

Die GNUPro Tool-Chain
unter besonderer Berücksichtigung der Embedded-ARM-ELF
Entwicklung

Andreas Bürgel¹

<http://www.andreas-buergel.de>

7. April 2003

¹Mail: andreas@andreas-buergel.de

Version \$Id: Toolchain.tex,v 1.3 2003/04/07 20:39:28 andreas Exp \$

eCosTM ist ein eingetragenes Markenzeichen von Red Hat, Inc.

RedHat[®], RedBootTM, GNUPro[®], InsightTM, CygwinTM sind eingetragene Markenzeichen von Red Hat, Inc.

Linux[®] ist ein eingetragenes Markenzeichen von Linus Torvalds

UNIX[®] ist ein eingetragenes Markenzeichen von The Open Group

Microsoft[®], Windows[®], Windows NT[®], Windows 95[®], Windows 98[®], Windows 2000[®] sind eingetragene Markenzeichen der Microsoft Corporation

ARM[®] ist ein eingetragenes Markenzeichen von ARM Ltd.

MIPS[®] ist ein eingetragenes Markenzeichen von MIPS, Inc.

Intel[®], StrongARM[®] sind eingetragene Markenzeichen von Intel Corporation

ATMEL[®] und AT91[®] sind eingetragene Markenzeichen der Atmel Corporation

Alchemy Semiconductor[®], Au1000[®], Au1500[®] sind eingetragene Markenzeichen von Alchemy Semiconductor Inc.

HyperStone[®] ist ein eingetragenes Markenzeichen der HyperStone AG

Alle in dieser Aufzählung nicht genannten Markenzeichen sind ebenfalls eingetragene Markenzeichen ihrer Besitzer. Der Autor will sich diese auf keinen Fall zu eigen machen.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Über dieses Dokument	1
1.2	Begriffsklärung	1
1.3	GNUPro-Tool-Chain Philosophie	2
1.4	Aus was besteht die Tool-Chain?	2
1.4.1	Compiler	3
1.4.2	Debugger	3
1.4.3	Binutils	3
2	Die Tools im Überblick	5
2.1	Installation	5
2.2	gcc - der Compiler	5
2.2.1	Wichtige Compiler-Optionen	6
2.3	ld - der Linker(Binder)	6
2.3.1	Linker-Skripte	7
2.3.2	Einbinden von „Fremd“-Daten	7
2.3.3	Wichtige Linker-Optionen	9
2.4	as - der Assembler (am Beispiel ARM)	9
2.4.1	Inline-Assembler in C/C++	10
2.5	Weitere Binutils	12
2.5.1	objdump - mehr über Binärdateien erfahren	12
2.5.2	readelf - ELF-Header von Binärdateien ansehen	12
2.5.3	objcopy - zwischen Binärformaten konvertieren	12
2.5.4	size - die wahre Größe von Binärdateien erfahren	12
2.5.5	strip - Debugger-Informationen aus Binärdateien entfernen	12
3	gdb/Insight - der Debugger	13
3.1	Aufruf	13
3.2	Das Insight Haupt-Fenster	13
3.3	Download vom Host zum Target	17
3.4	Watches - den Inhalt von Variablen beobachten	17
3.5	Breakpoints und Tracepoints	21
3.5.1	Breakpoints	21
3.5.2	Tracepoints	21
4	Cygin - unter Windows arbeiten wie unter Unix	23
4.1	Was ist Cygin?	23
4.2	Installation	23

A Probleme und Lösungen	25
A.1 Obskure Fehlermeldungen beim Umgang mit Text-/Quell-Dateien unter Cygwin	25
A.2 Der Build-Prozess unter Cygwin ist langsam	25
A.3 Beim Linken tritt eine <code>undefined reference</code> bezüglich einer Funktion auf obwohl diese im Programm vorhanden ist	25

Kapitel 1

Einleitung

1.1 Über dieses Dokument

Dieses Dokument ist ein Fragment. Es erhebt z.Z. keinen Anspruch auf Vollständigkeit und Richtigkeit. Der Autor übernimmt keinerlei Haftung für Schäden, die durch Anwendung der in diesem Dokument enthaltenen Informationen entstanden sind.

Sollte ein Leser dieses Dokuments auf die glorreiche Idee kommen, die Steuerungs-Software für die Speisewasserpumpen des Primär-Kreislaufs eines Atomkraftwerks umzuschreiben und damit ein mehr oder weniger zivilisiertes Land von der Weltkarte entfernen, so ist das nicht die Schuld des Autors. Der Autor übernimmt auf keinen Fall irgendwelche Haftung.

Dieses Dokument darf frei benutzt und weiter verbreitet werden. Die kommerzielle Weiterverbreitung ist jedoch untersagt. D.h. man darf mit Hilfe der Informationen in diesem Dokument Produkte entwickeln und diese verkaufen ohne für das Dokument zu bezahlen. Es ist jedoch nicht gestattet für die Weiterverbreitung dieses Dokuments Geld zu verlangen. Ebenso ist die Veränderung des Dokuments sowie die Verwendung des Dokuments in eigenen Werken, dies betrifft auch Auszüge des Dokuments, nur mit schriftlicher Einwilligung des Autors gestattet.

Der Autor behält sich vor die Nutzungsbedingungen ohne vorherige Bekanntmachung jederzeit zu ändern.

1.2 Begriffsklärung

Als *binär* oder als *Binärdatei* wird hier alles verstanden, was Produkt eines Übersetzer, d.h. eines Assemblers oder eines Compilers, ist. *Executable* ist eine ausführbare Applikation für eine bestimmte Hardware und letztlich auch eine Binär-Datei. Eine *Objekt-Datei* ist Ergebnis des Übersetzungs-Vorgangs einer einzelnen Quell-Datei und damit ebenfalls binär. Das, was auf .o endet ist meist eine Objekt-Datei. Eine *Bibliothek* ist eine Zusammenfassung mehrerer logisch zusammen gehörender Objekt-Dateien. Lader eines Betriebssystems, der Linker und andere Teile der Tool-Chain benötigen Informationen über Binärdateien um mit diesen funktionieren zu können. Diese Informationen sind in den Binär-Dateien abgelegt. Deshalb müssen alle Binär-Dateien in einem bestimmten *Binär-Format* vorliegen.

Als *Host* wird der Rechner bezeichnet, auf dem die Entwicklung stattfindet bzw. auf dem der Debugger läuft. Als *Target* oder *Ziel-Hardware* wird der Rechner bezeichnet, für den die Applikation entwickelt wird. Wenn man „normale“ Unix- oder Cygwin-Applikationen entwickelt, so ist diese Unterscheidung natürlich irrelevant.

Ein in spitze Klammern gefasster <BEGRIFF> ist ein Platzhalter, z.B. für einen Namen oder einen Programm-Parameter.

Alle Dateinamen und Benutzer-Eingaben werden serifenlos gedruckt. In Schreibmaschinenschrift werden Programm- und Skript-Dateien, sowie Funktions-Namen und teilweise auch Ausgaben von Programmen gedruckt. Menü-Punkte und Auswahl-Optionen in Bildschirm-Masken werden (*stark*) *kursiv* dargestellt.

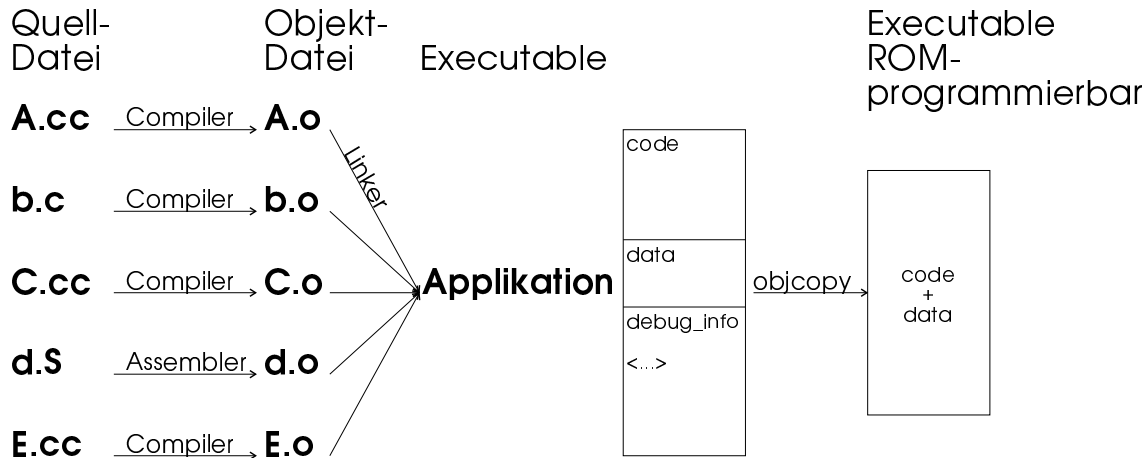


Abbildung 1.1: Von der Quell-Datei zur fertigen Applikation

1.3 GNUPro-Tool-Chain Philosophie

Die GNUPro-Tools folgen der Unix-Filosofie ein Programm nur für einen bestimmten Zweck zu entwerfen und komplexere Funktionalität durch das Zusammenwirken mehrerer Einzel-Programme zu erreichen. Bspw. erzeugt der GNU-C-Compiler nicht direkt Binär-Code für einen bestimmten Prozessor, sondern zunächst Assembler-Code, der danach vom Prozessor-spezifischen Assembler in Binär-Code übersetzt wird.

Ein Vorteil gegenüber einem einzigen monolithischen Werkzeug zur Programm-Erzeugung - neben Aspekten der Entwicklung dieser Werkzeuge - ist, dass man z.B. einzelne Teile, wie den Compiler, gegen andere Teile, wie z.B. einen kommerziellen Compiler, austauschen kann.

Für jeden Ziel-Prozessor, für den man Programme entwickeln will, benötigt man eine eigene Version der GNUPro-Tools. Dabei ist es unwichtig, ob der Host-Rechner einen zum Ziel-Prozessor kompatiblen Prozessor hat. Ist das nicht der Fall, so kompiliert man die gesamte Toolchain für Cross-Entwicklung. Das übliche Szenario hierbei dürfte ein Intel 80386 kompatibler Host-Rechner und ein ARM- oder MIPS-basiertes Embedded-System sein.

Jedes einzelne Programm der Tool-Chain erhält einen Präfix, der eindeutig Ziel-Prozessor-Architektur und Binärformat des Executables identifiziert. Der Name eines Tools hat die Form <CPU_ARCHITEKTUR>-<BINÄR_FORMAT>-<PROGRAMM>, der (Cross-)Compiler für die ARM-Architektur und das ELF-Binärformat heißt also arm-elf-gcc. Das Binärformat ELF¹ ist mittlerweile in der Unix-Welt das verbreitetste, andere Binärformate haben als veraltet zu gelten.

Das Bild zeigt welche Programme der Tool-Chain beteiligt sind, bis man eine in den nicht-flüchtigen Speicher eines Embedded-Systems programmierbare Stand-Alone - d.h. ohne Betriebssystem - Applikation erhält. Entwickelt man Applikationen für (Embedded-)Linux, so ist der objcopy-Schritt überflüssig.

1.4 Aus was besteht die Tool-Chain?

Die Tool-Chain besteht aus diversen einzelnen Programmen, die diversen Paketen - auch hinsichtlich der Installation - zu zuordnen sind.

¹ELF wird angeblich auch von einigen kommerziellen Entwicklungssystemen als Binärformat verwendet. Oft jedoch nur in einer sehr stark veränderten oder abgespeckten Form, die nicht mehr kompatibel ist.

1.4.1 Compiler

gcc	C-Compiler
g++	C++-Compiler

1.4.2 Debugger

gdb	Source-Level-Debugger, als RedHat-Insight mit grafischer Benutzeroberfläche
-----	---

1.4.3 Binutils

ar	erzeugt Bibliotheken aus Objekt-Dateien
as	Assembler
ld	Linker
nm	
objdump	
objcopy	
ranlib	
readelf	
size	zeigt die wahre Größe einer Objekt-Datei bzw. eines Executables, d.h. die Größe Code und Daten
strings	gibt alle für Menschen lesbaren Zeichenketten einer Objekt-Datei oder eines Executables aus
strip	entfernt Sektionen aus Objekt-Dateien und Executables, kann Debugger-Informationen aus einem Executable entfernen

Kapitel 2

Die Tools im Überblick

2.1 Installation

Die Installation ist in [6] genauestens beschrieben und wird hier deshalb nicht weiter behandelt. Man sollte *wirklich* alles - bis auf den Namen des Zielverzeichnis, diesen kann man frei bestimmen - so machen, wie es dort beschrieben ist. Ansonsten kann es sein, dass der Build-Prozess nicht funktioniert und abenteuerliche Fehlermeldungen produziert, insbesondere unter Cygwin/Windows.

2.2 gcc - der Compiler

Der Compiler ist der zentrale Punkt bei der Entwicklung mit der GNUPro-Tool-Chain. Sowohl Assembler, als auch Linker werden normalerweise nicht direkt aufgerufen, sondern implizit über den Compiler. Dies hat z.B. für Assembler-Dateien den Vorteil, dass man in diesen C-#defines benutzen kann, da der Compiler vor dem Aufruf des Assemblers die Quelldatei durch den Präprozessor vorverarbeiten läßt.

Außerdem weiß der Compiler mehr über die System-Umgebung, bspw. Namen und Pfad zu bestimmten Bibliotheken, als z.B. der Linker. Ruft man den Linker direkt auf, so muss - bzw. kann - man dieses Wissen als Parameter-Satz übergeben, was flexibel aber auch komplex ist.

Der Build-Prozess einer Applikation wird normalerweise über ein oder mehrere Makefiles und make gesteuert. Das folgende Beispiel zeigt ein Makefile für ein kleines eCos-Projekt, ist jedoch mit kleinen Änderungen auf Cygwin- oder Linux-Projekte übertragbar.

```
1 CC      = arm-elf-gcc
2 INCDIR  = -I$$HOME/develop/eCos_work/sa1110dimm/install/include \
3          -I$$HOME/irgendwas_anderes
4 LIBDIR  = -L$$HOME/develop/eCos_work/sa1110dimm/install/lib
5 OPT     = -O2
6 CCFLAGS = -ggdb -fno-rtti -fno-exceptions $(OPT)
7 LDFLAGS = $(LIBDIR) -nostdlib -Ttarget.ld -Wl,-Map,hello.mapfile
8
9 all: hello
10
11 hello: hello.o asmpart.o
12     @$ (CC) $(LIBDIR) $(INCDIR) $(LDFLAGS) -o $@ $<
13
14 hello.o: hello.cc
15     @$ (CC) -c $(CCFLAGS) $(INCDIR) $<
16
17 asmpart.o: asmpart.S
18     @$ (CC) -c $(CCFLAGS) $<
```

Interessant sind hier besonders die Zeilen 18 und 12. In Zeile 18 wird eine Assembler-Quell-Datei übersetzt. Das besondere ist, dass der Assembler `arm-elf-as` nicht explizit aufgerufen wird, sondern implizit über den Compiler `arm-elf-gcc`. Dieser erkennt an der Datei-Endung `.S` oder auch `.s` dass es sich bei der Quell-Datei um Assembler-Code handelt und ruft - nach Vorverarbeitung durch den Präprozessor - den Assembler auf. In Zeile 12 werden alle Objekt-Dateien zu einer Applikation `hello` gebunden. Auch hier wird der Linker nicht explizit aufgerufen, sondern wieder implizit vom Compiler.

2.2.1 Wichtige Compiler-Optionen

Optionen für den Compile-Prozess

- O<X>
Diese Option steuert die Optimierungsstufe des Compilers. <X> ist eine Zahl von 1 bis 6, wobei größere Zahlen aufwändigere Optimierung bedeuten. Ist die Optimierung aktiviert, so wird der Compiler versuchen, möglichst kompakten und Zeit-effizienten Code zu erzeugen. Beim Debuggen kann die Optimierung den Benutzer verwirren, da bei der Optimierung häufig auch die Reihenfolge von Code geändert wird, der aus der Hochsprachen-Übersetzung resultierende Maschinen-Code also häufig eine andere Reihenfolge hat, als eine Eins-Zu-Eins-Übersetzung des Hochsprachen-Quell-Codes erwarten läßt.
- g oder -ggdb
Ist eine dieser Optionen gesetzt, so wird der Compiler Debugger-Information beim Kompilieren einer Quell-Datei erzeugen.

Optionen für den Link-Prozess

- nostdlib
Diese Option ist hauptsächlich interessant für die Entwicklung von Embedded-Applikationen. Ist diese Option gesetzt, so werden beim Linken einer Applikation wirklich nur die als Parameter und vom Linker-Skript spezifizierten Objekt-Dateien und Bibliotheken zur Applikation gelinkt. Anderenfalls würden auch implizit definierte Bibliotheken, wie `libc`, und Objekt-Dateien, wie `crto.o`¹ zur Applikation gelinkt.
- Wl,<OPTION>
Übergibt die Option <OPTION> an den Linker. Wenn <OPTION> Kommata enthält, so bricht der Compiler <OPTION> an den Stellen der Kommata vor der Übergabe an den Linker in Einzelwörter auf.

2.3 ld - der Linker(Binder)

Der Linker bindet alle einzelnen Objekt-Dateien, aus denen das fertige Binär-Programm bestehen wird, zusammen. D.h. er berechnet Sprung- und Datums-Adressen und passt alle Address-Referenzen in den einzelnen Objekt-Dateien entsprechend an.

Jede Objekt-Datei besteht aus verschiedenen Sektionen (SECTIONS) für z.B. Code (`.text`), Daten (`.data`, `.rodata`) oder auch vom Compiler für den Debugger generierte Daten (`.debug_*`). Im fertigen Executable verschmilzt der Linker die einzelnen Sektionen der Objekt-Dateien gleichen Typs zu jeweils einer einzigen Sektion. D.h. aus allen einzelnen Code-Sektionen wird z.B. eine einzige Code-Sektion.

Der GNU-Linker kann auch mit den Entwicklungs-Tools anderer Hersteller zusammen verwendet werden, sofern diese sich beim Objekt-Format an die Standards halten. Ebenso kann der Linker Objekt-Dateien verschiedener Programmiersprachen linken. D.h. man kann Teile eines Projekts in C oder C++ schreiben und andere Teile in Assembler oder Pascal und diese trotzdem zu einem Executable linken.

¹Create Runtime 0 ist der Startpunkt von Linux-Applikationen und erzeugt eine minimale Laufzeit-Umgebung wie Stack etc. für eine Applikation.

2.3.1 Linker-Skripte

Der Link-Prozess wird von einem Skript gesteuert, das festlegt an welche Adresse welche Sektionen eines Programms reloziert werden, in welcher Objekt-Datei gestartet wird bzw. welche Funktion gestartet wird, d.h. wo der Start-Punkt des Programms liegt.

Normalerweise benutzt der Linker ein implizit definiertes Linker-Skript, wenn man eine Linux-Applikation kompiliert wird vom Linker ein Skript aus dem Verzeichnis `lib/ldscripts`¹. Deshalb kommt man bei der Entwicklung von Linux-Applikationen meist nicht mit Linker-Skripten in Berührung.

Ein einfaches Beispiel für ein RAM-Startup Linker-Skript:

```

1 MEMORY
2 {
3     ram : ORIGIN = 0, LENGTH = 0x2000000
4     rom : ORIGIN = 0x40000000, LENGTH = 0x400000
5 }
6
7 SECTIONS
8 {
9     . = 0x1000;
10    .text : { *(.text) } > ram
11    .data : { *(.data) } > ram
12    .bss  : { *(.bss) } > ram
13 }
```

In einem Linker-Skript werden zuerst Speicher-Regionen definiert. Meist läßt sich der Speicher so einfach wie in Zeile 3 und 4 beschreiben. Die Position der Speicher-Regionen im Adress-Raum hängt vom Betriebszustand der CPU ab. Nach dem Einschalten liegt bei allen CPUs eine Bank des nicht-flüchtigen Speichers an der logischen Adresse Null. Nach einem speziellen Umschalt-Befehl, bzw. dem Einschalten der MMU, liegen die einzelnen Speicher-Regionen jedoch an anderen Adressen. Abhängig vom Startup-Typ([5]) einer Applikation benötigt man also verschiedene Linker-Skripte.

Im Hauptteil eines Linker-Skripts werden die einzelnen Sektionen definiert und was in diese Sektionen hinein soll (Zeilen 10, 11, 12). Jede Sektion wird schließlich mit dem Operator `>` REGION einer Speicher-Region zugeordnet, d.h. der Linker berechnet die Adressen innerhalb dieser Sektion so, dass sie in der spezifizierten Speicher-Region liegen. In Zeile 9 wird definiert, dass die Code-Sektion an Adresse `0x1000` beginnen soll.

Die Reihenfolge, in der die Sektionen im Linker-Skript aufgeführt sind, entspricht der Reihenfolge der Sektionen in der Applikation.

Mehr über Linker-Skripte erfährt man in [1].

2.3.2 Einbinden von „Fremd“-Daten

Man kann nicht nur Objekt-Dateien, also Ergebnisse eines Compiler-Laufs über eine Programm-Quell-Datei, binden, sondern auch noch andere beliebige Dateien. Voraussetzung ist, dass diese Dateien in einem bestimmten Datenformat vorliegen. Ein Aufruf von `<CPU_ARCHITEKTUR>-<BINÄR_FORMAT>-objdump -i` zeigt die von dieser Version der Binutils unterstützten Objekt- bzw. Datei-Formate. Daten-Dateien hinzu zu linken, die nicht im Intel-Hex-Format (`ihex`) sind, scheint jedoch zumindest schwierig² zu sein.

Benutzt man bspw. ein ARM-Modul mit FPGA, so muss man das oder die FPGA-Programme beim Booten des Moduls in das FPGA laden, da das FPGA seine Programmierung nach einem Reset oder beim Abschalten der Versorgungs-Spannung verliert. Die FPGA-Daten müssen also im nicht-flüchtigen Speicher - EPROM oder Flash - des Moduls vorgehalten werden.

¹Dieses Verzeichnis befindet sich im Basis-Verzeichnis der GCC/Binutils-Suite. Der Name des Basis-Verzeichnisses variiert zwischen den Linux-Distributionen bzw. hängt der Name des Verzeichnisses davon ab was beim Kompilieren der Toolchain angegeben wurde. Unter SuSE Distributionen ab 7.0 heißt das komplette Verzeichnis `/usr/i486-suse-linux/lib/ldscripts`.

²Es ist dem Autor bis jetzt nicht gelungen Daten-Dateien in anderen Formaten als Intel-Hex erfolgreich hinzu zu linken.

Man könnte natürlich die FPGA-Daten an eine Adresse hinter der Applikation ins ROM programmieren. Dies hat jedoch mehrere Nachteile:

1. Man benötigt mindestens zwei Programmier-Vorgänge, einen für die Applikation und einen zweiten für die FPGA-Daten.
2. Der normalerweise sehr begrenzte nicht-flüchtige Speicherplatz wird nicht effizient ausgenutzt, da sich mit Sicherheit eine nicht nutzbare Lücke zwischen Applikation und FPGA-Daten ergibt.
3. Wächst die Applikation im Laufe ihrer weiteren Entwicklung, so müssen die FPGA-Daten im ROM irgendwann an eine andere Adresse als vorher einmal programmiert werden. Das bedeutet, dass in der Applikation gespeicherte Adress-Referenzen - die FPGA-Lade-Routine der Applikation muss ja wissen an welchen Adressen die FPGA-Daten liegen - angepasst werden müssen. Hierbei *wird* es zu Fehlern kommen.

Der GNU-Linker kann FPGA- und andere Daten zu einem Binär-Programm hinzu linken und sogar die Start- und Ende-Adresse dieser Daten in Form von Variablen exportieren. Diese Variablen sind dann innerhalb der Applikation verwendbar.

```

1 <...>
2
3 SECTIONS
4 {
5   <...>
6
7   .fpga_data ALIGN (4) : { __fpga_data_start = ABSOLUTE (.); \
8                           dimm_sall110_fpga_data1.ihex \
9                           __fpga_data_end   = ABSOLUTE (.); \
10                          . = ALIGN (4); \
11                          __fpga_data_start2 = ABSOLUTE (.);
12                          dimm_sall110_fpga_data2.ihex \
13                          __fpga_data_end2 = ABSOLUTE (.);} > ram
14
15   <...>
16 }
```

In Zeile 7 des obigen Linker-Skript-Teils wird eine neue, zusätzliche Sektion mit dem Namen `.fpga_data` definiert. Das Alignment für die Start-Adresse dieser Sektion ist gleich vier Byte, d.h. der Linker wird den Beginn dieser Sektion auf eine durch vier teilbare Adresse legen. Die Start-Adresse dieser Sektion soll in einer Variablen mit dem Namen `__fpga_data_start` exportiert werden. Der Linker wird dann Referenzen auf diese Variable gegen die von ihm berechnete Adresse ersetzen.

In Zeile 8 steht der Name der FPGA-Daten-Datei, die zu dem Programm hinzu gebunden werden soll, in Zeile 9 wird eine Variable für die End-Adresse des ersten FPGA-Daten-Blocks definiert. Auch hier wird der Linker Referenzen auf diese Variable gegen die entsprechende Adresse ersetzen.

Benötigt mehr als ein FPGA-Programm - es ist ja denkbar in verschiedenen Betriebszuständen verschiedene FPGA-Programme zu laden - so ist die FPGA-Daten-Datei analog zur ersten einzubinden. Wichtig ist für diesen Fall Zeile 10. Diese sorgt dafür, dass der Anfang des zweiten FPGA-Daten-Blocks ebenfalls auf einer durch vier teilbaren Adresse liegt.

```

1 <...>
2
3 extern unsigned int __fpga_data_start;
4 extern unsigned int __fpga_data_end;
5 extern unsigned int __fpga_data_start2;
6 extern unsigned int __fpga_data_end2;
7
8 <...>
```

```

9
10 void mainThread ( cyg_addrword_t pThreadData) {
11     unsigned int* lStart = (unsigned int*) &__fpga_data_start;
12     unsigned int* lEnd   = (unsigned int*) &__fpga_data_end;
13     unsigned int lProgramSize = (unsigned int) lEnd - (unsigned int) lStart;
14     unsigned int* lStart2 = (unsigned int*) &__fpga_data_start2;
15     unsigned int* lEnd2   = (unsigned int*) &__fpga_data_end2;
16     unsigned int lProgramSize2 = (unsigned int) lEnd2 - (unsigned int) lStart2;
17
18     SA1110DIMMFPGAAdapter* lAdapter = new SA1110DIMMFPGAAdapter ();
19     EP20K* lApex = new EP20K ( lAdapter);
20
21     setupGPIO ( lAdapter);
22     lStatus = lApex->enterProgrammingMode ();
23     lStatus = lApex->loadProgram ( lStart, lProgramSize);
24
25     <...>
26
27     lStatus = lApex->enterProgrammingMode ();
28     lStatus = lApex->loadProgram ( lStart2, lProgramSize2);
29
30     <...>
31
32 }
33
34 <...>

```

In der Applikation werden die vom Linker erzeugten Variablen als extern unsigned int (Zeile 3-6) deklariert und können nach Umwandlung in Zeiger-Typen direkt weiter verwendet werden.

2.3.3 Wichtige Linker-Optionen

- T <SKRIPT_DATEI>
Der Linker benutzt kein implizites Linker-Skript, sondern das Skript <SKRIPT_DATEI>. Bei der Entwicklung von Embedded-Applikationen wird man wahrscheinlich immer ein spezielles, an die jeweilige Hardware angepasstes Skript benutzen.
- Wl,-Map,<MAP_DATEI>
Der Linker sammelt Daten über Position und Größe aller Objekte, d.h. Funktionen und Daten, im Executable und speichert diese in der Datei <MAP_DATEI> ab. Mit dieser Option läßt sich sehr genau nachprüfen wieviel Speicherplatz der Code einer Funktion bzw. einer Methode verbraucht. Der Präfix -Wl, sowie das Komma zwischen -Map und <MAP_DATEI> sind notwendig, wenn der Linker implizit vom Compiler aufgerufen wird. Dies ist der Regelfall.

2.4 as - der Assembler (am Beispiel ARM)

Der GNU Assembler (für ARM) benutzt folgende Syntax (Drei-Adress-Befehl)

Befehl Ziel-Operand, Quell-Operand1, Quell-Operand2

Der ARM-Befehl `add r1, r2, r3` bedeutet also „Addiere den Inhalt von CPU-Register 2 zum Inhalt von CPU-Register 3 und speichere das Ergebnis in CPU-Register 1 ab“.

(Quell-)Operanden können mit speziellen Zeichen ausgezeichnet werden. Abhängig von dieser Auszeichnung wird der Operand anders interpretiert.

[<REGISTER>] Es wird nicht der Inhalt des referenzierten Registers verwendet, sondern der Inhalt des Registers wird als Speicher-Adresse interpretiert. Beispiel: `ldr r1, [r0]` lädt das CPU-Register 1 mit der in CPU-Register 0 referenzierten Speicherzelle.

#<KONSTANTE> Die dem # folgende Konstante wird als Immediate-Wert interpretiert, d.h. vom Assembler in das entsprechende Befehlswort hinein codiert. Beispiel: `add r1, r1, #7` inkrementiert den Inhalt von CPU-Register 1 um 7.

=<KONSTANTE> Der Assembler legt implizit in der Objekt-Datei eine Konstante mit dem Wert KONSTANTE an.

Beispiel: `ldr r0, =0x1000000` erzeugt eine Konstante mit dem Wert `0x1000000`, diese wird danach in das CPU-Register 0 geladen. Letztendlich wird also der einzelne Befehl zur Sequenz (ähnlich)

```
mov rX, <KONSTANTEN_ADRESSE>
ldr r0, [rX]
```

expandiert, wobei `rX` ein vom Assembler ausgewähltes CPU-Register ist. Dieser Ausdruck ist sehr nützlich für Konstanten, die so groß sind, dass sie nicht als Immediate-Wert in ein Befehls-Wort codiert werden können.

2.4.1 Inline-Assembler in C/C++

Statt reine Assembler-Dateien zu schreiben, ist es auch möglich Assembler-Code in Hochsprachen-Code³ einzubetten.

```
1 void inlineTest ( void) {
2     unsigned int a = 2;
3     unsigned int b = 3;
4     unsigned int c = 0;
5
6     asm volatile (
7         "sub %1, %1, #1
8         add %0, %1, %2"
9         : "=r" (c)
10        : "r" (a), "r" (b));
11
12    printf ( " --> %d\n", c); // gibt " --> 4" aus
13 }
```

Ein Inline-Assembler-Block beginnt mit dem Schlüsselwort `asm`. Folgt diesem das Schlüsselwort `volatile`, so wird der Block genau in der gegebenen Form in den Code eingefügt und vom Compiler nicht weiter optimiert. Ohne `volatile` würde der Compiler auch im Inline-Code Optimierungen vornehmen, falls er es für richtig erachtet.

Die ersten Zeile, bzw. die ersten Zeilen, des Blocks enthalten den Assembler-Code. Die erste mit dem : beginnende Zeile beschreibt in welchen Variablen die Ergebnisse des Assembler-Blocks gespeichert werden sollen, die zweite mit : beginnende Zeile beschreibt die Eingabe-Parameter des Blocks. Alle Ausdrücke der Form `%<ZAHL>` im Block sind Platzhalter für Eingabe-Parameter bzw. für die Rückgabe von Ergebnissen. Die Zahl im Namen der Platzhalter legt die Reihenfolge fest, in der diese durch die Parameter ersetzt werden, wobei die Ausgabe-Parameter mitzählen. `%1` ist der zweite Platzhalter und wird beim ersten Auftreten

³sofern man bei C und C++ von Hochsprachen sprechen kann

durch den insgesamt zweiten Parameter des Ausdrucks "r" (a), d.h. durch den Inhalt von Variable a, ersetzt. Taucht %1 im weiteren Verlauf noch einmal auf, so wird es nicht mehr ersetzt, da ja bereits ein Rechenergebnis darin gespeichert ist, ein nochmaliges Ersetzen durch a also das Gesamtergebnis verfälschen würde. Der dritte Platzhalter %2 wird durch den Inhalt von Variable b ersetzt (Zeile 10).

Das Ergebnis der Berechnung steht in %0. Der Ausgabe-Parameter "=r" (c) bedeutet, dass das Ergebnis in der Variablen c gespeichert werden soll (Zeile 9). Insgesamt berechnet der Assembler-Block nichts anderes als $c = (a - 1) + b$.

Das Attribut "r" bei den Eingabe-Parametern bedeutet, dass der Compiler ein CPU-Register an dieser Stelle verwenden soll. Für RISC-Maschinen muss man fast immer das Attribut "r" verwenden, da RISC-Maschinen im Gegensatz zu CISC-Maschinen, wie den Intel 80x86 kompatiblen, Operationen nur auf Registern ausführen können (Load/Store-Architektur). In [8] werden die möglichen Attribute aufgelistet und erklärt. Das Gleichheitszeichen, wie bei "=r" (c), ist bei Ausgabe-Parametern immer notwendig.

Noch ein Beispiel aus der Praxis:

```

1  cyg_uint32 camera_isrGetFrame ( cyg_vector_t pIntrVector,
2                                  cyg_addrword_t pDataAddress) {
3
4  <...>
5
6  #define NUM_REGS_USED (8)
7
8  for ( unsigned int lPixelIndex = 0;
9        lPixelIndex < CHUNK_SIZE_UINT;
10       lPixelIndex += NUM_REGS_USED) {
11
12  #if ( NUM_REGS_USED == 8)
13      asm volatile ( "ldmia %0!, { r3, r4, r5, r6, r7, r8, r9, r10}"
14                   :
15                   : "r" ( gLineBuffer + gHalfLineOffset)
16                   : "r3", "r4", "r5", "r6", "r7", "r8", "r9", "r10");
17
18      asm volatile ( "stmia %0!, { r3, r4, r5, r6, r7, r8, r9, r10}"
19                   :
20                   : "r" ( lFrameBuffer + <...> + gImageOffset)
21                   : "r3", "r4", "r5", "r6", "r7", "r8", "r9", "r10");
22  #else
23  #error rework assembly part of camera frame grabbing
24  #endif // NUM_REGS_USED == 8
25  }
26
27  <...>
28  }
```

In dieser Interrupt-Routine muss bei jedem Interrupt eine (Halb-)Zeile eines Kamera-Bildes ausgelesen und in den Arbeitsspeicher kopiert werden. Hierfür werden die besonders schnellen Multiple-Register-Befehle der ARM-CPU verwendet, die mit einem einzigen Opcode-Fetch mehrere Register lesen bzw. schreiben können. Ausgabe-Parameter werden hier nicht benötigt, deshalb sind die Zeilen 14 und 19 leer. Eingabe-Parameter ist beim Lesen die Adresse des Zeilenpuffers (Zeile 15) und beim Schreiben die entsprechende Zeile im gegenwärtig benutzten Bildpuffer (Zeile 20).

Da explizit die CPU-Register 3, 4, 5, 6, 7, 8, 9, und 10 benutzt werden, sind diese als sogenannte „clobbered“⁴-Register zu markieren (Zeile 16 und 21). Im ersten Inline-Assembler Beispiel wurden keine clobbered-Register deklariert, weil dort auch keine Register explizit benutzt worden sind und die Register-Verteilung dem Compiler überlassen wurde. Wird ein Register clobbered deklariert, so geht der Compiler davon

⁴clobbered := verhägelt, erschlagen, zerbombt

aus, dass es nach dem Inline-Assembler-Block einen anderen Inhalt hat als vor dem Block und kann den Register-Inhalt gegebenenfalls sichern.

2.5 Weitere Binutils

2.5.1 objdump - mehr über Binärdateien erfahren

2.5.2 readelf - ELF-Header von Binärdateien ansehen

Das Programm `readelf` gibt den ELF-Header einer ELF-Binärdatei in lesbarer Form aus.

2.5.3 objcopy - zwischen Binärformaten konvertieren

Hat man ein fertiges, für ROM- oder ROMRAM-Startup kompiliertes Executable und will dieses in einen Flash- oder EPROM-Baustein programmieren, so muss man das Executable in ein anderes Format umwandeln.

Ein normales ELF-Executable enthält nämlich Informationen für den Programm-Lader eines Betriebssystems und kann ohne einen solchen Lader⁵ nicht gestartet werden. Das Executable muss also in ein Format umgewandelt werden, das direkt ohne einen Lader ausgeführt werden kann. Mit dem Programm `objcopy` kann man Objekt-Dateien, Executables und Daten-Dateien in andere Formate konvertieren.

Beispiel: Das ARM-ELF-Executable `my_application(.exe)` soll in einen Flash-Baustein programmiert werden. Das Zielformat hierfür ist `binary`.

```
arm-elf-objcopy -O binary my_application(.exe) my_application.bin
```

Die Datei `my_application.bin` kann nun mit einem Programmiergerät oder einem JTAG-Interface in den nicht-flüchtigen Speicher programmiert werden. In einer `binary`-Datei sind nur noch „nackter“ Code und Daten enthalten.

2.5.4 size - die wahre Größe von Binärdateien erfahren

Sieht man sich die Größe einer Binär-Datei im Datenträger-Verzeichnis an, so bekommt man zuerst einen Schrecken, denn selbst primitive Applikationen scheinen sehr groß zu sein. Insbesondere Debug-Informationen, die ja innerhalb des Executables gespeichert sind, benötigen jedoch erheblichen Speicherplatz.

Beispiel: Die einfache eCos-Applikation „Hello world!“ hat laut Verzeichnis-Eintrag eine Größe von 1461100 Bytes. Der Aufruf `arm-elf-size hello` gibt folgendes aus:

```
text  data  bss   dec   hex filename
89776 3232  9556 102564 190a4 hello
```

Das bedeutet, dass `hello` in der Code-Sektion (`.text`) 89776 Bytes und in der Daten-Sektion (`.data` + `.rodata`) 3232 Bytes groß ist. Die Sektion `.bss` ist 9556 Bytes groß. In ihr werden nach dem Start der Applikation statisch angelegte Felder etc. angelegt.

Die tatsächliche Größe der Applikation im Arbeitsspeicher ist also 102564 Bytes.

2.5.5 strip - Debugger-Informationen aus Binärdateien entfernen

Wie der vorherige Abschnitt zeigt, kann ein Executable viel Ballast enthalten, den man in einer fertigen Nicht-Entwicklungs-Version nicht mehr benötigt.

⁵ROM-Monitore, wie der GDB-Stub oder RedBoot, sind in dieser Hinsicht auch Lader.

Kapitel 3

gdb/Insight - der Debugger

3.1 Aufruf

Die allgemeine Aufruf-Syntax ist:

```
<<ARCH>-<BIN_FMT>->gdb <OPTIONEN> <EXECUTABLE> <<REF>>
```

Wobei gilt:

ARCH	CPU-Architektur
BIN_FMT	Binär-Format
REF	Eine Prozess-ID oder ein Core-Dump ² . Gibt man eine Prozess-ID an, so verbindet sich der Debugger mit einem laufenden Prozess. Gibt man einen Core an, so kann man die Absturz-Ursache ergründen. EXECUTABLE muss zum laufenden bzw. abgestürzten Programm geführt haben d.h. mit diesem identisch sein.

Wichtige Optionen sind:

-nw	Ist der Debugger der um eine grafische Oberfläche erweiterte Insight, so kann man diesen mit dieser Option im reinen Text-Modus starten, wie vom nicht-erweiterten gdb gewohnt.
-d <DIR>	Fügt das Verzeichnis DIR zur Menge der Suchpfade nach Quell-Dateien hinzu. Dies kann beim Source-Level-Debugging wichtig werden, wenn der Debugger Quell-Dateien nicht finden kann.

Aber meist reicht einfach folgender Aufruf:

```
gdb <EXECUTABLE> bzw. arm-elf-gdb <EXECUTABLE>
```

3.2 Das Insight Haupt-Fenster

Abbildung 3.1 zeigt das Hauptfenster von Insight, der grafischen Variante des GNU-Debuggers gdb. Im Tabelle 3.1 werden die markierten Funktionen erläutert.

²UNIX-Betriebssysteme können das Speicher-Abbild eines abgestürzten Prozesses in einer Datei speichern. Diese nennt man Core bzw. Core-Dump.

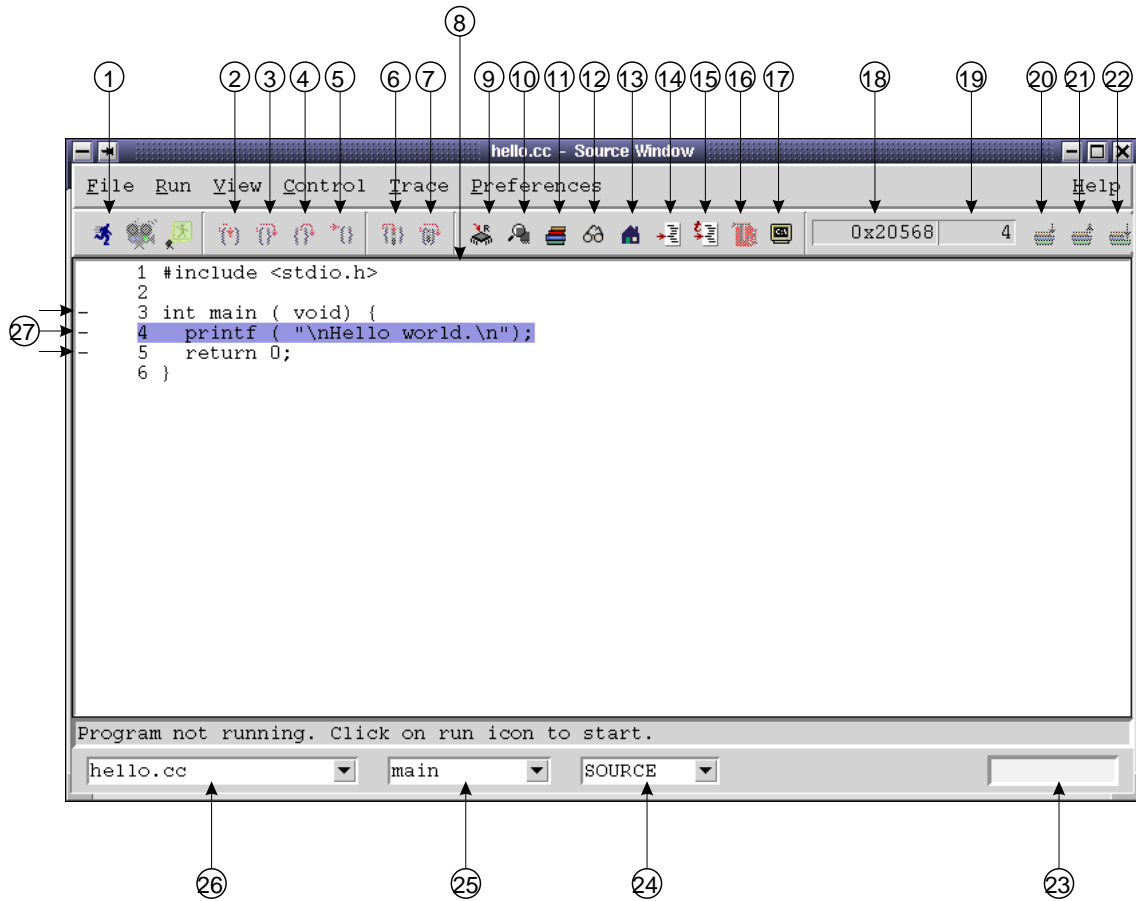


Abbildung 3.1: Das Insight-Hauptfenster

Nr.	Hotkey	Bedeutung
1	R	Startet man das Programm. Soll das Programm auf einem anderen Rechner als dem Host laufen, so wird es ggf. vorher auf das Target herunter geladen. Läuft das Programm, so wechselt das Symbol vom rennenden Männchen zu einem Stopp-Zeichen. Ein Maus-Klick auf Stopp-Zeichen hält das Programm an der gegenwärtig ausgeführten Instruktion an und versetzt es in den Einzelschritt-Modus. Fast alle der folgenden Funktionen machen nur im Einzelschritt-Modus Sinn, bzw. sind ansonsten gar nicht aufrufbar.

Ablaufsteuerung im Einzelschritt-Modus:

- 2 S Führt eine einzelne Hochsprachen-Instruktion aus, wenn die Ansicht *SOURCE*, *MIXED* oder *SRC+ASM* eingeschaltet ist und sich der Maus-Fokus im Hochsprachen-Quelltext-Fenster befindet. Um den Einzelschritt-Modus zu benutzen, muss das Programm vorher durch einen Breakpoint, Ctrl-C oder einen Klick auf das Stopp-Zeichen unterbrochen worden sein. Ist die gegenwärtige Instruktion ein Unterprogramm-Aufruf, so wechselt der Kontext zum Unterprogramm. Der Programmzeiger springt dann auf die erste Zeile des Unterprogramms.
- 3 N Die gegenwärtige Instruktion wird ausgeführt und der Programmzeiger springt in die nächste Zeile, auch wenn die Instruktion ein Unterprogramm-Aufruf war. In diesem Fall wird das Unterprogramm nicht im Einzelschritt-Modus abgearbeitet, sondern „normal“. D.h. der Kontext wechselt nicht in das Unterprogramm, sondern bleibt der, der gerade ausgeführten Instruktion.
- 4 F Beendet den Programm-Block, in dem man sich gerade befindet. Befindet man sich gerade innerhalb einer Schleife, so wird die Schleife komplett, d.h. nicht im Einzelschritt-Modus, abgearbeitet und der Programmzeiger springt zur ersten Instruktion, die der Schleife folgt. Befindet man sich in einem Unterprogramm, so wird das gesamte Unterprogramm abgearbeitet und der Programmzeiger springt zur Rücksprung-Adresse des Unterprogramms.
- 5 C Setzt die Programm-Ausführung bis zum nächsten Breakpoint bzw. bis zum Programm-Ende, einer Endlos-Schleife oder einem Absturz fort.
- 6 S Führt eine einzelne Assembler-Instruktion aus, wenn die Ansicht *MIXED*, *SRC+ASM* oder *ASSEMBLY* eingeschaltet ist und sich der Maus-Fokus im Assembler-Quelltext-Fenster befindet.
- 7 N Dies entspricht 3, lediglich bezogen auf Assembler-Code.
- 8 In diesem Fenster wird im Einzelschritt-Modus der Quelltext des gegenwärtig ausgeführten Programm-Moduls angezeigt. Für alle Tastatur-Shortcuts muss dieses Fenster den Fokus haben. Hat es den Fokus, so ist es von einem schwarzen Rahmen umgeben.

Die Funktionen der Buttons 9 bis 17 sind auch über das Menü *View* verfügbar:

- | | | |
|----|--------|---|
| 9 | CTRL-R | Öffnet das CPU-Register-Fenster. Im Einzelschritt-Modus wird in diesem Fenster der Inhalt aller CPU-Register angezeigt. |
| 10 | CTRL-M | Öffnet das Speicher-Inhalt-Ausgabe-Fenster. In diesem Fenster kann man sich den Inhalt beliebiger Speicher-Adressen ansehen. |
| 11 | CTRL-S | Öffnet das Stack-Ausgabe-Fenster. In diesem Fenster kann man sich den Inhalt des gegenwärtigen (Threads!) Programm-Stacks ansehen. |
| 12 | CTRL-W | Öffnet das Ausdruck-Ausgabe-Fenster. In diesem Fenster wird der gegenwärtige Inhalt von unter Beobachtung stehenden Ausdrücken, z.B. Strukturen, ausgegeben. |
| 13 | CTRL-L | Öffnet ein Fenster, in dem gegenwärtig benutzte lokale/Stack-Variablen und deren Wert aufgelistet werden. |
| 14 | CTRL-B | Öffnet das Breakpoint-Übersichts-Fenster. In diesem Fenster werden alle Breakpoints aufgelistet. Existierende Breakpoints können in diesem Fenster auch deaktiviert oder aktiviert werden. |
| 15 | CTRL-T | Entspricht 14 (? , möglicherweise Programm-fehler) |
| 16 | CTRL-U | Öffnet ein Fenster in dem der Tracedump an |
| 17 | CTRL-N | Öffnet das Konsolen-Fenster. Alle Programm-Ausgaben nach Standard-Ausgabe und Standard-Fehler-Ausgabe tauchen in diesem Fenster auf. Außerdem erscheinen hier die Text-Ausgaben des Debuggers. |
| 18 | | die Adresse der aktuellen Instruktion |
| 19 | | die aktuelle Zeilennummer im Quell-Code |
| 20 | | Hat in der Theorie die gleiche Bedeutung wie 21, lediglich umgekehrt, führt in der Praxis jedoch zum Programmabsturz. |
| 21 | | Klickt man auf dieses Symbol, so wird die Stelle im Source-Code orange markiert, die das aktuelle Unterprogramm aufgerufen hat. Der Stack-Frame ³ wechselt zum Frame des aufrufenden (Unter-)Programms (Da der Stack in Richtung kleinerer Adressen wächst, bedeutet aufwärts gehen sich in die Programm-Vergangenheit zu bewegen.). Diese Funktion führt bei der benutzten Insight-Version 5.00 häufig zum Programmabsturz. |
| 22 | | Programmabsturz |

23	Eingabefeld für Text-Suche, Fortschrittsanzeige beim Download
24	Auswahlfeld für den Ansichts-Modus des Quell-Fensters. <i>SOURCE</i> zeigt den Quell-Code der aktuell ausgewählten Datei, bei einer C-Datei also C-Code. <i>ASSEMBLY</i> zeigt die in Assmblar-Code übersetzte Datei. <i>MIXED</i> und <i>SRC+ASM</i> zeigen sowohl den Quell-Code als auch den übersetzten Assembler-Code. Bei <i>SRC+ASM</i> wird das Quell-Fenster in zwei übereinander liegende Fenster für Quell- bzw. Assembler-Code aufgeteilt.
25	Auswahlfeld für eine Funktion des aktuellen Quell-Textes
26	Auswahlfeld für eine Quell-Dateien aus denen die Applikation besteht
27	In allen Zeilen, in denen dieser Strich steht, kann ein Breakpoint oder ein Tracepoint gesetzt werden. Man muss lediglich den Strich anklicken einen normalen Breakpoint zu setzen. Ist ein Breakpoint gesetzt, so erscheint ein kleines rotes Quadrat statt des Strichs und ein Eintrag im Breakpoint-Übersichts-Fenster. Will man einen temporären Breakpoint oder einen Tracepoint setzen, so muss man mit der rechten Maustaste auf den Strich klicken. Es öffnet sich ein Kontext-Menü in dem man Tracepoint oder Breakpoints auswählen kann.

Tabelle 3.1: Legende zur Abbildung 3.1

3.3 Download vom Host zum Target

Vor dem Donwload zu einem Target muss man den Typ des Targets, die Art der Verbindung zwischen Host und Target und, abhängig von der Art der Verbindung, die Geschwindigkeit der Verbindung einstellen. Das entsprechende Dialog-Fenster wird in Abbildung 3.2 gezeigt. Man erreicht es über das Menü *File->Target Settings...* bzw. wird es implizit geöffnet, wenn man *Download* oder *Run* aufruft ohne vorher ein Target ausgewählt zu haben.

Der Target-Typ ist der Typ des ROM-Monitors auf der Ziel-Hardware und bestimmt das Protokoll mit dem Debugger und ROM-Monitor kommunizieren. Welche ROM-Monitore von Insight unterstützt werden, hängt auch davon ab für welches CPU-Architektur Insight kompiliert worden ist. Ist der ROM-Monitor RedBoot oder der GDB-Stub, so wählt man als Target *Remote/Serial* bei einer RS232-Verbindung zwischen Host und Target, bzw. *Remote/TCP* bei einer Netzwerk-Verbindung mit TCP/IP-Protokoll.

Hat der Verbindungsaufbau zwischen Host und Target funktioniert, so erscheint im Konsolen-Fenster eine ROM-Adresse des Targets im Hex-Format gefolgt von zwei Fragezeichen. Während des Downloads wird für jede Sektion des Executables eine Meldung ausgegeben.

3.4 Watches - den Inhalt von Variablen beobachten

Indem man im Quell-Code-Fenster des Hauptfensters (Abbildung 3.1) mit der rechten Maustaste auf eine Variable klickt, öffnet man ein Kontext-Menü, das den Punkt *Add <VARIABLE> to Watch* enthält (Abbildung 3.3). Wählt man diesen aus, so steht die ausgewählte Variable bis auf weiteres unter Beobachtung und erscheint im in Abbildung 3.4 gezeigten Fenster. Kann man die Variable nicht Auswählen, so hat der Compiler den Code soweit optimiert, dass sie im Executable nicht mehr enthalten ist. In diesem Fall sollte man das Programm ohne Optimierung übersetzen.

Man kann alle Variablen, Objekte (Struktur- bzw. Klassen-Instanzen) und Arrays beobachten bzw. ihren Inhalt betrachten. Variable wird im folgenden synonym für Objekt, Array und Variablen atomaren Typs verwendet.

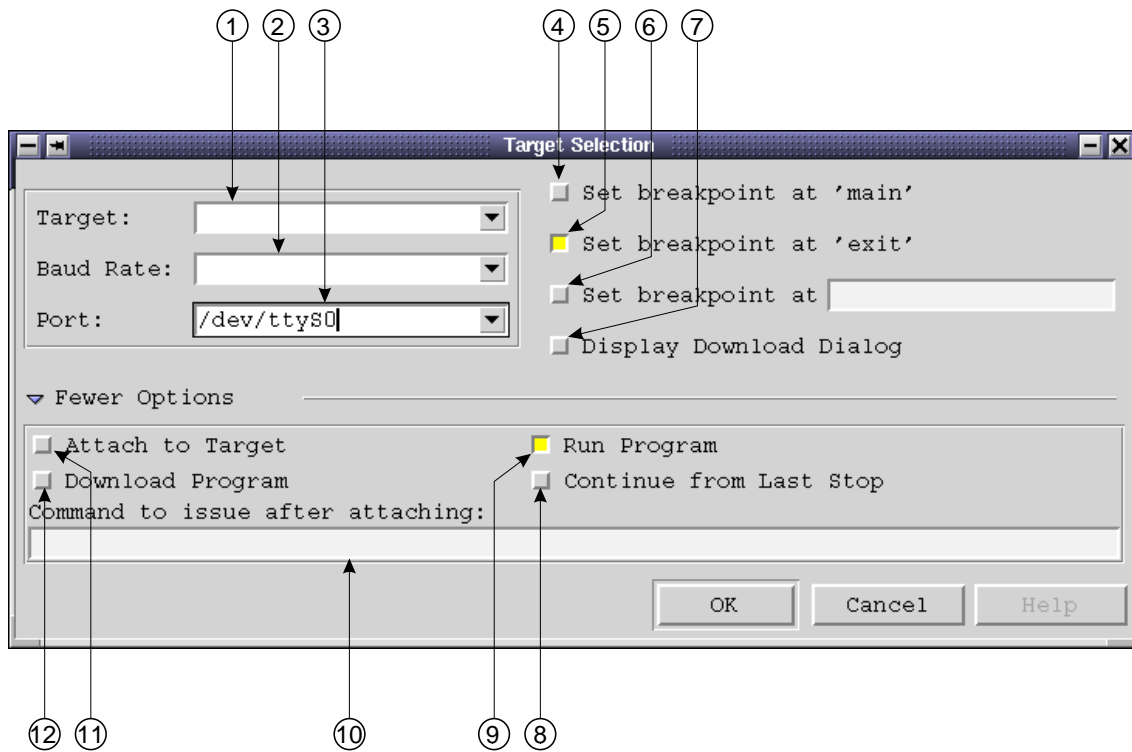


Abbildung 3.2: Der Target-Settings Dialog

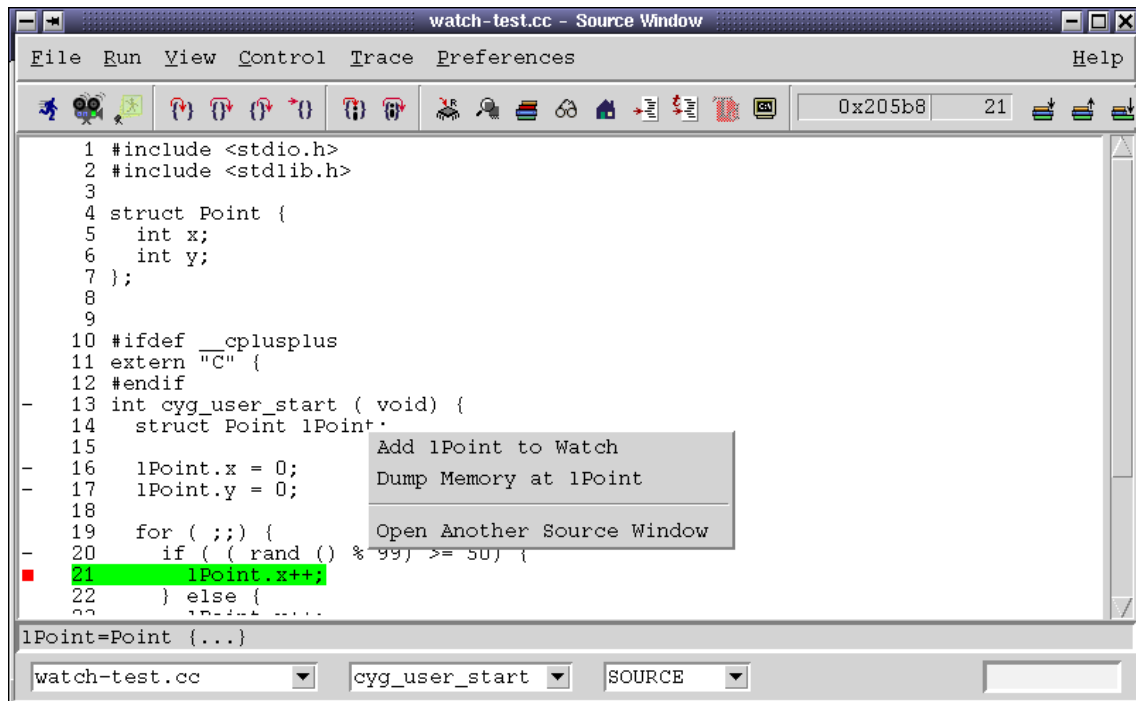


Abbildung 3.3: Eine Variable unter Beobachtung stellen

- 1 Auswahlfeld für den Target-Typ
- 2 Auswahlfeld für die Verbindungsgeschwindigkeit bei serieller Verbindung
- 3 Auswahlfeld für das logische Gerät, über das die Verbindung läuft, also z.B. der Name einer RS232-Schnittstelle
- 4 Setzt man diese Checkbox, so wird implizit ein Breakpoint am Beginn der Funktion `main` gesetzt. Dies ist z.B. für eCos-Applikationen nutzlos, da diese meist mit der Funktion `cyg_user_start` beginnen und `main` erst danach aufgerufen wird.
- 5 Setzt man diese Checkbox, so wird implizit ein Breakpoint am Beginn der Bibliotheks-Funktion `exit` gesetzt.
- 6 Setzt man diese Checkbox, so werden implizit am Beginn aller im nachfolgenden Eingabefeld angegebenen Funktionen Breakpoints gesetzt. Die Funktionsnamen sind ohne nachfolgende Klammern oder Parameter einzugeben. Die einzelnen Funktionsnamen werden durch Leerzeichen getrennt.
- 7 Setzt man diese Checkbox, so wird während des Downloads ein Status-Fenster geöffnet, das den Fortschritt des Downloads für jede Sektion des Executables einzeln anzeigt.

Der Target-Settings Dialogs bietet noch erweiterte Funktionalität an, die aber nur selten benötigt wird:

- 8 Setzt man diese Checkbox, so wird das Programm nach dem letzten Stopp fortgeführt. Diese Checkbox ist mit der Checkbox 9 XOR-verknüpft.
- 9 Setzt man diese Checkbox, so wird das Programm nach dem Download direkt am Anfang gestartet. Diese Checkbox ist mit der Checkbox 8 XOR-verknüpft.
- 10 In diesem Eingabefeld eingegebene GDB-Kommandos werden nach dem Verbindungsaufbau (Handshake) ausgeführt.
- 11 Setzt man diese Checkbox, so wird immer ein Handshake durchgeführt.
- 12 Setzt man diese Checkbox, so wird das Executable auf jeden Fall auf das Target herunter geladen.

Tabelle 3.2: Legende zur Abbildung 3.2

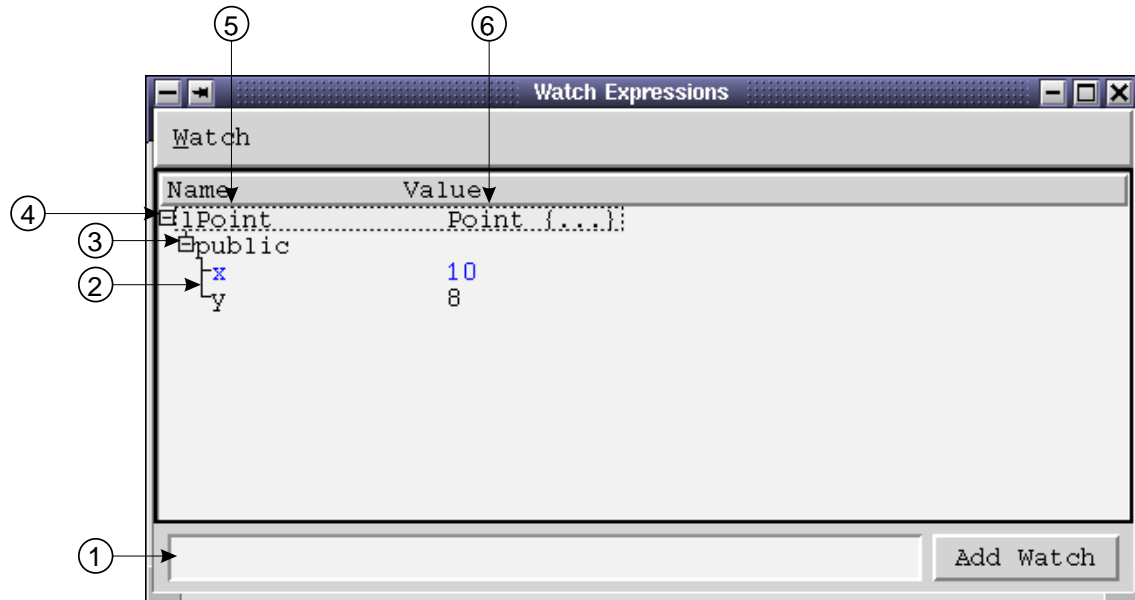


Abbildung 3.4: Das Watch-Expressions Fenster

- 1 In diesem Eingabefeld kann man den Namen von Variablen eingeben, die man ebenfalls beobachten will. Der Button *Add Watch* führt die Aktion aus.
- 2 *x* und *y* sind in diesem Beispiel die Atome der Struktur, die rechte Spalte zeigt deren derzeitige Werte.
- 3 Klickt man auf diesen Schalter, so öffnet bzw. schließt sich der Baum der *public*-Elemente dieser Struktur bzw. Klasse. Existieren *protected*- oder *private*-Elemente, so werden diese entsprechend in einem eigenen Teilbaum abgebildet. Ist das Programm ein reines C-Programm, so gibt es diese Sichtbarkeits-Unterscheidungen natürlich nicht.
- 4 Klickt man auf diesen Schalter, der nur sichtbar ist, wenn die Variable nicht atomar ist, so klappt der Inhalt der Variablen als Baum auf bzw. zu.
- 5 Name der Variablen, die unter Beobachtung stehen.
- 6 Typ der Variablen in der gleichen Zeile

Tabelle 3.3: Legende zur Abbildung 3.4

3.5 Breakpoints und Tracepoints

3.5.1 Breakpoints

Der gdb unterscheidet zwei Arten von Breakpoints. Ein normaler Breakpoint wird gesetzt wenn man mit der *linken* Maustaste auf den kleinen Strich am Anfang einer Source-Zeile im Source-Fenster klickt. Solch ein Breakpoint wird durch ein kleines rotes Quadrat sichtbar gemacht. Immer wenn das Programm die Zeile mit diesem Breakpoint erreicht, wird es angehalten und ist im Einzelschritt-Modus.

Oft benötigt man jedoch einen Breakpoint nur einmal. Um diesen nicht explizit deaktivieren zu müssen, was aufwändig sein kann, weil man vielleicht erst die Quell-Datei suchen muss, gibt es den temporären Breakpoint. Diesen setzt man, indem man mit der *rechten* Maustaste auf den Strich klickt und *Set Temporary Breakpoint* aus dem Popup-Menü auswählt. Statt eines roten Quadrats erscheint ein orangenes. Erreicht das Programm zum ersten Mal eine Zeile mit einem temporären Breakpoint, so wird es an dieser Zeile angehalten, der Breakpoint wird jedoch automatisch deaktiviert, sodass das Programm beim nächsten Erreichen dieser Zeile nicht mehr anhält.

3.5.2 Tracepoints

Gerade bei der Entwicklung von Echtzeit-Applikationen ist es oft nicht sinnvoll den Programm-Ablauf zum Debuggen zu unterbrechen, da ein solcher Eingriff das Timing einer Applikation völlig verfälscht. Um dieses Problem zu umgehen, kennt der gdb den Tracepoint. Erreicht eine Applikation einen Tracepoint, so wird der Inhalt bestimmter, d.h. für diesen Tracepoint spezifizierter, Variablen aufgezeichnet. Die Aufzeichnung kann dann später betrachtet werden.

Um einen Tracepoint zu setzen geht man genauso vor wie beim Setzen eines temporären Breakpoints, wählt jedoch *Set Tracepoint* aus dem Popup-Menü. Ein Tracepoint wird im Source-Fenster durch ein kleines lila Quadrat dargestellt. Da Tracing instabil bzw. unvollständig in Insight integriert ist, ist es teilweise notwendig die Kommandos per Tastatur über die gdb-Konsole einzugeben. Die Benutzung von Tracepoints wird in [3], Kapitel 9 ausführlich erklärt.

Kapitel 4

Cygwin - unter Windows arbeiten wie unter Unix

4.1 Was ist Cygwin?

Cygwin ist ein Software-Paket von RedHat, das eine Unix-Konsolen-Umgebung unter „höher entwickelten“ Windows-Betriebssystemen, wie Windows NT/2000/XP, emuliert und die Portierung von Unix-Text-Anwendungen vereinfacht. Cygwin ist die Grundlage für die gesamte GNUPro-Toolchain unter Windows und muss zuerst, vor allen anderen GNU-Tools installiert werden.

Im Paket enthalten sind u.a. Shells (z.B. bash), make, der gcc für die Übersetzung von Programmen unter Windows (Dieser ist nicht zu verwechseln mit dem arm-elf-gcc oder anderen Cross-Compilern, diese müssen gesondert installiert werden!), diversen üblichen Text- und Makro-Werkzeugen (z.B. awk und sed), Editoren und natürlich die üblichen Datei-Utilities, wie ls, cp, rm usw.

Cygwin abstrahiert vom (dämlichen) Laufwerks-Buchstaben-Konzept unter Windows und erlaubt es Festplatten-Partitionen, bzw. alle logischen Windows-Laufwerke, wie unter Unix an beliebige Stellen des Datei-Systems zu mounten. Allerdings funktioniert diese Abstraktion nur für Cygwin-Applikationen.

4.2 Installation

Das Paket ist ausschließlich als selbstentpackende EXE-Datei mit Installations-Assistent verfügbar. Man muss lediglich die Datei <http://sources.redhat.com/cygwin/setup.exe> herunterladen, ausführen und die Fragen des Installations-Assistenten beantworten.

Anhang A

Probleme und Lösungen

A.1 Obskure Fehlermeldungen beim Umgang mit Text-/Quell-Dateien unter Cygwin

Unter Cygwin werden Windows-Laufwerke mittels `mount`, ähnlich wie unter Linux, in die Verzeichnis-Hierarchie eingehängt. `mount` unterstützt dabei zwei Modi, *binary* und *text*. Generell sollte man den Text-Modus benutzen, da dieser wohl intern einen Filter verwendet, um die DOS/Windows Zeilen-Ende-Marke in die UNIX Zeilen-Ende-Marke zu übersetzen und umgekehrt.

A.2 Der Build-Prozess unter Cygwin ist langsam

Compiler-Läufe unter Cygwin dauern deutlich länger als auf der selben Maschine unter Linux. Dies liegt zum (geringeren) Teil an der schlechteren Performance von Windows und zum größten Teil an der Implementation von Cygwin. Cygwin bildet nämlich die UNIX-System-Aufrufe mit einer Wrapper-Bibliothek um die Windows-API herum nach. D.h. ruft eine Cygwin-Applikation eine System-Funktion auf, so wird erst eine Cygwin-Funktion aufgerufen, die wiederum eine Windows-Funktion aufruft. Hierfür gibt es keine Lösung, außer Linux zu benutzen.

A.3 Beim Linken tritt eine undefined reference bezüglich einer Funktion auf obwohl diese im Programm vorhanden ist

Das Symbol-Format der vom Compiler generierten Symbole für Funktionen bzw. Methoden in Objekt-Dateien unterscheidet sich zwischen C und C++. Wird von einem C-Modul eine Funktion eines C++-Moduls aufgerufen, so generiert der Compiler im C-Modul eine Referenz zu dieser C++-Funktion im C-Format. Das bedeutet, dass der Linker diese Referenz nicht auflösen kann. Die Lösung für dieses Problem ist, die von C-Modulen aus aufgerufenen C++-Funktionen als `extern "C"` zu deklarieren, damit der C++-Compiler für diese Funktionen C-kompatible Symbole erzeugt. Beispiel:

```
#ifdef __cplusplus
extern "C" {
#endif
void aFunction (<...>) {
    <...>
}
#ifdef __cplusplus
}
#endif
```

Der C++-Compiler wird für die Funktion `aFunktion` ein C-kompatibles Symbol generieren, sodass diese auch von C-Code aus aufgerufen werden kann.

Literaturverzeichnis

- [1] **GNU linker ld version 2.10 documentation**
<http://sources.redhat.com/binutils/docs-2.10/ld.html>
- [2] **GNU assembler as version 2.10 documentation**
<http://sources.redhat.com/binutils/docs-2.10/as.html>
- [3] **GNU gdb documentation**
<http://sources.redhat.com/gdb/documentation>
- [4] **Cygwin Homepage**
<http://sources.redhat.com/cygwin>
- [5] Andreas Bürgel
eCos Applikations-Entwicklung
http://www.andreas-buergel.de/documents/ecos_applications.pdf
- [6] **GNU-Pro Installation**
Linux: <http://sources.redhat.com/ecos/install-linux.html>
Windows: <http://sources.redhat.com/ecos/install-windows.html>
- [7] **linuxassembly.org**
<http://www.linuxassembly.org>
- [8] Robin Miyagi
Introduction to GCC Inline ASM
<http://www.linuxassembly.org/rmiyagi-inline-asm.txt>
- [9] E. Siever, S. Spainhour, S. Figgins, J.P. Hekman
Linux in a Nutshell
Verlag O'Reilly
ISBN 0-596-00025-1