Workshop: Crosscompiling für Embedded-Systeme

GNU-Tools mal kreuzweise

René Rebe

© Bruce Amos, Fotolia

Beim Cross-Kompilieren für ein Embedded-System übersetzen Sie Ihren Quellcode für eine andere CPU als die im PC des Entwicklers eingebaute. Der folgende Workshop erklärt zwei Wege, wie das mit dem GCC als Crosscompiler zu bewerkstelligen ist.

Wenn Sie für ein Embedded-System, zum Beispiel einen Router, Telefon, Steuergerät und so weiter, Software entwickeln wollen, brauchen Sie eine zur Hardware passende Kette aus Werkzeugen, also Crosscompiler, Assembler, Linker und Debugger. Die gibt es fertig als Open Source oder proprietär, es kann aber auch zweckmäßig sein, selbst einen GCC [1] zu konfigurieren. Entweder weil es für die Zielplattform keinen vorkompilierten Compiler gibt, weil dieser zu alt ist und neue CPU-Funktionen nicht unterstützt oder um selbst am GCC Veränderungen vorzunehmen.

Obwohl die GNU Compiler Collection und die Binutils (Assembler, Linker und so weiter, [2]) Autoconf verwenden, ist ein Cross-GCC nicht ganz einfach einzurichten, denn er will genau auf das Zielbetriebssystem und die CPU abgestimmt sein: Als Erstes extrahieren Sie die System-Header des Betriebssystems und der C-Bibliothek. Das ist notwendig, weil diese Header die Datenstrukturen und Systemaufrufe des Betriebssystems für die jeweilige CPU definieren und bereits der Compiler sie benötigt.

Entweder zu Fuß oder mit dem Crosstool-Projekt

Danach kompilieren Sie mit den extrahierten Headerdateien die Binutils und den GCC. Mit dem neuen Crosscompiler übersetzen Sie wiederum die C-Bibliothek. Der Kasten „Cross-GCC selbst übersetzen“ liefert Ihnen eine Schritt-für-Schritt-Anleitung. Wegen des erheblichen Konfigurationsaufwands rief Dan Kegel, der zurzeit bei Google arbeitet, das Crosstool-Projekt [3] ins Leben. Crosstool automatisiert die Konfigurationsschritte und integriert zudem diverse Patches sowie Tests, um die korrekte Funktion zu überprüfen.

Cross-GCC selbst übersetzen

Das folgende Beispiel benutzt nicht das fertige Kit Crosstool [3], sondern Sie übersetzten den GCC selbst. Die Anleitung in Listing 1 benutzt »$DESTDIR« als Pfad, in dem Sie das Embedded-System erstellen wollen, und »$TOOLSDIR« als Aufenthaltsort für den Crosscompiler und die anderen Werkzeuge. »$TOOLSDIR« könnte »/home/user/arm-tools« sein und »$DESTDIR« »/home/user/arm-linux« - ganz nach Belieben. Den Pfad »$TOOLSDIR/bin« müssen Sie der »$PATH«-Variablen hinzufügen, damit die Shell den Crosscompiler und Assembler findet.
Das Beispiel verwendet ARM als Zielsystem (Abbildung 1). Da der Compiler Details des Zielsystems benötigt, extrahieren Sie mit den Zeilen 1 bis 4 zunächst einige Header aus

dem Linux-Kernel. Die Zeilen 5 bis 10 erledigen das Gleiche für die Glibc. Jetzt können Sie die gewohnten Configure-Skripte der Binutils und des GCC-Pakets verwenden, müssen jedoch darauf achten, die richtige Zielplattform anzugeben. Für die Binutils gehen Sie wie in den Zeilen 12 bis 15 vor, für den GCC gelten die Kommandos aus den Zeilen 16 bis 20.

Der ARM-Compiler ist geschafft

Nun haben Sie den Crosscompiler »arm-linux-gnu-gcc« fertig gebaut und können andere Pakete übersetzen. Als erstes Paket müssen Sie in den Zeilen 22 bis 25 unbedingt die C-Bibliothek kompilieren, damit andere Programme dagegen linken können. Ab jetzt lässt sich eigener C- und Assembler-Code für das Zielsystem übersetzen und auch linken. Um zu überprüfen, ob das wirklich klappt, füttern Sie den »arm-linux-gnu-gcc« mit einem gängigen Testprogramm:

```
arm-linux-gnu-gcc -o hello hello.c
file hello
```

Auf das File-Kommando von eben müsste eine Ausgabe der Art

```
hello: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically li
```

erfolgen, die ein ARM-Binary meldet. Geschafft: Sie haben Ihr eigenes Embedded-Kit!

Sind Sie auf einem der beiden Wege zu einem Crosscompiler gelangt, können Sie Ihre Software für die Zielplattform übersetzen. Zuerst setzen Sie die Umgebungsvariable »$CC« auf den Namen des Crosscompiler-Binary und beim Configure die Zielarchitektur hier für ARM:

```
export CC=arm-linux-gnu-gcc
./configure --host=arm-linux-gnu
```

Die verschiedenen Arten von Zuweisungen überschreiben eventuell innerhalb der Makefiles die Umgebungsvariable »$CC« wieder. In diesen Fällen reicht es gewöhnlich, »$CC« mittels »make«-Argument zu setzen:

```
make CC=arm-linux-gnu-gcc
```

Zuweisungen in Makefiles sind nun sicher, es sei denn, eine explizite »override«-Anweisung hagelt dazwischen.

Wenn Sie neben dem Compiler auch Linker (LD), Assembler (AS) oder andere Programme in der Toolchain verwenden wollen, geben Sie diese mit

```
export LD=arm-linux-gnu-ld
export AS=arm-linux-gnu-as
make CC=arm-linux-gnu-gcc LD=arm-linux-gnu-ld AS=arm-linux-gnu-as
```

bekannt. Das Installations-Zielverzeichnis erfährt Make mit:

```
make DESTDIR=/home/user/arm-linux
```

Da ohne »DESTDIR« schnell Dateien des eigenen Systems dran glauben, sollten Sie in der Sektion »install:« des Makefile prüfen, ob es die Angabe unterstützt oder ob die

Variable dort nicht anders heißt, zum Beispiel »INSTALL_ROOT«. Achtung: Verwechseln Sie »DESTDIR« nicht mit dem »--prefix«-Argument von Configure! Obwohl der Aufruf

```
./configure --prefix=/home/user/arm-linux
```

die Dateien, die aus dem »make install«-Lauf resultieren, in dasselbe Zielverzeichnis installiert, ist die Variante mit

```
./configure ; make installU
 DESTDIR=/home/user/arm-linux
```

zu bevorzugen. Diese setzt nämlich die Pfade in Ihrem Programm. Fehlt sie, kann es Daten wie Bilder, Plugins, Einstellungen und so weiter nicht laden, da sie auf dem Zielsystem nicht relativ zu »/home/user/arm-linux« liegen.

Ein funktionierender Compiler ist ein guter Anfang, allerdings gilt es darüber hinaus noch, die eigentlichen Programme zu übersetzen. Viele Projekte entwickeln Software allerdings ohne Rücksicht auf Crosscompiling, und so lauern in den Buildsystemen allerlei Hindernisse.

Listing 1: Schritte zum GCC

```
01 tar xvfj linux-$ver.tar.bz2
02 cd linux-$ver
03 mkdir -p $DESTDIR/usr/include/asm
04 make ARCH=arm INSTALL_HDR_PATH=$DESTDIR/usr headers_install
05 tar xvfj glibc-$ver.tar.bz2
06 cd glibc-$ver
07 mkdir objdir; cd objdir
08 CC=gcc ../configure --host=arm-linux-gnu --prefix=/usr --with-heade
09 make -k cross-compiling=yes DESTDIR=$DESTDIR install-headers
10 touch $DESTDIR/usr/include/gnu/stubs.h
11
12 tar xvfj binutils-$ver.tar.bz2
13 cd binutils-$ver
14 ./configure --target=arm-linux-gnu --prefix=$TOOLSDIR
15 make; make install
16 tar xvfj gcc-$ver.tar.bz2
17 cd gcc-$ver
18 mkdir objdir; cd objdir
19 ../configure --target=arm-linux-gnu --disable-cpp --disable-shared
   --enable-languages="c" --prefix=$TOOLSDIR
   --with-headers=$DESTDIR/usr/include
20 make all; make install
21
22 cd glibc-$ver/objdir
23 ../configure --host=arm-linux-gnu --prefix=/usr --with-header=$DEST
24 make
25 make DESTDIR=$DESTDIR install
```

Abbildung 1: Der Workshop verwendet als Beispiel für die Zielplattform ARM. Im Bild der ARM-Singleboard-Computer Taskit Portux920T mit dem Controllerchip Atmel AT91RM9200, zwei seriellen Schnittstellen, 100 MBit/s Ethernet und JTAG-Unterstützung.

Alle Achtung

Ein Problem ist der Mix von Host- und Target-Compiler. Einige Makefiles enthalten »gcc«- oder »cc«-Aufrufe für einzelne oder alle Dateien. Auch vom Entwickler bevorzugte maschinenabhängige Optimierungen stellen sich oft als problematisch heraus. Das führt entweder zu Programmen, die nicht auf dem Zielsystem ausführbar sind, oder - falls nur einige Dateien betroffen sind - zu Inkompatibilitätsfehlern beim Linken. Ein möglicher Ausweg ist das Ändern von Regeln sowie das Setzen der »$(CC)«-Variablen.

Einige Autoconf-Skripte versuchen übersetzte Testprogramme auszuführen, um Charakteristika des Systems zu erkennen. Cross-kompilierte Programme lassen sich aber nicht auf dem Entwicklungsrechner starten. Hier hilft es, das »configure«-Skript anzupassen oder zur Not fixe Werte einzutragen, entweder direkt in das Configure-Skript oder in die Cachedatei »config.cache«.

Einige Programme generieren wiederum Quellcode, zum Beispiel systemspezifische Werte, aber auch vorbereitete Tabellen für mathematische Berechnungen. Falls das Programm hierfür nicht auf Skripte, sondern auf das Übersetzen von Hilfsprogrammen angewiesen ist, kommt es dann zu Problemen, wenn der Systemcompiler durch andere Wortbreiten, Endianness oder Genauigkeiten inkompatible Werte generiert. Als letzte Möglichkeit bleibt auch hier, das cross-kompilierte Programm auf dem Zielsystem auszuführen, um eine korrekte Ausgabe zu erhalten.

Zielkonflikte an der Quelle

Eine weitere Hürde stellen »pkg-config« sowie die älteren »*-config«-Skripte zur Ermittlung von installierten Programmen und deren Optionen auf. Ohne manuelle Nachhilfe würden die Configure-Skripte einfach die Daten des Buildsystems verwenden - manche der Pakete sind jedoch auf dem Embedded-System vielleicht gar nicht vorhanden. Durch Standardisierung von »pkg-config« reicht es, wenn Sie zumindest für diese eine Variable die Pfade des Zielsystems setzen:

```
PKG_CONFIG_PATH="/home/user/arm-linux/usr/lib/pkgconfig"
```

Alle weiteren Pfade, in die Sie Bibliotheken installieren, können Sie entsprechend durch Doppelpunkte getrennt angeben. Für ältere »*-config«-Skripte der Art »sdl-config« ist meist der einzige Ausweg, sie mit

```
./configure --with-gcrypt-prefix="/home/user/arm-linux/usr"
```

den Configure-Skripten schließlich manuell zu übergeben.

Sobald relevante Teile Ihres Programms laufen, werden Sie sich ans Debuggen machen wollen. Für die Suche nach reinen Logikfehlern reicht meist ein Debugger direkt auf der Workstation. Tauchen die Fehler jedoch nur auf dem Zielsystem auf, etwa Endianness-bedingte oder durch die Optimierung des Crosscompilers oder durch andere architekturspezifische Details hervorgerufene, ist ein Debugger auf dem Zielsystem nötig.

GNU-Debugger auch remote

Der GNU Debugger (GDB, [4]) unterstützt Sie gut beim Remote-Debugging. Sie müssen ihn aber passend zum Ziel konfigurieren und das kleine Monitorprogramm »gdbserver« auf die Zielhardware kopieren. Der Server behandelt Breakpoints und inspiziert Speicheradressen. Sie starten den Remoteserver ähnlich wie einen gewöhnlichen Debugger. Er erwartet lediglich den Netzwerk- oder den seriellen Port, über den er mit dem GDB auf dem Entwicklungsrechner kommunizieren wird:

```
gdbserver :1024 ./hello
gdbserver /dev/ttyS0 ./hello
```

Den Cross-GDB auf Ihrem PC starten Sie dann in der Art

```
arm-linux-gnu-gdb hello
```

und geben die IP samt Port der Gegenstelle oder die serielle Schnittstelle an, an der die Zielhardware hängt:

```
(gdb) target remote 192.168.1.250:1024
(gdb) target remote /dev/ttyS1
```

Statt auf der Kommandozeile können Sie das Gleiche auch mit einem der grafischen GDB-Frontends erledigen (siehe Abbildung 2).

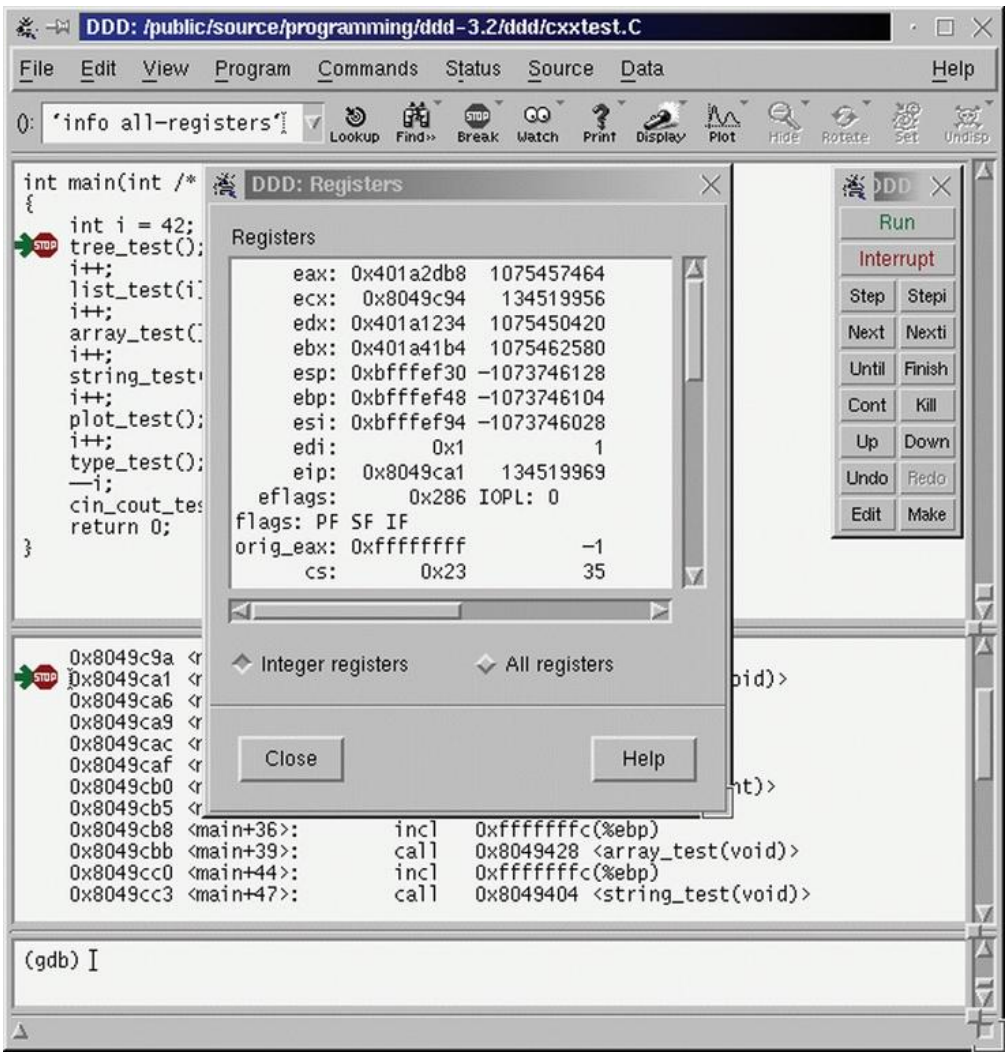

Abbildung 2: Der GNU DDD ist ein grafisches GDB-Frontend, das Sie statt der Kommandozeilen-Version zum Debuggen Ihres Systems benutzen können.

Selbst ein entsprechend gepatchter Linux-Kernel lässt sich bei einigen Architekturen auf Quellcode-Ebene remote mit dem GDB debuggen.

Sollten GDBs Fähigkeiten im Einzelfall mal nicht ausreichen, etwa wenn Sie den Bootloader oder einen Linux-Port auf einer neuen Chip-Plattform debuggen müssen, bleibt Ihnen noch der Griff zum JTAG-Debugger [5]. Der beherrscht auf Hardware-Ebene Single Stepping und Boundary Scans [6] vieler Pins ab dem ersten Taktimpuls.

Fazit

Die Open-Source-Landschaft bietet ausgereifte Entwicklungstools und einen immensen Vorrat an quelloffenen Softwarepaketen, die für eingebettete Geräte unter Einhaltung der Lizenzbestimmungen einsetzbar sind. Die meisten Entwickler verwenden der Einfachheit halber Systeme wie das vorgestellte Crosstool [3] oder Buildroot [7], Open Embedded [8] oder die T2 SDE [9].

Das hier im Kasten „Cross-GCC selbst übersetzen“ beschriebene manuelle Zusammenstellen der gesamten Toolchain gestaltet sich zwar zeitintensiver, hilft jedoch den Zusammenhang von Tools und Hardware besser zu verstehen und damit im Embedded-Entwicklungsprozess auftauchende Probleme schneller zu meistern. *(Heike Jurzik, jk)*

Infos

[1] GCC: [http://gcc.gnu.org]
[2] Binutils: [http://www.gnu.org/software/binutils/]
[3] Crosstool: [http://kegel.com/crosstool/]
[4] GNU Debugger: [http://www.gnu.org/software/gdb/]
[5] JTAG: [http://de.wikipedia.org/wiki/Joint_Test_Action_Group]
[6] Boundary Scan Test: [http://de.wikipedia.org/wiki/Boundary_Scan_Test]
[7] Buildroot: [http://buildroot.uclibc.org]
[8] Open Embedded: [http://www.openembedded.org]
[9] T2 System Development Environment: [http://www.t2-project.org]

Der Autor

René Rebe ist einer der Geschäftsführer der Exact CODE GmbH in Berlin und durch die tägliche Arbeit mit Linux auch in diverse Open-Source-Projekte involviert.