Systemsoftware

Toolchain + Co

Prof. Dr. Michael Mächtel

Informatik, HTWG Konstanz

Version vom 03.04.17

Übersicht

Übersicht

Komponente + Distribution

- Komponentenentwicklung
  - Konzeption und Realisierung (Programmierung) von Systemteilen
    - Treiber
    - Kernel
    - Applikation
    - Webserver
    - Datenbank
- Distributionsentwicklung
  - Zusammenstellung der Komponenten zu einem Gesamtsystem
    - Ausgangsbasis: Pakete inklusive einer Anweisung zur Zusammenstellung
    - Ergebnis: Images

Komponentenentwicklung

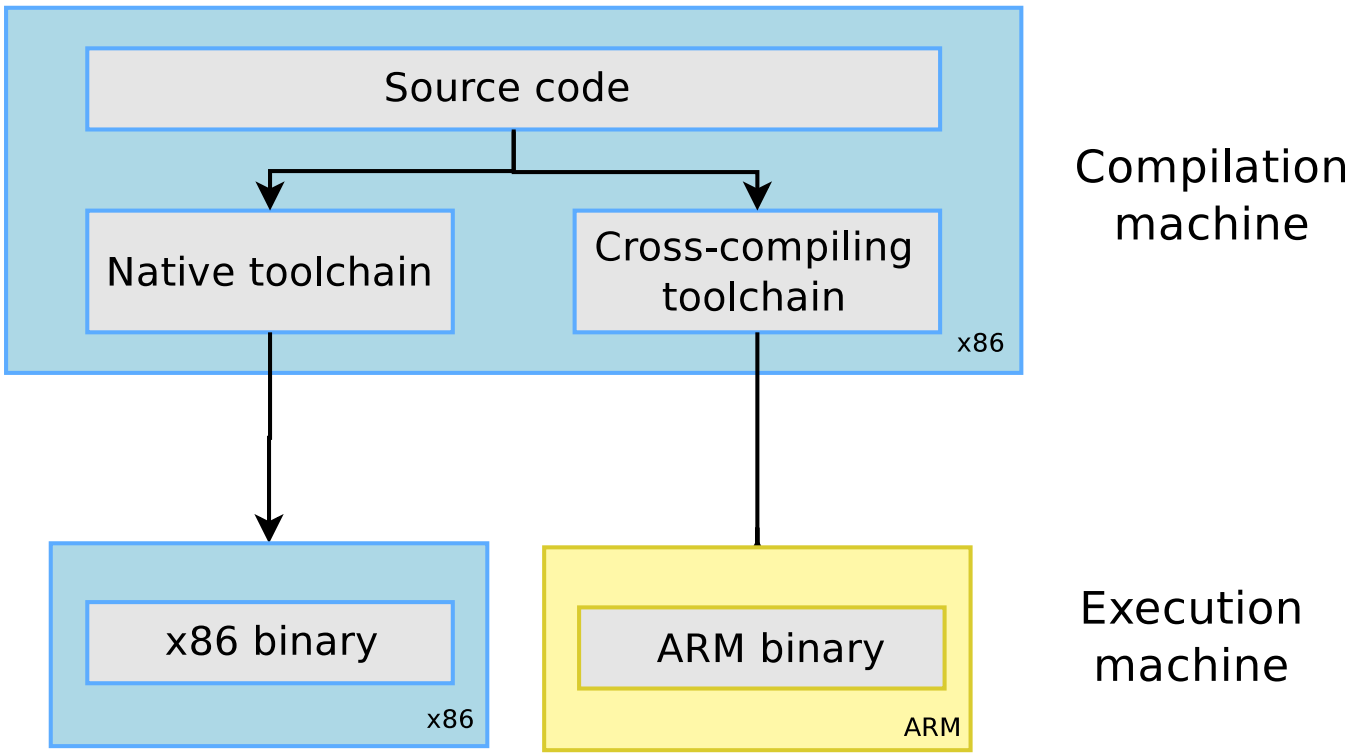
- Im Prinzip bekannt (wie im Studium gelernt/geübt)
  - Editor, Compiler, Linker, ...
  - Host-/Target-Entwicklung
- Linux
  - Host und Target haben das gleiche Betriebssysteme
  - Entwicklung kann über lange Strecken auf dem Host-System erfolgen
- Für die Target-Entwicklung:
  - (Cross-) Compiler
  - (Cross-) Linker
  - Emulator (Qemu)
- **Generierung** der Cross-Entwicklungswerkzeuge notwendig

Distributionsentwicklung

- Aufgabe
  - Auswahl und Zusammenstellung der versch. Software Pakete
  - Konfiguration der Pakete
  - Generierung der Distribution
- Distributionsgenerator
  - Generierung der Komponenten
  - Zusammenbau des Root-Filesystems
  - Ergebnis: Für den Target geeignete Imagedateien
  - dazu nötig: *Generierung der* (Cross)-Toolchain
  - Vorteil:
    - Distributionsgeneratoren erstellen eigene Toolchain

Übersicht

Code Erzeugung



Was ist eine Toolchain?

- Werkzeugkette, die für die Programmierung von Anwendungen und Betriebssystemen eingesetzt wird:
  - Make, für die Automatisierung des Build- und Kompilierungsvorgangs.
  - Compiler Collection, mit Compilern für verschiedene Programmiersprachen.
  - Binutils, Linker, Assembler und andere Tools.
  - Debugger.
  - Build System (z.B. GNU Autotools): Autoconf, Autoheader, Automake, Libtool, ...
- Cross-Toolchain:
  - Wenn Target andere Plattform als Host System hat.
  - Beispiel: ARM Binaries auf X86 Systemen erstellen.

binutils

- Binutils ist ein Set von Tools, um Binaries für eine CPU zu erzeugen:
  - **as**, the assembler, generiert binary code aus assembler source code
  - **ld**, the linker
  - **ar**, **ranlib**, generiert .a archives, wird zur Erstellung von Libraries benutzt
  - **objdump**, **readelf**, **size**, **nm**, **strings**: Tools zur Untersuchung von Binaries.
    - Sehr nützliche Analyse-Tools!
  - **strip**, entfernt überflüssige Teile im Binary, um die Größe zu reduzieren
  - Quelle: http://www.gnu.org/software/binutils
    - GPL Lizenz

Anforderungen

- die verschiedenen Software-Komponenten der Toolchain stellen zahlreiche Parameter und Optionen zur Verfügung.
- Optionen für die jeweilige Plattform müssen betrachtet und verstanden werden.
- die jeweils passenden Kernel Headers werden benötigt.
- je nach Plattform müssen verschiedene Patches verwendet werden.
- Hilfreich ist die Erfahrung mit Building- und Konfigurations-Tools (make, autoconf …).

ABI (1)

- Für die Erzeugung der ARM Toolchain muss ein entsprechendes Application Binary Interface definiert werden.
- Das Application Binary Interface (ABI) definiert die Aufruf Konvention des Programms:
  - Wie werden Parameter bei Funktionsaufrufen weitergegeben?
  - Wie wird der Rückgabewert der Funktion übergeben?
  - Wie werden Systemfunktionen aufgerufen (z.B. Per TRAP …)?
  - …

ABI (2)

- Alle Binaries im System müssen mit der gleichen ABI compiliert werden
- Der Kernel muss das entsprechende ABI unterstützen
- Für die ARM Architektur unterscheidet man zwischen
  - OABI und EABI
- Für MIPS
  - o32, o64, n32, n64
- Quelle: http://en.wikipedia.org/wiki/Application_Binary_Interface

Kommerzielle Toolchains

- Hersteller gehen mit der GPL im Allgemeinen sehr fair um und geben ihre Änderungen am Source-Code auch wieder frei.
- Neben der eigentlichen Toolchains gibt es je nach Distribution eigen entwickelte Tools.
- Die von den Herstellern entwickelten 'grafischen Tools' sehen proprietär aus und nur als potentieller Kunde erfährt man oft das Lizenzmodell hintern diesen Toolkits.

Vorteile kommerzieller Toolchains

- Technische Vorteile
  - Gut getestete und unterstützte Kernel und Tool Versionen.
  - Beinhalten frühe Patches, die teilweise im Mainstream-Kernel noch nicht zu finden sind.
- Komplette Development Toolsets: von Konfiguration bis fertigem Software Image, incl. grafischer Entwicklungswerkzeuge.
- Entwicklungswerkzeuge für verschiedene Host-Systeme erhältlich: GNU/Linux, Solaris, Windows ...
- Support Service:
  - Nützlich, wenn man keine eigenen Support Ressourcen hat.
  - 'Long term support commitment'

Freie Binary Toolchains

- verschiedene freie fertige Toolchains (Binary) im Internet zum Download.
- Nachteile:
  - Toolchains müssen genau an der Stelle im Dateisystem installiert werden, an welcher sie kompiliert wurden (feste Pfade).
  - Es muss sichergestellt sein, dass die ausgewählte Toolchain auch genau den gewünschten Anforderungen (Konfiguration) entspricht.
  - oft veraltete Version der Toolchain.
- Vorteil:
  - ‘Ready To Go’

Scripte zur Erstellung von Toolchains

- Verschiedene Skriptsammlungen im Internet erleichtern das Erstellen einer Cross-Entwicklungsumgebung.
- Vorteil:
  - kann genau auf eigene Bedürfnisse konfiguriert werden.
  - patched die Sourcen **automatisch**, abhängig von gewählter Zielarchitektur
- Beispiele:
  - crosstool, crosstool-NG
  - ptxdist
  - emdebian
  - **buildroot**
  - openembedded
- Einige dieser Tools gehen weit über die Erstellung der Toolchain hinaus und erstellen komplette Distributionen incl. Paketmanager (z.B. openembedded)

Crosstool

- Crosstool
  - http://kegel.com/crosstool/
  - Crosstool von Dan Kegel erstellt über Skripte automatisch eine Cross-Entwicklungsumgebung
  - wird derzeit nicht mehr gepflegt, benutzt veraltete Sourcen für Toolchain
- Crosstool Next Generation
  - http://crosstool-ng.org
  - aktuelles und gepflegtes Projekt
  - Hilft bei Konfiguration und Erstellung einer Toolchain (incl. Kernel Konfiguration und Erstellung für Target)

Crosstool Matrix

gcc-4.1-20050709 cgcc-3.3.6 glibc-2.3.2 binutils-2.15 linux-2.6.11.3 hdrs-2.6.11.2	gcc-4.1-20050709 cgcc-3.3.6 glibc-2.3.2 binutils-2.15 linux-2.6.11.3 hdrs-2.6.11.2	gcc-4.0.1 cgcc-2.95.3 glibc-2.2.2 binutils-2.16.1 linux-2.6.11.3 hdrs-2.6.11.2	gcc-4.0.1 cgcc-2.95.3 glibc-2.2.2 binutils-2.16.1 linux-2.6.11.3 hdrs-2.6.11.2	gcc-4.0.1 cgcc-2.95.3 glibc-2.2.2 binutils-2.16.1 linux-2.6.11.3 hdrs-2.6.11.2	gcc-4.0.1 cgcc-3.3.6 glibc-2.3.2 binutils-2.16.1 linux-2.6.11.3 hdrs-2.6.11.2	gcc-4.0.1 cgcc-3.3.6 glibc-2.3.2 binutils-2.16.1 linux-2.6.11.3 hdrs-2.6.11.2	gcc-4.0.1 cgcc-3.3.6 glibc-2.3.2 binutils-2.16.1 linux-2.6.11.3 hdrs-2.6.11.2	gcc-4.1-20050716 cgcc-2.95.3 glibc-2.2.2 binutils-2.16.1 linux-2.6.11.3 hdrs-2.6.11.2	gcc-4.1-20050716 cgcc-2.95.3 glibc-2.2.2 binutils-2.16.1 linux-2.6.11.3 hdrs-2.6.11.2	gcc-4.1-20050716 cgcc-2.95.3 glibc-2.2.2 binutils-2.16.1 linux-2.6.11.3 hdrs-2.6.11.2	gcc-4.1-20050716 cgcc-3.3.6 glibc-2.3.2 binutils-2.16.1 linux-2.6.11.3 hdrs-2.6.11.2	gcc-4.1-20050716 cgcc-3.3.6 glibc-2.3.2 binutils-2.16.1 linux-2.6.11.3 hdrs-2.6.11.2	gcc-4.1-20050716 cgcc-3.3.6 glibc-2.3.2 binutils-2.16.1 linux-2.6.11.3 hdrs-2.6.11.2	
kernel fail gdb ok	kernel fail gdb ok	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	ok gdb ok	ok gdb ok	ok gdb ok	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	alpha
ICE gdb FAIL	ICE gdb FAIL	ok gdb ok	ok gdb ok	ok gdb ok	ok gdb ok	ok gdb ok	ok gdb ok	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	arm
ICE gdb FAIL	ICE gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	ok gdb ok	ok gdb ok	ok gdb ok	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	arm9tdmi
FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	arm-iwmmxt
FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	arm-softfloat
ICE gdb FAIL	ICE gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	ok gdb ok	ok gdb ok	ok gdb ok	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	arm-xscale
FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	armeb
FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	armv5b-softfloat
kernel ICE gdb ok	kernel ICE gdb ok	ok gdb ok	ok gdb ok	ok gdb ok	ok gdb ok	ok gdb ok	ok gdb ok	ok gdb ok	ok gdb ok	ok gdb ok	ok gdb ok	ok gdb ok	ok gdb ok	i686
FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	ia64
kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	m68k
kernel ICE gdb ok	kernel ICE gdb ok	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	mips
kernel ICE gdb ok	kernel ICE gdb ok	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	mipsel
kernel ICE gdb ok	kernel ICE gdb ok	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	ok gdb ok	ok gdb ok	ok gdb ok	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	ok gdb ok	ok gdb ok	ok gdb ok	powerpc-405
kernel ICE gdb ok	kernel ICE gdb ok	ok gdb ok	ok gdb ok	ok gdb ok	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	ok gdb ok	ok gdb ok	ok gdb ok	ok gdb ok	ok gdb ok	ok gdb ok	powerpc-750
kernel ICE gdb ok	kernel ICE gdb ok	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	ok gdb ok	ok gdb ok	ok gdb ok	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	ok gdb ok	ok gdb ok	ok gdb ok	powerpc-860
kernel ICE gdb ok	kernel ICE gdb ok	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	powerpc-970
kernel fail gdb ok	kernel fail gdb ok	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	FAIL gdb FAIL	FAIL gdb FAIL	FAIL gdb FAIL	kernel fail gdb ok	kernel fail gdb ok	kernel fail gdb ok	s390

Übersicht

Buildroot

- Sammlung von Skripten zur
  - Konfiguration,
  - zur Generierung und
  - zum Zusammenbau einer Distribution
- Download des Quellcodes (notwendige Pakete)
- Erstellen einer Cross-Development-Toolchain
  - gcc, binutils, uclibc, gdb (für Host und Target)
- Open-Source
- Download unter http://buildroot.uclibc.org

Buildroot Software Komponenten

- enthält zahlreiche Softwarepakte (700+):
  - Busybox
  - Netzwerk
  - Grafik (GUI)
  - Audio
  - Kernel
  - Bootloader
  - Eigene Software
  - …

Buildroot konfigurieren

- **make menuconfig**
  - Konfiguriert buildroot selbst
    - konfiguriert Parameter der Toolchain Erstellung
    - ermöglicht auch externe Toolchains
  - Auswahl des Targets
  - Auswahl und Konfiguration verschiedener Softwarekomponenten
    - Set von Applications und Bibliotheken
  - Auswahl des zu erzeugenden Filesystem (Images) - Kernel und Bootloader Konfiguration

Busybox konfigurieren

- **make busybox-menuconfig**
  - Konfiguriert busybox
  - Sollte erst nach einem ersten Compilationslauf aufgerufen werden
  - Nach der Konfiguration ist ein erneuter Compilationslauf zu starten
    - Aufruf von **make**

uclibc konfigurieren

- **make uclibc-menuconfig**
  - Default-Konfig reicht in vielen Fällen
  - Sollte ebenfalls erst nach einem ersten Generierungslauf aufgerufen werden.
  - Nach der Konfiguration der uclibc muss die Konfigurationsdatei an die richtige Stelle kopiert werden
    - cp .config toolchain/uClibc/uClibc.config

Linux Kernel konfigurieren

- **make linux-menuconfig**
  - Konfiguration des Linux-Kernels

Generierung der Distribution

- Benötigte Quellcodepakete werden von dem jeweiligen Quellcodeserver heruntergeladen.
- Die Cross-Development-Toolchain wird konfiguriert, gepatcht, generiert und installiert.
- Quellcode wird – wo notwendig – gepatcht.
- Pakete werden konfiguriert und generiert.
- Der Betriebssystemkern wird generiert.
- Das Root-Filesystem wird generiert und mit der erstellten Software gefüllt.

Buildroot Verzeichnisstruktur 1

- *output/images/*
  - Enthält die generierten Imagedateien (kernel, bootloader, root-Filesystem).
  - Hier findet sich also das Ergebnis des Build-Prozesses.
- *output/target/*
  - Hier befindet sich das ROOT-FS des Targets
- *output/build/*
  - Verzeichnis, in dem die zur Generierung eines Images notwendige Werkzeuge abgelegt werden.
- *output/host/usr*
  - Verzeichnis, das die generierten Cross-Development-Werkzeuge enthält.

Buildroot Verzeichnisstruktur 2

- *target/*
  - In diesem Verzeichnis befinden sich Konfigurationsoptionen für einzelne Projekte (beispielsweise Linux-Kernel oder u-boot).
- *dl/*
  - In diesem Verzeichnis werden die Download-Dateien abgelegt (Sourcecode).
- *docs/*
  - Dokumentation zu buildroot.
  - Einstieg über *docs/buildroot.html* .

Buildroot Verzeichnisstruktur 3

- *configs/*
  - Konfigurationsinformationen für einige Pakete.
- *support/*
  - Skripte, die von buildroot selbst verwendet werden.
- *package/*
  - Enthält zu jedem Paket notwendige Patches.

Buildroot Verzeichnisstruktur 4

- *toolchain/*
  - Enthält Konfigurationen beziehungsweise Patches für die Entwicklungssoftware (Toolchain) selbst.
- *stamps/*
  - Enthält von Buildroot intern generierte und verwendete Zustandsinformationen des Generierungsprozesses.

Anpassungen am Target Filesystem (RootFS)

- Das target Filesystem liegt unter *output/target/*
- Das Default Skeleton für die Erstellung des Root-FS liegt unter *fs/skeleton*
  - ein eigenes Skeleton kann auf Basis des Default Skeleton erstellt werden
  - BR2_ROOTFS_SKELETON_CUSTOMIZE und BR2_ROOTFS_SKELETON_CUSTOM_PATH erlauben die freie Platzierung eines eigenen Skeletons
- *package/customize/source/*
  - Dateien, die hier abgelegt werden, werden später ins Root-Filesystem kopiert, wenn das Customize Paket ausgewählt ist
- Weitere Infos siehe BuildRoot Doku: “Customizing The Generated Target Filesystem”

Weitere Anpassungen am RootFS

- Zugriffsrechte des zu erzeugenden Filesystems
  - Lassen sich über die Datei *target/generic/device_table.txt* einstellen.
- post-build-script
  - Wird aufgerufen nach der Generierung der Pakete aber bevor das Root-Filesystem zusammengebaut wird.
  - Mit Hilfe des Skripts können beispielsweise Dateien in das Root-Filesystem kopiert werden.

Paketaufbau

- Die Generierung eines Softwarepakets wird über zwei Dateien gesteuert:
  - Dateierweiterung *.mk*
    - Ein Makefile, das den Download-, Konfigurations-, Patch-, Generierungs- und Installationsprozess beschreibt.
  - config.in
    - Beschreibt die Auswahlmöglichkeiten (Konfiguration eines Paketes).

Eigene Pakete: *Config.in*

- Um eigene Pakete in buildroot zu integrieren gehen Sie folgendermaßen vor:
  - Verzeichnis mit dem Paketnamen unterhalb des Verzeichnisses *package/* erzeugen.
  - *Config.in* unterhalb des neuen Verzeichnisses anlegen
  - Beispiel:

config BR2_PACKAGE_WGET

bool "wget"
help

Network utility to retrieve files from
http/ftp/etc...
http://wget.sunsite.dk/

Eigene Pakete: *Makefile*

- Makefile "...mk" anlegen.

```
<name_des_pakets>_VERSION
<name_des_pakets>_SOURCE
<name_des_pakets>_PATCH
<name_des_pakets>_CONFIGURE_CMDS
<name_des_pakets>_BUILD_CMDS
<name_des_pakets>_INSTALL_TARGET_CMDS
<name_des_pakets>_CLEAN_CMDS
<name_des_pakets>_POST_BUILD_HOOKS
<name_des_pakets>_POST_INSTALL_HOOKS
```

- über Defines (Hooks) werden die einzelnen Aktionen gesteuert.

Weiterführende Informationen

- Doku beim Paket selbst: docs/buildroot.html
- http://www.elektronikpraxis.vogel.de/themen/embeddedsoftwareengineering/implementierung/articles/173855/index.html
- http://buildroot.uclibc.org
- http://lists.busybox.net/pipermail/buildroot/2010-February/032488.html

Übersicht

Open Embedded: Übersicht

- Projekt Open Embedded
  - kann neben U-Clibc- auch Glibc-basierte Systeme cross-kompilieren.
  - Den Buildvorgang übernimmt das im Rahmen von Open Embedded entwickelte Programm ‘bitbake’.
  - Administrator die Datei ‘build/conf/local.conf’ an.
  - Das Programm ‘bitbake’
    - zusammen mit dem Paketsatz aufgerufen
    - führt den eigentlichen Übersetzungsvorgang aus.
  - bitbake task-base übersetzt die Basispakete,
  - ‘Bitbake world’ kompiliert alle.

Open Embedded: Vorteile/Nachteile

- Vorteile:
  - Grosse Auswahl an Softwarepaketen für das Embedded Device
  - die komplexe Abhängigkeiten der Softwarepakete untereinander wird von openembedded verwaltet
  - Grosse Entwickler und Benutzer Community rstellt z.B. ipkg Pakete für Paketverwaltung auf dem Embedded Device
  - flexibel
- Nachteile:
  - schwierig aufzusetzen und zu konfigurieren
  - hoher Einarbeitungsaufwand, wenig Dokumentation

bitbake

- verarbeitet die Informationen der Open Embedded Metadaten
- erstellt die gewünschten Outputs wie z.B. Images oder Pakete ‘from scratch’
- erstellt dazu automatisch die benötigte Cross-Entwicklungsumgebung (toolchain)
- lädt die Sourcen der benötigten Pakete von den original Quellen aus dem Internet
- patched die Sourcen automatisch (entspr. der Zielarchitektur)
- compiliert für das Target das Root-FS und die Pakete (ipkg, rpm, deb …).

Software Pakete in Open Embedded

- Die Informationen über die Software Pakete **packages/** für das Embedded Device stehen in den entsprechenden Bitbake Dateien **packages//*.bb**.
- In diesen Dateien ist beschreiben:
  - die Paket Informationen (Description, License, Maintainer, usw.)
  - Quelle der Sourcen (http://oder ftp://)
  - benötigte bitbake Module (autotools, pkgconfig, usw.)
  - speziell angepasste Build Skripte

Spinoff: Yocto

- Da OpenEmbedded eine sehr komplexe Umgebung darstellt, gab es in der Vergangenheit immer wieder SpinOffs davon
- Der derzeit aktivste Spinoff ist Yocto
- Quelle: https://www.yoctoproject.org

Übersicht

Mögliche Themen der Vertiefung

- Analyse von Toolchains
  - Erstellung einer geeigneten Cross-Toolchain und Benchmarks als Gegenüberstellung
  - Kriterien wie Hard- vs. Soft-Float Optionen prüfen
- Analyse von Distributionen
  - Anforderungen von Eingebetteten Systemen an Distributionen
  - Erstellung eines RootFS vollständig kompiliert aus Quellen mit der o.g. Toolchain
- Konfigurationsoptionen für Quellen der Distributionsgenerierung
- Aspekte der Binär-Paket Verteilung auf mehrere Devices
- Optimierung der Bootzeit (läuft als Teamprojekt)
- Optimierung des Energieverbrauchs

Entwurfsgrundsätze Embedded System (1)

- Einfaches HMI (Human Machine Interface), einfach zu bedienen.
- Keine Interaktion mit dem User notwendig: AUTO-Mode.
- Interaktion mit dem User ist möglich.
- Alle Eingaben werden auf Gültigkeit und Sinnhaftigkeit überprüft (syntaktische und semantische Überprüfung).
- Konfigurationen werden automatisch durchgeführt.
- Updates werden automatisch installiert.

Entwurfsgrundsätze Embedded System (2)

- Einheitliche, standardisierte Schnittstellen verwenden.
  - Keine eigenen Stecker verwenden, vorhandene Stecker auswählen.
  - Selbstdefinierte Schnittstellen offenlegen und lizenzkostenfrei zur Verfügung stellen.
  - Beispiel: Stromversorgung über USB.
- Lesbare und vor allem interpretierbare XML-Dateien verwenden.
- Wartungsfreundlich.
  - Einfacher Austausch von Batterien.
- Erweiterbarkeit (Folgegeschäfte).
- Denken Sie radikal!

Auto-Update: Anforderungen

- Geräte, die mit dem Internet verbunden sind suchen regelmäßig im Internet nach Updates und spielen diese automatisiert ein.
  - Gerät darf durch ein fehlgeschlagenes Update nicht unbrauchbar werden.
  - Updates müssen eine digitale Unterschrift tragen, die Unterschrift muss vom Gerät überprüft werden.
  - Beim Update dürfen keine User-Daten/Konfigurationen verloren gehen.

Auto-Update: Realisierungsmöglichkeiten (1)

- System liegt in Form eines Images vor, das auf das Gerät transferiert wird.
  - Internet
  - SD-Karte
- Beim Booten wird das neue Image erkannt und als „jünger“ und damit zu installieren identifiziert.
- Die Unterschrift und damit Gültigkeit und Unversehrtheit der Daten wird verifiziert.
- Es wird überprüft, ob auf ein funktionierendes Image (im Fall eines Fehlschlages) zurückgegriffen werden kann.

Auto-Update: Realisierungsmöglichkeiten (2)

- Das neue Image wird installiert (z.B. durch Kopieren in den Flashspeicher).
- Die Installationsroutine setzt im Flash ein Flag das besagt, dass die Installation „fehlgeschlagen“ ist.
- Das neue Image wird gestartet. Nach dem Booten wird das Flag „fehlgeschlagen“ gelöscht.

Auto-Config: Anforderungen

- Initiale Systemparameter müssen automatisch bestimmt werden.
- Parameter, die sich nicht automatisch bestimmen lassen, müssen automatisch bestimmt werden …
- Analyse:
  - Warum lassen sich Parameter nicht automatisch bestimmen?
  - Was für Informationen fehlen?
  - Woher können diese Informationen kommen?
  - Wie kann man (zur Not) diese Informationen hinterlegen?

Auto-Config: Realisierungsmöglichkeiten

- Daten werden automatisch bestimmt/berechnet.
- Vorhandene Informationen werden hierzu genutzt.
- Daten werden hinterlegt, über eine leicht zu merkende „Kennung“ findet eine automatische Installation statt.
- Konfigurationsdaten/Kennung werden in Form einer Datei (SD-Karte, USB-Stick, usw.) dem Gerät zur Verfügung gestellt.
- Weitere Ideen?

Betrieb (1)

- Zustandsüberwachung
  - Daten müssen für den User einsehbar sein.
  - Es muss klar sein, welche datenschutzrechtlichen Informationen erfasst (und gespeichert) werden.
- Ausgabe von Fehlerinformationen im Klartext
  - Keine obskuren Fehlercodes (F16).
  - Direkte Angabe von Reparaturanweisungen. (Positive) Beispiele:
    - Kopierer.
    - Auto.

Betrieb (2)

- Automatisierte Ersatzteilbeschaffung.
- Weitergabe von Daten anonymisiert.
- Datenübermittlung an (zentrale) Server nur nach Freigabe durch den User.
- Schnittstellen für die Ausgabe von Daten vorsehen.
  - Webinterface
  - ???

Beispiel

- Entwurf einer Waschmaschine:
  - Welche innovativen Feature hat Ihre Waschmaschine? Was ist das Alleinstellungsmerkmal?
  - Wie sieht das Bedienkonzept aus?
  - Welche Interfaces gibt es?
  - Was benötigen Sie an Hardware?
  - Welche Softwarekomponenten werden benötigt?