einem Funktionsaufruf die nächste Zeile die erste der Funktion.
nexti [anzahl], stepi ↳ [anzahl]	Wie <i>next</i> bzw. <i>step</i> nur auf Instruktionsebene.
until zeile * adresse	Fortsetzen des Ausführung bis die zeile bzw. adresse ^{#4} erreicht wird.
finish	Fortsetzen der Ausführung bis die akt. Funktion beendet worden ist. Der Rückgabewert - falls vorhanden - wird ausgegeben.
return [wert]	Sofortiges Verlassen der akt. Funktion. Der Rückgabewert kann mittels wert angegeben werden.

3.5. Breakpoints (BP)

Breakpoints stoppen den Programmfluß zu einem vom Benutzer vom Benutzer vorgegebenen Zeitpunkt (z.B. Adresse).

break zeile	Das Programm wird vor der Ausführung der zeile angehalten.
break funktion	Stoppt am Anfang von funktion , der “call/branch” wurde bereits ausgeführt. Zusätzlich Ausgabe der Funktionsparameter.
break * adresse	Stoppt an einer bestimmten Instruktions adresse ^{#4} . Sinnvoll bei Assemblercode bzw. bei Nichtvorliegen der Debugsymbole.
break zeile funktion ↳ * adresse ↳ if bedingung	Der Breakpoint an einer bestimmten Position wird nur bei Erfüllen einer bedingung . Syntax der bedingung ist wie in C.
tbreak zeile funktion ↳ * adresse	Unterbrechungszeitpunkt wie bei <i>break</i> . Jedoch wird der BP nach Eintritt automatisch gelöscht.
info breakpoints	Ausgabe der aktuellen BPs, ihrer internen Nummer und ihres Status.
delete nummer	Entfernen den BP mit der Nummer nummer , wobei nummer mittels “info breakpoint” herausgefunden werden kann.
enable disable ↳ nummer	Scharfschalten (enable)bzw deaktivieren (disable) eines BP. Siehe auch “delete nummer ”.

Nachtrag: Das setzen eines BP auf eine leere Zeile, obgleich sie natürlich eine Zeilennummer besitzt, ist nicht möglich!

3.6. Watchpoints

Im Gegensatz zu Breakpoints beziehen sich Watchpoints auf Speicherzugriffe.

watch variable ↳ * adresse	Anhalten, sobald das Programm versucht den Inhalt der variable ^{#6} bzw. adresse ^{#4} zu verändern.
rwatch	Wie <i>watch</i> - Versuch zu lesen. Wird nicht von jeder Plattform unterstützt.
awatch	Wie <i>watch</i> - Versuch zu lesen oder schreiben. Wie auch <i>rwatch</i> nicht von jeder Plattform unterstützt.

3.7. Hardwareebene

disassemble	Assemblerlisting des aktuellen Frames ^{#5} .						
disassemble adresse ↳ [, endadresse]	Assemblerlisting von adresse ^{#4} bis endadresse bzw. Ende des Frames ^{#5} .						
info registers ↳ [register 1, ↳ ..., register N]	Ausgabe des Inhalts der Register. register beschränkt die Ausgabe auf die zutreffenden Register.						
x/[anzahl][format][typ] ↳ adresse	Formatierte Speicherausgabe der adresse ^{#4} . Parameter:						
	<table border="1"> <tr> <td>anzahl</td> <td>Anzahl der auszugebenden typen</td> </tr> <tr> <td>format</td> <td>x = hex (standard), s = 0-String, i = instruktion</td> </tr> <tr> <td>typ</td> <td>Alle unsigned: b = char, h = short, w = long</td> </tr> </table>	anzahl	Anzahl der auszugebenden typen	format	x = hex (standard), s = 0-String, i = instruktion	typ	Alle unsigned: b = char, h = short, w = long
anzahl	Anzahl der auszugebenden typen						
format	x = hex (standard), s = 0-String, i = instruktion						
typ	Alle unsigned: b = char, h = short, w = long						

3.8. Konstante Informationen

Diese können bereits nach Einladen der Executable angesehen werden.

info file	Dateiname der geladenen Executable ^{#1} und ihrer Segmente.
info functions [regexp]	Alle Funktionen, d.h. auch Library Funktionen. Eindämmung der Informationsflut mittels eines reg. Ausdrucks regexp , z.b. Teilstring.
info variables [regexp]	Alle globalen Variablen. Einschränkung mit regexp .
info scope funktion ↳ zeile * adresse	Ausgabe der in funktion , in der zeile oder an der adresse ^{#4} sichtbaren Variablennamen.

3.9. Informationen zur Laufzeit

<code>print variable</code>	Ausgabe einer variable ^{#6} , wobei dies auch ein zusammengesetzter Ausdruck (z.B. Array-Element, Struct) sein kann.
<code>whatis variable</code>	Ausgabe des Typs von variable ^{#6} .
<code>info locals</code>	Alle gerade lokalen Variablen.
<code>info args</code>	Ausgabe der akt. Funktionsargumente.
<code>frame</code>	Anzeige der akt. Position.
<code>info frame</code>	Informationen über den akt. Frame ^{#5} , d.h. der akt. Stack.
<code>backtrace</code>	Anzeige der Funktionsaufrufshierarchie inkl. der übergebenen Argumente.

3.10. Werte verändern

<code>set variable = wert</code>	Zuweisen von wert der variable ^{#6} .
----------------------------------	--

3.11. Automatische Anzeige

<code>display variable *adresse</code>	Ausgabe von variable ^{#6} bzw. adresse sobald das Programm stoppt.
<code>display</code>	Anzeige der akt. definierten <i>displays</i> .
<code>undisplay nummer</code>	Löschen eines <i>displays</i> nummer .

3.12. Benutzerbefehle

<code>define funktion</code> : <code>end</code>	Beginn und Ende einer Benutzerfunktion mit dem Namen funktion . Falls ihr Argumente übergeben werden, so werden sie in \$arg0..9 gespeichert. Erlaubt sind zwischen den Tags alle GDB und Benutzerbefehle - einer pro Zeile.
<code>define hook-funktion</code>	Definition eines Hooks für funktion , d.h. die enthaltenen Befehle werden immer vor funktion ausgeführt. Dies ist erlaubt für alle internen und auch Benutzerbefehle. Es existieren drei Sonder funktionen :

stop	Sobald der Programmfluß gestoppt (z.B. BP) wird
run	Ausführungsbeginn
continue	Fortsetzen der Ausführung

<code>define hookpost- ↳ funktion</code>	Wie <i>hook</i> , die Ausführung jedoch erst nach funktion .
--	---

Nachtrag: Damit man die Befehle nicht immer neu definieren muß, kann man sie in der Initialisierungsdatei “~/gdbinit” ablegen.

4. Sonstiges Erwähnenswertes

- Da GDB auf die Readline Library zugreift, gibt es Tab-Automcompletion für die Befehle und Dateinamen.
- “help” hilft, wenn auch meist kryptisch;-)
- Ein Enter ohne Befehl wiederholt den vorherigen Befehl.
- Ctrl+C stoppt ein laufendes Programm. Nachteil: Man befindet sich meist mitten in einer Libraryfunktion.
- Auf interne Variablen (z.B. Register) kann mittels eines vorangestellten “\$” zugegriffen werden, z.B. “\$eax”.
- Bei den Zahlenangaben (z.B. Adressen) dürfen einfache math. Ausdrücke angegeben werden, z.B. “\$eax+20”.
- “shell” erlaubt das Aufrufen von externen Befehlen.
- Ein “alias gdb=gdb -q” im Shell-Profile schaltet die Copyright-Message am Anfang aus.

5. Glossar

- #1 Executable:
Das ausführbare Programm mit Debuginformationen - siehe →#3 (“gcc -g -o programm sources”).
- #2 Argumente:
“argv” bei main() in C.
- #3 Symbole:
Die benutzten dynamischen Libraries, die im-/exportierten Funktionen, Variablen, etc.
- #4 Adresse:
Alle Zahleneingaben sind im GDB standardmäßig dezimal, d.h. heximalen Adressen muß ein “0x” vorangestellt werden. Dieses Verhalten des GDB kann mit “radix” verändert werden. Zusätzlich kann man mittels Typcasting auf die konkreten Datentypen zugreifen.
- #5 Frame:
Allgemein ein Codeblock “{ ... }” in dem Variablen vereinbart werden, d.h. der Funktionskörper ist auch ein Frame.
- #6 Variable:
Eine Variable muß sichtbar sein, damit man auf sie zugreifen kann.