Using Yocto to Build an IoT OS
Targeting a Crossover SoC

Ryan Fairfax
Principal Software Engineering Lead
Microsoft
Agenda

- Intro to Crossover SoCs
- Example use case: Azure Sphere
- Building with Yocto
- Lessons learned
What is a Crossover SoC?

Many embedded SoCs fall into one of two broad categories:

• **Application processors**
  - Highly capable CPUs with high clock speeds, MMUs, and features of modern phones or PCs
  - Often run Linux, Android, or another high-level OS (HLOS)
  - Usually includes networking (Wi-Fi, Cellular, etc.).
  - More expensive ($10s)

• **Microcontrollers (MCUs)**
  - Feature limited CPUs designed for real time or highly predictable applications
  - Often run real time OSes (RTOS) such as Zephyr, ThreadX, or FreeRTOS
  - Less than 1% include networking
  - Very inexpensive ($1s)
What is a Crossover SoC?

Crossover SoCs combine the best of both worlds via multiple CPU cores on a single chip

- Application cores for networking, advanced graphics, machine learning
- Microcontroller cores for real time sensors, motors, and simple displays

Examples:
- MT3620 (MediaTek) – Cortex-A7 + 3x Cortex-M4F + 802.11 Wi-Fi
- STM32MP1 (ST Microelectronics) – 2x Cortex A7 + Cortex-M4 + Gigabit Ethernet
- i.MX 8M (NXP) – 4x Cortex-A53 + Cortex-M4 + GPU + Gigabit Ethernet
Why Crossover SoCs

Why do products use crossover SoCs?

• Lower cost versus two chips on board
• Leverage Linux without having to try to do real-time app logic in Linux
• Add security and features to an MCU product

Crossover SoCs are increasing in popularity and availability

• Cost has dropped. Now available for < $10
• Easier adoption curve – port your existing RTOS to a core and run Linux on the other
• More silicon vendors producing crossover chips
Use Case: Azure Sphere

Azure Sphere is a solution for building secure IoT devices

- Runs a custom crossover OS that uses custom firmware, a secure enclave, the Linux kernel
- There are many OS components in the system:
  - Pluton, a dedicated security core (M4)
  - Security Monitor, a TrustZone based secure enclave (A7)
  - Linux (A7)
  - Real time applications (M4)
Challenges of building for Crossover SoCs

**Embedded Linux development is focused on symmetric multiprocessing (SMP)**
- Assume that all CPU cores are “the same” or the same architecture
- Run your HLOS on all CPU cores, let the OS handle scheduling
- An app can run on any of the available CPU cores

**Crossover SoCs imply asymmetric multiprocessing (AMP)**
- Different types of CPU cores with different capabilities
- Run multiple different OSes or firmware that cooperate to form a final product
- Apps are targeted to specific sets of CPU cores

**Most build systems are optimized for SMP, not AMP**
Building for Crossover SoCs using Yocto

Yocto is primarily designed to build embedded Linux

- There is minimal support out of box for non-Linux targets or multiple target architectures
- But Yocto is *very* extensible

There are three main techniques for building for your non-Linux targets:

- Recipe level overrides
- Custom classes
- Multi-config
Recipe Level Overrides

The simplest approach to build your non-Linux targets is to change flags on a per recipe basis

• You may already be doing this in recipes like uboot
• This works well if you have only a few non-Linux recipes and they’re self contained

There are some challenges to watch out for

• If you need to build a recipe for multiple targets it gets complicated quickly
• Build output ends up in the same package arch as other recipes (can mostly control)
• Dependencies can get complicated, especially if you produce the same file in multiple recipes like common headers

```bash
do_compile () {
    # Remove Linux values
    unset LDFLAGS
    unset CFLAGS
    unset CPPFLAGS

    # Set target specific values
    export CFLAGS="-mcpu=cortex-m4 ...

    # Run make
    oe_runmake
}
```
Custom Classes

Yocto already has a mechanism to handle different targets at build time – classes

- Controlled via BBCLASSEXTEND
- You’ve seen it for “-native” and “nativesdk-” targets
- Fully extensible with only a few caveats

To create a new class

- Create a new class definition in “<class>.bbclass”
- In the class, override DEFAULTTUNE and other common flags
- Set CLASSOVERRIDE and add a virtclass handler
- [https://gist.github.com/rfairfax/c7a3258b63300372ee130dc5a75833d3](https://gist.github.com/rfairfax/c7a3258b63300372ee130dc5a75833d3)

```bash
# Setup our tune settings
DEFAULTTUNE = "cortexm4"

# Define our target settings
TARGET_ARCH = "${TUNE_ARCH}"
TARGET_OS = "${ABIEXTENSION}"

# We are a bare metal setup, not a full libc
TCLIBC = "baremetal"
LIBCOVERRIDE = ":libc-baremetal"

# Setup FW defaults
TARGET_CFLAGS = "-Wall -Werror -Wno-unknown-pragmas -Wno-comment -Os"
TARGET_CXXFLAGS = "-Wall -Werror -Wno-unknown-pragmas -Wno-comment -Os"
TARGET_LDFLAGS = "-static -nostartfiles -nostdlib -Wl,--gc-sections"

# Must be built explicitly
EXCLUDE_FROM_WORLD = "1"

# Setup our new class
CLASSOVERRIDE = "class-securityfw"
```
Custom Classes

To fully support some configurations you may need some extra items:

- Newly defined tunes for microcontroller CPUs
- Custom gcc & binutils to target the core
- [https://gist.github.com/rfairfax/c5c3a44913213fde35fceac51b660663](https://gist.github.com/rfairfax/c5c3a44913213fde35fceac51b660663)

To use in your recipes

- Add BBCLASSEXTEND="<class>"
- Build with "bitbake <class>-recipe" or add to DEPENDS

DEPENDS_append_class-securityfw = "fw-runtime"
DEPENDS_append_class-hlosfw = "hlos-runtime"
BBCLASSEXTEND = "native nativesdk hlosfw securityfw"
Custom Classes

**Pros**

- Can build a recipe for multiple targets (helpful for common libraries)
- Opt-in on targets – can’t attempt a build on a class not in BBCLASSEXTEND
- Rest of the system “just works” – dependencies, packaging, tunes, etc.

**Cons**

- A bit harder to reason about if you’re not familiar with the product. Should I be building my_recipe or securityfw-my_recipe?
- May need to modify setscene_depdevvalid in sstate.bbclass to handle cross-target dependency graphs like how it handles “-native” recipes.
Multi-config

Recent versions of Yocto have included support for “multiconfig” builds

- Multi-config enables you to target more than one Yocto machine at once
- Each logical CPU is modelled as a machine in this model
- Often needs no recipe changes

To use multi-config

- Create a multiconfig folder with a machine.conf for each machine
- Update local.conf to have BBMULTICONFIG="machine1 machine2 ..."
- To build use the extended syntax “bitbake multiconfig:machine1:my_recipe"
- Declare dependencies using the mcdepends flag:

  task[mcdepends] = "multiconfig:from_multiconfig:to_multiconfig:recipe_name:task_on_which_to_depend"
Multi-config

Pros
• Very little modification needed to support
• Easy to build recipes for multiple machine targets
• Continued investment by Yocto team

Cons
• Very verbose bitbake syntax
• Can attempt to build any recipe for any machine, even if it’s not applicable
• Multi-config dependencies are limited in what they can share
Lessons Learned

We learned a few lessons along the way

• Design for debugging. Knowing that a build was targeting Cortex-M vs A without having to go deep into logs and look at compile flags is critical.

• Keep your targets as isolated as possible. Accidental sharing of libc headers or other items can cause very confusing build breaks.

• Don’t assume one compiler can target all available CPU cores. Cross vendor SoCs are starting to look more likely (ARM + RISC-V, for example).

• Try to keep as much in Yocto as possible. Avoid the temptation to build your final flash image via custom scripts, for example.
Conclusion

Check out our examples and published source:

- https://gist.github.com/rfairfax/c5c3a44913213fde35fceac51b660663
- https://gist.github.com/rfairfax/c7a3258b63300372ee130dc5a75833d3
- https://3rdpartysource.microsoft.com/download/Azure%20Sphere/19.07/Core%20OS%20components.zip

Q & A