Softwareportierung auf eingebetette Systeme

Thomas Kunze

3. Dezember 2007

Zusammenfassung

In dieser Studienarbeit geht es um die Methoden und Probleme, die beim Portieren von Linuxsoftware auf ein eingebettes Linuxsystem entstehen. Insbesondere wird auf Software eingegangen, welche durch autotools verwaltet wird. Es wird nicht darauf eingegangen, wie man Toolchains erstellt, oder wie man bestimmte Werkzeuge für eingebettete Linuxsysteme wie buildroot oder OpenEmbedded benutzt, jedoch ist das hier dargestellte Wissen nützlich falls bei den Werkzeugen ein Fehler auftritt.

Inhaltsverzeichnis

1 Einleitung

Will man auf einem Desktoplinux ein Programm übersetzen, so ist dies meist nicht weiter kompliziert, man öffnet eine Konsole und übersetzt das Programm direkt mit dem Compiler:

```
>gcc -o hello hello.c
>./hello
Hello World!
```

Da auf einem eingebetteten Linux im allgemeinen kein Compiler installert ist, scheitert diese Methode dort. Man kann also Software nicht direkt auf dem System übersetzen, auf dem sie später laufen wird. Die Lösung ist ein Crosscompiler. Direkt am Beispiel:

```
>mipsel-linux-gcc -o hello hello.c
>scp hello user@embeddedlinuxsystem:/tmp
Password:
hello
>ssh user@embeddedlinuxsystem
Password:
>/tmp/hello
Hello World!
```

Man muss sich also zusätzlich zum eigentlichen Übersetzen auch das Programm auf das Zielgerät übertragen. Da dies im Entwicklungsprozess oft stört empfiehlt es sich auf dem Zielsystem ein Verzeichnis des Hostsystem durch NFS zu mounten, einen Emulator wie qemu einzusetzen.

2 Toolchains

Ein Crosscompiler läuft auf einer bestimmten Platform, dem Hostsystem, und erzeugt Code für eine andere Platform, dem Zielsystem. Als Toolchain wird ein Paket aus Crosscompiler, Bibliotheken und anderen Tools wie zum Beispiel readelf und objdump bezeichnet. Dabei sind die anderen Tools natürlich auch für die gleiche Zielplatform wie der Crosscompiler. Man kann sich aus dem Internet viele Toolchains herunterladen oder auch einfach selbst eine Toolchain erstellen. Eine Möglichkeit dafür ist [5] . Hier wird jedoch nicht weiter darauf eingegangen.

2.1 cross-compile-prefix

Eine Toolchain hat im allgemeinen einen bestimmten Prefix, den sogenannten 'cross-compile-prefix'. Dieser Prefix hat die Form CPU-Hersteller-Kernel-Betriebssystem. Die einzelnen Felder bedeuten jeweils:

CPU Die Prozessorarchitektur des Zielsystems. Also zum Beispiel 'i386' für PCs oder 'mipsel' für little endian Mips Prozessoren.

Hersteller Diese Feld ist nicht standartisiert. Hier steht oft 'pc', 'sun' oder 'unknown'. Im Fall 'unknown' wird das Feld auch oft weggelassen.

Betriebssystem In diesem Feld steht das Betriebssystem des Zielsystems, also z.B. 'solarisx.y', 'aixa.b.c' oder auch 'linux'. Bei Systemen ohne Betriebssystem steht hier 'elf' oder 'coff'.

Kernel Dieses Feld wird hauptsächlich bei Linux verwendet und auch dort nicht immer. Hier wird der Kernel vom Betriebssytem getrennt. Man schreib dann also 'linux-glibc' oder 'linux-uclibc'

Der Prefix 'arm-unknown-linux-glibc2.6' bedeutet also, dass es sich um eine Arm-Toolchain handelt, die für Linuxsysteme mit der glibc in Version 2.6 Programme erzeugt. Der Hersteller wurde nicht spezifiziert und steht deshalb auf 'unknown' .
Wenn man sich eine Toolchain installiert hat muss man sie nur noch in seinen PATH hinzufügen um einfache Programme zu übersetzen.

2.2 Bibliotheken

Ein weiterer Aspekt von Toolchains sind die mitgelieferten Bibliotheken. Je nach Umfang liefert eine Toolchain Biblioteken mit. Das reicht von der C-Bibliothek, die jede Toolchain mitliefert, bis hin zu Bibliotheken für X oder GTK. Diese Bibliotheken befinden sich an einer Stelle, die der Linker automatisch nach ihnen durchsucht. Diese Pfade sind meist relativ zur den Binärdateien der Toolchain und werden beim Übersetzen des Linkers festgelegt. Also benutzt der Crosscompiler diese Bibliotheken automatisch, wenn man die entsprechenden Compileroptionen angibt:

```
>mipsel-linux-gcc -o threadtest threadtest.c -lpthread
```

Will man gegen eine Bibliothek linken, welche nicht von der Toolchain mitgeliefert wird, muss man zunächst diese übersetzen und in ein beliebiges Verzeichnis installieren. Es ist sinnvoll ein Verzeichnis anzulegen in das man alle zusätzlich benötigten Bibliotheken installiert. Dieses Verzeichnis wird oft 'staging-dir' oder Platform-Verzeichnis genannt. Ist man damit fertig kann man seine Programme gegen die neue Bibloethek linken indem man die entsprechenden Pfade übergibt:

```
>mispel-linux-gcc -o glibtest glibtest.c -lglib2.0
-L/stageing-dir/lib -I/staging-dir/include
```

2.3 Make

Beim Crosscompilieren kann man die eingebauten Regeln von make wiederverwenden. Dazu belegt man die Umgebungsvariable CC mit dem entsprechenden Programm der Toolchain:

```
>CC=mipsel-linux-gcc LD=mipsel-linux-ld make hello
mipsel-linux-gcc     hello.c   -o hello
>scp hello user@embeddedlinuxsystem:/tmp
Password:
hello
>ssh user@embeddedlinuxsystem
Password:
>/tmp/hello
Hello World!
```

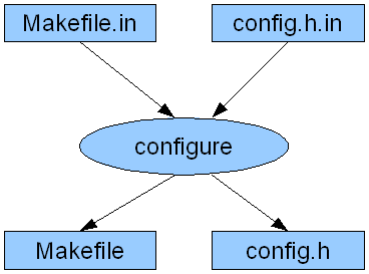

Abbildung 1: configure (vereinfacht)

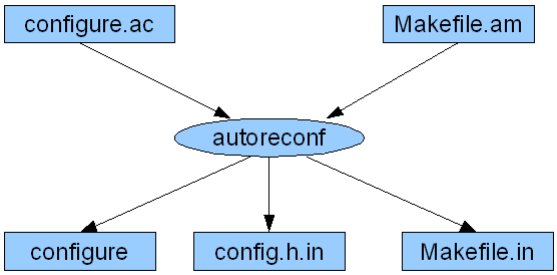

Abbildung 2: autoreconf

Weitere wichtige Umgebungsvariablen für make sind auch CXX, AR, RANLIB, AS und STRIP (Siehe auch [2]).

3 Autotools

Autotools sind eine Sammlung von Tools welche es ermöglichen im Quellcode vorliegende Programme auf verschiedenen Unixsystemen zu übersetzen.

3.1 vereinfachte Einführung

Eine typische Installation eines solchen Programms verläuft meist wie folgt:

```
>tar xzf programm.tgz
>cd programm
>./configure
...
>make
...
>make install
```

Da 'make' und 'tar' bekannt sind, ist der interessante Schritt dabei 'configure'. Wie auf Abbildung 1 gezeigt erzeugt das 'configure' Skript aus den Templatedateien mit der Endung '.in' normale Dateien. Wollte man den Buildprozess

verändern, müßte man also die Templatedateien und das 'configure'-Skript anpassen. Doch auch diese Dateien werden durch andere Programme erzeugt. Wie in Abbildung 2 dargestellt werden die Templatedateien und das 'configure'-Skript durch 'autoreconf' erzeugt. Damit kann durch das anpassen folgender Dateien den Buildprozess manipulieren.

configure.ac Diese Datei heißt manchmal auch configure.in und benutzt den GNU M4 Makro Prozessor [3]. Aus ihr werden das 'configure'-Skript und 'config.h.in' generiert. Ein kurzes Beispiel:

```
AC_INIT([hello],[1.0],[bugreport@provider])
AM_INIT_AUTOMAKE([-Wall -Werror foreign])
AC_PROG_CC
AC_CONFIG_HEADERS([config.h])
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Makefile.am Aus diesen Dateien werden die 'Makefile.in' Dateien generiert. In ihnen wird alles beschrieben was später für Makefiles wichtig ist. Das ist zum Beispiel welche Programme welche Quellen haben und wohin welche Dateien installiert werden. Ein kurzes Beispiel:

```
bin_PROGRAMS = hello
hello_SOURCES = hello.c
```

3.2 Probleme und Lösungen

3.2.1 configure

Die grundlegende Vorgehensweise beim Installieren von Softwarepaketen unterscheidet sich beim Crosskompilieren nur leicht:

```
>tar xzf programm.tgz
>cd programm
>./configure --target=mipsel-linux --prefix=/usr
...
>make
...
>make install DESTDIR=staging-dir
```

Damit teilt man configure mit, dass man für mipsel-linux die Zielplatform ist und dass nach '/usr' statt nach '/usr/local/ installiert wird. Durch die letzte Zeile installiert man das Programm schließlich in das staging-dir. Alternativ kann man auch einfach das fertige Programm in Binärform auf die Zielplatform kopieren. Da man die Man-pages u.ä. oft nicht auf einem eingebetteten System braucht verfolgt z.B. buildroot-ng[8] diesen Ansatz.

Wenn der 'configure'-Schritt eines Pakets mit der Fehlermeldung

```
...
 configure: error: cannot run test program while cross compiling
...
```

abbricht, ist der Grund dafür das M4 Makro AC_TRY_RUN mit der Syntax

```
AC_TRY_RUN (PROGRAM, [ACTION-IF-TRUE [, ACTION-IF-FALSE [, ACTION-IF-CROSS-COMPILING]]])
```

Einige Projekte setzen die ACTION-IF-CROSS-COMPILING nicht oder nicht korrekt. Beispielsweise ist ACTION-IF-CROSS-COMPILING oft einfach auf gesetzt. In diesem Fall compiliert man PROGRAM für die Zielplatform, führt es dort aus und setzt dann die ACTION-IF-CROSS-COMPILING entsprechend. Alternative kann man die entsprechende Option auch in einer Cache-Datei setzen die man mit '-C' an 'configure' übergibt.
Folgendes Beispiel stammt aus der configure.in des Mono-Projekts:

```
AC_TRY_RUN(
[
#include <sys/types.h>
#include <stdio.h>
#include <sys/un.h>

int main(void) {
struct sockaddr_un sock_un;
FILE *f=fopen("conftestval", "w");
if(!f) exit(1);
fprintf(f, "%d\n", sizeof(sock_un.sun_path));
exit(0);
}
],
cv_mono_sizeof_sunpath=`cat conftestval`,
cv_mono_sizeof_sunpath=0,
cv_mono_sizeof_sunpath=0
)
```

Wenn man das Programm getrennt übersetzt und auf der Zielplatform ausführt wird eine Datei 'conftestval' erzeugt. Bei der Mipsel-Platform steht in dieser Datei '108'. Wenn das AC_TRY_RUN Makro in Verbindung mit AC_CACHE_VAL verwendet wird, kann man eine Cache-Datei mit dem Eintrag 'cv_mono_sizeof_sunpath=108' erzeugen und an 'configure' übergeben. Alternativ kann man auch die letzte Zeile 'cv_mono_sizeof_sunpath=0' in 'cv_mono_sizeof_sunpath=108' ändern und 'configure' durch 'autoreconf' neu erzeugen. Damit ermittelt 'configure' für diese Zielplatform den richtigen Wert für 'cv_mono_sizeof_sunpath'. Für andere Zielplatformen m"usste man den Wert aber möglicherweise anpassen. Daher ist die Variante mit der Cache-Datei zu bevorzugen.

3.2.2 Native Tools

Zum erstellen von aufführbaren Binäredateien bernötigt man eine Reihe von Tools. Im einfachsten Fall ist dies ein Compiler. Manchmal werden Tools benötigt, welche in dem Softwarepaket enthalten sind und erst während des Übersetzungsprozesses selbst übersetzt werden. Bei falscher Konfiguration wird dieses Tool beim Crosscompilieren für die Zielplatform übersetzt und kann deshalb nicht auf der Hostplatform ausgeführt werden. Dies erkennt man durch eine Fehlermeldung wie 'program : cannot execute binary file'. In einem solchen Fall

muss man das entsprechende Makefile so ändern dass 'CC_FOR_BUILD' anstatt 'CC' verwendet wird. Bei einem autotools-Projekt ändert man dazu das 'Makefile.am' und führt 'autoreconf' aus. Folgender Ausszug stammt aus einem 'Makefile.am' des Mono-Projekt und zeigt eine richtige Konfiguration:

```
monoburg$(BUILD_EXEEXT): $(srcdir)/monoburg.c $(srcdir)/monoburg.h parser.c
        $(CC_FOR_BUILD) -o $@ $(srcdir)/monoburg.c parser.c $(am_CFLAGS)\
        $(LDFLAGS) $(BUILD_GLIB_LIBS)
```

Es ist zu beachten, dass zum Linken die richtigen Pfade der Hostbibliotheken zu übergeben sind.

3.2.3 pkgconfig Probleme

Ein weiters Tool welches beim Crosscompilieren oft Probleme bereitet ist pkg-config. Dieses Tool liefert die richtigen Kommandozeilenargumente um gegen eine Bibliothek zu kompilieren und zu linken. So kann ein Programm gegen eine Bibliothek zu Beispiel mit

```
>gcc -o test test.c `pkg-config --libs --cflags glib-2.0`
```

kompilieren und linken statt die Pfade an der sich die Bibliothek befindet fest einzufügen. Dateien für pkg-config werden meist nach '/usr/lib/pkgconfig' installiert. Es folgt eine Beispieldatei:

```
prefix=/usr
exec_prefix=/usr
libdir=${exec_prefix}/lib
includedir=${prefix}/include

glib_genmarshal=glib-genmarshal
gobject_query=gobject-query
glib_mkenums=glib-mkenums

Name: GLib
Description: C Utility Library
Version: 2.12.12
Libs: -L${libdir} -lglib-2.0 -lintl -liconv
Cflags: -I${includedir}/glib-2.0 -I${libdir}/glib-2.0/include
```

Die letzten beide Zeilen geben an, was 'pkg-config –libs glib-2.0' bzw. 'pkg-config –cflags glib-2.0' zurückgibt. Das Verzeichnis in dem pkg-config nach .pc Dateien sucht kann man durch die Umgebungsvariable PKG_CONFIG_DIR beeinflussen. Diese wird meist auf 'staging-dir/usr/lib/pkg-config' gesetzt, da gegen die Biblotheken der Zielplatform gelinkt werden soll. Wie man in der Beipspieldatei sieht, ist in der .pc-Datei der Pfad der Bibliothek kodiert. Aber es ist der falsche. Ein Versuch z.B. Mono gegen diese glib zu linken endet mit:

```
/bin/sh ../../libtool --tag=CC --mode=compile mipsel-linux-uclibc-gcc\
 -DHAVE_CONFIG_H
   ...
 -I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include \
   ...
```

```
 -c -o mono-hash.lo mono-hash.c; \
mono-hash.c:39:18: error: glib.h: No such file or directory
```

Die Includedatei 'glib.h' wird nicht gefunden, da die Pfade in der glib-2.0.pc auf die Hostplatform und nicht auf das staging-dir verweisen. Wenn auf dem Hostsystem eine glib installiert wäre, würde der Compiler versuchen die Hostbibliotheken beim Kompilieren und Linken zu verwenden und dabei spätestens beim Linken mit einer Fehlermeldung wie 'could not read symbols: File in wrong format' abbrechen. Die Lösung des Problem ist denkbar einfach: Man ändert die entsprechenden Einträge der .pc-Datei. Dies wird von guten Buildsystemen vollautomatisch durch sed erledigt.

3.2.4 libtool Probleme

Eines weiteres Probleme beim Crosscompilieren verursacht libtool. Dies ist ein Tool, welches es erlaubt gemeinsam benutzt Bibliotheken (shared libaries) unabhängig von der Art der gemeinsam benutzten Bibliothek zu benutzen. Über .la-Dateien wird eine Schnittstelle bereit gestellt über die man diese Bibliotheken portabel nutzen kann. Beim Crosscompilieren treten dabei oft Fehler auf wie:

```
/bin/sh ../../libtool --tag=CC --mode=link mipsel-linux-uclibc-gcc \
-O -Os -pipe -mips32 -mtune=mips32 -funit-at-a-time -I/vol/fob-vol4\
/mi03/tkunze/openwrt/staging_dir_mipsel/usr/include -I/vol/fob-vol4\
/mi03/tkunze/openwrt/staging_dir_mipsel/include -fno-strict-aliasing\
 -Wdeclaration-after-statement -g -Wall -Wunused -Wmissing-prototypes\
 -Wmissing-declarations -Wstrict-prototypes  -Wmissing-prototypes\
 -Wnested-externs -Wpointer-arith -Wno-cast-qual -Wcast-align\
 -Wwrite-strings  -L/vol/fob-vol4/mi03/tkunze/openwrt/\
 staging_dir_mipsel/usr/lib -L/vol/fob-vol4/mi03/tkunze/openwrt\
 /staging_dir_mipsel/lib -o pedump  pedump.o libmonoruntime.la\
  ../io-layer/libwapi.la ../utils/libmonoutils.la ../../libgc\
  /libmonogc.la -pthread -L/vol/fob-vol4/mi03/tkunze/openwrt\
  /staging_dir_mipsel/usr/lib -lgthread-2.0 -lglib-2.0 -lintl -liconv\
   -lm -ldl -lpthread -lm
libtool: link: cannot find the library `/usr/lib/libglib-2.0.la' or\
 unhandled argument `/usr/lib/libglib-2.0.la'
```

Der Linker kann also die Bibliothek '/usr/lib/libglib-2.0.la' nicht finden. Wenn diese auf dem Hostsystem installiert wäre, würde man stattt dessen die Fehlermeldung 'could not read symbols: File in wrong format' erhalten. Das Problem ist, dass der Linker versucht die Bibliotheken des Hostsystems zu benutzen. Aber es ist nicht klar warum: Die Pfade in dem Kommando enthalten keine Referenz auf das Hostsystem. Das Problem tritt nur auf, wenn mehrere beteidigte Bibliotheken libtool verwenden. Um das Problem zu verstehen muss man sich ein .la-Datei anschauen:

```
# libgthread-2.0.la - a libtool library file
# Generated by ltmain.sh - GNU libtool 1.5.22 (1.1220.2.365 2005/12/18 22:14:06)
#
# Please DO NOT delete this file!
# It is necessary for linking the library.
```

```
# The name that we can dlopen(3).
dlname='libgthread-2.0.so.0'

# Names of this library.
library_names='libgthread-2.0.so.0.1200.12 libgthread-2.0.so.0 libgthread-2.0.so'

# The name of the static archive.
old_library='libgthread-2.0.a'

# Libraries that this one depends upon.
dependency_libs=' -L/vol/fob-vol4/mi03/tkunze/openwrt/staging_dir_mipsel/usr/lib \
-L/vol/fob-vol4/mi03/tkunze/openwrt/staging_dir_mipsel/lib \
-lpthread /usr/lib/libglib-2.0.la -lintl -liconv'

# Version information for libgthread-2.0.
current=1200
age=1200
revision=12

# Is this an already installed library?
installed=yes

# Should we warn about portability when linking against -modules?
shouldnotlink=no

# Files to dlopen/dlpreopen
dlopen=''
dlpreopen=''

# Directory that this library needs to be installed in:
libdir='/usr/lib'
```

Die letzte Zeile ist der Kern des Problems: Sie teilt libtool mit, dass sich die Systembibliotheken in dem Verzeichnis '/usr/lib' befinden. Deshalb sucht libtool schließlich in diesem Verzeichnis nach der nächsten .la-Datei 'libglib-2.0.la'. Zur Lösung des Problems ändert man die Zeile in 'libdir='staging-dir/usr/lib'. Da man auf einem eigebetteten System keine Software kompiliert ist dies problemlos möglich. Die meisten Buildsysteme erledigen das auch durch sed.

4 Kernel und Rootimage

Manchmal hat man es mit einer Zielplatform zu tun, auf der Linux prinzipiell läuft, aber für die es noch keine fertigen Images gibt. Um zu einem Image zu kommen integriert ist es am einfachsten Unterstützung für die Zielplatform in eine Buildsystem wie OpenEmbedded [6] zu integrieren. Man kann allerdings auch mit Linux Bordmitteln ein Image erzeugen. Bevor man jedoch damit beginnt sollte man jedoch einen funktionierenden Kernel übersetzt haben.

4.1 Kernel

Der Kernel ist gleichzeitig der wichtigste und schwierigste Teil bei der Portierung von Linux. Wenn man mit eingebetteten Systemen keine weitere Erfahrung hat neigt man schnell zu dem Fehlschluss: "Linux ist auf meine Architektur portiert, es sollte also alles funktionieren.". Bei eingebetteten Systemen sind Portiereungen des Linuxkernels oft spezifisch für eine bestimmtes Unterarchitektur. Unterarchitekturen sind spezielle Prozessorfamilien eines Herstellers, sogenannte SSystem-on-a-Chip"(SoC). Diese Prozessoren haben sehr viel Peripherie wie Speicherkontroller, USB-Controller, veschiedene Serielle Ports, SPI und I2C Controller direkt im Prozessor integriert. Jede dieser Komponentene benötigt einen eigenen Treiber. Diese Treiber sind jedoch noch recht generisch. Ein Desktopsystem kann man meist wie gewünscht aus verschiedenen Komponenten zusammensetzten. Ganz im Gegensatz dazu kauft mein bei einem eingebetteten System meist ein Board auf dem bereits alle Komponenten enthalten sind. Jede diese Komponenten wie Bluetooth und Wlan Chips benötigt wieder einen Treiber. Wenn man vom Hersteller des Boards die Kernelquellen mit Treibern erhält hat man es einfach, ansonsten gibt es verschiedene Möglichkeiten an Treiber zu kommen:

- Der Treiber befindet sich bereits im offiziellen Linux Kernel. Dies ist der bestmögliche Fall. Man muss den Treiber nur noch in der Kernelkonfiguration aktivieren.
- Man erhält durch Internetrecherche oder Nachfrage beim Hersteller einen Kernelpatch, welcher den Treiber enthält. In diesem Fall muss man darauf achten, dass man die richtige Kernelversion zu dem Patch wählt. Ansonsten patcht man den Kernel wie üblich und aktiviert seinen Treiber.
- Man erhält durch Internetrecherche oder Nachfrage beim Hersteller Dokumentation der Hardware. In dem Fall muss man einen Treiber schreiben. Wie man dies macht wird in dem online verfügbaren Buch "Linux Device Drivers"[4] beschrieben.
- Man erhält nur einen Treiber in Binärform möglicherweise für ein anderes Betriebssystem. In diesem Fall kann man die Binärdatei disassemblieren und versuchen durch reverse engineering Informationen über die Komponente zu erhälten. Mit diesen Informationen kann man dann einen Treiber schreiben.

Wenn man seine Kernelquellen und Patches schließlich gesammelt hat, werden die Kernelquellen ausgepackt und die Patches angewendet:

```
>tar xzf kernel.tar.bz2
>cd kernel
>patch -p1 <../kernel-patch1
```

Dann wird der Kernel wie gewohnt konfiguriert und compiliert. Wichtig ist hierbei dass die Umgebungsvariablen ARCH auf die Architektur des Zielsystems und CROSS_COMPILE auf das Cross-compile-prefix der Toolchain gesetzt wird.

```
>ARCH=arm CROSS_COMPILE=arm-unkown-linux- make menuconfig
...
>ARCH=arm CROSS_COMPILE=arm-unkown-linux- make
```

Nun befindet sich das ferige Kernelimage irgentwo Kernelverzeichnis/arch/Architektur. Der genaue Ort ist je nach Architektur unterschiedlich. Dieses Kernelimage kann man dann über den Bootloader des System starten. Dann sollte man auf der seriellen Konsole Kernelausschriften sehen. Fall der Kernel nicht richtig startet sieht man hier auch was das Problem ist. Dazu muss allerdings der Treiber des seriellen Ports funktionieren und die serielle Konsole in der Kernelkonfiguration aktivert sein.

4.2 Rootimage

Ein Rootimage zu erstellen ist recht einfach wenn man eine fertige Toolchain zur Verfüng hat:

- Man legt ein Verzeichnis an, in dem das Rootimage angelegt wird. In diesem Verzeichnis legt man die Unterverzeichnisse etc,bin,sbin,lib,dev,proc und sys an. Dieses Verzeichnis ist das Staging Verzeichnis stage_dir.
- Man kopiert die Bibliotheken der Toolchain nach stage_dir/lib. Es ist nicht notwendig alle Bibliotheken zu kopieren, aber jede Bibleothek gegen die gelinkt wird muss kopiert werden.
- Man installiert eine Shell, udev und schreibt ein init-Skript und Speichert es nach stage_dir/bin/init. Bei einem eingebetten System empfiehlt sich busybox als Shell, init und Grundprogramme. Die Dateien in stage_dir/etc werden wie gewünscht angepasst oder angelegt. Welche Dateien benötigt werden und welche Software auf einem Desktopsystem installiert wird kann sich beim Linux-from-Scratch Projekt [7] anschauen.
- Zusätzliche Software wird ist das Staging Verzeichnis installiert.
- Das Staging Verzeichnis wird in ein Dateisystem Image umgewandelt. Je nachdem welches Dateisystem gewünscht wird, werden die Programme mkfs.squashfs oder mkfs.jffs2 benutzt. Alternativ kann man das Verzeichnis auch durch NFS exportieren.

Das so erhaltene Image kann man dann auf sein Board übertragen und starten. Dabei muss man natürlich darauf achten, dass man in der Kernelkonfiguration das Dateisystems des Rootimages aktiviert hat. Außerdem muss die Kernelkommandozeile durch den Bootloader oder bei der Konfiguration richtig gesetzt werden. Wenn alles richtig gemacht wurde startet nun das eigene Linux auf dem eingebettet Gerät.

Literatur

[1] Alexandre Duret-Lutz. Using gnu autotools. `http://www-src.lip6.fr/homepages/Alexandre.Duret-Lutz/dl/autotools.pdf`, August 2006.

[2] Free Software Foundation. Gnu make manual. `http://www.gnu.org/software/make/manual/html_node/Implicit-Variables.ht%ml#Implicit-Variables`, April 2006.

[3] Free Software Foundation. Gnu m4- gnu macro processor. `http://www.gnu.org/software/m4/manual/html_node/index.html`, July 2007.

[4] Greg Kroah-Hartman Jonathan Corbet, Alessandro Rubini. Linux device drivers. http://lwn.net/Kernel/LDD3/, February 2005.

[5] Dan Kegel. crosstool-howto. http://www.kegel.com/crosstool/crosstool-0.43/doc/crosstool-howto.html, March 2006.

[6] OpenEmbedded Project. Open embedded. http://www.openembedded.org/, November 2007.

[7] LFS Projekt. Linux from scratch. http://www.linuxfromscratch.org/, November 2007.

[8] OpenWRT Projekt. Buildroot-ng. http://wiki.openwrt.org/OpenWrtDocs/BuildrootNg, October 2006.

[9] Ian Lance Taylor. The gnu configure and build system. http://www.airs.com/ian/configure/configure_toc.html, July 1998.

Abbildungsverzeichnis