First Experiences with the Embedded Debian Build System Isar

Jan Kiszka | Embedded Linux Conference, February 21, 2017
Agenda

Why Yet Another Build System?
Isar Introduction
First Steps
Customizations
To-Dos & Outlook
Summary
### Types of Embedded Linux Build Systems

- **“Roll your own” (OE, Yocto, buildroot, ...)**
  - Usually implies cross-build & toolchain bootstrap
  - Highly customizable
  - Production times explode with long package lists
  - Non-zero dependencies on host for reliable reproducibility

- **Distribution-based (ELBE, OBS, Ubuntu Core...)**
  - Reuse standard desktop/server distribution
  - Install pre-built binary packages
  - Larger images & slower boots – unless extra customization is applied

- **Hybrid approach (meta-debian)**
  - Use distribution packages
  - Rebuild from source under Yocto
  - New package recipes required (replicates poky + OE)
## Requirements on Embedded Linux Build Systems

<table>
<thead>
<tr>
<th>General needs</th>
<th>Our additional requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Generate ready-to-use device firmware image</td>
<td>• Avoid building from source where possible</td>
</tr>
<tr>
<td>• Ensure reproducible builds</td>
<td>• Integrate binary packages</td>
</tr>
<tr>
<td>• Support for integration of business logic as well as third-party components</td>
<td>• Do not invalidate upstream QA</td>
</tr>
<tr>
<td>• Enable product-line development: reusable components &amp; configurations</td>
<td>• Access to large package pool for increasingly complex systems</td>
</tr>
<tr>
<td>• Easy for beginners, yet powerful for advanced</td>
<td>• Reuse well-established long-term maintenance</td>
</tr>
<tr>
<td></td>
<td>• Clean and well-documented package licenses</td>
</tr>
</tbody>
</table>
Choosing a Base Distribution

Why Debian?

- Large community-driven ecosystem
- Popular in embedded (Raspbian, Armbian, …)
- Successfully used in several of our products
  - But no standardized image builder
- Long-term support
- Strict license checks
  - To ensure “free software only”
- Scales up and down
Isar – New Project, Long History

- 2004: SLIND (Siemens Linux Distribution, Debian-based, cross-building) + build.sh
- 2011: SLIND + bitbake, used in Siemens products
- 2015: Debian + bitbake
- 2016: Released as Isar open source project
  - Developed by ilbers GmbH
  - Sponsored by Siemens Corporate Technology
- Isar?
  - Integrated System for Automated Root filesystem generation
  - River along traditional BBQ site in Munich

Copyright Usien, CC BY-SA 3.0
Combining the Best of 3 Worlds

Debian Base System
Integration via Bitbake
Isar
Yocto Structure & Workflow
How Isar Works (ARM Targets)
# First Steps into the Isar

## Testing in QEMU
- Requires Debian on the host or in a VM
- Clone https://github.com/ilbers/isar
- `isar-init-build-env /path/build-dir`
- `bitbake multiconfig:qemuarm:isar-image-base`
- `start_armhf_vm`

## Testing on Raspberry Pi 1
- Uses Raspbian instead of Debian
- `bitbake`
  ```bash
  multiconfig:rpi:isar-image-base
  ```
- `cd tmp/deploy/images; dd if=isar-image-base.rpi-sdimg of=/dev/mmcblk0 bs=4M`
Structure of Isar

Top-level view

- **bitbake** – Recipe interpreter
  (copy, updated from time to time)
- **meta** – Core layer
- **meta-isar** – Template layer
- **scripts** – Helpers scripts
- **isar-init-build-env** –
  Build environment setup script
  (equivalent to `oe-init-build-env`)
# Starting an Isar Project

## Basic steps
- Clone Isar repository
- Derive from meta-isar as template
- Add your own
  - ...image
  - ...packages
  - ...board (machine)

- Or create separate repo with own layer, including Isar layer in configuration (bblayers.conf)
- Configuration management via repo etc.

## Options when organizing in layers
- meta-VENDOR1-bsp (U-Boot, kernel, ...)
- meta-VENDOR2-libs (codecs, ...)
- meta-COMPANY: Company-wide common bits
- meta-PRODUCT1 (app1, ...)
- meta-PRODUCT2 (app2, ...)
Adding Your Own Image

### Basic steps
- **Setup**
  - Derive from templates (see `meta-isar/recipes-core/images`)
  - Extend base image
- **Typical tasks**
  - Add Debian packages → IMAGE_PREINSTALL
  - Add self-built packages → IMAGE_INSTALL
  - Add files to rootfs → do_rootfs task
  - Modify rootfs → .../images/files/debian-configscript.sh

```makefile
my-image.bb

require \n  recipes-core/images/isar-image-base.bb

IMAGE_PREINSTALL += "dropbear"

do_rootfs_append() {
  install -m 600 \n    ${THISDIR}/files/dropbear_ecdsahost_key\n    ${S}/etc/dropbear
}
```
## Adding Your Own Application

<table>
<thead>
<tr>
<th>Basic steps</th>
<th>Source-based example recipe</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Two options</td>
<td></td>
</tr>
<tr>
<td>• Build out of source via Isar</td>
<td></td>
</tr>
<tr>
<td>• Install pre-built Debian package from own repo</td>
<td></td>
</tr>
<tr>
<td>• Source-based</td>
<td></td>
</tr>
<tr>
<td>• Code repo must be Debianized (debian/ folder containing metadata files)</td>
<td></td>
</tr>
<tr>
<td>• Build natively, using QEMU in cross setups</td>
<td></td>
</tr>
<tr>
<td>• Add package to IMAGE_INSTALL in local.conf or your own image recipe</td>
<td></td>
</tr>
</tbody>
</table>

### Source-based example recipe

```bash
DESCRIPTION = "Sample application for ISAR"
LICENSE = "gpl-2.0"
LIC_FILES_CHKSUM = \n  "file://${LAYERDIR_isar}/licenses/CO[...]"
PV = "1.0"
SRC_URI = \n  "git://github.com/ilbers/hello.git"
SRCREV = "ad7065ecc484..."
SRC_DIR = "git"
inherit dpkg
```
Using a Custom Kernel

Kernel or app – no major differences

- Debianize your kernel tree
- Let Isar build it
- See example in custom_kernel branch (needs fix-up for URI to work)

- Or build separately (e.g. `make deb-pkg`) and pull from local repo
Custom Kernel + Debianization

<table>
<thead>
<tr>
<th>Carry metafiles in Isar</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Goal: Keep kernel tree unmodified (e.g. pull from external source repo)</td>
</tr>
<tr>
<td>• Copy metafiles and config from recipe into unpacked Linux tree</td>
</tr>
<tr>
<td>• Make pattern reusable via include</td>
</tr>
<tr>
<td>• Pattern may also be applied to self-built applications</td>
</tr>
</tbody>
</table>
Reusable files

meta/recipes-kernel/
  └── linux
      ├── files
      │    └── debian
      │        ├── changelog
      │        │    └── compat
      │        ├── control
      │        ├── control.in
      │        └── README.debian
      └── rules

linux.inc

DESCRIPTION = "Linux Kernel"
FILES_PATH_prepend := "${THISDIR}/files:
LICENSE = "gpl-2.0"
LIC_FILES_CHKSUM = "...
SRC_URI += "file://debian/ \  
             file://defconfig"
SRC_DIR = "git"
inherit dpkg

do_build_prepend() {
    cp ${BUILDROOT}/defconfig \  
       ${BUILDROOT}/${SRC_DIR}/arch/arm/... ...configs/isar_defconfig
    cp -r ${BUILDROOT}/debian \  
       ${BUILDROOT}/${SRC_DIR}
### Custom Kernel Recipe – Specific Bits

<table>
<thead>
<tr>
<th>Self-provided files</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>meta-mylayer/recipes-kernel/linux/</code></td>
</tr>
<tr>
<td>├── files</td>
</tr>
<tr>
<td>│   └── <code>defconfig</code></td>
</tr>
<tr>
<td>├── <code>linux_4.9.bb</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>linux_4.9.bb</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>require recipes-kernel/linux/linux.inc</code></td>
</tr>
</tbody>
</table>
| `SRC_URI += "git://git.kernel.org/[…]
  …/linux-stable.git;branch=linux-4.9.y"
SRCREV = "390caeedd4fd"` |
Adding Custom Bootloader

See custom_uboot branch

- Debianize u-boot
  → see custom_uboot branch in Isar repo
- Or apply kernel pattern presented before
How to make it bootable?

Image layout definition

• # meta-isar/conf/machine/rpi.conf
  IMAGE_TYPE = "rpi-sdimg"
• # meta-isar/classes/rpi-sdimg.bbclass
  inherit ext4-img
  do_rpi_sdimg () {
    # shell commands
  }
  addtask rpi_sdimg before do_build \ 
    after do_ext4_image
• Logically the same what Yocto & Co. do
• Vision: use common tools and descriptions – wic?
# Lessons Learned

## The good
- Similarities to Yocto / OpenEmbedded helps writing recipes
  - Same language
  - Same structuring
- Recipes can become very simple
- Image generation out of packages is nicely fast
- Structure is simple, code complexity still low
  - About 350 LOC Python and shell code
  - OK, +Bitbake (46K Python code)

## ...and the not-yet-so-good
- Needs root-privileges for image generation
- Room for improvements for recipe development
  - Some recipe changes do not trigger rebuilds as expected
  - No clean-up tasks implemented yet
- QEMU-based build can be too slow for large packages (e.g. kernels)
  - Selective cross-building, at least during development
  - ARM-based build servers can help
What's next?

Isar changes in the queue

- Address findings reported earlier
- x86 support, adding reference board
- Jessie integration & fixes
- Image creation via `wic`
- Documentation improvements
One size fits all?
Out-of-Source Build Systems Remain Relevant

- SIMATIC IOT2000 (industrial IoT platform)
  - Erratum workaround for Intel Quark in toolchain
  - Many distro packages do not work
- Also: systems highly optimized
  - for size (package configurations)
  - for performance (compiler tunings, boot times)
Summary

- Isar: Promising framework for building embedded Debian images
- Some rough edges remaining, but none seem unfixable
- Code & recipe sharing is in the center
  - ...between Isar-based images
  - ...with related build system projects (Yocto, ELBE, meta-debian)
- Smooth path for switching between Yocto-based and Debian-based projects
Resources

- Code: https://github.com/ilbers/isar/
- Mailing lists
  - Isar specifics: https://groups.google.com/d/forum/isar-users
  - Collaboration topics: https://lists.debian.org/debian-embedded/
Thank you!

Jan Kiszka <jan.kiszka@siemens.com>