Distribution im Eigenbau

Alexander Vogt
25. Januar 2010

Outline

Outline

Einführung | Motivation

Wann baut man eine eigene Distribution?

- Wenn man ein spezielles System benötigt:
  - Smartphone
  - Echtzeitsystem
  - Router
  - Mailserver

Einführung | Inhalt

Worum geht es in diesem Vortrag?

- Methodik: Was sind die Schritte zum Bau eines Linux-Systems?
- Wie haben wir “linexa” gebaut?
- Was macht eine Distribution aus?

Wo finde ich mehr Infos?

- “Linux from Scratch” http://www.linuxfromscratch.org
- “Cross Linux from Scratch” http://cross-lfs.org
- “Do-It-Yourself Linux” http://www.diy-linux.org
- K. Yahmour et al., “Building Embedded Linux Systems”, O’Reilly 2008
- linexa homepage http://www.linexa.de

Einführung | Zielsystem

Bevor wir ein System bauen können, müssen wir uns einige Gedanken über das Ziel machen:

- Welche Architektur hat unser Zielsystem?
- Ist dessen Architektur die gleiche wie die unseres Gastgebersystems?
- Welche Anforderungen haben wir an das System? Ist es ein embedded System, oder ein Destop System? Single- oder Multi-User?
- Soll ein Sicherheits-Framework integriert werden?

Einführung | Gastgebersystem

Auch an das Gastgebersystem haben wir ein paar Anforderungen:

- Wir wollen ein System bauen - dazu benötigen wir das richtige Werkzeug: Bash, GCC, File, Libc, Binutils
- Wollen wir das System in einer Virtuellen Maschine bauen, benötigen wir entsprechende Anwendungen
- Ist unsere Hardware überhaupt in der Lage, in angemessener Zeit umfangreiche Pakete zu übersetzen?
- Bau als **user** — oder: Wie schützen wir unser Gastgebersystem?

Einführung | Tools

Es ist auch empfehlenswert, ein wenig Planung in die Infrastruktur zu stecken:

- Dokumentation
- Versionsverwaltung
- Bug-Tracker

Wir setzen eine Kombination aus Subversion (Versionsverwaltung) und Trac (Wiki / Bug-Tracker / Subversion Frontend) ein.

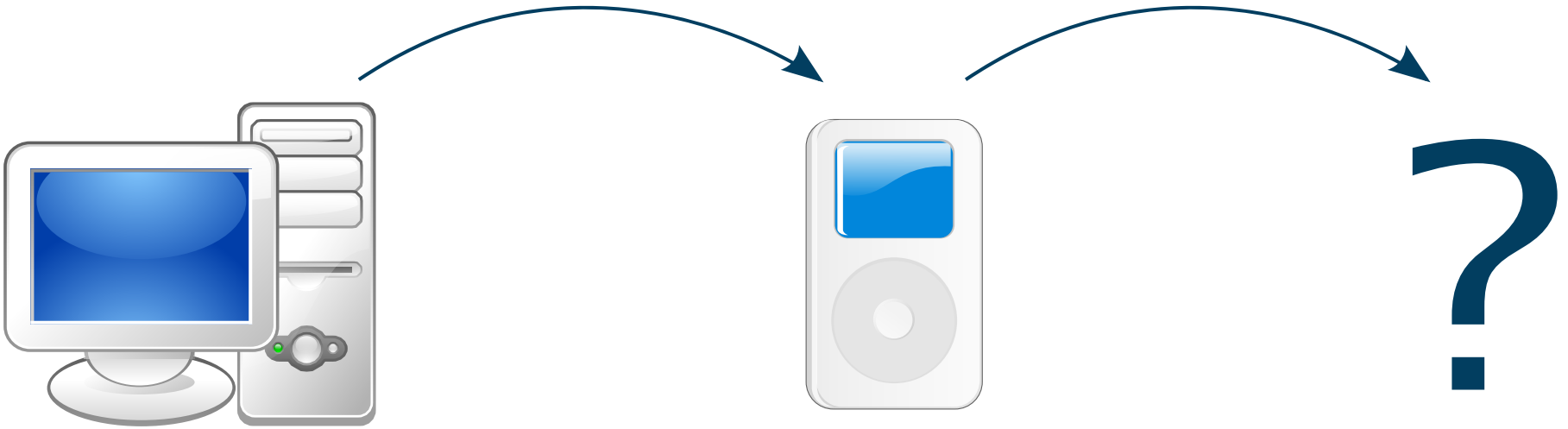
Einführung | Systeme

Nomenklatur:

build das System, das die Software übersetzt und zusammenbaut

host hier soll die Software einmal laufen

target für dieses System erstellt die Zielsoftware selbst ihre binären Programme



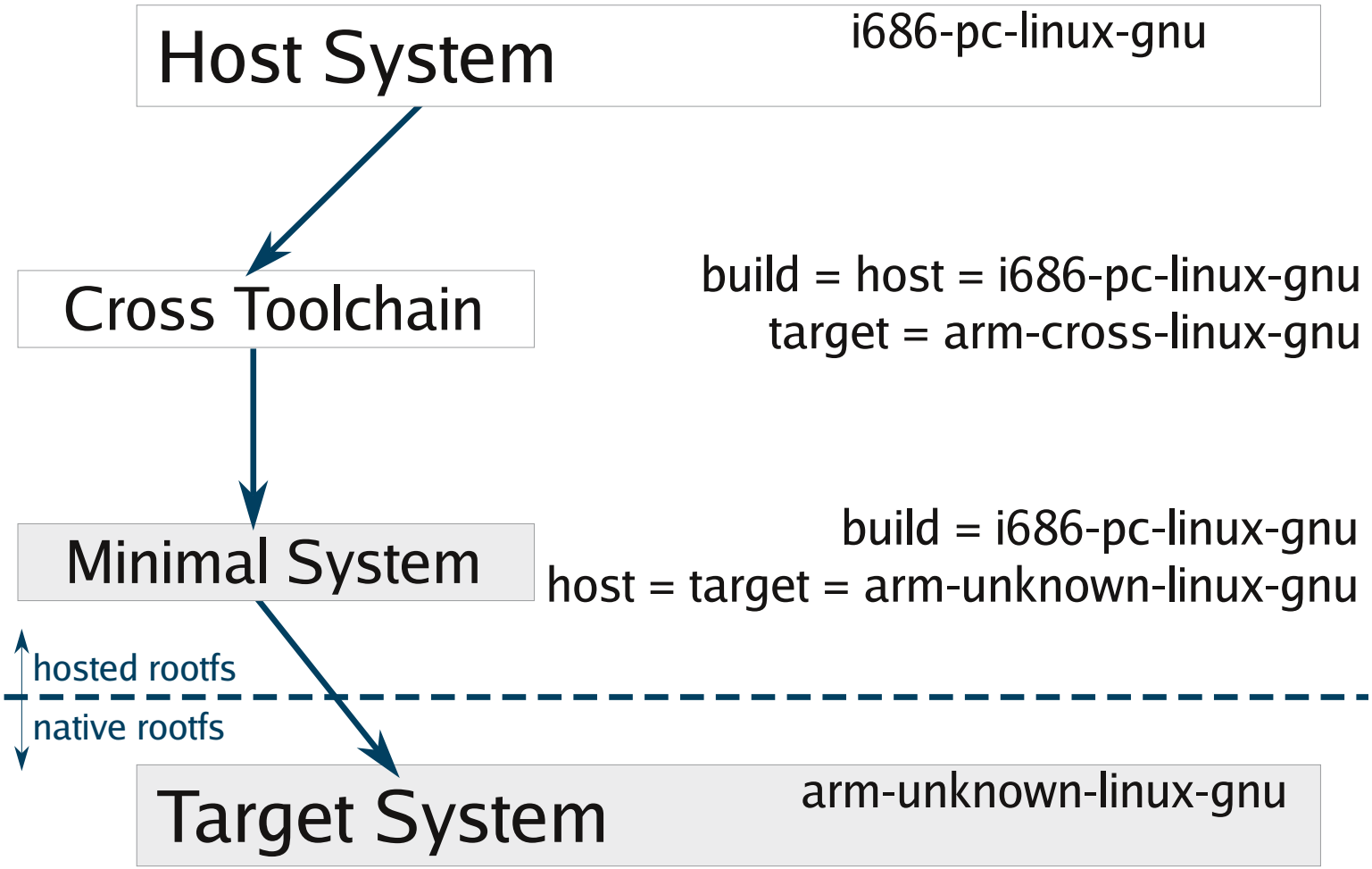
Systeme werden in folgendem Format beschrieben:

```
arch-vendor-kernel-os
```

arch	Architektur des Systems
vendor	Hersteller oder Produktfamilie
kernel	Verwendeter Betriebssystemkern
os	Verwendetes Betriebssystem oder ABI

Outline

Überblick | Bauplan



Outline

GNU Toolchain | Definition

Was ist eigentlich eine Toolchain (TC)?

Eine Sammlung der grundlegenden Entwicklungswerkzeuge zum Kompilieren (Übersetzen) von Software.

Was beinhaltet eine TC?

Compiler übersetzt Quellcode in (maschinenspezifischen) Binärcode

grundlegene Bibliotheken enthalten standardisierte Funktionen und Routinen, sowie deren Header

Linker verlinkt vom Compiler übersetzte Objekte (z.B. Programme und Bibliotheken) miteinander

Assembler, Archiver, etc.

GNU Toolchain | Pakete

Welche GNU-Pakete werden benötigt?

binutils liefert alle Tools, die das System zum Umgang mit binären Objektdateien benötigt

GNU Compiler Collection Enthält u.A. den C/C++ Compiler

C Library Standard C Bibliothek und Header

Kernel Header Schnittstelle zum Kernel

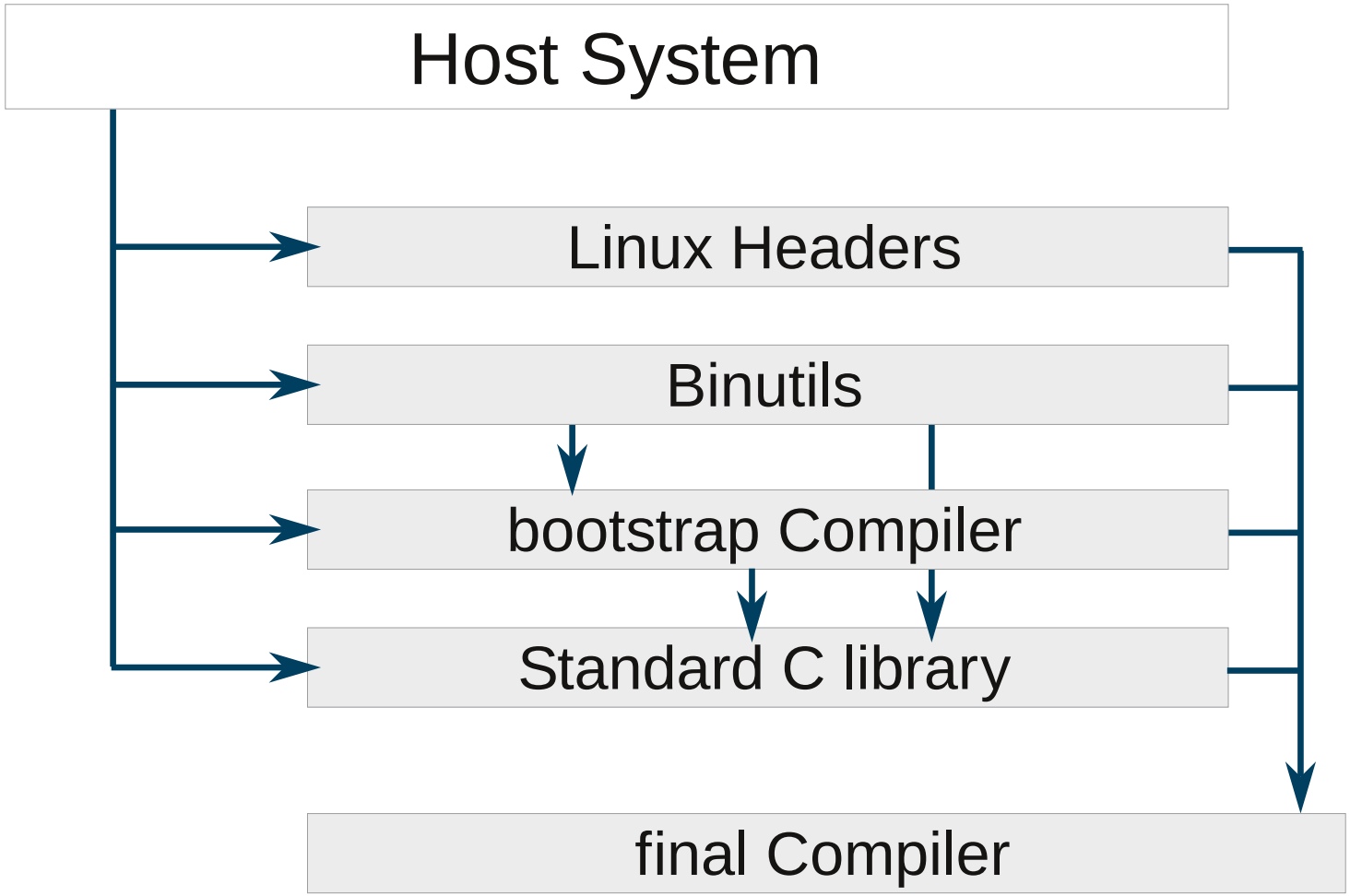
GNU Toolchain | Pakete - cont.

Welche Versionen und Varianten gibt es?

binutils	"passend zum Compiler", neueste Version (2.20) aber eigentlich unproblematisch
GNU Compiler Collection	4.2.x, 4.3.x, 4.4.x
C Library	Glibc (2.7 – 2.10), eGlibc, uClibc
Kernel Header	aktueller Kernel

Wir haben für unser aktuelles System die binutils-2.19, GCC-4.4.0 und Glibc-2.10 verwendet.

GNU Toolchain | Schema



Outline

Cross-Toolchain | Vorbereitungen

Wo baut man am besten ein Linux-System?

eigene Partition Boot mit gleicher Hardware

Verzeichnis, NFS Zielsystem inkompatibel, Cross-Build

Image Boot mit Emulator / Virtueller Maschine, nativer Bau

Was muss man beachten?

- volle Zugriffsrechte für den user “build”
- Platzbedarf etwa 10GB
- Transfer auf das Zielsystem

Cross-Toolchain | Vorbereitungen - cont.

Unser Ziel ist es, unser endgültiges System nativ in einer virtuellen Maschine zu bauen. Wir haben uns für einen Bau im Image entschieden. Dafür sprechen folgende Punkte:

- ein einfaches (mit `dd` erstelltes) Raw-Image kann problemlos als loop-device in das Gastgebersystem eingehängt werden
- QEMU benötigt irgendwann ohnehin ein Image und kann ebenfalls mit raw-Images umgehen - wir sparen uns also einen Kopiervorgang
- Images sind sehr handlich - für ein BackUp muss nur eine Datei kopiert werden, bzw. mit Snapshots (für qcow2)

Cross-Toolchain | Vorbereitungen - cont.

```
export BUILD_ROOT=/pfad/zum/build/verzeichnis
```

Vorbereitung des Arbeitsverzeichnisses:

```
${BUILD_ROOT}
|-- cross-tools/
|-- sources/
`-- tools/
```

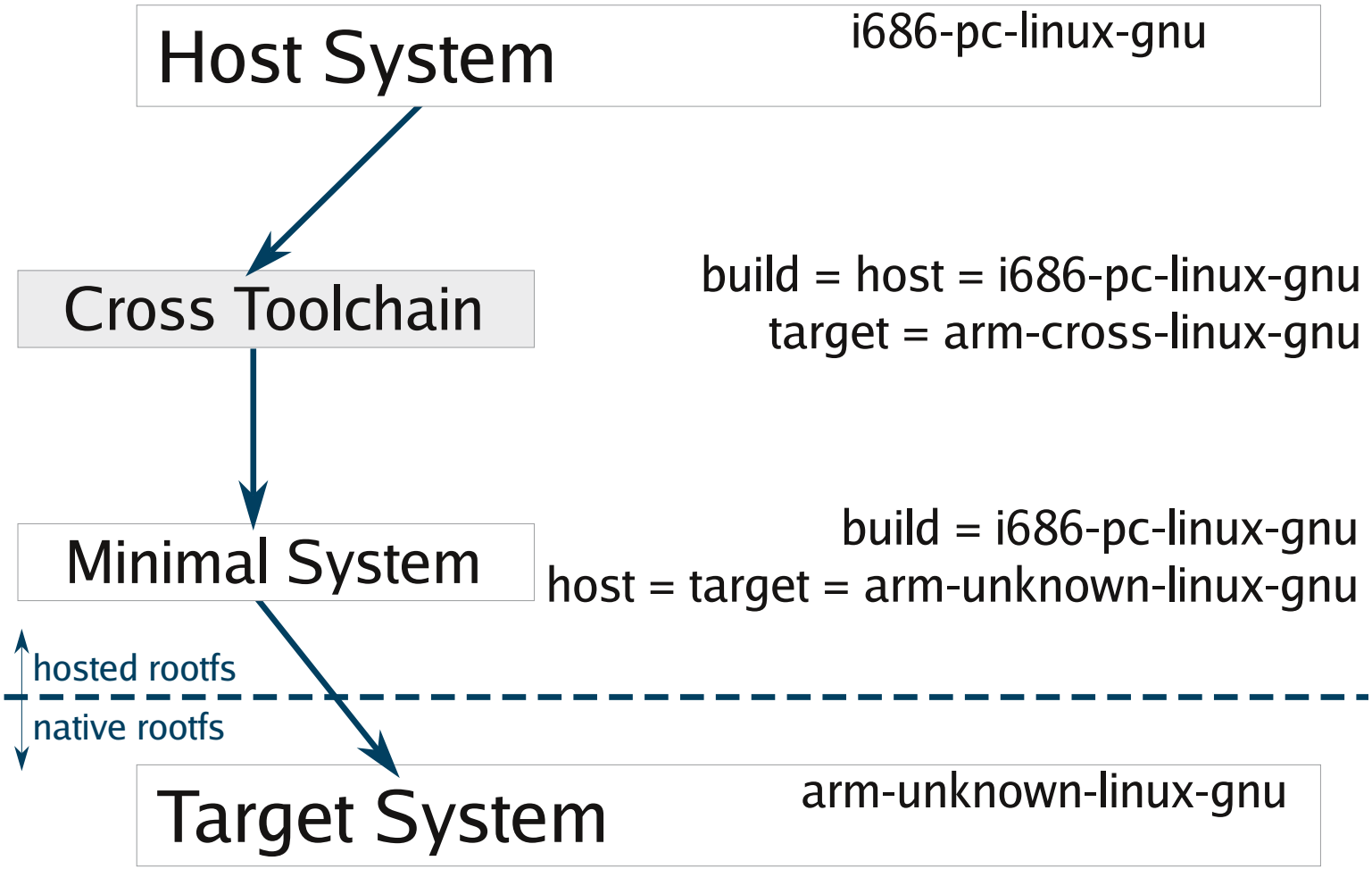
```
:> mkdir -p ${BUILD_ROOT}/{cross-tools,tools,sources}
```

Um das gleiche Installationsziel wie auf dem Zielgerät vorzutäuschen, legen wir Links nach / an:

```
:> ln -s ${BUILD_ROOT}/{cross-,}tools /
```

```
:> chown -R build ${BUILD_ROOT}
```

Cross-Toolchain | Bauplan

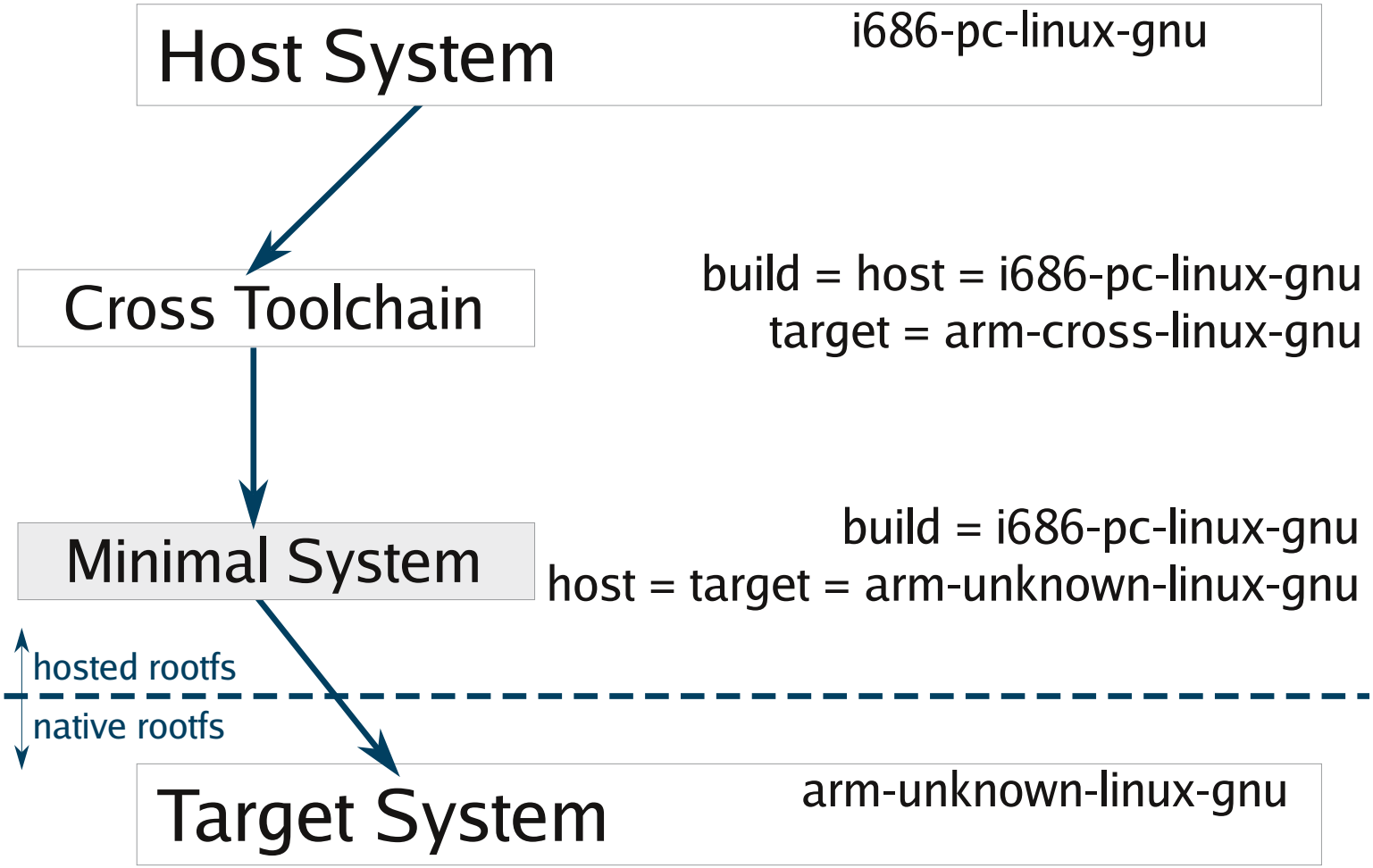


Cross-Toolchain | allg. Vorgehen

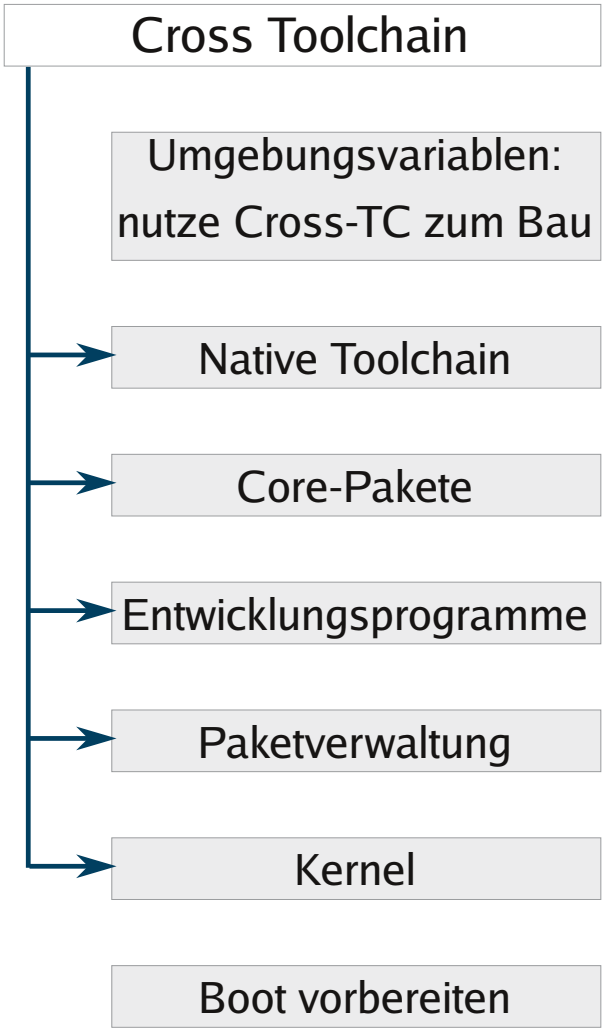
```
:> ./configure --build=i686-pc-linux-gnu
→ --host=i686-pc-linux-gnu
→ --target=arm-cross-linux-gnu
→ --prefix=/cross-tools
:> make
:> make DESTDIR=${BUILD_ROOT}/cross-tools install
```

Outline

Minimalsystem | Bauplan



Minimalsystem | Schema



allgemeines Vorgehen:

```
:> ./configure
→ --build=i686-pc-linux-gnu
→ --host=arm-cross-linux-gnu
→ --target=arm-unknown-linux-gnu
→ --prefix=/tools
:> make
:> make DESTDIR=${BUILD_ROOT}/tools
→ install
```

Minimalsystem | Umgebungsvariablen

Nutzen der Cross-TC:

```
CC=‘‘arm-cross-linux-gnu-gcc’’, AR=,
LD=, usw.
```

Optionen der TC:

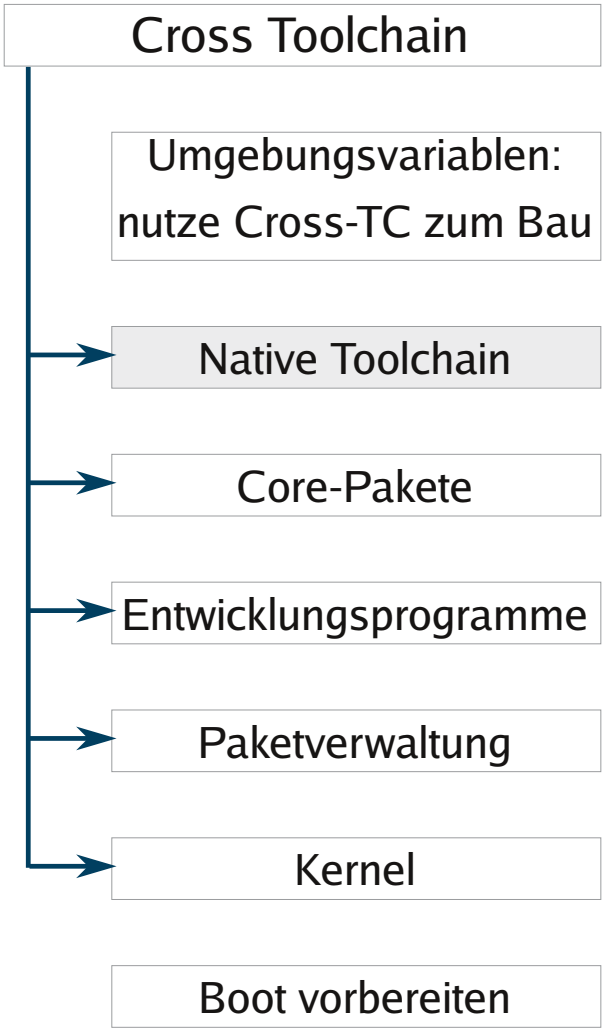
```
LC_ALL=POSIX, MAKEFLAGS=‘‘-j2’’
```

Setzen der Pfade:

```
BUILD_ROOT=/pfad/zum/build/verzeichnis
PATH=${BUILD_ROOT}/cross-tools:$PATH
```

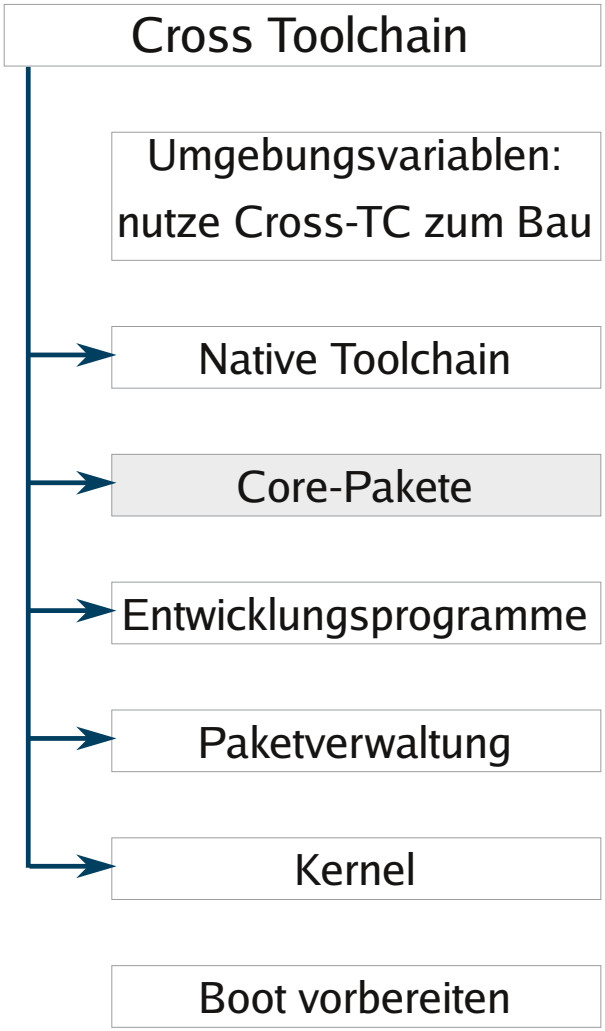
Wichtig:
/cross-tools **vor** `$PATH`

Minimalsystem | Native Toolchain



ähnlich wie Cross-TC

Minimalsystem | Core-Pakete

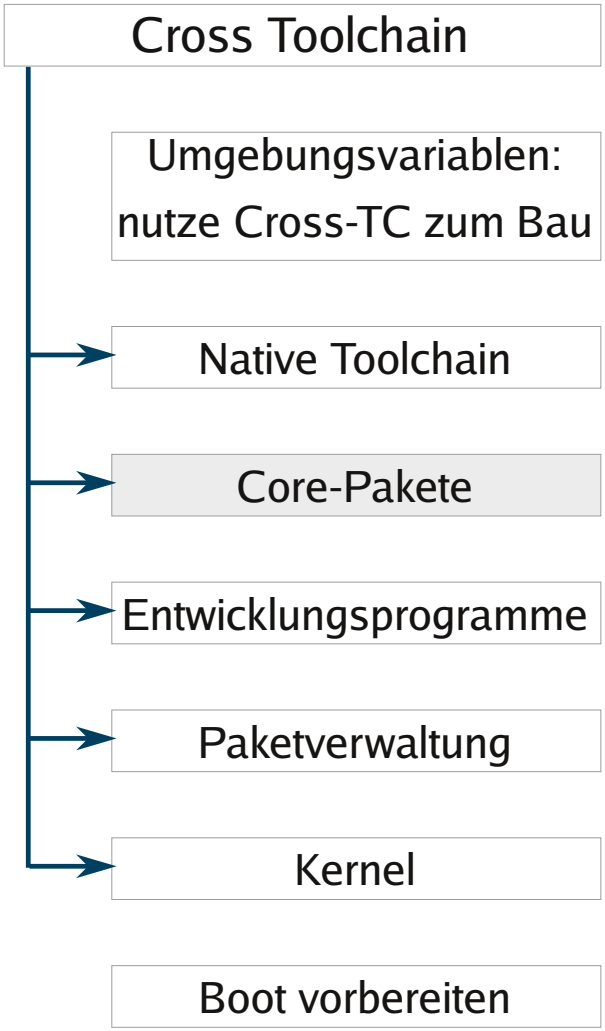


Hier finden sich alle auf einem Linux-System zwingend erforderlichen Tools.
2 Möglichkeiten:

GNU Grundpakete core-utils, diffutils, findutils, bash, etc.

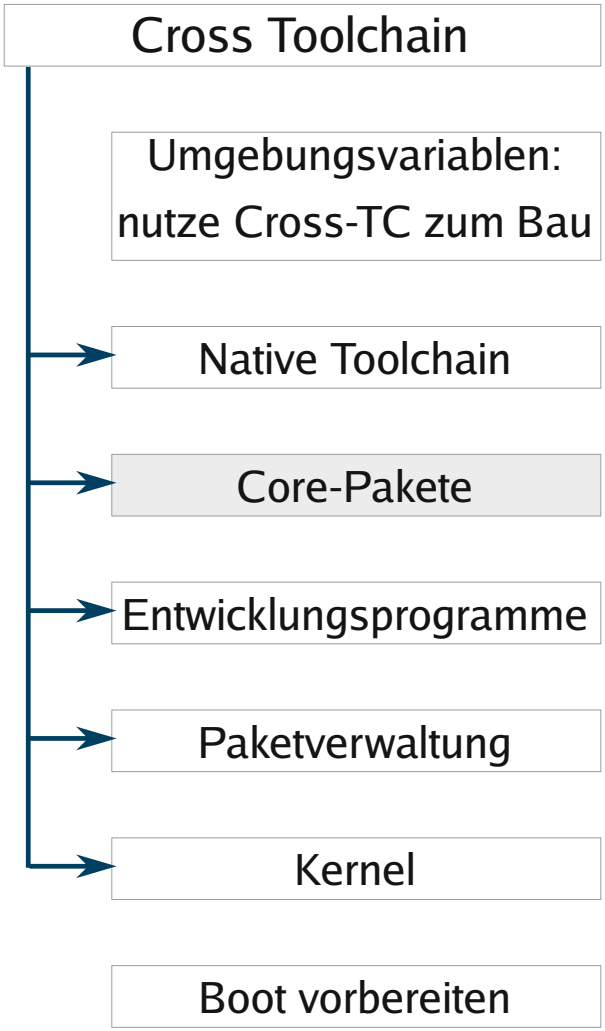
busybox v.A. für embedded Systeme, liefert minimalistische Versionen aller gängigen Grundpakete

Minimalsystem | GNU Grundpakete



Zlib	(De-)Komprimungsroutinen
Ncurses	
Bash	Bourne Again Shell
Bison	
Coreutils	Anzeigen und Verändern der grundlegenden Systemeinstellungen
Diffutils	Anzeigen der Unterschiede von Dateien und Ordnern
Findutils	Finden von Dateien
Util-Linux-ng	Sammlung essentieller Tools
E2fsprogs	Verwalten von ext[234] Dateisystemen

Minimalsystem | GNU Grundpakete - cont.



Module-Init-Tools (Ent-)Laden von Kernel-Modulen

File

Flex

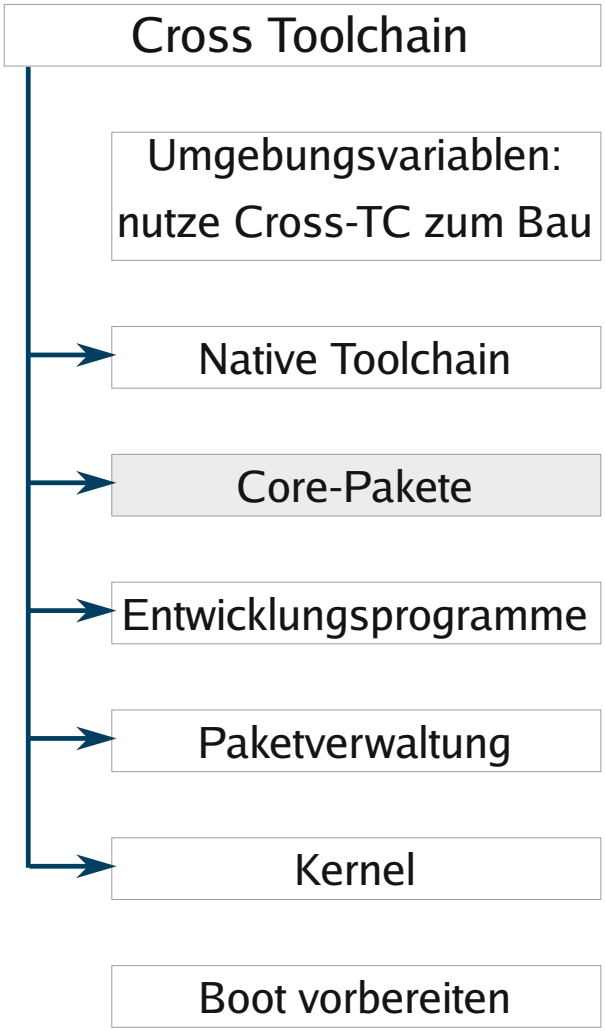
Gettext

Grub Bootloader

Gawk, Grep, Sed Manipulieren von Textdateien

Iana-Etc, Inetutils, IPRoute2 essentielle Netzwerk-Tools

Minimalsystem | GNU Grundpakete - cont.



M4

Make benötigt zum Kompilieren von Dateien

Patch

Texinfo Erzeugen, Ausgeben und Verändern von Info-Seiten

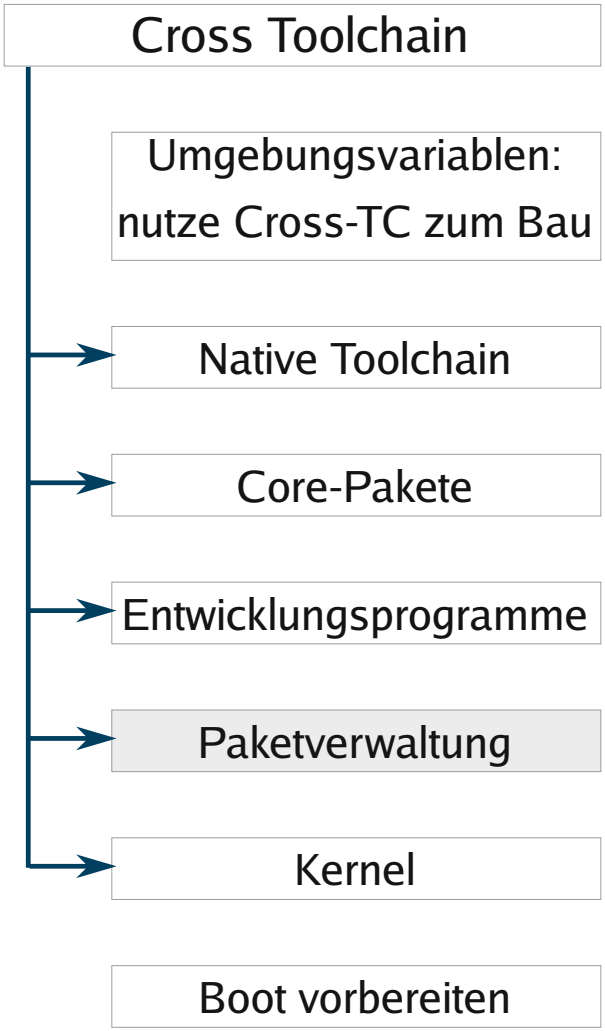
Bzip2, Gzip, Tar, XZ Utils (lzma) Tools zum (Ent-)Packen von Paketen

Minimalsystem | Entwicklungsprogramme

Diese Tools sind nicht zwingend erforderlich, erleichtern einem aber die folgenden Schritte erheblich.

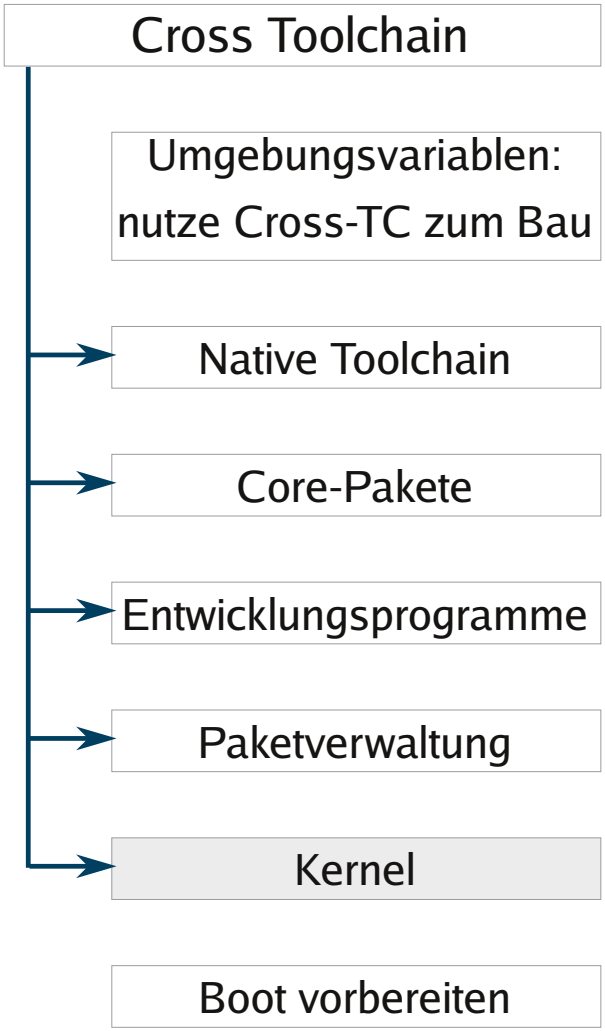
Vim, nano	Editor
wget, rsync	Programme zum Download/Synchronisieren von entfernten Dateien
cvs, svn, git	Schnittstelle zur Versionsverwaltung

Minimalsystem | Paketverwaltung



Nun muss eine Paketverwaltung und ihre Abhängigkeiten installiert werden, mit der wir im nächsten Schritt schon die Core-Pakete bauen und verwalten können.

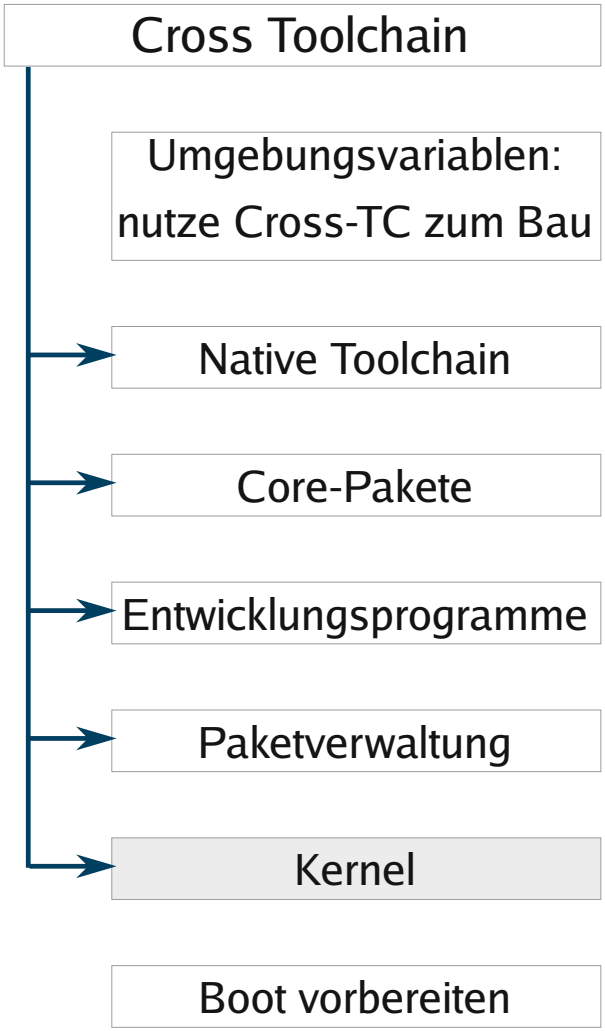
Minimalsystem | Kernel



Konfiguration und Bau:

```
:> make ARCH=arm
→ CROSS_COMPILE=arm-cross-linux-gnu-
→ EXTRAVERSION=-cross menuconfig
:> make ARCH=arm
→ CROSS_COMPILE=arm-cross-linux-gnu-
→ EXTRAVERSION=-cross bzImage
:> make ARCH=arm
→ CROSS_COMPILE=arm-cross-linux-gnu-
→ EXTRAVERSION=-cross modules
```

Minimalsystem | Kernel - cont.



Installation:

```
:> cp arch/arm/boot/bzImage
→ ${BUILD_ROOT}/boot/vmlinuz-cross
:> cp arch/arm/boot/bzImage
→ /home/build/vmlinuz-cross
:> make ARCH=arm
→ CROSS_COMPILE=arm-cross-linux-gnu-
→ EXTRAVERSION=-cross
→ INSTALL_MOD_PATH=${BUILD_ROOT}
→ modules_install
```

Minimalsystem | Kernel - cont.

Was muss beachtet werden?

EXTRAVERSION Die Kernel-Module müssen ins System installiert werden. Um sie später sauber wieder entfernen zu können, setzen wir hier ein Suffix "-cross", das ans Installationverzeichnis angehängt wird.

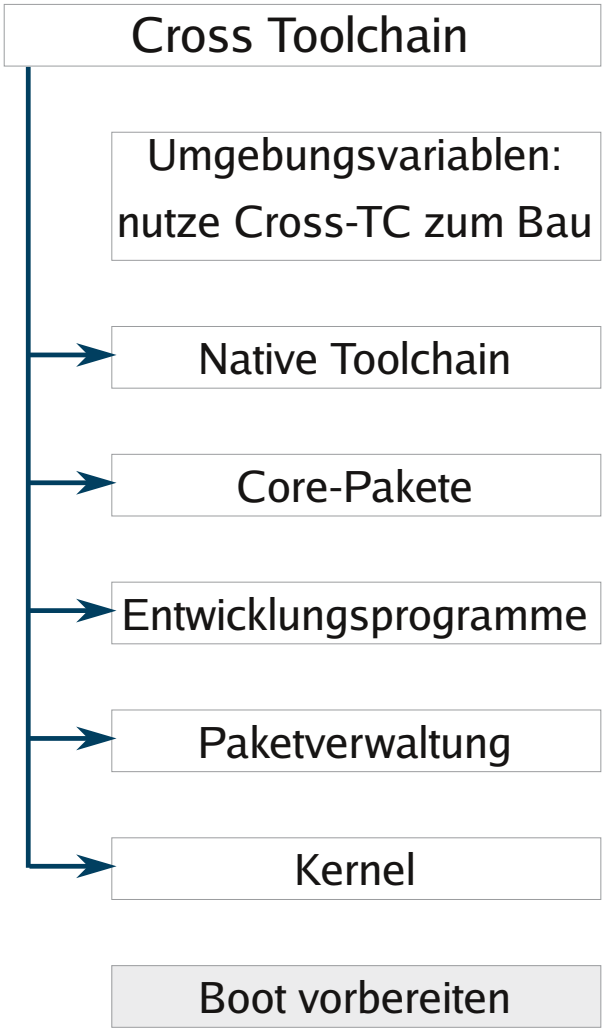
CROSS_COMPILE TC, die der Kernel nutzen soll (mit "-" am Ende!)

make ... menuconfig Zum Boot benötigte Treiber **nicht** als Modul kompilieren - ein InitRD wäre an dieser Stelle absoluter Overkill!

bzImage Kopieren des Kernels in das Verzeichnis und neben das Image, da wir beim ersten Booten noch keinen Bootloader haben, und mit QEMU einen externen Kernel laden können.

INSTALL_MOD_PATH Die Module sollen in das Image installiert werden

Minimalsystem | Boot vorbereiten

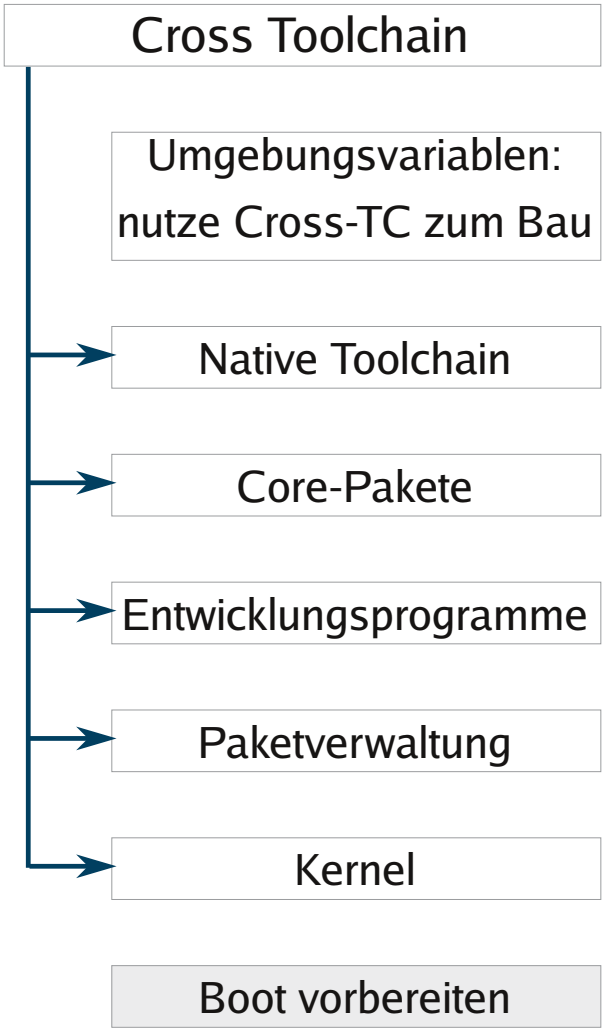


Ordnerstruktur anlegen Erstellen der grundlegenden Linux Ordnerstruktur

Links anlegen viele Make-Skripte erwarten Programme an festen Orten (z.B. `/bin/sh`), die des Minimalsystems liegen aber unter `/tools`

Devices Linux benötigt zum Booten minimal `/dev/null` und `/dev/console` sowie das Device der Root-Partition, diese müssen noch angelegt werden

Minimalsystem | Boot vorbereiten - cont.



/etc/fstab dem System muss mitgeteilt werden, wo es /, `/proc` und `/sys` findet

Bootskript zum Starten des Systems

Minimalsystem | Bootscript

Das Minimalsystem wird ohne ein init-System gestartet, stattdessen wird dem Kernel später über die Option `init=/boot/linuxrc` direkt ein Bootskript angegeben.
Aufgaben:

1. `/proc` und `/sys` einhängen
2. `/` schreibfähig mounten
3. das Netzwerk starten
4. Umgebungsvariablen setzen
5. eine Konsole starten

Outline

Exkurs: QEMU | Übersicht

Was ist QEMU?

Qemu ist ein Open Source Maschinen Emulator und Virtualisierer.

QEMU gibt es in drei "Geschmacksrichtungen":

QEMU Qemu selbst ist ein Emulator, der (fast) alle Architekturen nachbilden kann, aber doch recht langsam ist - HPU (Hardware Emulation)

KQEMU KQemu ist eine speziell beschleunigte Variante von Qemu, in der eine x86 Architektur nativ virtualisiert werden kann

KVM Für KVM (Kernel-based Virtual Machine) benötigt man spezielle Hardware (Intels Vanderpool, AMDs Pacifica), diese Variante der "vollen Virtualisierung" ist zwar die schnellste, aber es kann wiederum nur nativ virtualisiert werden.

Exkurs: QEMU | Architekturen

Welche Architekturen unterstützt QEMU?

```
:> qemu-system-
qemu-system-arm          qemu-system-ppc
qemu-system-cris         qemu-system-ppc64
qemu-system-m68k         qemu-system-ppcemb
qemu-system-microblaze   qemu-system-sh4
qemu-system-mips         qemu-system-sh4eb
qemu-system-mips64       qemu-system-sparc
qemu-system-mips64el     qemu-system-sparc64
qemu-system-mipsel       qemu-system-x86_64
```

Exkurs: QEMU | Grundlagen

Anlegen eines Images:

```
qemu-img create -f qcow2 build_image.img 10G
```

Konvertieren eines Images:

```
qemu-img convert -f raw /pfad/zum/image.iso
→ -O qcow2 build-qcow2.img
```

Boot eines Images von der virtuellen Festplatte:

```
qemu-system-x86_64 -hda build_image.img -cdrom bootCD.iso
→ -boot c
```

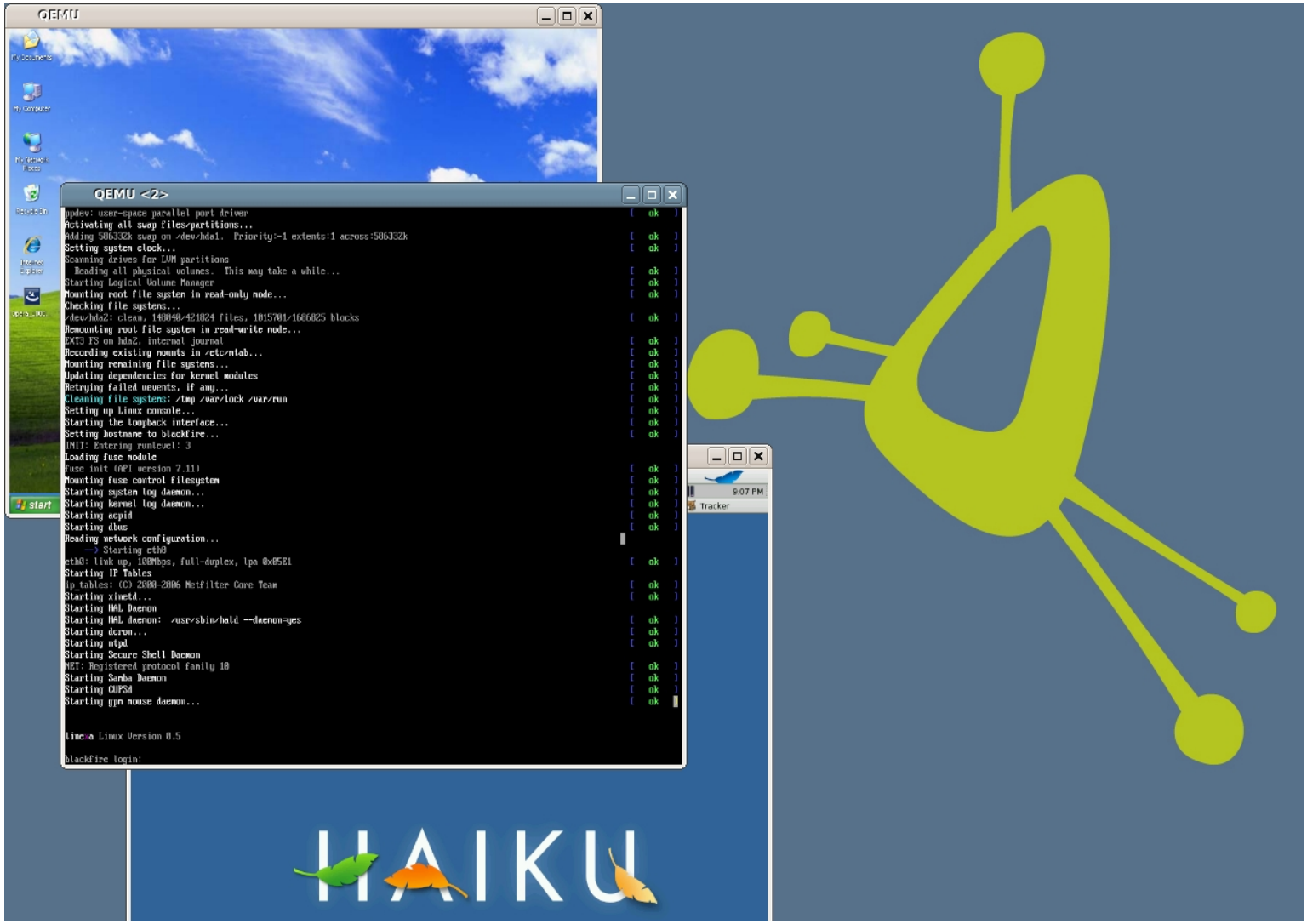
Boot eines Images von einer DVD:

```
qemu-system-x86_64 -hda build_image.img -cdrom /dev/dvd
→ -boot d
```

Boot mit externem Kernel:

```
qemu-system-x86_64 -hda build_image.img -kernel vmlinuz
→ -append ‘‘root=/dev/hda1 ro’’
```

Exkurs: QEMU | Screenshot



Outline

Boot des Systems | mit externem Kernel

In dem bisher gebauten System ist noch kein eigener Bootloader eingerichtet, es kann daher noch nicht von allein starten. Wir starten QEMU daher mit dem externen Kernel, den wir dafür auch außerhalb des Images gelagert haben:

```
QEMU -hda /pfad/zum/image.img
→ -kernel /home/build/vmlinuz-cross
→ -append ‘‘root=/dev/hda1 ro init=/boot/linuxrc’’
```

Hinweis:
QEMU ist durch qemu-systems-xxx oder qemu-kvm zu ersetzen!

Boot des Systems | Grub

Achtung: Grub unterstützt nur die Architekturen x86, PPC und IA32!

Wir haben Grub, den "GRand Unified Bootloader" oben bereits gebaut, nun verankern wir ihn auf der virtuellen Festplatte:

```
:> grub --no-floppy
```

Suche die boot-Partition:

```
$ find /boot/grub/stage1
```

Lege die zu bootende Partition fest:

```
$ root (hd0,1)
```

Installiere Grub auf der Partition:

```
$ setup (hd0)
```

Fertig!

```
$ quit
```

Boot des Systems | mit internem Kernel

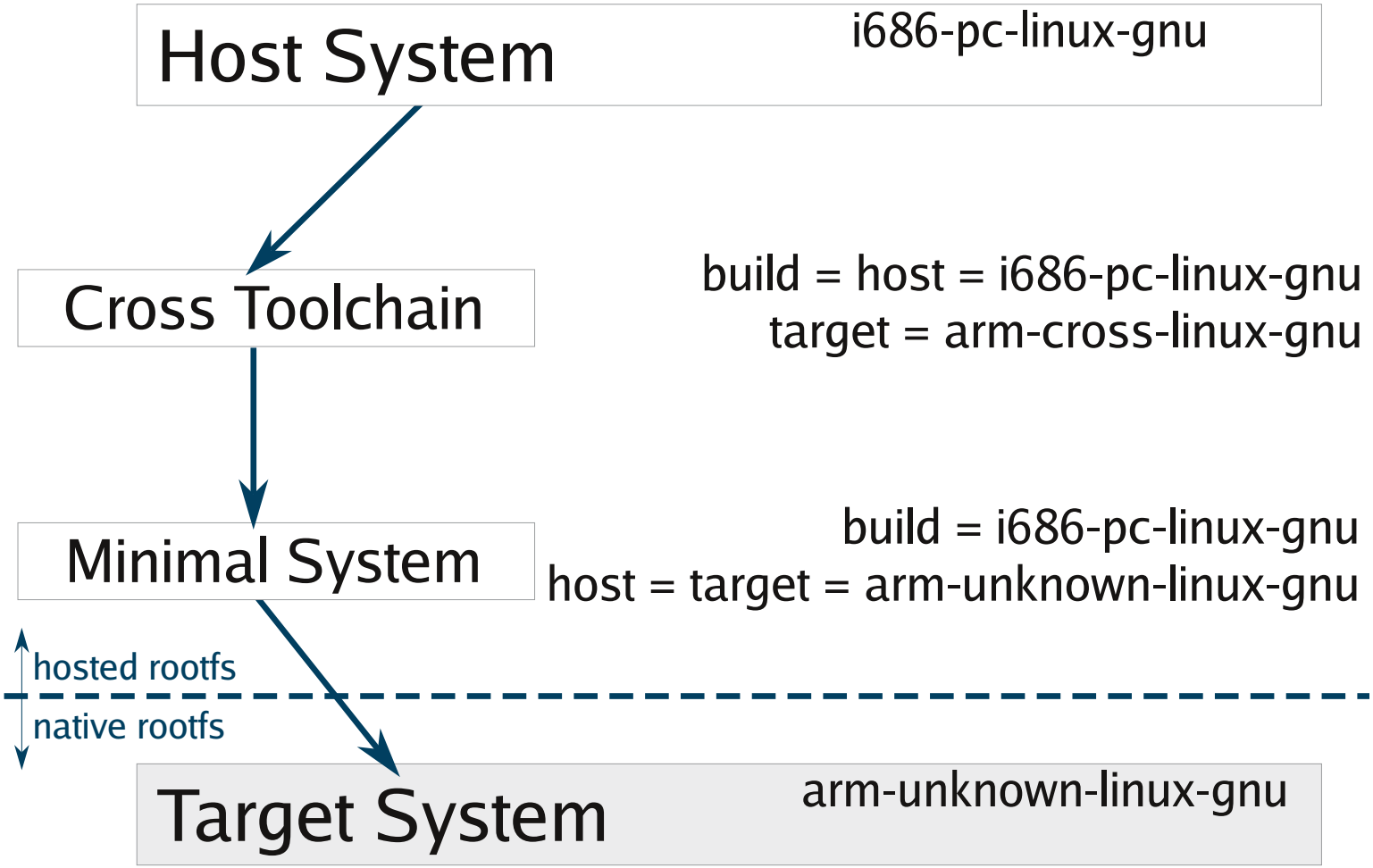
Nun, da ein Bootloader installiert wurde, kann das Image normal gestartet werden:

```
QEMU -hda /pfad/zum/image.img -boot c
```

Jetzt wäre ein guter Zeitpunkt, das Image zu konvertieren, um sich den Geschwindigkeitsvorteil von qcow2 zu Nutze zu machen!

Outline

Grundsystem | Bauplan



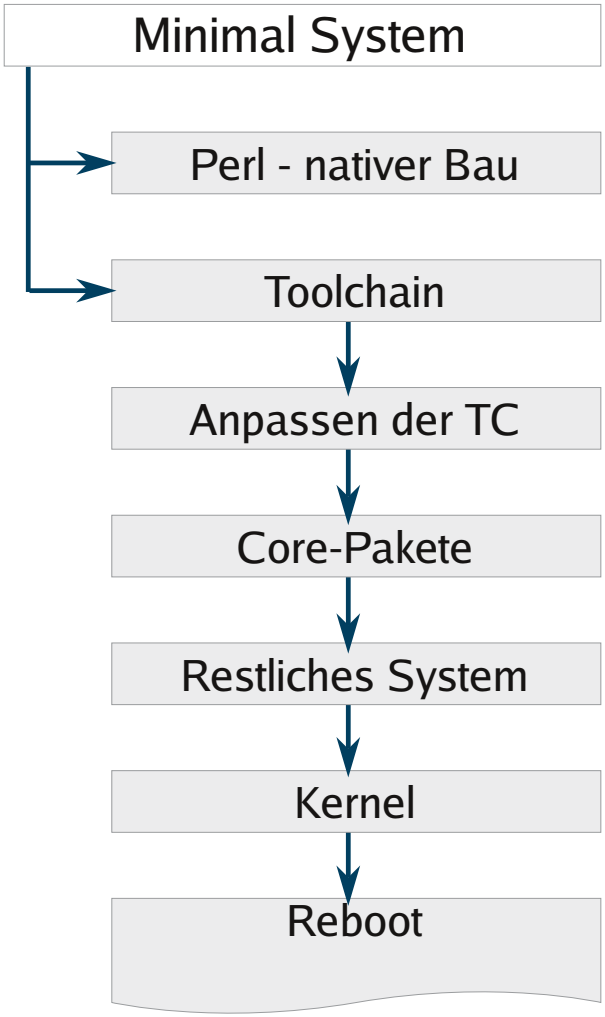
Grundsystem | Bau der Pakete

Das System ist nun gebootet, ab jetzt wird nativ gebaut.
Außerdem bauen und installieren wir die Pakete nun nicht mehr manuell, sondern benutzen dafür einen Paketmanager. Wie ein Paket gebaut werden soll, welche Patches eingepflegt werden müssen und welche zusätzlichen Dateien wir dem Paket mitgeben wollen, beschreiben wir in einem "Buildskript".

allgemeines Vorgehen:

1. Erstellen des Buildskriptes auf der Workstation
2. Transfer auf das Zielsystem
3. Bau des Paketes, evtl. Modifikation des Buildskriptes
4. Installation des Paketes
5. Kopieren des Paketes in ein Repository

Grundsystem | Schema



Wie schon für das Minimalsystem müssen auch hier wieder die Umgebungsvariablen gesetzt werden:

```
LC_ALL=POSIX, MAKEFLAGS=‘‘-j2’’
```

Setzen der Pfade:

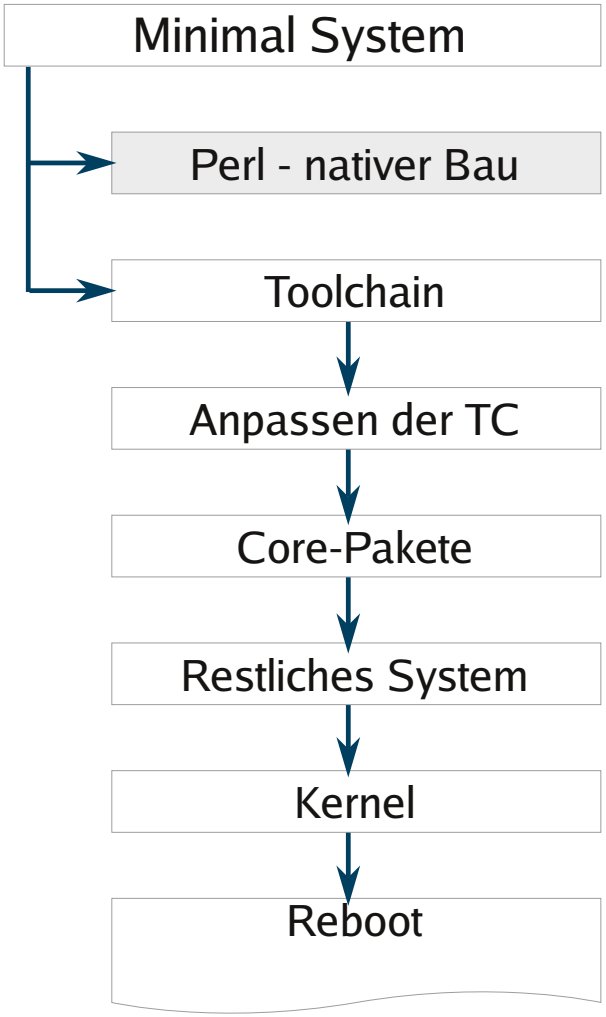
```
PATH={,/usr}/bin:{,/usr}/sbin:/tools/{bin,sbin}
```

`/tools` ist **hinter** `$PATH`

Wir müssen in den folgenden Schritten nun kein System mehr angeben. Da wir nativ bauen wird dieses durch das Ausführen der ./configure - Skripte automatisch erkannt.

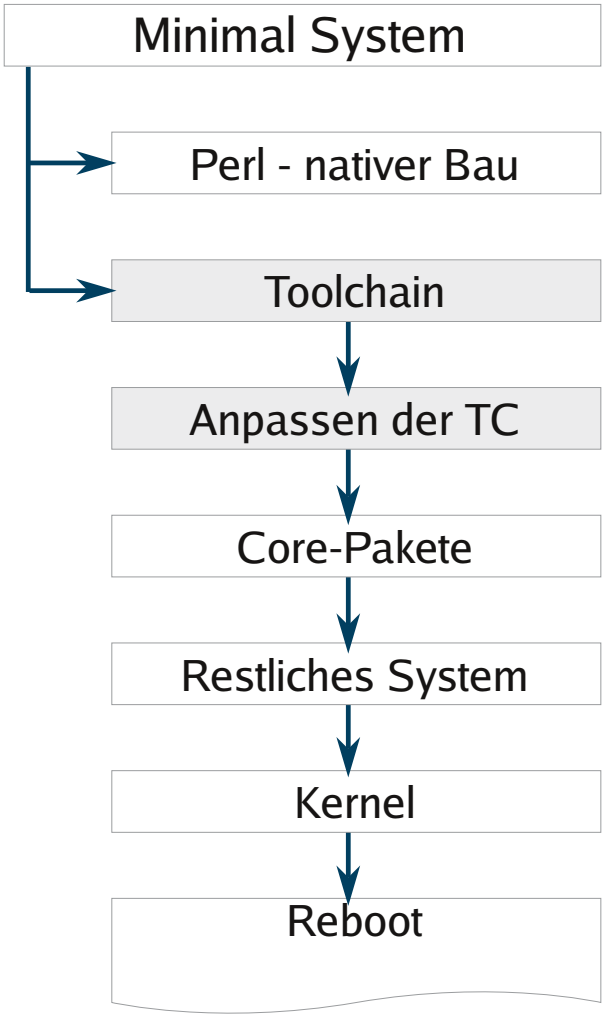
Die Pakete bauen wir in einem dedizierten Verzeichnis `/usr/src/buildtree`, dieses muss noch angelegt werden.

Grundsystem | Sonderrolle Perl



Perl wird von einigen Paketen benötigt und gehört damit ins Minimalsystem. Leider lässt es sich nur mit sehr viel Aufwand cross-kompilieren, daher wird es hier nativ gebaut und nach `/tools` installiert.

Grundsystem | Toolchain



Die endgültige Toolchain, die auch später auf dem System laufen wird. Diese Toolchain soll sofort verwendet werden, dies muss dem System mitgeteilt werden.

```
:> gcc -dumpspecs | sed -e 's@/tools@@g' \
-e '/\*startfile_prefix_spec:/{n;s@.*@/usr/lib/ @}' \
-e '/\*cpp:/{n;s@$@ -isystem /usr/include@}' > \
$(dirname $(gcc --print-libgcc-file-name))/specs
```

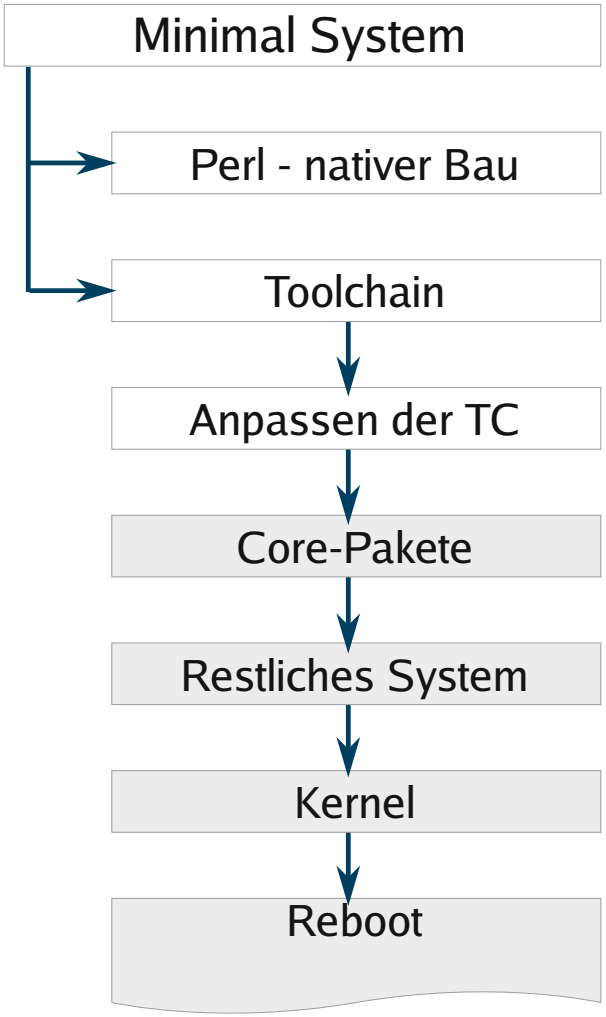
Dies sollte unbedingt überprüft werden, das System ist sonst nicht lauffähig!

```
:> echo 'main(){}' > dummy.c
cc dummy.c -v -Wl,--verbose &> dummy.log
readelf -l a.out | grep ':  /lib'
```

Die Antwort sollte so aussehen:

```
[Requesting program interpreter: /lib/ld-linux.so.2]
```

Grundsystem | Restliches System



- Nun wird der Rest des Systems gebaut, die Pakete ersetzen sukzessive die Programme im `/tools` Verzeichnis.
- Sind alle grundlegenden Pakete gebaut, so wird der endgültige Kernel gebaut. In der Datei `/boot/grub/menu.lst` ist der neue Kernel einzutragen und das neue System kann gebooted werden.

Funktioniert alles wie vorgesehen, so kann nun aufgeräumt werden:

- Löschen der temporären Verzeichnisse
- Zurücksetzen der Variablen `PATH`
- Entfernen der Quellen in `/sources`
- Löschen des temporäre Kernels

Outline

Distribution | Anforderungen

Was muss eine Paketverwaltung leisten?

- Zusammenbau von Quellcode zu individuellen, **wiederverwertbaren** Paketen
- **sicheres** Installieren, Deinstallieren und Erneuern von Paketen
- Schutz vor versehentlichem Überschreiben, evtl. BackUp von Konfigurationsdateien
- Verteilung der Pakete über ein Netzwerk

Distribution | Vergleich: Installation

Debian

```
apt-get install Paket
```

RPM

```
rpm -i Paket
```

Arch Linux

```
pacman -S Paket
```

Slackware

```
installpkg Paket
```

Distribution | Vergleich: Deinstallation

Debian

```
apt-get remove Paket
```

RPM

```
rpm -e Paket
```

Arch Linux

```
pacman -R Paket
```

Slackware

```
removepkg Paket
```

Distribution | Vergleich: Update

Debian

```
apt-get upgrade Paket
```

RPM

```
rpm -U Paket
```

Arch Linux

```
pacman -S Paket
```

Slackware

```
upgradepkg Paket
```

Distribution | Build-Skripte

Debian:

- Makescript mit Targets
- mehrere Dateien

RPM:

- Beschreibungsdatei mit eigener Syntax

Arch Linux:

- Shell-Skript mit Variablen und build-Funktion

Slackware:

- Shell-Skript, das direkt ausgeführt wird
- keine Möglichkeit, Abhängigkeiten abzubilden

Distribution | Fazit

Wir haben uns für “pacman” von Arch Linux entschieden, da

- die Build-Skripte sehr einfach sind
- die Pakete einfache tar-Archive mit wenigen Kontroll-Dateien sind
- die Tools zum Bau eines Repositories gleich mitgeliefert werden

Distribution | Überblick

Welche Init-Systeme gibt es?

Sys-V-Init ist das klassische Init-System für Linux. Für jeden Runlevel existiert ein Verzeichnis mit Start- und Kill-Skripten, die alle (alphabetisch) nach einander ausgeführt werden.

BSD-Style wie der Name schon sagt, ist es das System unter BSD. Beim Wechsel in einen Runlevel wird ein Skipt gestartet, das alle benötigten Applikationen startet und die anderen beendet.

Busybox liefert auch ein Init-System. Da auf Busybox-Systemen die meisten Anwendungen von in dem Programm Busybox selbst implementiert sind, kann auf eine komplizierte Initialisierung verzichtet werden - nur externe Programme werden hierüber geladen. Besonderheit: Es gibt nur einen Runlevel!

Upstart Upstart ist noch relativ neu und wurde von Ubuntu entwickelt. Anwendungen werden "On Demand" gestartet und entladen.

Distribution | Vielen Dank!

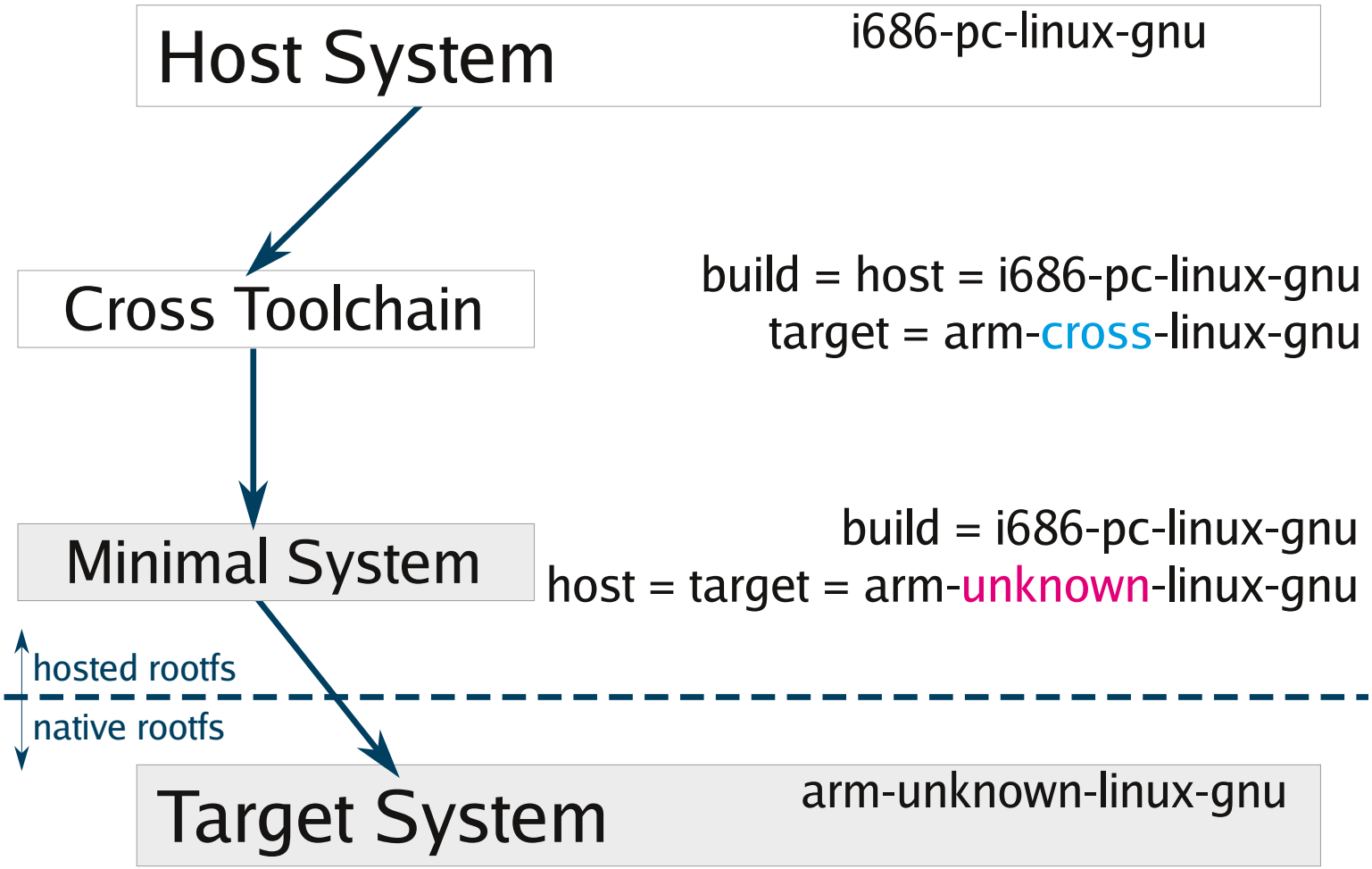
Quellen:

- “Linux from Scratch” http://www.linuxfromscratch.org
- “Cross Linux from Scratch” http://cross-lfs.org
- “Do-It-Yourself Linux” http://www.diy-linux.org
- K. Yahmour et al., “Building Embedded Linux Systems”, O’Reilly 2008
- linexa http://www.linexa.de

Kontakt:

- a.vogt@linexus.de

Zusatz | zur Cross-TC



Zusatz | Erstellen eines RAW-Images

```
:> dd if=/dev/zero of=/pfad/zum/image.iso bs=1M count=10k
:> losetup /dev/loop0 /pfad/zum/image.iso
:> mkfs.ext3 /dev/loop0
:> mkdir -p /pfad/zum/build/verzeichnis
:> mount /dev/loop0 /pfad/zum/build/verzeichnis
```

Zusatz | Bootskript

```
#!/bin/sh
# mount the kernel virtual fs
mount /proc
mount /sys

# remount the rootfs rw
mount -n -o remount,rw /

# set up network
ip addr add 127.0.0.1/8 label lo dev lo
ip link set lo up

ip addr add 192.168.0.100/24 dev eth0
ip route add default via 192.168.0.1 dev eth0
ip link set eth0 up

# set PATH
export PATH=/bin:/usr/bin:/sbin:/usr/sbin:/tools/bin:/tools/sbin

# start shell
/bin/sh
```

Zusatz | Build-Skript Debian

```
#!/usr/bin/make -f

package = diff
docdir = debian/tmp/usr/share/doc/$(package)

CC = gcc
[...]
build:
$(touchfiles)
./configure --prefix=/usr \
--build=$(DEB_BUILD_GNU_TYPE) --host=$(DEB_HOST_GNU_TYPE)
$(MAKE) CC="$(CC)" CFLAGS="$(CFLAGS)"
touch build

clean:
[...]
binary-indep: build

binary-arch: build
rm -rf debian/tmp
[...]
gzip -r9 debian/tmp/usr/share/man
dpkg-shlibdeps debian/tmp/usr/bin/*
dpkg-gencontrol -isp -p$(package)
cd debian/tmp && \
md5sum `find * -type f ! -regex "DEBIAN/.*"` > DEBIAN/md5sums
chown -R root:root debian/tmp
chmod -R u+w,go=rX debian/tmp
dpkg --build debian/tmp ..

binary: binary-indep binary-arch
```

Zusatz | Build-Skript RPM

```
Summary: A GNU collection of diff utilities
Name: diffutils
Version: 2.8.1
[...]
URL: http://www.gnu.org/software/diffutils/diffutils.html
Source: ftp://ftp.gnu.org/gnu/diffutils/diffutils-%{version}.tar.gz
[...]
License: GPLv2+
Requires(post): /sbin/install-info
Requires(preun): /sbin/install-info
[...]
%description
[...]
%prep
[...]
%build
%configure
make PR_PROGRAM=%{_bindir}/pr

%install
rm -rf $RPM_BUILD_ROOT
make DESTDIR=$RPM_BUILD_ROOT install

( cd $RPM_BUILD_ROOT
  gzip -9nf .%{_infodir}/diff*
  mkdir -p .%{_mandir}/man1
  for manpage in %{SOURCE1} %{SOURCE2} %{SOURCE3} %{SOURCE4}
  do
    install -m 0644 ${manpage} .%{_mandir}/man1
  done
)
[...]
```

Zusatz | Build-Skript Arch Linux

```
pkgname=diffutils
pkgver=2.8.1
pkgrel=1
pkgdesc="Utility programs used for creating patch files"
arch=('i686')
url="http://www.gnu.org/software/diffutils"
license=('GPL')
groups=('core')
depends=('glibc')
source=(http://www.linuxfromscratch.org/patches/lfs/development/diffutils-2.8.1-i18n-1.patch
http://ftp.gnu.org/gnu/$pkgname/$pkgname-$pkgver.tar.gz)
md5sums=(
'c8d481223db274a33b121fb8c25af9f7'  #   diffutils-2.8.1-i18n-1.patch
'71f9c5ae19b60608f6c7f162da86a428') #   diffutils-2.8.1.tar.gz


build() {
  cd $startdir/src/$pkgname-$pkgver

# POSIX requires the diff command to treat whitespace characters
# according to the current locale.
  patch -Np1 -i ../diffutils-2.8.1-i18n-1.patch || return 1

  touch man/diff.1

  ./configure   \
  --prefix=/usr

  make || return 1
  make DESTDIR=$startdir/pkg install
}
```

Zusatz | Build-Skript Slackware

```
#!/bin/sh

# Copyright 2005-2009  Patrick J. Volkerding, Sebeka, MN, USA
# All rights reserved.
[...]
VERSION=2.8.1
ARCH=${ARCH:-x86_64}
BUILD=${BUILD:-3}

CWD=$(pwd)
TMP=${TMP:-/tmp}
PKG=$TMP/package-diffutils
[...]
rm -rf $PKG
mkdir -p $TMP $PKG

cd $TMP
rm -rf diffutils-$VERSION
tar xzvf $CWD/diffutils-$VERSION.tar.gz
cd diffutils-$VERSION
[...]
CFLAGS="$SLKCFLAGS" \
./configure \
  --prefix=/usr \
  --program-prefix="" \
  --program-suffix="" \
  $ARCH-slackware-linux

make || exit 1
make install DESTDIR=$PKG || exit 1
[...]
makepkg -l y -c n $TMP/diffutils-$VERSION-$ARCH-$BUILD.txz
```