Flashimages und YADDs mit newmake

Table of contents

1. Versionen

An English version of this document is available here.

Diese Version ist eine Übersetzung des englischsprachiges Originaldokument. Bitte kontrollieren Sie ggfl. falls das Originaldokument in eine neuere Version vorliegt. Diese Version ist vom 19. Juli 2006.

Die Versionsgeschichte befindet sich nur in der englischsprachige Version.

2. Einleitung

Dieses Dokument behandelt newmake aus Sicht des Benutzers. Es behandelt Image- und yadd-Herstellung und einfache Benutzeranpassungen ("customization"). Die Architektur von *newmake* wird in einem anderen Dokument beschrieben.

2.1. Zur Geschichte

Vor einigen Jahren war Imageherstellen für die Tuxbox eine schwarze kunst. Die Makefile-Unterstützung war, insbesondere für andere Images als cramfs-images, ziemlich lückenhaft. Die CVS Werkzeuge waren schlecht, oder unvollständig. Noch schlimmer, einige Teile wurden absichtlich geheim gehalten, nämlich das Werkzeug, jetzt als `mkflfs` bekannt, jetzt im CVS-directory `.../hostapps/mkflfs` vorhanden. Laut einer Forumsbeitrag von dieser Zeit waren die meisten Entwickler nicht in der Lage, eigene Images herzustellen. Die "Gilde der Imagehersteller" wurde geboren. Von dieser Zeit sind die "AlexW-Images" weithin am bekanntesten: hauptsächlich bestehend aus CVS-sources, aber mit einigem mehr-oder-weniger den geheim gehaltenen "Fixes", (vermutlich) notwendig für das Herstellen eines funktionierendes Images aus dem CVS-Quellen.

Im August 2003, in einem Projekt, das sich "GNU DBox2 Software-Projekt" nannte, wurde es in zunehmendem Maße peinlich, `mkflfs` geheim zu halten, und die Quellen für `mkflfs` wurden in CVS eingecheckt. Auch die Funktionalität des Makefiles wurde stufenweise verbessert. Noch wurde viel zu wünschen übrig: Funktionalität, Pflegbarkeit, gesunde Software-Design. Ein Image von reinen CVS-Dateien zu bauen war nicht wirklich möglich.

In 2004 wurde das YADI ("Yet Another DBox Image") Projekt geboren. (Bitte nicht "YADI" und "YADD" verwechseln!) Sein Ziel war, Imagebuilden zu automatisieren und zu vereinfachen. Für dieses Zweck wurden eine Anzahl von Skripte und Patches gesammelt und/oder geschrieben. Zusätzlich wurden flashfertige Images zur Verfügung gestellt.

YADI war ein grosser Erfolg. Das Ziel wurde erreicht. Images wurden zur Verfügung gestellt, die sich (fast?) vollständig auf freier Software basierte -- sowohl in seinem Inhalt als auch bezüglich die involvierte Werkzeugen, in eine Weise, die für den Benutzer

transparent war. Mit dem YADI-Skript war das automatische Imagebuilden möglich. Jedoch, statt die grundlegende Schwäche im CDK Imageprozeß zu adressieren, stellten sie Skripte zum Imagebauen zur Verfügung. Sie stellten nicht eine Buildsystem für ein Software-Projekt zur Verfügung. Software-Projekt werden mit einem Software-Buildsystem gehandhabt, wie make, oder ein neuerer Nachfolger, wie Ant oder Maven, nicht mit Shellskripte.

newmake, momentan als alternative Branch in CVS, versucht diese Schwächen zu adressieren.

Ich möchte mich hier bei jedem, der Bugreports und Feedback geliefert haben, bedanken. Insbesonderes gilt dies für dietmarw, der *newmake* benutzt, um die dietmarw-Images zu erzeugen.

2.2. Ziel

Das Ziel des vorliegenden Artikels ist, dem Leser mit grundlegendem Know-how zu versehen. Es ist nicht das Ziel, eine idiotensichere Schritt-bei-Schritt Anweisung bereitzustellen (wie die sogenannte HOWTOs). Kenntnis in der Umgang mit Shellskripte wird für viele Teile, insbesondere das Customization-Kapitel und der Anhang, aber nicht für image/yadd Kreation in seiner einfachsten Form gefordert.

Das vorliegende Dokument versucht nicht die innere Funktion des Makefile und des Makeprozesses zu beschreiben. Für dieses wird der Leser auf den Quellen (die sogar ein wenig kommentiert sind) hingewiesen, und zu den relevanten Threads im cross development kit Forums des Tuxbox Forums. Alle Optionen für `configure` werden auch nicht beschrieben, nur die Allgemeinste und Wichtigste.

2.3. Wie schwierig ist es?

Meine Antwort würde sein: Es ist so schwerig wie diesen Artikel zu lesen und verstehen. Der Leser, der ohne Probleme den Inhalt versteht sollte keine Probleme haben; für den Leser, für den das Meiste von diesem Text Kauderwelsch ist, sollte vielleicht bei fertige Images bleiben.

2.4. Allgemeines

Es gibt zwei mögliche Ziele: Entweder "YADD" oder Image. "Ein YADD" besteht aus einigen Files, die das dBox vom TFTP-Service lädt, sowie einem Filesystem, von einem NFS-Server (siehe diesen Artikel) für die dBox zur Verfügung gestellt. Diese Betriebsart hat viele Vorteile, insbesonderes während Softwareentwicklung oder beim Erlernen des Systems. Der Name "YADD" bedeutete einmal "Yet Another DBox Distribution" ("noch eine dBox Distribution"). Leider hat sich dieser irreführende und durchaus alberne Namen durchgesetzt.

Mein Vorschlag für den angehende Image/YADD-Lehrling ist: Bauen Sie zuerst ein

YADD mit Ihrem Favorit-GUI, und lernen Sie, mit ihm zu arbeiten. Folgender Schritt würde sein, ein jffs2-image, wieder mit Ihrem Liebling GUI zu erstellen.

Die meisten Leute möchten die unten beschriebenen Schritte kombinieren und/oder automatisieren. In Unterschied zu "HOWTOs", versucht diesen Artikel grundlegende Know-How zu vermitteln, und überlassen Scripting dem Leser. Der Leser oder die Leserin mit Vorkenntnisse über Skriptprogrammierung soll nach diesem Text in der Lage sein, seine/ihre eigene Build-Skripts zu verfassen.

In diesem Artikel bezeichnet "GUI" entweder Neutrino oder Enigma. "Filesystem" im Kontext eines kompletten Images bezeichnet das Dateisystem, in dem die Wurzel liegt: Dieses kann cramfs (ein komprimiertes, Read-only filesystem für embedded Systeme) sein, squashfs (ein anderes komprimiertes read-only-Dateisystem, oft als leistungsfähiger als cramfs betrachtet) oder jffs2 (ein "journalled" Read-Write-Filesystem). Ein "cramfs (voll-)Image" besteht aus einem Wurzeldateisystem mit dem cramfs Dateisystem und ein (kleineres) jffs2-Filesystem, das an `/var` gemounted werden soll. Die analoge Aussage hält für "squashfs (voll), das Images", während ein "jffs2-(voll-)Image" nicht ein separates `/var`-Dateisystem enthält, weil das Wurzeldateisystem schon jffs2 ist, und deswegen schreibbar. Zusätzlich enthalten die vollen Images ein zusätzliche Partition, die den u-boot Bootloader enthält. Dieses Teil ist zwischen dBoxen mit einen und zwei Flashchips unterschiedlich. Dieses wird durch "1x" und "2x" angezeigt. Ein komplettes Image trägt den Namen `[neutrino, enigma]-[cramfs, squashfs, jffs2].img[1, 2]x`, z.B. `neutrino-jffs2.img2x`.

3. Buildsystem Voraussetzungen

Die Voraussetzungen auf dem Buildhost können in etwa so zusammengefasst werden: Ein modernes Unix/Linux System mit ca. 2 GB freiem Speicherplatz. Das Tuxbox Projekt hat keine favorisierte Buildumgebung. In Allgemen bekommt Fragen wie "geht es mit Redhat x.y?" keine klare Antwort. Der Grund für dieses ist, dass; niemand sich wirklich interessiert, die Eigenschaften der bestimmten Distributionen zu verfolgen. Anforderung werden anstatt für Versionen der Werkzeuge, wie autoconf, automake, make, usw. formuliert. Die offizielle erfordeliche Toolversione beim Zeit dieses Texts wird in der folgenden Tabelle zusammengefasst:

Tool	Required Version
autoconf	2.57a
automake	1.8
libtool	1.4.2
gettext	0.12.1
make	3.80
makeinfo	irgendwelche

tar	irgendwelche
bunzip2	irgendwelche
gunzip	irgendwelche
patch	irgendwelche
infocmp	irgendwelche
gcc	= 2.95 or >= 3.0
g++	= 2.95 or >= 3.0
flex	irgendwelche
bison	irgendwelche
pkg-config	irgendwelche
wget	irgendwelche

Der Bauprozeß überprüft automatisch einige dieser Anforderungen. Wenn Ihnen eins der Programme fehlt, oder, wenn Ihre Version zu alt ist, ist es in Allgemein viel schneller, die erforderliche Version zu installieren (entweder alles kompilierte Paket, z.B. im rpm-Format von Ihrem Distributor, oder die Quellen zu besorgen, compilieren und installieren), als zu versuchen, herauszufinden ob die oben genannten Anforderungen *wirklich* notwendig sind.

Note:

Andere Beschreibungen erfordern dass Tools wie `fakeroot`, `mksquashfs`, `mkcramfs`, `mkjffs2fs` (oder `mkfs.jffs2`), vielleicht auch `mlibs`, auf Ihrem System installiert sind. In unserer Umgebung ist dies nicht erfordelich.

Auf meinem SuSE System 10.0 war es notwendig, die folgende Pakete nachzuinstallieren: `autoconf`, `automake`, `gcc`, `bison`, `flex`, `gcc-c++`, `newcurses-develop` sowie `zlib-develop`.

Builden auf einem Unix, non-Linux System sollte vermutlich möglich sein, so weit die erforderlichen GNU Werkzeuge vorhanden sind. Mit einem anderen make als GNU wird es fast sicher nicht laufen, da GNU-Erweiterungen ungehemmt verwendet werden.

Ebenso, das Compilieren unter Cygwin "soll" funktionieren, obwohl niemand es während der modernen Zeiten getan hat (soweit ich weiss).

4. Die Quellen auschecken

Die Tuxbox Quellen wird durch den Tuxbox CVS Server distribuiert. Regelmäßige Quellreleases sind niemals gemacht worden, und sind auch nicht für die Zukünft geplant. Für unseren Zwecken werden die Quelle anonym "ausgecheckt" (= kopiert zu die lokalen

Festplatte), indem man zuerst ein leeres Verzeichnis erstellt, z.B. `/tuxbox/head`, auf einer (lokalen) Festplatte mit "ordentlich" freiem Platz, cd-ing zu ihr, und den Befehl

```
cvs -d anoncvs@cvs.tuxbox.org:/cvs/tuxbox -z3 co -f -r newmake -P .
```

eingeben. Merken Sie die Periode am Ende der vorhergehenden Zeile! Dieser Befehl checkt die *newmake* Files aus; in den Fälle, in denen keine *newmake* Version vorhanden ist, wird die HEAD-Version genommen.

Im HEAD gibt es zwei Files `cdk/root/etc/init.d/rcS` und `root/etc/init.d/rcS.insmod`. Im *newmake* sind diese anstatt *Produkte*, die von seiner Quelle `root/etc/init.d/rcS.m4` *erzeugt* werden. Es ist ratsam, `cdk/root/etc/init.d/rcS` und `cdk/root/etc/init.d/rcS.insmod` zu löschen, um auf der sicheren Seite zu sein.

An diesem Punkt kann es wünschenswert sein, einige Patches an den Quellen anzuwenden. Wenn Sie zum ersten Mal kompilieren, ist es ratsam, Patches nicht anzuwenden. Wenn Probleme auftreten, ist es viel einfacher (technisch sowohl als auch sozial) jemand zu helfen, das die "unveränderten CVS Quellen" verwendet.

5. Konfiguration

Demnächst werden einige Zwischenprodukten erzeugt. Ändern Sie zum `cdk` Unterverzeichnis, und geben Sie den Befehl

```
./autogen.sh
```

ein (ohne Argumente). Dieses erzeugt unter anderem einen Shellskript namens `configure`. Dieser wird ausgeführt; dabei wird eine Anzahl von Optionen übergeben, um das System für das Builden eines Image/einer YADD entsprechend den Benutzerwünschen vorzubereiten. Für eine komplette Liste von Optionen, benutzen Sie das Befehl `./configure --help`. Dieser Guide beschreibt nur einen typischen Verwendung, und einige Optionen die der Author für nützlich halten. Der Geist der Konfigurationsoptionen sind wie in den typischen GNU Werkzeugen. Ein typischer Gebrauch, der mit den Pfadnamen oben kompatibel ist, kann sein

```
./configure --prefix=/tuxbox --with-cvsdir=/tuxbox/head
--enable-maintainer-mode
```

`--with-cvsdir` sagt wo die Quellen zu finden sind, (sollte ein Unterverzeichnis cdk haben), während `--prefix` bedeutet, dass eine Anzahl von wichtigen Verzeichnissen als Unterverzeichnisse des besagten Verzeichnisses erstellt werden sollen. Ihre Position kann durch andere Konfigurationsoptionen weiter beeinflußt werden; `./configure --help` produziert die volle Liste der Optionen. `--enable-maintainer-mode` ist, auch für Nichtmaintainers praktisch, da er den hergestellten Makefiles ermöglicht, sich automatisch neu zu erzeugen, sobald die Notwendigkeit entsteht, zum Beispiel nach einem Software-Update.

Es gibt andere nützliche Optionen; einige werden unten besprochen.

Überprüfen Sie bitte den Ausgaben von autogen für Fehler ("Error") und Warnungen. Die

Warnung

```
/usr/local/share/aclocal/pkg.m4:5: warning: underquoted definition of
PKG_CHECK_MODULES
```

from `autogen.sh` kann ignoriert werden, sowohl als folgende Warnungen von `configure`:

```
configure: WARNING: using tuxbox mklibs
checking for mkcramfs... no
configure: WARNING: using tuxbox cramfs
checking for mkjffs2... no
checking for mkfs.jffs2... no
configure: WARNING: using tuxbox mkfs.jffs2
checking for mksquashfs... no
configure: WARNING: using tuxbox squashfs
```

Note:

Der Leser, der dieses Dokument mit ähnlichen Beschreibungen "von den dunklen Zeiten" vergleicht, haben gemerkt, das die Option `--with-targetruleset=[standard,flash]` nicht mehr vorhanden ist. Während "des dunklen Zeitalters" war es notwendig, beim Konfigurationszeitpunkt sich auf Builds von *entweder* YADDs *oder* Images sich einzuschränken. Im *newmake* ist dieses nicht mehr notwendig.

Warning:

Versuchen Sie nicht, als root zu builden!

6. Kompilieren

Die high-level make Targets, die für das Builden von (voll-)Images relevant sind, sind: `flash-[neutrino, enigma, all]-[cramfs, squashfs, jffs2, all]-[1x, 2x, alle ]`. Für YADD-Builds, sind diese: `yadd-[neutrino, enigma, all]`. Z.B. der Befehl

```
make flash-neutrino-jffs2-all yadd-enigma
```

erzeugt flashbare jffs2-only Images mit Neutrino, für 1x-boxes und für 2x-boxes (Dateinamen `neutrino-jffs2.img1x` und `neutrino-jffs2.img2x`). Auch ein YADD, das Enigma enthält, wird erzeugt.

Auf meinem Athlon XP 1800 dauert ein Befehl wie `make yadd-neutrino` in einem leeren Verzeichnisses etwa eins und in einer halben Stunde.

7. Wohin gehen wir von hier?

7.1. Booting der YADD

Wenn ein YADD gerade erzeugt worden ist, fahren Sie zu diesem Artikel, über die Einrichtunng eines YADD-Servers, fort. Merken Sie auch das make-Target `serversupport`, das einige Konfigurationsfiles für den Server erzeugt, und den YADD-Build nahtlos an den Server-Setup anknüpft.

7.2. Flashen des Images

Wenn ein Image gebaut worden ist, ist nächste Schritt das Einspielen des Images in den nichtflüchtige Speicher der dBox, "Flashen" genannt. Für dies empfehle ich entweder, das interaktive Flashen von Neutrino (dBox -> Service -> Software-Aktualisierung -> Expertenfunktionen -> ganzes Flashimage einspielen) zu benutzen, oder der dboxflasher zu verwenden, das hier beschrieben wird. Der dboxflasher wird durch das Make-Target `serversupport` erzeugt. Andere Möglichkeiten des Flashens werden in Tuxbox Wiki beschrieben.

8. Inkrementelle Builds

Im allgemeinen sind Leute nicht an einem einmaligen Build der Software interessiert. Verbesserungen zu den Quellen werden in CVS täglichen eingecheckt. Viele Leute möchten die Software verbessern, indem sie Patches anwenden (z.B. von meiner Patchseite :-), oder durch eigene Programmierung. Es ist dabei wünschenswert, dass genau die Teile neu erzeugt wird, die neu erzeugt werden muss, nicht mehr und nicht weniger. Das vorliegende "*newmake*" geht ein langer Weg in dieser Richtung. Um ein Target `target` neu zu bauen, benutzen Sie das Befehl `make target`, und make wird es, falls notwendig, neu erzeugen. Es kann dann passieren, dass make zusätzlich ein vollständig anderer Bestandteil neu erzeugt! Dieses ist, am mindestens in der Regel, die richtige Sache, da das Target von anderen Teilen abhängen kann, die sich geändert haben, und machen deswegen einen erneuerten Build von diesem Bestandteil notwendig.

In einige Situationen kann es wünschenswert sein, ein erneutes Build eines Komponents zu erzwingen. Einige Komponente werden in einem Distributionsfile zum Verzeichnis `cdk/Archive` downloadet, und wenn das Build stattfindet, ausgepackt, Patches werden angewendet (nur in einigen Fällen) , konfiguriert, kompiliert, installiert, und die Quellen dann wieder gelöscht. Alles findet automatisch statt. Die Installation des bestimmten Pakets wird durch das Anlegen einer Markerdatei im Verzeichnis `cdk/.deps` notiert. Verwendet auf Auspacken-kompilieren-installieren-löschen-paketen, ist diese Technik nicht so schlecht wie wenn in anderen Kontexten (miss-)braucht (wie in den MAIN-Branch in CVS). Falls gewünscht, kann solch eine Markiererdatei entfernt werden um das Neuerzeugen des verbundenen Komponents zu erzwingen.

9. Cleaning targets

Es gibt mehrere unterschiedliche Aufräum-Targets:

`distclean`
Das drastischste Reinigungs-Target, (fast) alles löschend, das nicht von CVS ausgecheckt wurde. Dieses ist selten notwendig.

`mostlyclean`
Ein intelligenteres Target ist `mostlyclean`, säubert in die Verzeichnisse, die Tuxboxquellen enthalten; lässt aber die Kompilationsumgebung und in alle

Auspacken-kompilieren-installieren-löschen-Komponente unberührt. Auch das `cdkroot` Verzeichnis, (d.h. die Yadd-Installation), sowie die TFTP-Files (Kernel und u-boot) werden nicht angefasst.

`depsclean`
Löscht alle Markerdateien im `.deps` Verzeichnis und zwingt so Neucompilation aller Auspacken-kompilieren-installieren-löschen-Komponente. Dieses ist selten sinnvoll: Sie hängen von ihren Quellen und vielleicht von einer Patchfile ab, und der Makefile kennt diese Abhängigkeiten.

`clean`
Kombiniert `mostlyclean`, `depsclean`, und `flash-clean`. Versucht auch soviel wie möglich im `cdkroot`-Verzeichnis zu löschen, das nicht während des Bootstrapdurchlaufes installiert waren. So wird er *versucht*, die Umgebung in einem Zustand zu bringen, wo die Buildumgebung gerade kompiliert worden ist, z.B. mit `make bootstrap`.

`flash-semiclean`
Dieses Target löscht die meisten Verzeichnisse in `$(flashprefix)`, mit Ausnahme von den Boot-Partitionen und den Kernelbauverzeichnisse. Dieses ist oft sinnvoll, da diese Bestandteile verhältnismässig selten sich ändern.

`flash-mostlyclean`
Zusätzlich zum`flash-semiclean` löscht dieses Target auch Bootfiles und die Kernbauverzeichnisse. Vollimages werden unberührt gelassen.

`flash-clean`
Dieses Target löscht Alles in $(flashprefix).

Einige Quellverzeichnisse können mit einem Befehl wie `make -C /tuxbox/head/apps/tuxbox/neutrino clean` gesäubert werden.

10. Aktualisierung des CVS

Um Ihre Quellen mit dem spätesten checkins zu aktualisieren, verwende Sie einen Befehl wie

```
cvs up -f -r newmake -dP > cvs.log 2>&1
```

vom toplevel CVS Verzeichnis (oder von einem anderen Verzeichnis, wenn Sie wissen, was Sie tun). Mögliche Fehler werden in das logfile `cvs.log` aufgeführt.

11. Customization

Die erzeugte Images und die yadds können angepasst ("customized") werden, ohne die Makefiles zu ändern. Erstmals, gibt es einige Konfigurationsoptionen: mit `--with-ucodesdir=DIR` kann ein Verzeichniss angegeben werden, das ucodes inthält, die im Images enthalten sein soll. (Ein Image das ucodes enthält kann nicht legal verteilt werden.) Zweiten, mit der Option `--with-logosdir=DIR` kann ein Verzeichniss angegeben werden, das boot logos (`logo-lcd` und `logo-fb`) enthält, die

im Image enthalten sein soll.

Durchdachtere Customization ist möglich. Für dieses ist es notwendig, etwas Wissen über die innere Funktion des Makefiles zu haben. In der Folge bezeichnet `$(flashprefix)` den Wert des Makefile Variablen `flashprefix` (mit Konfiguration wie oben `/tuxbox/cdkflash`), `$(targetprefix)` bezeichnet den Wert des Makefile Variablen `targetprefix` (mit Konfiguration wie oben `/tuxbox/cdkroot`), und `$(buildprefix)` bezeichnet den Wert des Makefile Variablen `buildprefix` (mit der Konfiguration oben `/tuxbox/head/cdk`).

Um z.B. `neutrino-cramfs.img2x` zu erzeugen, werden die folgenden Verzeichnisse erstellt: `$(flashprefix)/root` (enthält Filesystem- und GUI-unabhängige Bestandteile), `$(flashprefix)/root-cramfs` (enthält den Kernel, für Wurzel-Filesystem auf cramfs konfiguriert, zusammen mit seinen Treibern) und `$(flashprefix)/root-neutrino` (enthält die Neutrinoinstallation). Aus diesen drei Verzeichnissen, werden das Rootfilesystemverzeichniss `$(flashprefix)/root-neutrino-cramfs` und das `var`-filesystemverzeichnis `$(flashprefix)/var-neutrino` gebaut.

Selbstverständlich ist es möglich, einen Befehl wie `make $(flashprefix)/root-neutrino-jffs2` einzugeben (wobei der Benutzer `$(flashprefix)` selbst ersetzen muss; es ist eine make-Variabel, aber nicht eine Shell-Variabel), dann manuell gewünschten Änderungen an `$(flashprefix)/root-neutrino-jffs2` durchzuführen, und dann, mit dem Befehl `make flash-neutrino-jffs2-2x` den Imagebau abschließen, um ein Image zu erstellen, das die manuellen Änderungen enthält. Dieses kann für den einmaligen Imagesbau sinnvoll sein. Jedoch in vielen Fällen wird eine automatischere und systematischere Methode gewünscht, zunächst beschrieben.

Viele der wichtigeren Targets rufen ein Customization-Script auf, falls vorhanden und ausführbar. Der Name des Customization Scriptes wird als das Nicht-Verzeichnis Teil der Regel genommen, mit `-local.sh` angefügt. Der Script soll im *customizationsdir* liegen. Dies ist mit der configure-Option `--with-customizationsdir` auswählbar. Default ist das cdk-Verzeichnis. Das Script wird zwei Argumente übergeben: Für Imagetargets sind diese `$(flashprefix)` und `$(buildprefix)`; für Yaddtargets sind diese `$(targetprefix)` und `$(buildprefix)`.

Die Bezeuchnung "Script" ist ein bisschen irreführend, da sie als normale Programme mit zwei Argumenten ausgeführt werden. Anstelle von einem Shell-Script können dies z.B. ein kompilierte C Programme, oder ein Perl-Script, sein.

Jedoch werden die Customization Filen für die Targets `version` und `flash-version` (die `/.version`-Files in YADD bzw. im Image erstellt) nicht am Ende der normale make-Actions ausgeführt, es *ersetzt* sie.

Der Custiomizationscripting wird durch das folgende Beispiel veranschaulicht.

11.1. Beispiel

In einem Image wird es gewünscht:

1. Eigene /etc/hosts benutzen,
2. Eigene neutrino.conf, bouquets.xml, services.xml benutzen
3. Inkludiere die lirc-Komponente, zusammen mit eigenen lirc Konfiguration Files.

1. und 3. sind Erweiterungen, die zu `$(flashprefix)/root` erfolgt werden sollten, während 2., seiend Neutrino-regeln, sollten zu `$(flashprefix)/root-neutrino-jffs2`, zu `$(flashprefix)/root-neutrino-cramfs` oder zu `$(flashprefix)/root-neutrino-squashfs` getan werden. Um 1. und 3. zu erzielen. schreiben wir das Script `root-local.sh`, z.B.:

```
#!/bin/sh

flashprefix=$1
buildprefix=$2
newroot=$flashprefix/root
myfiles=/home/somewhere/dbox/myfiles

cp -f  $myfiles/etc/hosts                  $newroot/etc
make flashlirc
cp -fr $myfiles/var/tuxbox/config/lirc
$newroot/var/tuxbox/config
```

Das Script für 2. nennen wir es `root-neutrino-local.sh`, ist völlig ähnlich:

```
#!/bin/sh

flashprefix=$1
buildprefix=$2
newroot=$flashprefix/root-neutrino
myfiles=/home/somewhere/dbox/myfiles

cp $myfiles/var/tuxbox/config/neutrino.conf  $newroot/var/tuxbox/config
cp $myfiles/var/tuxbox/config/zapit/bouquets.xml
$newroot/var/tuxbox/config/zapit
cp $myfiles/var/tuxbox/config/zapit/services.xml
$newroot/var/tuxbox/config/zapit
```

Note:

Diese Scripte sollen als Beispiele dienen und können vermutlich nicht ohne Anpassung verwendet werden.

11.2. Ändern der Partitionierung

Ab 2006-03-19, kann die Rootpartitionsgröße für cramfs und squashfs Images mit der Configure-Option `--with-rootpartitionsize=SIZE` angegeben werden. Die Größe des `var`-Partitions wird automatisch berechnet, um den restlichen Flashspeicher zu benutzen, der nicht durch die anderen Partitionen benutzt wird. Defaultgröße ist 0x660000. Diese Zahl sollte eine Multiple der Erasesize, momentan 0x20000 sein. Wird ignoriert (wenn sinnlos) beim jffs2imageerstellung.

12. Einige "best practices"

In diesem Abschnitt sammeln wir einige Richtlinien, die nicht "notwendig" sind um korrekte Ergebnisse zu erhalten, jedoch werden sie langfristig helfen, um bessere, zuverlässigere und pflegbare Software zu erstellen. Sie wenden sich sowohl an den Customizationen an, sowie zukünftige änderungen am Makefile (und zu seinen Bestandteilen) selbst.

Wenn Sie nicht diese Richtlinien mögen, ignoriern Sie sie, am mindestens wenn Sie Customization Scripte für Ihren eigenen Brauch schreiben.

12.1. Idempotens

Es ist fast immer eine gute Idee zu versuchen, ein Installationsscript *idempotent* zu schreiben. Dies bedeutet, dass das mehrmalige Ausführen den gleichen Effekt hat wie das einmalige Ausführen.

12.2. Benutze "make install", mopse nicht einzelne Files!

In der Vergangenheit hat das Tuxbox Makefile die Komponente zuerst in `$(targetprefix)` installiert, und dann die Imageverzeichnissen durch Kopieren der einzelnen Files von der `$(targetprefix)` Hierarchie erstellt. Dieses ist nicht sehr gute Softwaretechnik. Zuerst gehört das Know-how bzgl. Installation des Paket dem Makefile des Pakets, und soll nicht einem allgemeinen Makefile sitzen, das einfach einzelne Files rüberkopiert. Wenn dieses Paket sich ändert, z.B. dadurch eine Konfiguration File zugefügt oder gelöscht wird, wird es auch notwendig, in das globale Makefile zu ändern.

Es ist häufig der Fall, dass das Makefile, das dem Paket gehört, include-Files, (statische) Bibliotheken, Info-Files etc. installiert, die nicht auf einem enbedded System mit beschränktem Speicher gewünscht sind. Die korrekte Lösung zu diesem (wirklichen!) Problem würde wurde sein, das Makefile des Pakets zu ändern, entweder, um ein `flashinstall`-Target zu schreiben, oder das Makefile mit einem Parameter wie `installsize=[full,flash]` zu versehen. Wenn dieses nicht durchführbar ist, ist es meine Meinung, daß `make -C ... install` gefolgt vom Löschen der unerwünschten Files besser ist als das kopierend einzelne Files. Zu erwähnen ist auch, daß in dem Schritt, der die Verzeichnisse `$(flashprefix)/root-gui-`*filesystem* erzeugt, das include-verzeichnis, sowie alle statischen Bibliotheken gelöscht werden und dynamische Bibliotheken von unbenutzten Symbolen gestrippt werden.

13. Antwort auf einige Fragen

13.1. Falls das Build nicht gelingt

Es gibt kein Standardverfahren auf was zu tun, wenn das Build schiefläft. Ich versuche hier einige Richtlinien zu geben, zu Lesen bevor Posten zum Forum.

Zuerst, überprüfen Sie den Output der ersten zwei Schritte, `autogen.sh` und `configure` für Fehler und Warnungen. Jede Warnung oder Fehler, außer den fünf Warnings, die oben verzeichnet werden, zeigt ein Problem an, das wahrscheinliche Build unmöglich macht.

Wenn ein Build bricht, kann es die Umgebung in einem inkonsequenten Zustand hinterlassen. Dies gilt insbesondere für die Verzeichnisse in `$(flashprefix)`. Wenn der Bau solch eines Make-Targets bricht, besteht das Verzeichnis, ist entsprechend ihrer Änderungszeit aktuell, und ein folgendes make Befehl behandelt ihn wie fertig und okay. Selbstverständlich wird ein fehlerhaftives Build das Ergebniss. Wenn ein Build eines Unterverzeichnisses von `$(flashprefix)` in den Brüchen geht, bitte lösche es, bevor Sie einen anderen Make Befehl versuchen.

Bei "es funktionierte gestern"-Probleme, ist vermutlich die Umgebung in einem inkonsequenten Zustand. Ein mehr-oder-wenige drastische Reinigungsbefehl (sehen Sie oben) ist hierbei oft schneller als eine systematische Problemsuche.

Wenn Sie Hilfe benötigen, sehen Sie unten.

13.2. Nach dem Flashen bekomme ich "Kein System" auf dem LCD/Was ist diese "bad magic byte" Zeugs?

Uhh, ich hoffte, daß die Frage würde nicht kommen wurde... Die kurze Antwort ist: Ich weiß es nicht. Wir wissen es nicht. Aber, wenn Sie diesen Artikel so weit lesen, erwarten Sie nicht "kurze Antworten", sondern "gute Antworten". O.K. Das Thema ist ausführlich hier besprochen worden. Kurz gesagt, das Image "ist" in Ordnung, es ist nur dass irgendwelche Firmware in der dBox es zurückweist, weil es einige "schlechte magische Bytes" auf bestimmten Adressen findet. Der Forumteilnehmer mogway hat ein Programm geschreiben, in CVS im Verzeichnis `hostapps/checkImage` zu finden. Das Programm ermittelt die "schlechten Bytes", aber es tut nichts, um sie zu beheben. Meine eigene Erfahrung sagt, daß Images, die `checkImages` für gut findet, wirklich laufen. cramfs-, oder squashfs Images, worüber sich `checkImage` beschweren, läufen im allgemeinen nicht, in einigen Fällen läufen sie aber doch. jffs2-images, worüber sich `checkImage` beschwert, läufen oft,aber nicht immer. Mit diesen empirischen Beobachtungen überlasse ich dem möglichen Benutzung von `checkImage` und den folgenden Entscheidungen, dem Benutzer, ohne weitere Empfehlungen.

newmake weisst, wie dieses Programm angeruft werden kann, um die erzeugten Images automatisch zu überprüfen. Die Konfigurationsoption `--with-checkImage=[none,rename,warn]` ist dazu zu verwenden. Falls `warn` gewählt ist, wird für jedes Image, das den Test nicht bestehen, eine leere Datei erzeugt, Name gleich Imagefile mit `_bad` angehängt. Wenn `rename` gewählt wird, wird anstatt die fragliche Imagefile umbenannt.

Es kann erwähnt werden, daß die "schlechten magischen Bytes" in einem (oder mehrere!) von den Partitionsimages sitzt, und werden nicht durch den abschließenden Schritt erzeugt (die `*.img1x` und/oder `*.img2x` Files erschaffen). Es ist möglich, `checkImage` auf den Partitionsfiles aufzurufen (`*.jffs2`, `*.cramfs`, `*.squashfs`, `*.flfs1x`, `*.flfs2x`). Schließlich hat checkImage eine Debugoption, die nützlich sein kann.

13.3. Ich habe ein Fehler gefunden!

Bugs, Unklarheiten, Verbesserungsvorschläge, etc. der Software sollten vorzugsweise zum Cross Development Kit Abteilung des Tuxbox forum gehen. Was diesen Text direkt betrifft -- Fehler (die technische Angelegenheiten, Rechtschreibung, Grammatik, pädagogisch), Verbesserungsvorschläge und Verlängerungen -- können zu mir direkt gehen. "Diskussion" ist vermutlich besser im Forum abgehoben.

13.4. Ich benötige Hilfe!

Supportanfragen können imCross Development Kit Abteilung des Tuxbox Forum geposted werden. Postings in deutsch oder englisch sind willkommen. Bitte vergessen Sie nicht, die benutzte Konfigurationsoptionen zu erwähnen.

Bitte mailen/PM-en Sie nicht mich persönlich, da ich nicht persönliche Gratissupport anbiete.

13.5. Parallel make?

Siehe die englische Version.

13.6. Kernel 2.6?

Siehe die englische Version.

13.7. Update images

Siehe die englische Version.

13.8. How do I convert 1x-images to 2x, or vice versa?

Siehe die englische Version.

14. Appendix. Einige nützlige customization script Fragmente

In diesem Anhang werden einige nützliche Customization Scripte gezeigt. Zwei Scripte sind bereits oben gezeigt worden.

Warning:

Auch falls in einige Fällen benutzbar wie sie sind, werden die Scripte als Beispiele, nicht Lösungen zu den realen Problemen gezeigt. Aus diesem Grund sind die Beispiele hier als Codefragmente, nicht als downloadbare Datein, veröffentlicht. Bitte verwenden Sie sie nicht, es sei denn Sie am mindestens ungefährlich verstehen wie sie funktionieren. Es wird am mindestens grundlegende Script-Erfahrung erfordelig.

14.1. Games und Languages nuker

Diese File löscht alle Spiele (definiert als plugins mit type=1 in ihrer Konfiguration File), sowie unerwünschte Sprachfiles (Neutrino angenommen). Das File sollte vermutlich von `root-neutrino-$filesystem-local.sh` aufgerufen werden.

```
#!/bin/sh

# Nukes all game plugins, as well as all locale files not listed in
LANGUAGES

newroot=$1/root-neutrino-jffs2
LANGUAGES="deutsch english"

for f in $newroot/lib/tuxbox/plugins/*.cfg; do
    grep 'type=1' $f>/dev/null && rm -f
$newroot/lib/tuxbox/plugins/`basename $f .cfg`.*
done

for f in $newroot/share/tuxbox/neutrino/locale/*; do
    (echo $LANGUAGES | grep -v `basename $f .locale` >/dev/null) && rm
-f $f
done
```

14.2. Customizing die /.version file

Ihre eigene `/.version`-File herzustellen (gezeigt von Neutrino durch dBox -> Services -> Image-Version) ist sicher ein allgemeinee Wunsch. Hier ist die File, die ich momentan für dieses benutze:

```
#/bin/sh

if [ $0 = ./flash-version-local.sh ] ; then
    outfile=$1/root/.version
    type="image"
else
    outfile=$1/.version
    type="yadd"
fi

echo Creating $outfile ...

echo "version=`./mkversion -snapshot -version 200`"      > $outfile
echo "creator=Barf"                                     >> $outfile
echo "imagename=Barf-$type"                             >> $outfile
echo "homepage=http://www.bengt-martensson.de"          >> $outfile
```

Dieses File kann sowohl mit dem Namen `flash-version-local.sh`, sowie mit dem Name `version-local.sh`, für das Erstllen des `/.version-` File für Images

beziehungsweise yadds benutzt werden. Merken Sie die Auswertung von `$0` (die den tatsächlichen Namen enthält, unter dem der Script aufgerufen worden ist). Das benannte Script `mkversion` stellt die etwas kryptische Versionszeichenkette her und ist einfach eine "Verkapselung" davon. Es wird hier gezeigt:

```
#!/bin/sh

releasetype=3
versionnumber=000
year=`date +%Y`
month=`date +%m`
day=`date +%d`
hour=`date +%H`
minute=`date +%M`

while expr $# > 0 ; do
        case "$1" in
        -release)
                releasetype=0
        ;;
        -snapshot)
                releasetype=1
        ;;
        -internal)
                releasetype=2
        ;;
        -version)
                versionnumber=$2
                shift
        ;;
        esac
        shift
done

echo $releasetype$versionnumber$year$month$day$hour$minute
```

14.3. Archivierung der Images

Es ist die Aufgabe des Buildprozesses, die flashbare Images zu erstellen, und nicht sie zu archivieren. Jedoch kann die Customization leicht missbraucht werden, um irgendeine Art vom Archivieren zu erschaffen, wie das folgende Beispiel zeigt:

```
#!/bin/sh

flashprefix=$1
imagefile=`basename $0|sed -e s/-local.sh//`
imagefilebase=`echo $imagefile|sed -e s/\.img.x//`
extension=`echo $imagefile|sed -e s/[-a-z0-9]*\.//`
newfilename="barf-"$imagefilebase-`date --iso-8601`.$extension

echo Copying $flashprefix/$imagefile to $flashprefix/$newfilename...
cp $flashprefix/$imagefile $flashprefix/$newfilename
```

Das Script sollte einen oder mehr der Namen `[neutrino, enigma]-[cramfs, squashfs,jffs2].[img1x, img2x]` haben. Es benennt die Files entsprechend dem Tagesdatum um. Wieder wird das Script gezeigt, um ein Konzept zu zeigen, nicht

gerade kopiert zu werden.

15. Referenzen

Siehe die englische Version.