Building Flash Images and YADDs with newmake

Table of contents

1. Revision history

Date	Description
2006-02-14	Initial version.
2006-03-02	Updated script fragments to take into account that customization scripts are now called as `./$@-local.sh` (previously `sh $@-local.sh`), making `$0` different.
2006-03-19	Minor changes. Added section on root partition size.
2006-04-15	Mention custiomizationsdir. Added paragraph on "script" as "misnormer". Added comment on bad magic bytes. Added link to the architecture document.
2006-04-17	Rewrote Cleaning targets (to reflect changes). Updated URL to the GNU Make manual.
2006-07-19	Minor spellfixes, reference to the German Version.

2. Introduction

Eine deutsche Version dieses Dokuments ist hier verfügbar.

This document covers newmake from the user's perspective, and covers image and yadd builds, and elementary customization. The architecture of newmake is described in another document.

2.1. Some history

A few years ago, image creation for the Tuxbox software was a black art. The Makefile support was quite incomplete, in particular for other images than cramfs-images. Not only were the CVS tools bad or incomplete, worse, some parts were deliberately kept secret, namely the tool, now known as `mkflfs`, available in the CVS-directory `.../hostapps/mkflfs`. According to a posting from this time, most developers were not able to build an image. The "Guild of the Image makers" was born. Most well-known from this time are the "AlexW-Images": mainly consisting of CVS-sources, but with some, more-or-less secretly held "fixes", (probably) necessary for building a functioning image out of the CVS-sources.

In August 2003, in a project that called itself "GNU DBox2 Software Project", it became increasingly embarrassing to keep `mkflfs` secret, and the sources for `mkflfs` were checked in to CVS. Also, the Makefile gradually improved in functionality. Still, much was left to be desired: functionality, maintainability, sound software design. Building an image from pure CVS was not really possible.

In 2004 the YADI ("Yet Another DBox Image") project was born. (Do not confuse "YADI" and "YADD"!) Its goal was to automate and to simplify image creation. For this, a number of scripts and patches were incorporated and/or written. Additionally, flash-ready images were provided.

YADI was a big success. The goal was achieved. Images were made available, based (almost?) completely with free software -- both in its content, and the tools involved, in a way that was open to the user. With the YADI-script, automatic image creation was possible. However, instead of addressing the fundamental weakness in the CDK build process, they provided scripts to build images. They did not provide a build mechanism for a software project. Software project are built with a software build system, like make, or a later successor, such as Ant or Maven, not with shell scripts.

newmake, presently as alternative branch in CVS, tries to address these weaknesses.

I would like to thank everyone who have provided bug reports and feedback, in particular dietmarw, who is using *newmake* to build the dietmarw-images.

2.2. Goal

The goal of the present article is to provide the reader with basic know-how. It is not the goal to provide an idiot-proof step-by-step instruction (like the so-called HOWTOs). Prior exposure to shell scripts is required for many parts, in particular the customization chapter and the appendix, however not for image/yadd creation in its simplest form.

The present document (at least in the present version) does not try to describe the inner working of the make file and the make process. For this, the reader is referred to the sources (which *are* somewhat commented!), and to the relevant threads in the Cross Development Kit section of the Tuxbox forum. Also, we do not describe all options to `configure`, only the most common and important ones.

2.3. How hard is it?

My answer would be: It is as hard as reading this article. The reader understanding most herein should not have any problems; for the reader for which most of this is gibberish, hmm, it may be wiser to stay with ready-made images.

2.4. General

There are two possible goals when compiling the source: Either "YADD" or image. "A YADD" consists of a few files that the dBox loads using the TFTP-Service, and a filesystem, made accessible to the dBox from a NFS-Server (see this article). This mode of operation has many advantages when developing the Tuxbox software, or when learning the system. The name "YADD" once meant "Yet Another DBox Distribution".

Unfortunately, this misleading and throughout silly name has stuck.

My suggestion to the apprentice image/yadd-builder is: First build a YADD with your favorite GUI, and get it to work. Next step would be to build a jffs2-image, again with your favorite GUI.

Most people would like to combine and/or automate the steps described below. As opposed to "HOWTOs", this guide is aimed at understanding the involved issues, and leaves scripting to the reader. The reader with reasonable prior exposure to shell scripts should have no problem writing his/her own build script after reading this guide.

In this article, "GUI" will denote either Neutrino or Enigma. "Filesystem" in the context of a complete image will denote the file system where the root resides: This may be *cramfs* (a compressed, read-only filesystem for embedded devices), *squashfs* (another compressed, read-only file system, often considered to be more efficient than cramfs), or *jffs2* (a journalled read-write filesystem). A "cramfs (full) image" consists of a root file system, using the cramfs file system, and a (smaller) jffs2 filesystem, that is to be mounted on `/var`. The analog statement holds for "squashfs (full) images", while a "jffs2 (full) image" does not have a separate `/var` file system, since the root files system, being jffs2, is writeable. Additionally, the full images contains an additional partition, containing the u-boot boot loader. This part is different between dBoxes with one and two flash chips. This is indicated by "1x" and "2x". A complete image carries the name `[neutrino,enigma]-[cramfs,squashfs,jffs2].img[1,2]x`, e.g. `neutrino-jffs2.img2x`.

3. Build system prerequisites

The prerequisites on the host for building Tuxbox images and yadds can be summarized in: A modern Unix/Linux system with some 2 GB of free disk space. The Tuxbox project does not have a favorite host environment, and in general, questions like "Can Redhat x.y build it?" will not get a definite answer. The reason for this is that no-one really cares to keep track of the features of particular distributions. Requirement are instead formulated for versions of the tool, like autoconf, automake, make, etcetera. The official tool version requirement at the time of this writing is summarized in the following table:

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

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

The build process will automatically check some of these requirements. If you miss one of the programs, of if your version is older than the above requirements, in general it is much quicker to installed the required version (either all compiled package, e.g. in rpm-format from your distributer, or getting it in source format, compiling and installing yourself), than to try to find out if the above requirements *really* are necessary.

Note:
Other descriptions require tools like `fakeroot`, `mksquashfs`, `mkcramfs`, `mkjffs2fs` (or `mkfs.jffs2`), and, possibly, `mklibs` to be installed on your system. In our setup, this is not required.

On my SuSE 10.0 system it was necessary to install these extra packages: `autoconf`, `automake`, `gcc`, `bison`, `flex`, `gcc-c++`, `newcurses-develop`, and `zlib-develop`.

Building on a Unix, non-Linux system should probably be possible, as long as the required GNU Tools are installed. Using a non-GNU make will almost surely not work, since GNU-extensions are used freely.

Likewise, compiling with Cygwin *"should"* work, although no-one has done it during modern times (as far as I am aware of).

4. Checking out the sources

The Tuxbox sources is distributed through the Tuxbox CVS server. Regular source releases are presently neither made, nor planned. For our purposes, the source are "checked out" (= copied to your local disk) anonymously by first creating an empty directory, say `/tuxbox/head`, at a (local) disk with "lots" of free space, cd-ing to it, and issuing the command

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

Note the period at the end of the previous line! This command checks out the *newmake* files, and for the cases where no *newmake* version is available, the HEAD version.

In HEAD, there are two files `cdk/root/etc/init.d/rcS` and `root/etc/init.d/rcS.insmod`. In *newmake*, these are instead *products*, which are *generated* from its source `root/etc/init.d/rcS.m4`. It is therefore advisable to delete `cdk/root/etc/init.d/rcS` and `cdk/root/etc/init.d/rcS.insmod`, just to be on the safe side.

At this point, it may be desirable to apply some source patches to the sources. If you are compiling for the first time, it is advisable not to apply patches. If problems occur, it is much easier (both technically *and* socially) to help someone who is using the "unmodified CVS sources".

5. Configuring

Next some intermediate files are generated. Change to the `cdk` subdirectory, and issue the command

```
./autogen.sh
```

(with no arguments). This creates, among other things, a shell script called `configure`. This script is executed, given a number of options, to set up the system for building an image/a yadd according to the users wishes. For a complete list of options, use the command `./configure --help`. This guide will only describe a typical use, and some other options the author happens to consider useful. The spirit of the configuration options are like in typical GNU tools. A typical use, compatible with the selection above, may be

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

The `--with-cvs-dir` states where the sources are located (should have a subdirectory `cdk`), while the `--prefix` states that a number of important directories are to be created as subdirectories of said directory. Their location can be further influenced by some other configuration options, `./configure --help` produces the full list. `--enable-maintainer-mode` is practical, also for not-maintainers, since it enables the created Makefiles to be automatically rebuild when the need arise (for example after some software updates).

There are other useful options available; some are being discussed below.

Please examine the output of `autogen` for errors and warnings. The warning

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

from `autogen.sh` can be ignored, as well as these warnings from `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:

The reader comparing this document to similar descriptions from "the dark ages" have noted, that the option `--with-targetruleset=[standard,flash]` is no longer used. During "the dark ages" it was necessary to, during configuration time, restrict yourself to building either yadds, or images. In *newmake* this is no longer necessary.

Warning:

Do not try to build as root!

6. Compiling

The high-level make targets relevant for building (full) images are: `flash-[neutrino,enigma,all]-[cramfs,squashfs,jffs2,all]-[1x,2x,all]`. For YADD-builds, these are: `yadd-[neutrino,enigma,all]`. For example, the command

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

will build flashable jffs2-only images with Neutrino, both for 1x-boxes and for 2x-boxes (filenames `neutrino-jffs2.img1x` and `neutrino-jffs2.img2x`). Also, a YADD containing Enigma will be built.

On my Athlon XP 1800 a command like `make yadd-neutrino` in a clean directory takes around one and a half hour.

7. Where do we go from here?

7.1. Booting the YADD

If a YADD just have been built, proceed to this article for setting up a YADD server. Note the make-target `serversupport` that generates some setup files for the server, interfacing the build with the server setup seamlessly.

7.2. Flashing the image

If an image has been build, next step would be to read it into the flash memory of the dBox, called "flashing". For this, I recommend either using the interactive flashing of Neutrino (dBox -> Services -> Software update -> Expert functions -> Write whole image), or the dboxflasher described here. The dboxflasher is built by the make-target `serversupport`. Other possibilities for flashing are described in Tuxbox Wiki.

8. Incremental builds

In general, people are not interested in just building the software once. Improvements to the sources are checked into CVS on a daily basis. Also, many people would like to improve the software, either by applying other peoples patches (e.g. from my patch page :-), or by programming themselves. It is then desirable for make to rebuild what is needed, no more and no less. The present "*newmake*" goes a long way in that direction. To rebuild a make-target `target`, just issue the command `make target`, and make will remake that target. It can then happen, that make starts (re-)building a completely different component! This is, at least most of the time, the right thing to do, since the target may depend on other parts, which have changed, making a renewed build of that component necessary.

In some situations, it may be desirable to *force* a rebuild of a component. Several components are downloaded in a distribution file to the directory `cdk/Archive`, and when the build takes place, unpacked, patches are applied (only in some cases), configured, compiled, installed, and the sources then deleted again. Everything takes place automatically. The installation of the particular package is recorded by a marker file in directory `cdk/.deps`. Used on unpack-compile-install-delete-packages, this technique is not as bad as when (mis-)used in other contexts (like the HEAD branch in CVS still does). If desired, such a marker file can be removed, forcing rebuild of the associated component.

9. Cleaning targets

There is a large number of different cleaning targets:

`distclean`
The most drastic cleaning target, deleting (almost) everything that was not checked out from CVS. This is seldomly necessary.

`mostlyclean`
A smarter target is `mostlyclean`, that cleans in the directories containing "tuxbox-sources", but leaves the compilation environment, and all unpack-compile-install-delete-components alone. Also, the cdkroot directory, (i.e. the yadd-installation), as well as the TFTP-files (kernel and u-boot) are not touched.

`depsclean`
Deletes all marker files in the `.deps` directory, thus forcing recompilation of all unpack-compile-install-delete-components. This is seldomly sensible: They depend on their sources, and, possibly, a patch file, and the Makefile knows these dependencies.

`clean`
Combines `mostlyclean`, `depsclean`, and `flash-clean`. Also tries to delete as much as possible in the cdkroot directory, that was not installed

during the bootstrap run. Thus, it is *attempted* to bring the environment to the stage when the build environment has just been compiled, for example by `make bootstrap`.

`flash-semiclean`
This target deletes most build directories in `$(flashprefix)`, but leaves the built boot-partitions and kernel build directories alone. This is often sensible, since these components change comparatively seldomly.

`flash-mostlyclean`
In addition to `flash-semiclean`, this target also deletes boot-partition files and the kernel build directories. Build full images are left untouched.

`flash-clean`
This target deletes all components in `$(flashprefix)`.

Some source directories can be cleaned with a command like `make -C /tuxbox/head/apps/tuxbox/neutrino clean`.

10. Updating the CVS

To update your sources with the latest commits, use a command like

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

from the top CVS directory (or from another directory, if you know what you are doing). Possible errors are put into the log file `cvs.log`.

11. Customization

The built images and yadds can be customized without changing the Makefiles. First of all, there are some `configure`-options: using `--with-ucodesdir=DIR` a directory, containing ucodes to be included in the image, can be given. (Note that an image containing ucodes can not legally be distributed.) Secondly, the option `--with-logosdir=DIR` can give a directory containing boot logos (`logo-lcd` and `logo-fb`) to be included.

More elaborate customization is possible. For this, it is necessary to have some knowledge about the inner working of the makefile. In the sequel, `$(flashprefix)` will denote the value of the makefile variable `flashprefix` (with the configure line above `/tuxbox/cdkflash`), `$(targetprefix)` will denote the value of the makefile variable `targetprefix` (with the configure line above `/tuxbox/cdkroot`), and `$(buildprefix)` will denote the value of the makefile variable `buildprefix` (with the configure line above `/tuxbox/head/cdk`).

In order to build, say, `neutrino-cramfs.img2x`, the following directories are being built: `$(flashprefix)/root` (containing filesystem and gui-independent components), `$(flashprefix)/root-cramfs` (containing the kernel, built for root filesystem on cramfs, together with its drivers), and `$(flashprefix)/root-neutrino` (containing the neutrino-installation). From these three directories, the root filesystem directory `$(flashprefix)/root-neutrino-cramfs` and the `var`-filesystem directory `$(flashprefix)/var-neutrino` are built.

Of course, it is possible to invoke a command like `make $(flashprefix)/root-neutrino-jffs2` (whereby the user have to expand `$(flashprefix)`, it is a make variable, but not a shell variable), then manually do the desired changes to `$(flashprefix)/root-neutrino-jffs2`, and then, with the command `make flash-neutrino-jffs2-2x` have the final image build, containing the manual changes. This can be desirable for the one-time image builder. However, in many cases a more automatic and systematic methodology is desired, described next.

Many of the major targets calls a customization script, if present and executable. The name of the customization script is taken as the non-directory part of the rule, with `-local.sh` appended. The script is supposed to reside in *customizationsdir*, which is selectable with the `./configure`-option `--with-customizationsdir`. It defaults to the cdk directory. The script is given two arguments: For image targets these are `$(flashprefix)` and `$(buildprefix)`; for yadd-targets these are `$(targetprefix)` and `$(buildprefix)`.

Actually, "script" is a bit of a misnormer, since they are just executed as any programs with two arguments. Instead of shell-scripts, these may be, e.g., compiled C programs or Perl-scripts.

However, the customization files for the make-targets `version` and `flash-version` (creating the `/.version` files in YADD and the image respectivelly) are not executed at the end of the normal actions, it *replaces* them.

The custiomization script facility is illustrated by the following example.

11.1. Example

In an image, it is desired to:

1. Use own /etc/hosts,
2. Use own neutrino.conf, bouquets.xml, services.xml
3. Include the lirc component, together with own lirc configuration files.

1. and 3. are extensions that should be done to `$(flashprefix)/root`, while 2., being a Neutrino-fix, should be done to `$(flashprefix)/root-neutrino-jffs2`, `$(flashprefix)/root-neutrino-cramfs`, or `$(flashprefix)/root-neutrino-squashfs`. To achieve 1. and 3. we write the script `root-local.sh`, say:

```
#!/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
```

The script for 2., say, `root-neutrino-local.sh`, is entirely similar:

```
#!/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:
These scripts are intended to serve as examples, and can probably not be used without modification.

11.2. Changing the partitioning

As of 2006-03-19, the root partition size for cramfs and squashfs images can be selected with the configure-option `--with-rootpartitionsize=SIZE`. The size of the `var`-partition is automatically computed to use all remaining flash space, i.e. everything not used by the other partitions. Default size is 0x660000. This number should be a multiple of the erase size, presently 0x20000. Ignored (while meaningless) when building jffs2-images.

12. Some "best practices"

In this section, we collect some rules that are not "necessary" to get the right result, however, they may in the long run lead to better and more reliable and maintainable software. They apply both to customizations and future changes to the Makefile (and its components) itself.

If you do not like these rules, feel free just to ignore them, at least if you are writing customization scripts for your own usage.

12.1. Idempotence

It is almost always a good idea to try to make a setup-script *idempotent*. That means, that executing it several times has the same effect as executing it once.

12.2. Use "make install", do not just snarf individual files!

"Traditionally", the Tuxbox Makefile first installed packages in `$(targetprefix)`, and then created the image directories by copying individual files from the `$(targetprefix)` hierarchy. This is not very good software engineering. First, the know-how on the installation of the package *package* should sit in *its* Makefile, not in a some general Makefile, just snarfing together already copied files. If that package changes in the sense that a configuration file is added or deleted, it is necessary to change also in the global makefile.

It is often the case, that the Makefile belonging to the package installs include files, (static) libraries, info-files etc., that are not wanted on an embedded system with restricted memory. The correct solution to this (real!) problem would be to modify the Makefile of *package*, either to write a `flashinstall` target, or to provide the Makefile with a parameter like `installsize=[full,flash]`. If this is not feasible, it is my opinion that `make -C ... install` followed by deletion of unwanted files still is better than copying individual files. Note that, in the step that makes the directories `$(flashprefix)/root-gui-filesystem`, the include directory, as well as all static libraries are deleted, and shared libraries are stripped of unused symbols.

13. Answers to some questions

13.1. What if it does not build?

There is no standard procedure on what to do when the build fails. I will here try to give some guidelines, to be read before posting to the forum.

First of all, examine the output of the first two steps, autogen.sh and configure for errors and warnings. Every warning or error, except for the five messages listed above indicates a problem that will most likely make build impossible.

If a build breaks, it may leave the build environment in an inconsistent state. This is in particular true for the directories in `$(flashprefix)`. If the build of such a make target breaks, the directory will exist, be up-to-date according to their file modification time, and a subsequent make command will treat it as finished and ok. Of course, an incorrect build will result. Therefore, if a build of a directory in `$(flashprefix)` breaks, please delete it before trying another make command.

By "it worked yesterday"-problems, probably the build environment is in an inconsistent state. Issuing a more-or-less drastic cleaning command (see above) and trying recompiling might be faster than systematic problem search.

If you need help, see below.

13.2. After flashing I get "Kein System" on the LCD/What is this "bad magic

byte" business?

Uhh, I hoped the question would not come up... The short answer is: I do not know. We do not know. But if you are reading this article this far, you do not expect "short answers", but "good answers". Ok. The issue has been discussed at length here. In short the image "is" ok, it is just that some firmware in the dBox rejects it because it finds some "bad magic bytes" on certain addresses. The forum participant mogway wrote a program, in CVS available in the directory `hostapps/checkImage`. The program detects these "bad bytes", but it does nothing to correct them. My own experience says that images `checkImages` says are OK really runs. cramfs or squashfs images which `checkImage` complain about, in general do not run, in some cases they do. jffs2-images that `checkImage` complains about *in general*, but not always, runs. With these empirical observations, I leave the possible usage of `checkImage`, and the subsequent decisions, to the user, with no further recommendations.

newmake knows how to build and how to invoke this program to automatically check the generated images. The configure-option `--with-checkImage=[none,rename,warn]` may be used. If `warn` is selected, then for every image that do not pass the test, a dummy, zero-length file is generated, name as image file with `_bad` appended. If `rename` is selected, the questionable image file is instead renamed.

It can be mentioned that the "bad magic bytes" sit in one (or more!) of the partition files, and are not generated by the final step (building the *.img1x and/or *.img2x files). It is possible to invoke `checkImage` on the image files (`*.jffs2`, `*.cramfs`, `*.squashfs`, `*.flfs1x`, `*.flfs2x`). Finally, `checkImage` has a debug-option that may be useful.

13.3. I have found a mistake or a bug!

Bugs, gripes, suggestions for improvement of the software should preferably go to the Cross Development Kit section of the Tuxbox forum. Issues regarding this text -- mistakes (technical matters, spelling, grammar, pedagogical), suggestions for improvements and extensions -- can go to me directly, however, "discussion" is probably better off in the forum.

13.4. I need help!

Requests for help can be posted in the Cross Development Kit section of the Tuxbox forum. Postings in German or English are welcome. Please include the configuration options used in the posting.

Please do not mail or PM me personally, since I do not provide personal free-of-charge support. (*After* having posted the problem, a PM/mail politely pointing to the thread and asking for my answer is ok.)

13.5. Parallel make?

Recently, when the GHz-explosion ceased, the idea of multiprocessor computers got popular again, in particular in the "budget" form of dual core processors. Builds are in general intrinsically parallelizeable, and should be able to take advantage of several processors. In particular, since many years GNU make supports parallel builds (issuing several commands in parallel) (the `-j` option, with or without an argument). Can this be used to compile the Tuxbox software in parallel?

The short answer is: "Parallel builds are not supported. But you are welcome to work on it." With the present setup, some components (kernel, u-boot, busybox,...) are built in different versions, for example, there are different kernels for YADD, and for the three different file systems. Different versions can not be build in parallel, since the same files/filenames are being used.

Also when no multiple versions of the same component are being built, a command like `make -j flash-neutrino-jffs2-2x` presently does not produce correct result. Feel free to work on it!

13.6. Kernel 2.6?

Kernel 2.6 is not supported (even if there are some lines regarding it in the source). Feel free to work on it.

13.7. Update images

Sometimes image builders have distributed "update images", consisting of the cramfs (sometimes squashfs) filesystem image, to be flashed as a partition -- in general partition 2. *newmake* also supports this habit. Just, e.g., `make $(flashprefix)/root-neutrino.cramfs`. Neutrino's "expert" flash function recognizes the *newmake* file extensions since 2006-01-02.

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

You don't. The here outlined procedure builds any, whatever the user desires. Also, all legal images are available in both 1x- and 2x-version.

14. Appendix. Some useful customization script fragments

In this appendix, some useful customization scripts will be shown. Two scripts have already been shown above.

Warning:

Although in many cases usable as they are, the scripts are intended as *examples*, not solutions to real problems. For this reason, the examples are included here as code snippets, not as downloadable files. Please do not use unless you understand how they work, at least roughly. To incorporate in the building/customization process requires at least elementary script-writing experience.

14.1. Games and Languages nuker

This file deletes all games (defined as plugins with `type=1` in their configuration file), as well as unwanted languages files (neutrino assumed). This file should probably be called from (or included in) `root-neutrino-$filesystem-local.sh`

```
#!/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 the /.version file

To create your own `/.version` file (shown by Neutrino by `dBox -> Services -> Image Version`) is surely a common requirement. Here is the file I am presently using for this:

```
#/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
```

This file can both be used with the name `flash-version-local.sh`, as well as the name `version-local.sh`, for creating the `/.version`-file for images and yadds respectively. Note the evaluation of `$0` (which contains the actual name, under which the script is called). The called script `mkversion` creates the somewhat cryptical version string, and is simply an "encapsulation" of its idiosyncrasies. It is shown here:

```
#!/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. Archiving the images

It is the task of the build process to create the flash images, not to archive them. However, the customization can easily be "mis"-used to make some sort of archiving, as the following example shows:

```
#!/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
```

The script should have one or more of the names `[neutrino,enigma]-[cramfs,squashfs,jffs2].[img1x,img2x]`. It will rename the file according to the current date. Again, the script is shown to demonstrate a concept, not to be just copied.

15. References

- The GNU Make manual, online version. Also contained in the software distribution.

- The CVS Manual online, known as "The Cederqvist".
- Open Source Development with CVS, 3rd Edition. A quite useful book, downloadable as PDF (among other formats).