Designing a distro from scratch

using OpenEmbedded

Part 2

Koen Kooi <koen.kooi@linaro.org>

ELCE Berlin 2016

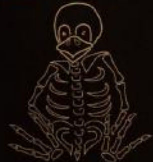
Overview

1. OpenEmbedded basics
2. Initsystem choices
3. Dealing with BSPs
4. Dealing with Binary blobs

Don’t hesitate to interrupt if you have questions or remarks!

Part one slides available at https://goo.gl/HiRhi5

Part two (the one you’re watching) slides available at https://goo.gl/qt3IKp


YÖP
ÏS
MÖAR MËTÄL!!

OpenEmbedded basics (1/2)

- OpenEmbedded is part of the Yocto Project umbrella organization
- OpenEmbedded is a buildsystem
- Closest equivalent: Buildroot

- OpenEmbedded is **NOT** a distribution

OpenEmbedded basics (2/2)

- OE consists of
  - Recipes
  - Config files
  - A task executor called bitbake
- Three orthogonal concepts
  - MACHINE.conf, a description of the target hardware (i.e. powerpc, screen, networking)
  - DISTRO.conf, a collection of policies for the build (i.e. systemd, PAM, rpm)
  - Image.bb, a description of the output filesystem in terms of packages and format (i.e traceroute, ext4.gz)

Init systems

- In theory you can use any init system you want
- In practice the recipes need to support your init system of choice
  - Massive collection of bbappends
  - Meta-systemd: Git log
- OE-core supports sysvinit and systemd

Picking systemd

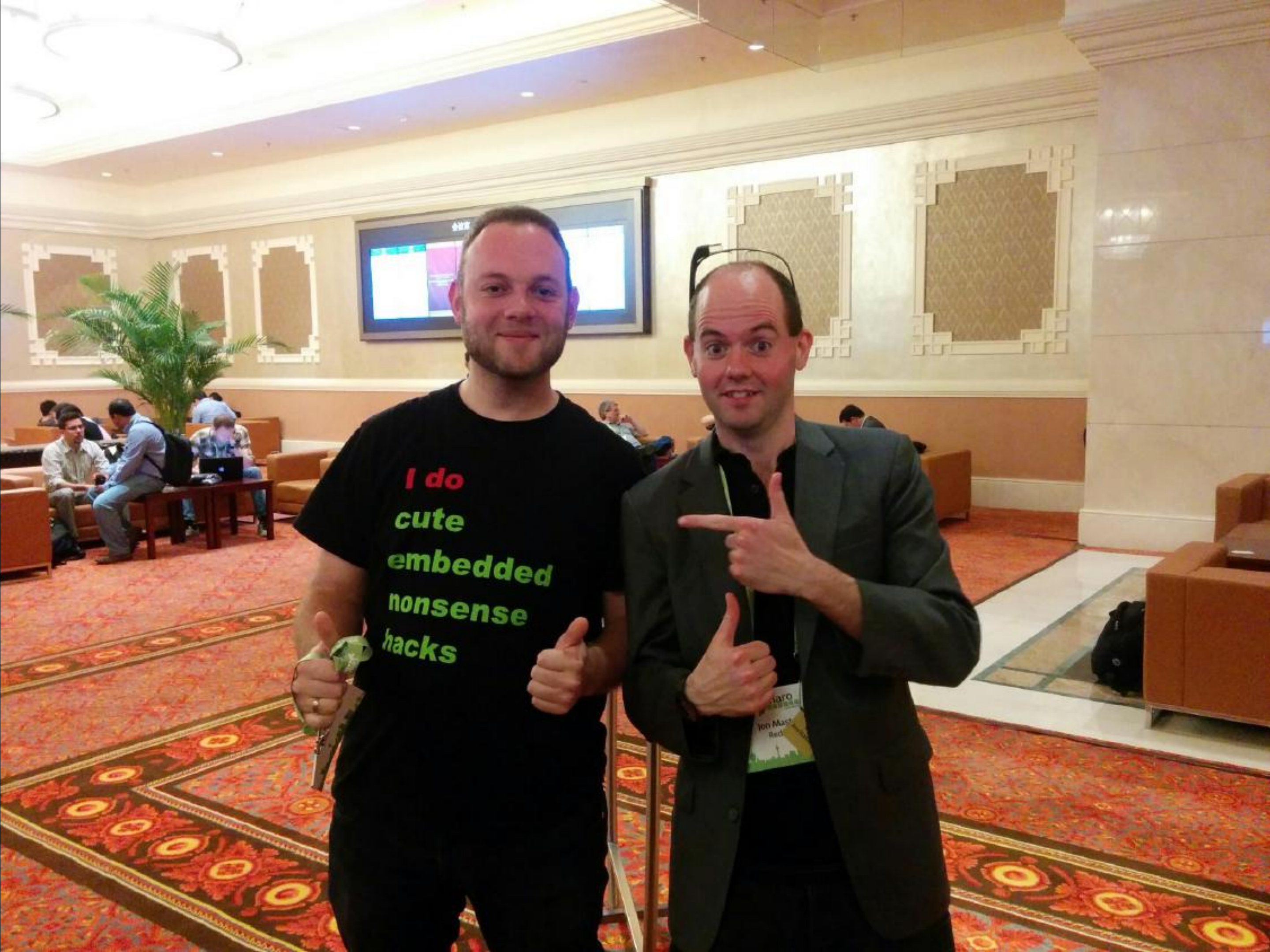
```
DISTRO_FEATURES_append = " systemd"
DISTRO_FEATURES_remove = "sysvinit"
VIRTUAL-RUNTIME_init_manager = "systemd"

PACKAGECONFIG_append_pn-systemd = " resolved networkd"
DISTRO_FEATURES_append = "pam"
```

C libraries

- Glibc and uclibc are supported in OE-core
- Musl is supported by meta-musl
- TCLIBC=musl bitbake my-image
- As with init systems: know what you’re getting into

C libraries

- Space savings won’t be as impressive as you’d expect
  - NLS is turned on by default
  - Libiconv is huge
- Musl seems to be displacing uclibc as glibc alternative


I do
cute
embedded
nonsense
hacks

Dealing with BSPs

- BSPs are a necessary evil for the embedded zoo
- No standard to live up to, very low metadata quality in general
- Using multiple BSPs in your $DISTRO is actively discouraged by ‘yocto’
- ARM and x86 BSPs are the worst offenders when it comes to ‘anti-social’ behaviour

Wait, actively discouraged?

- Documentation says “take poky, add your BSP”
- Reporting BSPs interactions gets met with “Well, don’t combine them.”
  - “I wrote a tool to automatically enable a BSP and disable all others”
  - “Hey, me too!”
- BSP maintainers generally don’t care or don’t want to understand the interaction issues being reported.

What’s ‘anti-social’ about my BSP?

- It pokes at DISTRO stuff, breaking ABI
- You set DEFAULTTUNE to something that changes PACKAGE_ARCH
- Your libdrm_%.bbappend has patches that fail to apply for every version that isn’t 2.4.66
- You have a glibc recipe that shadows the OE-core one
- You have a linux-libc-headers bbappend without overrides
- Your mesa bbappend deletes all mesa libraries in do_install, without override safeguards
- You have a more recent linux-yocto recipe than OE-core

DEFAULTTUNE = “Gcc -OMG -noatime”

- DEFAULTTUNE is used for 2 things:
    a. Selecting the ISA
    b. Selecting the ABI
-

```
# This function changes the default tune for machines which
# are based on armv7a to use common tune value, note that we enforce hard-float
# which is default on Ångström for armv7+
# so if you have one of those machines which are armv7a but can't support
# hard-float, please change tune = 'armv7athf' to tune = 'armv7at'
# below but then this is for your own distro, Ångström will not support
# it
# - Khem
def arm_tune_handler(d):
   features = d.getVar('TUNE_FEATURES', True).split()
   if 'armv7a' in features or 'armv7ve' in features:
       tune = 'armv7athf'
       if 'bigendian' in features:
           tune += 'b'
       if 'vfpv3' in features:
           tune += '-vfpv3'
       if 'vfpv3d16' in features:
           tune += '-vfpv3d16'
       if 'neon' in features:
           tune += '-neon'
       if 'vfpv4' in features:
           tune += '-vfpv4'
   else:
       tune = d.getVar('DEFAULTTUNE', True)
   return tune
DEFAULTTUNE_angstrom := "${@arm_tune_handler(d)}"
```

Dealing with BSPs

Ideally a BSP would consist of multiple layers:

1. A base layer with kernel, bootloader, firmware
2. A second layer with codec, Wi-Fi, DSP support
3. Another layer with tweaks to recipes

That 3rd layer is where most integration problems will be:

- ‘*My special snowflake MACHINE really needs rfkill support in busybox’*
- *‘My MACHINE has a 2D engine, so I disabled pixman support everywhere’*

GPU blobs

- Do they belong in the BSP?
- Is it DISTRO policy?
- No ‘best practices’ around

conf/distro/include/mali.inc

```
MALI_USERLAND_LIBARIES ?= "mali450-userland"

# Helper function for overloading the default EGL/GLES implementation.
# The Mali libraries provided by ARM are compatible with the Mesa headers
# and it is safe to use with user space applications linked against Mesa.

def get_mali_handler(d, target):
   """  Overloading the default EGL/GLES implementation."""
   features = d.getVar('MACHINE_FEATURES', True).split()
   mali_libs = d.getVar('MALI_USERLAND_LIBARIES', True);

   if(mali_libs):
       mali_libs = mali_libs.split()

   if 'mali450' in features and mali_libs:
       provider = mali_libs[0]
   else:
       provider = "mesa"

   return provider;

PREFERRED_PROVIDER_virtual/egl := "${@get_mali_handler(d, 'egl')}"
PREFERRED_PROVIDER_virtual/libgles1 = "${@get_mali_handler(d, 'libgles1')}"
PREFERRED_PROVIDER_virtual/libgles2 = "${@get_mali_handler(d, 'libgles2')}"
```

mali450-userland_r6p0_01rel0.bb

```
# Disable for non-MALI machines
python __anonymous() {
   features = bb.data.getVar("MACHINE_FEATURES", d, 1)
   if not features:
       return
   if "mali450" not in features:
       pkgn = bb.data.getVar("PN", d, 1)
       pkgv = bb.data.getVar("PV", d, 1)
       raise bb.parse.SkipPackage("%s-%s ONLY supports machines with a MALI iGPU" % (pkgn, pkgv))
}
```

Outside perspective

http://lwn.net/Articles/681651/

There's a set of simple things projects can do the be more friendly (or unfriendly) to distributions... (speaking as someone who builds a distribution).

Good things - http://lwn.net/Articles/681651/

- Use a standard build/make system (autoconf, cmake, python setuptools, whatever, something that is pretty widely used)

- Clear license declaration (COPYING file)

- include unit tests (make test/make check); a distribution can and will use this this to verify they integrated the component correctly

- use pkg-config for dependencies

- regular releases, at least for bugfixes and security fixes (bonus points for having maintenance releases against latest stable in addition to more major releases, but rolling release is fine)

- Know what an "ABI break" is if you are providing a library (Note: C++ makes it much harder to keep ABI, but it can be done, see the Qt folks)

Bad things - http://lwn.net/Articles/681651/

- Custom Makefile hackery that is not parallel build safe or ignores DESTDIR etc etc
- Unit tests that fail always on the official release
- No clear declaration of license
- Have "creative" ideas on where files go... when in doubt, please just follow the FHS.
- Not using the system CFLAGS, but thinking you know better (expanding by adding things to the system CFLAGS is fine, but don't throw the distro provided flags away)
- Adding -Werror to CFLAGS.... newer versions of compilers add warnings, and -Werror will just require distros to patch -Werror away again