Building Embedded Userlands

Ned Miljevic & Klaas van Gend
Solution Architects, MontaVista Europe / USA
Introduction Ned

- FAE for MV Europe since 2008

- Best sounding home network audio player ever
  - Linux powered!

- Father of twin little guys with Californian masks, plastic Japanese swords and Mongolian outfits
Who is Klaas van Gend?

Klaas-the-Geek:

- Started programming age 13
- First encountered Linux 1993
- Software Engineer since 1998
- Lead developer of umtsmon
- Program Committee member for various open source conferences

Klaas-the-Sales-Guy:

- Joined MontaVista as FAE (not sales) 2004
- Was part of European MontaVista Team
- Awarded FAE of the year 2006
- Working in USA until July 1st, 2009
Kernel vs User Space

Today: building the file system(s)

- 90% userland
- 10% kernel

“desktop/server” vs “embedded”
Agenda

- The big decisions in User Space
  - So many packages, so many choices
- Build procedures and challenges
  - Horizontals, Verticals and “diagonals”
- Existing user land build mechanisms
- Summary
Some big decisions in user space

• Choose a libc
• Shell commands vs busybox
• Startup Mechanisms
Pick a libc

There are many C libraries to choose from, and some obvious criteria.

Impact is tremendous:
- quality of C++ support
- completeness
- stability
- size / configurability
  - libopt!
  - upgrade in the field?
- availability for architecture
- compiler modifications (uClibc!)
- do not underestimate community!
  - Security / bugfixes / future

<table>
<thead>
<tr>
<th></th>
<th>POSIX / NPTL</th>
<th>i18n / L10n</th>
<th>RT</th>
<th>size</th>
<th>Actv comm</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU glibc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eglibc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uClibc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Newlib</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dietlib</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSD libc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Busybox

479 kB

Contains 177 commands:

addgroup adduser ash cat chgrp chmod chown cp cpio date dd
delgroup deluser df dmesg echo egrep false fgrep grep gunzip gzip
hostname ip ipcalc kill ln login ls mkdir mklnum mknod mktmp more mount
mv netstat nice pidof ping ping6 ps pwd rm rmdir run-parts sed sh
sleep stty su sync tar touch true umount uname usleep vi watch zcat
linuxrcdevfsd fdisk getty halt hdparm hwclock ifconfig ifdown ifup init
insmod klogd loadmodmap losetup lsmod makedevs mkswap modprobe
namei pivot_root poweroff reboot mmmod route start-stop-daemon
su log in swapoff swapon syslogd vconfig
[[ arping awk basename bunzip2 bzcat chvt clear cmp crontab cut dc dealloctv dirname
dos2unix du env expr find fold free ftpget ftpput head hexdump
hostid id install killall last length logger lowner md5sum msg
mkfifo nc nslookup od openvt passwd patch printf readlink realpath
renice reset rpm2cpio rx seq sha1sum sort strings tail tee telnet test
tftp time top tr traceroute tty uniq unix2dos unzip uptime uudecode
uuname vlock wc wget which who whoami xargs yes chroot crond
tfset httpd inetd rdate telnetd

sh 312k (tcsh), 656k (bash), 86k (dash)
cp 55k
grep 105k
login 34k
mkdosfs 24k
mkfifo 17k
mkfs.ext3 39k
mount 78k
mv 63k
nice 18k
rm 38k
setserial 20k
sleep 18k
stty 42k
getty 15k (agetty), 93k (mgetty)
ifconfig 61k
vi 352k (nvi), 1003k (vim)
---------
TOTAL 1291 kB

Source: MontaVista Pro 5.0 for x86_pentium3
System V approach:
- `/etc/init.d/rcX.d/*`
- `/etc/inittab`
  - Runlevels !!!
  - Many options like wait, respawn, powerwait/powerfail
  - Scripts usually require sed, grep, awk present
  - Many fork/exec

“Busybox approach”:
- `/etc/inittab` (optional)
  - Much simpler, no concept of runlevels, will start console
- 1 simple `/etc/rcS` file
  - can run other files if needed

New developments:
- Upstart
- runit
Build procedures and their challenges
Feature selections

System Designer must map requirements to packages

For a non-Linux user, this poses a significant challenge

• Finding packages that fulfill requirements:
  – Why is the BGP router called “zebra”? 
  – Why are there multiple, different “zebra” projects?
• What’s the advantage of package A over B? (Gnome vs KDE?)
• Do I need all features or strip down?

  – “need threads” - use libc with NPTL / uClibc with LinuxThreads?
  – “need webserver” - use Apache / boa / thttp / busybox (etc)
  – “need SNMPv3” - use NetSNMP, Level9, write own
  – “need GUI” - QT, DirectFB, GTK, LinuxPEG, etc
Naïve embedded user land building

Package selection

I need it quick. Let’s do it by hand

Create Image

File selection

Ubuntu, openSUSE, Debian

Montavista
Disadvantages

Limited architecture support
- Usually x86 only

No other features than the standard ones
- Rebuilding packages is probably beyond this user’s knowledge

No reproducibility, high risk for human errors
- More or less solvable
  - use scripts for each transition
  - Generate manifests
  - Still not future proof

A lot of work !!!
- And no guarantee of success

But I had fun and learned a lot!
Examples:
- LICENSE files
- no man pages
- SSL support
- python bindings
- IPv6 awareness
- SELinux compat.
- LDAP support
Dependencies

Verticals often are dependencies:
- You need other packages to make it work

Diagonals *may* add dependencies:
- Adding LDAP as requirement probably should add OpenLDAP to the target file system
- Adding “no_man” should remove all man pages from the target file system – and *man* itself

Dependency resolution can be a daunting task
- “RPM hell” anyone?

*At least* two types of dependencies exist:
- Build-time dependencies, e.g: `cmake`, `glade`, `qt-devel`
- Run-time dependencies, e.g: `libcrypto`, `perl`, `PAM`, `libqt`
“Pristine” vs patched: recipes

Building your user land from source gives more choices
  - It doesn’t make smaller binaries by definition!

Building from source also introduces new problems
  - Bug fixes / Feature adds
  - Tracking upstream releases

Make sure to differentiate between the original released source tarball (“upstream”) and any patches on top of that
  - Patches change or (hopefully) go away with newer releases!
  - Kernel folks: look at quilt by Andrew Morton

Definition:
A recipe describes how to patch & build a certain package from source
  - Preferably taking dependencies and diagonals into account
  - E.g. RPM’s SPEC files and Gentoo’s ebuilds are examples of recipes
Cooking with recipes (1)

Grocer / Butcher / etc

Recipe suggestions

Chef

Create menu

Menu

recipes

Sous Chef

Ingredients

Good food

Customers
Community

public repos

Sync sources + recipes

Add patches + custom recipes

Define manifest (what to build)

System Designer

Build machine

Packages, Tools, File Systems

Developers
The cross compilation challenge

- Variable types / sizes:
  - 32-bit: `sizeof(int) == sizeof(long)`
  - 64-bit: `sizeof(int) == sizeof(long long)`
  - Endianness
  - Packing of structs
  - Different ABIs

- The include directories:
  - `/usr/include` usually contains system specifics and inline assembly
  - Careful with location + output from scripts like `pkgconfig` or `qmake`

- Compiler names / arguments:
  - Name: `gcc` vs `arm_v6t_vfp-gcc`
  - Arguments: `--march=x86` vs `--mcpu=mips2_fp_le`
  - GNU Autotools
`./configure` usually tests for system/architecture-specific topics by running small test programs – on the host???
Don’t reinvent the wheel - learning curve

If you do things by hand you need to know all details presented here

But:

• **Recipes** should contain most of the “what to build from what” knowledge
  – How difficult is creating/modifying new recipes?
• A **tool** should do all the work – take the recipes and brew the file system
  – How difficult is the tool to setup and how self-explanatory are the error messages?
  – How does a master chef differ from my mother?

The **combo** should hide many of the ugly complexities
  – like cross-compiling and autotools

**Quiz 1:** What is a Linux distribution?
**Quiz 2:** Why have most distributions invented new tools?
**Quiz 3:** Which is best for embedded?
Some analysis of available build systems
To save time during the presentation, we skipped all the slides in this chapter and ran immediately to the summary table at the end.
Buildroot

- Essentially a set of Makefiles and patches that generate the toolchain and the target FS
- patches for packages and compiler to ensure proper cross-compile
- **uClibc based** – no other libraries supported
- good support for different architectures and boards
- cross-compilation environment works across different hosts
- adding your own packages: modify the example Makefile
- OK for building a single executable – for building libraries we have to find our own recipe
A sandbox system for building Linux systems from scratch

- ARM and x86 supported
- Development system for maemo
- Debian centric
- ./configure test for gcc producing executable files = 'cpu transparency' (qemu or real target [sbrsh])
- Prescribed toolchain
- glibc and uClibc support
- Libraries provided by toolchain or rootstraps
- Package management through apt
- Reported having been used to build Slackware for ARM
- ./configure – make cycle
- Well documented (web site)
- Succeeded by “scratchbox2”
scratchbox2

- more host OS agnostic
- documentation??
RPMbuild

- Build-your-own RPMs
- Used by Fedora / Red Hat, openSuse, MontaVista, Mandriva
- Works with a RPM spec file (metadata) – build tool agnostic
- Contains instructions how to prepare, compile and install the package
- Works within the usual RPM directory structure

Example:

```bash
#define buildroot <my_root>
# below is a standard RPM macro
#define _prefix <my_prefix>
%prep
%setup
%build
%configure
make

%install
rm -rf %{buildroot}
make install DESTDIR= %{buildroot}

%clean
rm -rf %{buildroot}
```
Fedora on ARM

- Fedora approach: native build – on ARM boards or qemu
- Provides a target file system – can run in qemu
- Target FS built with rfsbuild - yum used for package maintenance
- Needs quite a capable Linux system to run:
  - Python 2.4 and several other packages
  - qemu for native compilation (distcc is possible)
- Mainly supported by Marvell

- Architecture: ARM
  - ARMv5, LE, Soft-Float, EABI
- What if we need something else??
Kconfig-based

- Stems from the Linux kernel configurator
- Examples:
  - uClibc
  - PTXdist
- Uses quilt for patching
- Configuration: `make menuconfig`
- Adding new packages implies writing new kconfig files
- Community:
  - At least 20 very active participants from various companies
  - Used in real embedded systems
  - Various architectures used
- Does allow for diagonals (sort of) through the kconfig mechanism
Ebuilds

Created by Gentoo project for Portage use
• Able to cross compile and/or perform a sandbox install

```
"portage"  "emerge"  "ebuild"
=concept    =tool      =recipe
```

The ebuild contains:
- Run time & compile time dependencies,
- Instructions for download,
- Instructions for patch,
- Compilation,
- Installation

• USE flags work roughly as diagonals
  - /etc/make.conf
  - /etc/portage/package.use
Ebuild for “beep”

# Copyright 1999-2006 Gentoo Foundation
# Distributed under the terms of the GNU General Public License v2
# $Header: /var/cvsroot/gentoo-x86/app-misc/beep/beep-1.2.2-r1.ebuild,v 1.3 2006/08/19 11:00:37 kloeri Exp $

inherit eutils base
DESCRIPTION="the advanced PC speaker beeper"
HOMEPAGE=http://www.johnath.com/beep/
SRC_URI="http://www.johnath.com/beep/${P}.tar.gz"
LICENSE="GPL-2"
SLOT="0"
KEYWORDS="alpha amd64 ~ppc ~ppc64 ~sparc ~x86"
IUSE=""

PATCHES="${FILESDIR}/${P}-nosuid.patch"
src_compile() {
    emake FLAGS="${CFLAGS}" || die "compile problem"
}

src_install() {
    dobin beep
    fperms 0711 /usr/bin/beep
    doman beep.1.gz
    dodoc CHANGELOG CREDITS README
}
Bitbake

- Tool for executing tasks and managing metadata
- Derived from Portage
- Basis of OpenEmbedded
- Distributions using it: Ångström, OpenMoko, Poky, SlugOS
- Learning curve is very steep

Example recipe:

```
DESCRIPTION = "hello world sample program"
PR = "r0"

DEPENDS = ""

SRC_URI = " file://hello.c "
```
S = "${WORKDIR}"

do_compile () {
    ${CC} ${CFLAGS} ${LDFLAGS} -o hello hello.c
}

do_install () {
    install -d ${D}/${bindir}/
    install -m 0755 ${S}/hello ${D}/${bindir}/
}

FILES_${PN} = "${bindir}/hello"
<table>
<thead>
<tr>
<th>Tool / Distro</th>
<th>Recipe</th>
<th>Hori</th>
<th>Vert</th>
<th>Diagonal</th>
<th>Cross compile</th>
<th>Multi Arch</th>
<th>Community</th>
<th>Learning Curve</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buildroot</td>
<td>Makefile</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Medium</td>
<td>Med/lo</td>
</tr>
<tr>
<td>Scratchbox</td>
<td>Makefile</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes?</td>
<td>Yes</td>
<td>Few?</td>
<td>Med</td>
</tr>
<tr>
<td>RPM</td>
<td>RPM SPEC</td>
<td>Yes</td>
<td>Runtime</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>???</td>
<td>Med/Hi</td>
</tr>
<tr>
<td>Fedora on ARM</td>
<td>RPM SPEC</td>
<td>Yes</td>
<td>Runtime</td>
<td>No</td>
<td>Yes?</td>
<td>Yes</td>
<td>Few</td>
<td>Med/Hi</td>
</tr>
<tr>
<td>KConfig</td>
<td>.config</td>
<td>Yes</td>
<td>~</td>
<td>~</td>
<td>Yes</td>
<td>Yes</td>
<td>Some</td>
<td>Med/Lo</td>
</tr>
<tr>
<td>Portage</td>
<td>Ebuild</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Big</td>
<td>Med</td>
</tr>
<tr>
<td>Bitbake</td>
<td>Recipe</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Some</td>
<td>😞</td>
</tr>
</tbody>
</table>
Conclusions & Summary
Summary

- Building a file system should not be an afterthought
  - It is complex, it has loads of impact on other features
  - It is system designer complexity – don’t leave to the junior engineer
- Product features hugely impact file system design
  - And vice versa: 64MB is cheaper than 256 MB

- Recipes and Diagonals
  - very important to platform products
  - Simplify a designer’s job
  - Require a community to work well

- There are many tools to simplify the task
  - At least, they claim to do so

- MontaVista is going to move away from RPMbuild
Questions & Thank You

nmiljevic@mvista.com
Klaas.van.gend@mvista.com