Advanced Class

Paul Barker, Marco Cavallini, Beth Flanagan, Sean Hudson, Joshua Lock,
Scott Murray, Tim Orling, David Reyna, Rudi Streif, Marek Vasut

Yocto Project Developer Day •
Prague• 26 October 2017

Advanced Class

- **Class Content:**
  - https://wiki.yoctoproject.org/wiki/DevDay_Prague_2017
- **Requirements:**
  - Wireless
  - SSH (Windows: e.g. “putty”)
  - Serial (Linux: “screen”, Windows: e.g. “Teraterm”)
- **Wireless Registration:**
  - TBD

Agenda – The Advanced Class

9:00- 9:15	Opening session, What's New (Jefro)
9:15- 9:30	Account setup (David Reyna)
9:30-10:15	Devtool: creating new content (Tim Orling)
10:15-10:30	Morning Break
10:30-11:15	DT overlays (Marek Vasut)
11:15-12:00	Userspace: packaging, installation, system services (Rudolf Streif, given by David Reyna)
12:00- 1:00	Lunch (and board pass-out)
1:00- 1:45	License audit of a release (Beth 'pidge' Flanagan)
1:45- 2:15	CROPS (Tim Orling)
2:30- 2:45	Afternoon Break
2:45- 3:15	Recipe specific sysroots (Joshua Lock, given by Sean Hudson)
3:15- 3:45	Maintaining your Yocto Distribution (Scott Murray)
3:45- 4:20	Kernel Modules with eSDKs (Marco Cavallini)
4:20- 5:00	Analytics and the Event System (David Reyna)
5:00- 5:30	Kernel/Security Forum, Q and A (All)

Activity One

Class Account Setup

Notes for the Advanced Class:

- **The class will be given with YP-2.4 (Rocko)**
- **Wifi Access:**
  - SSID: <TBD>
  - Password: <TBD>
- **Your account's IP access addresses**
  - SSH (password "devday"):

    `ssh ilab01@devday-a.yocto.io -p 10000(+your session number)`
  - HTTP:
    http://**`devday-a.yocto.io:30000(+your session number)`**

Yocto Project Dev Day Lab Setup

- **The virtual host's resources can be found here:**
  - Your Project: "/scratch/working/build-arm“
  - Extensible-SDK Install: "/scratch/sdk/arm“
  - Sources: "/scratch/src“
  - Poky: "/scratch/poky"
  - Downloads: "/scratch/downloads"
  - Sstate-cache: "/scratch/sstate-cache"
  - QEMU/Toaster Install: "/scratch/build“
- **You will be using SSH to communicate with your virtual server.**
- **You may want to change the default password (“devday”) after you log on, in case someone accidently uses the same account address as yours.**

FYI: How class project was prepared

```
$ cd /scratch
$ git clone -b rocko git://git.yoctoproject.org/poky.git
$ cd poky
$ ./scratch/poky/oe-init-build-env  build
$ echo "MACHINE = \"qemuarm\"" >> conf/local.conf
$ echo "SSTATE_DIR = \"/scratch/sstate-cache\"" >> conf/local.conf
$ echo "DL_DIR = \"/scratch/downloads\"" >> conf/local.conf
$ echo "IMAGE_INSTALL_append = \" gdbserver openssh libstdc++ \
    curl \"" >> conf/local.conf
$ # Capture the build into a Bitbake/Toaster database
$ . toaster start webport=0.0.0.0:8000
$ # Build the project
$ bitbake core-image-base
$ # Prepare eSDK
$ bitbake core-image-base -c populate_sdk_ext
$ cd /scratch/sdk
$ /scratch/poky/build/tmp/deploy/sdk/poky-glibc-*-toolchain-ext-*.sh \
   -d `pwd` -y
$. environment-setup-armv5e-poky-linux-gnueab
$ devtool build-image
```

NOTE: Clean Shells!

- **We are going to do a lot of different exercises in different build projects, each with their own environments.**
- **To keep things sane, you should have a new clean shell for each exercise.**
- **There are two simple ways to do it:**
  1. Close your existing SSH connection and open a new one -- or –
  2. Do a “bash” before each exercise to get a new sub-shell, and “exit” at the end to remove it, in order to return to a pristine state.

Activity Two

Devtool

Tim Orling

Activity Three

DT overlays

Marek Vasut

Device Tree

- **Data structure describing hardware**
- **Tree with cycles**
- **Example:**

```
/dts-v1/;
#include <dt-bindings/interrupt-controller/arm-gic.h>
/ {
        model = "V2F-1XV7 Cortex-A53x2 SMM";
        arm,hbi = <0x247>;
[...]
        chosen {
                stdout-path = "serial0:38400n8";
        };
[...]
        gic: interrupt-controller@2c001000 {
                interrupt-controller;
                reg = <0 0x2c001000 0 0x1000>,
                      <0 0x2c002000 0 0x2000>,
[...]
```

Problem – Variable hardware

- DT started on machines the size of little fridge
  - HW was mostly static
  - DT was baked into PROM, optionally modified by bootloader
- DT was good, so it spread
  - First PPC, embedded PPC, then ARM …
- There always was slightly variable hardware
  - Solved by patching DT in bootloader
  - Solved by carrying multiple DTs
  - Solved by co-operation of board files and DT
  - ^ all that does not scale

Problem – Variable hardware – 201x edition

- Come 201x, variable HW became easy to make:
  - Cheap devkits with hats, lures, capes, …
  - FPGAs and SoC+FPGAs became commonplace …
  - => Combinatorial explosion of possible HW configurations
- Solution retaining developers' sanity
  - Describe only the piece of HW that is being added
  - Combine these descriptions to create a DT for the system
  - Enter DT overlays

Device Tree Overlays

- DT: Data structure describing hardware
- DTO: necessary change(s) to the DT to support particular feature
  - Example: an expansion board, a hardware quirk,...
- Example DTO:

```
/dts-v1/;
/plugin/;
/ {
        #address-cells = <1>;
        #size-cells = <0>;
        fragment@0 {
                reg = <0>;
                target-path = "/";
                __overlay__ {
                        #address-cells = <1>;
                        #size-cells = <0>;
                        hello@0 {
                                compatible = "hello,dto";
                                reg = <0>;
};      };      };      };
```

Advanced DTO example

```
/dts-v1/;
/plugin/;
[...]
        fragment@2 {
                reg = <2>;
                target-path = "/soc/usb@ffb40000";
                __overlay__ {
[...]
                        status = "okay";
                };
        };

        fragment@3 {
                reg = <3>;
                target-path = "/soc/ethernet@ff700000";
                __overlay__ {
[...]
                        status = "okay";
                        phy-mode = "gmii";
                };
        };
```

DTO Hands-on

- Use pre-prepared meta-dto-microdemo layer
- meta-dto-demo contains:
  - Kernel patch with DTO loader with ConfigFS interface
  - Kernel config fragment to enable the DTO and loader
  - Demo module
  - Demo DTO source (hello-dto.dts)
  - core-image-dto-microdemo derivative from
  - core-image-minimal with added DTO examples and DTC

DTO Hands-on 1/2

- **Add meta-dto-demo to bblayers.conf BBLAYERS:**

```
$ ${EDITOR} conf/bblayers.conf
```

- **Rebuild virtual/kernel and core-image-dto-microdemo**

```
$ bitbake -c cleansstate virtual/kernel
$ bitbake core-image-dto-microdemo
```

- **Start the new image in QEMU**

```
$ runqemu qemuarm
```

DTO Hands-on 2/2

- **Compile DTO**

```
$ dtc -I dts -O dtb /lib/firmware/dto/hello-dto.dts > \
      /tmp/hello-dto.dtb
```

- **Load DTO**

```
$ mkdir /sys/kernel/config/device-tree/overlays/mydto
$ cat /tmp/hello-dto.dtb > \
      /sys/kernel/config/device-tree/overlays/mydto/dtbo
```

- **Unload DTO**

```
$ rmdir /sys/kernel/config/device-tree/overlays/mydto
```

DTO encore

- **DTOs can be used to operate SoC+FPGA hardware**
- **Done using FPGA manager in Linux**

```
fragment@0 {
    reg = <0>;
    /* controlling bridge */
    target-path = "/soc/fpgamgr@ff706000/bridge@0";
    __overlay__ {
        #address-cells = <1>;
        #size-cells = <1>;
        area@0 {
            compatible = "fpga-area";
            #address-cells = <2>;
            #size-cells = <1>;
            ranges = <0 0x00000000 0xff200000 0x00080000>;
            firmware-name = "fpga/bitstream.rbf";
                fpga_version@0 {
                    compatible = "vendor,fpgablock-1.0";
                    reg = <0 0x0 0x04>;
                };
```

Activity Four

Userspace: Advanced Topics

Rudi Streif

(given by David Reyna)

What We Are Going To Do

- Most of your development work will likely be developing your own software packages, building them with the Yocto Project and installing them into a root file system built with the Yocto Project.
- Let’s look at some typical tasks beyond creating the base recipe:
  - Customizing Packaging
  - Package Installation Scripts
  - System Services

Activity Setup

- **Initialize the Build Environment (IN A CLEAN SHELL)**
  - cd /scratch/working
  - source ../poky/oe-init-build-env build-userspace
- *Adjust Configuration (DONE FOR YOU)*
  - vi conf/local.conf

```
MACHINE = "qemuarm"
DL_DIR ?= "/scratch/downloads"
SSTATE_DIR ?= "/scratch/sstate-cache"
EXTRA_IMAGE_FEATURES ?= "debug-tweaks dbg-pkgs dev-pkgs package-
management"
```

- *Build (DONE FOR YOU)*
  - bitbake -k core-image-base
- **Test**
  - runqemu qemux86-64 nographic

Activity Setup - Continued

- Create Layer
  - devtool create-workspace meta-uspapps
- Copy Source Files
  - cd ..
  - cp -r /scratch/src/userspace/uspsrc .

Packaging

- Packaging is the process of putting artifacts from the build output into one or more packages for installation by a package management system.
- Packaging is performed by the package management classes:
  - package_rpm – RPM style packages
  - package_deb – Debian style packages
  - package_ipk – IPK package files used by the OPK package manager
- You configure the package management in conf/local.conf:

```
PACKAGE_CLASSES ?= "package_rpm"
```

  - You can add more than one of the package classes.
  - Only the first one will be used to create the root file system.
  - However, this does not install the package manager itself.
- Install the package manager in conf/local.conf:

```
EXTRA_IMAGE_FEATURES ?= "package-management"
```

Package Splitting

- Packaging Splitting is the process of putting artifacts from the build output into different packages.
- Package splitting allows you to select what you need to control the footprint of your root file system.
- Package splitting is controlled by the variables:
  - PACKAGES – list of package names, default:

```
PACKAGES = "${PN}-dbg ${PN}-staticdev ${PN}-dev ${PN}-doc \
   ${PN}-locale ${PACKAGE_BEFORE_PN} ${PN}"
```

  - FILES – list of directories and files that belong into the package:

```
SOLIBS = "*.so.*"
FILES_${PN} = "${bindir}/* ${sbindir}/* ${libexecdir}/* \
   ${libdir}/lib* {SOLIBS} ${sysconfdir} ${sharedstatedir} \
   ${localstatedir} ${base_bindir}/* ${base_sbindir}/* \
   ${base_libdir}/*${SOLIBS} ${base_prefix}/lib/udev/rules.d \
   ${prefix}/lib/udev/rules.d ${datadir}/${BPN}\
   ${libdir}/${BPN}/* ${datadir}/pixmaps \
   ${datadir}/applications ${datadir}/idl ${datadir}/omf \
   ${datadir}/sounds ${libdir}/bonobo/servers"
```

Package Splitting - Continued

- The package classes process the PACKAGES list from left to right, producing the ${PN}-dbg package first and the ${PN} package last.
- The order is important, since a package consumes the files that are associated with it.
- The ${PN} package is pretty much the “kitchen sink”: it consumes all standard leftover artifacts.
- BitBake syntax only allows prepending (+=) or appending (=+) to variables:
  - Prepend PACKAGES – place standard artifacts into different packages
  - Append PACKAGES – place any leftover packages in non-standard installation directories those packages.
- The variable PACKAGE_BEFORE_PN allows you to insert packages right before the ${PN} package is created.

Packaging QA

- The insane class adds plausibility and error checking to the packaging process.
- You can find a list of the checks in the Reference Manual: http://www.yoctoproject.org/docs/2.3/ref-manual/ref-manual.html#ref-classes-insane
- Some of the more common ones:
  - already-stripped – debug symbols already stripped
  - installed-vs-shipped – checks for artifacts that have not been packaged
  - ldflags – checks if LDFLAGS for cross-linking has been passed
  - packages-list – same package has been listed multiple times in PACKAGES
- Sometimes the checks can get into your way…
  - INSANE_SKIP_<packagename> += “<check>”
  - Skips <check> for <packagename>.

Example – The Fibonacci Library

- Source code in /scratch/working/uspsrc/fibonacci-lib
  - Builds static and dynamic libraries to calculate the Fibonacci series and an application to test it.
- Create development environment
  - cd /scratch/working/build-userspace
  - devtool add fibonacci-lib /scratch/working/uspsrc/fibonacci-lib
- Build the recipe
  - bitbake fibonacci-lib
- Add to your image (conf/local.conf):

```
IMAGE_INSTALL_append = " fibonacci"
```

- Build and test image
  - bitbake core-image-minimal
  - runqemu qemux86-64 nographic

Example – The Fibonacci Library (continued)

- Edit the recipe meta-uspapps/recipes/fibonacci-lib/fibonacci-lib.bb and place the fibonacci test application into its own package ${PN}-examples

```
PACKAGE_BEFORE_PN = "${PN}-examples"
FILES_${PN}-examples = "${bindir}/fibonacci"
```

- Add to your image (conf/local.conf):

```
IMAGE_INSTALL_append = " libfibonacci libfibonacci-examples"
```

- Build and test image
  - bitbake core-image-minimal
  - runqemu qemux86-64 nographic

Package Installation Scripts

- Package management systems have the ability to run scripts before and after a package is installed, upgraded, or removed.
- These are typically shell scripts and they can be provided by the recipe using these variables:
  - `pkg_preinst_<packagename>`: Preinstallation script that is run *before the package is installed*.
  - `pkg_postinst_<packagename>`: Postinstallation script that is run *after the package is installed*.
  - `pkg_prerm_<packagename>`: Pre-uninstallation script that is run *before the package is uninstalled*.
  - `pkg_postrm_<packagename>`: Post-uninstallation script that is run *after the package is uninstalled*.

```
pkg_postinst_${PN}() {
#!/bin/sh
# shell commands go here
}
```

Script Skeleton

```
pkg_postinst_${PN}() {
#!/bin/sh
if [ x"$D" = "x" ]; then
   # target execution
else
   # build system execution
fi
}
```

Conditional Execution

Example – Conditionally running ldconfig

- The Fibonacci library installs a dynamic library libfibonacci.so.1.0 on the target system in /usr/lib.
- For ld to be able to locate the library it must be added to the ld cache and its symbolic name (soname) must be linked. That is done by running ldconfig on the target.
- Add a post installation script to the ${PN} package that only runs ldconfig when it is run on the target but not when the build system creates the root file system.

```
pkg_postinst_${PN}() {
#!/bin/sh
if [ x"$D" = "x" ]; then
   # target execution
   ldconfig
   exit 0
else
   # build system execution
   exit 1
fi
}
```

Installation for Packaging

- Makefile Installation

```
INSTALL ?= install
.PHONY: install
Install:
        $(INSTALL) -d $(DESTDIR)/usr/bin
        $(INSTALL) -m 0755 $(TARGET) $(DESTDIR)/usr/bin
```

- Recipe Installation
  - Providing/overriding the do_install task

```
do_install() {
   install -d ${D}${bindir}
   install -m 0755 ${B}/bin/* ${D}{bindir}
}
```

  - The build system defines a series of variables for convenience:

bindir = "/usr/bin"
sbindir = "/usr/sbin"
libdir = "/usr/lib"
libexecdir = "/usr/lib"
sysconfdir = "/etc/"
datadir = "/usr/share"
mandir = "/use/share/man"
includedir = "/usr/include"

Debugging Packaging

- Check the packaging logfiles in ${WORKDIR}/temp
- Check installation of artifacts in ${WORKDIR}/image
  - The do_install task installs the artifacts into this directory.
  - If artifacts are missing they are packaged.
- Check packaging artifacts in ${WORKDIR}/package
  - This where the artifacts are staged for packaging, including the ones created for the debug packages.
- Check package splitting in ${WORKDIR}/packages-split
  - Packages and their content are staged here by package name before they are wrapped by the package manager.
  - Allows you to verify if the artifacts have indeed been placed into the correct package.
- Check created packages in ${WORKDIR}/deploy-<pkgmgr>

Package Architecture

- The build system distinguishes packages by their hardware dependencies into three main categories:
  - Tune – Generic CPU architecture such as core2_32, corei7, armv7, etc. This is the default and typically appropriate for userspace packages.
  - Machine – Specific machine architecture. Appropriate for packages that require particular hardware features of a machine. Typically applicable to kernel, drivers, and bootloader.
  - All – Package applies to all architectures such as shell scripts, managed runtime code (Python, Lua, Java, …), configuration files, etc.
- Package architecture is controlled by the PACKAGE_ARCH variable:
  - Tune (default) – PACKAGE_ARCH = "${TUNE_PKGARCH}"
  - Machine – PACKAGE_ARCH = "${MACHINE_ARCH}"
  - All – inherit allarch
- Note: Package architecture does not simply determine into what category a package is placed but determines compiler and linker flags and other build options.

System Services

- If your software package is a system service that eventually needs to be started when the system boots you need to add the scripts and service files.
- **SysVInit**
  - Inherit `update-rc.d` class.
  - INITSCRIPT_PACKAGES - List of packages that contain the init scripts for this software package. This variable is optional and defaults to INITSCRIPT_PACKAGES = "${PN}".
  - INITSCRIPT_NAME - The name of the init script.
  - INITSCRIPT_PARAMS - The parameters passed to update-rc.d. This can be a string such as "defaults 80 20" to start the service when entering run levels 2, 3, 4, and 5 and stop it from entering run levels 0, 1, and 6.
- **Systemd**
  - Inherit `systemd` class.
  - SYSTEMD_PACKAGES - List of packages that contain the systemd service files for the software package. This variable is optional and defaults to SYSTEMD_PACKAGES = "${PN}".
  - SYSTEMD_SERVICE - The name of the service file.

Example – The Fibonacci Server

- Source code in /scratch/working/uspsrc/fibonacci-srv
  - Builds a TCP socket server listening on port 9999 for the number of terms and responds with the list of Fibonacci terms.
- Create development environment
  - cd /scratch/working/build-userspace
  - devtool add fibonacci-srv /scratch/working/uspsrc/fibonacci-srv
- Add system service startup to the recipe meta-uspapps/recipes/fibonacci-srv/fibonacci-srv.bb

```
inherit update-rc.d systemd
INITSCRIPT_NAME = "fibonacci-srv"
INITSCRIPT_PARAMS = "start 99 3 5 . stop 20 0 1 2 6 ."
SYSTEMD_SERVICE = "fibonacci-srv.service"
```

- Build the recipe
  - bitbake fibonacci-lib
- Add to your image (conf/local.conf):

```
IMAGE_INSTALL_append = " fibonacci-srv"
```

- Build and test image
  - bitbake core-image-minimal
  - runqemu qemux86-64 nographic
  - nc localhost 9999

Changing the System Manager

- SysVInit is the default system manager for the Poky distribution.
- To use systemd add it to your conf/local.conf file, or better, to your distribution configuration:

```
DISTRO_FEATURES_append = " systemd"

VIRTUAL-RUNTIME_init_manager = "systemd"
```

- If you exclusively want to use systemd, you can remove SysVInit from you root file system image with:

```
DISTRO_FEATURES_BACKFILL_CONSIDERED = "sysvinit"

VIRTUAL-RUNTIME_initscripts = ""
```

Activity Five

License audit of a release

Beth 'pidge' Flanagan, Paul Barker

Activity Six

CROPS

Tim Orling

Activity Seven

Recipe Specific Sysroots

Joshua Lock

(given by Sean Hudson)

Recipe Specific Sysroots - Overview

Topics

- **Definitions**
- Determinism improvements in YP 2.3 +
- Future reproducibility work

Recipe Specific Sysroots

- **Reproducible**
  - **Repeatable**: rerun a build and have it succeed (or fail) in the same way
  - **Deterministic**: given the same inputs the build system should produce equivalent outputs
  - **Binary reproducible**: given the same inputs the system should produce bit-for-bit identical outputs

Reproducibility and Yocto Project

- **Repeatability** was a founding goal of the Yocto Project
  - Not as common place at the time of the project's inception
- **Determinism** of the YP build system has improved over time
  - Vast leap forward with most recent, Pyro, release
- Being able to build **binary reproducible** artefacts is a goal for future development
  - Some concrete tasks planned for 2.4

Recipe Specific Sysroots

Binary Reproducible

- **Fully deterministic build system, producing bit-for-bit identical output given the same inputs**
- **Build environment is recorded or pre-defined**
- **Mechanism for users to:**
  - Recreate the environment
  - Repeat the build
  - Verify the output matches

https://reproducible-builds.org/

Recipe Specific Sysroots

Yocto Project Reproducibility Features

- DL_DIR – shareable cache of downloads
- Easily replicated build environment - configuration in known locations, printed build header
- Shared state mechanism – reusable intermediary objects when inputs haven't changed
- SSTATE_MIRRORS – remotely addressable cache of
- Uninative – static libc implementation for use with native tools, improves sstate reuse
- Fixed locale – ensures consistent date/time format, sort order, etc

Recipe Specific Sysroots

Topics

- Definitions
- **Determinism improvements in YP 2.3 +**
- Future reproducibility work

Recipe Specific Sysroots

Shared sysroots – a long-standing source of non-determinism

- Shared sysroot used by YP build system until 2.3/Pyro release
- Cause of non-determinism, particularly with long-lived workspaces
  - automatic detection of items in the sysroot which weren't explicitly marked as a dependency
  - items which appear lower in common YP build graphs such as libc, kernel or common native dependencies such as glib-2.0-native

Recipe Specific Sysroots

Recipe specific sysroots improve determinism

- **per-recipe sysroot** which only includes sysroot components of explicit dependencies
- sysroot artefacts are installed into a **component specific location**
- built by hard-linking dependencies files in from their component trees
- reinstall sysroot content when the task checksum of the dependency changes
- resolves the issue of autodetected dependencies and implicit dependencies through build order

Recipe Specific Sysroots

Implementations challenges

- Artefacts in the component sysroots can include hard-coded paths – we need to be able to fix them for installed location
  - The code knows to look at certain common sites for hard-coded paths and can be taught to fixup in more locations by appending to the EXTRA_STAGING_FIXMES variable
- A recipe is composed of several tasks to run in the course of building its output; fetch, unpack, configure, etc.
  - Many of these tasks have task-specific dependencies, we need to re-extend the sysroot when tasks explicitly require items in the sysroot. i.e. do_package_write_deb need dpkg-native do_fetch for a git repo requires git-native

Recipe Specific Sysroots

Implementations challenges (II)

- post-install scriptlets need to be executed for each recipe-specific sysroot
  - We handle this by installing postinst scriptlets into the recipe-specifc sysroot with a postinst- prefix and running all of the scriptlets as part of the sysroot setup
- Still need to be able to replicate old shared-sysroot behaviour in certain scenarios, i.e. eSDK
  - bitbake build-sysroot recipe target takes everything in the components directory which matches the current MACHINE and installs it into a shared sysroot

Recipe Specific Sysroots

Adapting to recipe specific sysroots

Would have liked to be pain-free transition, but there is some conversion required for recipe-specific sysroots.

- fix missing dependencies – commonly native dependencies, i.e. glib-2.0-native
- SSTATEPOSTINSTFUNCS → SYSROOT_PREPROCESS_FUNCS
  - SSTATEPOSTINSTFUNCS are a hook to call specific functions after a recipe is populated from shared state, commonly used for fixing up paths.
  - As shared state objects will now be installed into the recipe-component location, then linked into the recipe specific sysroot, we need to be able to perform such fixes in each constructed sysroot.
  - SYSROOT_PREPROCESS_FUNCS: is list of functions to run after sysroot contents are staged and the right place to perform relocation in RSS world

Recipe Specific Sysroots

Adapting to recipe specific sysroots (II)

- **Add PACKAGE_WRITE_DEPS for any postinsts requiring native tools at rootfs construction**
  - YP build system tries to run preinst and postinsts at rootfs construction time, deferring any which fail to first boot.
  - Any special native tool dependencies of pkg_preinst and pkg_postinst must be explicitly listed in PACKAGE_WRITE_DEPS to ensure they are available on the build host at rootfs construction time.

Recipe Specific Sysroots

Unexpected consequences

- Recipe specific sysroots aggravated an existing source of non-determinism
- PATH included locations in the host for boot-strapping purposes
- Host tools were being used, where available, when native dependencies were missing

Recipe Specific Sysroots

Resolved with PATH filtering

- All required host utilities must be explicitly listed
- These are all symlinked into a directory
- PATH is then cleared and set to this filtered location
  - HOSTTOOLS: being unavailable causes an early failure (when they can't be linked in place)
  - HOSTTOOLS_NONFATAL: aren't a build failure when absent, i.e. optional tools like ccache or proxy helpers

Recipe Specific Sysroots - Overview

Topics

- Definitions
- Determinism improvements in YP 2.3 +
- **Future reproducibility work**

Recipe Specific Sysroots

Improved build system determinism

Next set our sights on the next level reproducible definition: binary reproducible builds.

Common issues that affect binary reproducibility include:

- Compressing files with different levels of parallelism
- Dates, times, and paths embedded in built artefacts
- Timestamps of outputs changing

Recipe Specific Sysroots

Future reproducibility work

- Layer fetcher/Workspace setup tool – to improve ease of build environment replication
- SOURCE_DATE_EPOCH – open spec to ensure consistent date/time stamps in generated artefacts
- strip-nondeterminism – post-processing step to forcibly remove traces of non-determinism
- etc...

Activity Seven

Maintaining Your Yocto Project Based Distribution

Scott Murray

Activity Nine

Kernel Modules with eSDKs

Marco Cavallini

Kernel modules with eSDKs – Overview

- **The Extensible SDK (eSDK) is a portable and standalone development environment , basically an SDK with an added bitbake executive via devtool.**
- **The "devtool" is a collection of tools to help development, in particular user space development.**
- **We can use devtool to manage a new kernel module:**
- Like normal applications is possible to importa and create a wrapper recipe to manage the kernel module with eSDKs.

Kernel modules with eSDKs – Compiling a kernel module

- **We have two choices**
- **Out of the kernel tree**

–When the code is in a different directory outside of the kernel source tree

- **Inside the kernel tree**

–When the code is managed by a KConfig and a Makefile into a kernel directory

Kernel modules with eSDKs – Pro and Cons of a module outside the kernel tree

- **Advantages**
  - Might be easier to handle than modify it into the kernel itself
  - Not integrated into the configuration/compilation process
  - Needs to be built separately
  - The driver cannot be built statically
- **Drawbacks**
  - Not integrated to the kernel configuration/compilation process
  - Needs to be built separately
  - The driver cannot be built statically

Kernel modules with eSDKs – Pro and Cons of a module inside the kernel tree

- **Advantages**
  - Well integrated into the kernel configuration and compilation process
  - The driver can be built statically if needed
- **Drawbacks**
  - Bigger kernel size
  - Slower boot time

Kernel modules with eSDKs – The source code

```
#include <linux/module.h>
#include <linux/kernel.h>

static int __init hello_init(void)
{
    printk("When half way through the journey of our life\n");
    return 0;
}

static void __exit hello_exit(void)
{
    printk("I found that I was in a gloomy wood\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Greeting module from the Divine Comedy");
MODULE_AUTHOR("Dante Alighieri");
```

Kernel modules with eSDKs – The Makefile

```
obj-m += hellokernel.o
SRC := $(shell pwd)

all:
 $(MAKE) -C $(KERNEL_SRC) M=$(SRC) modules

modules_install:
 $(MAKE) -C $(KERNEL_SRC) M=$(SRC) modules_install
```

Kernel modules with eSDKs – Devtool setup

- **Start a new Shell!** Otherwise, the existing bitbake environment can cause unexpected results

- Here is how the eSDK was prepared (and you can too!):

```
$ cd /scratch/sdk
$ /scratch/poky/build/tmp/deploy/sdk/poky-glibc-*-toolchain-ext-*.sh \
   -d `pwd` -y
$. environment-setup-armv5e-poky-linux-gnueab
$ bash
$ devtool build-image
```

Kernel module with eSDKs – Hooking a new module into the build

- **Run the devtool 'add recipe-name /path/to/source'**

```
$ devtool add --version 1.0 simplestmodule \
        /scratch/src/simplestmodule
```

- This generates a minimal recipe in the workspace layer
- This adds EXTERNALSRC in an workspace/appends/simplestmodule_1.0.bbappend file that points to the source
- In other words, the source tree stays where it is, devtool just creates a wrapper recipe that points to it
- ***Note: this does not add your image to the original build engineer's image, which requires changing the platform project's conf/local.conf***

After the add

Workspace layer layout

```
qemu> tree workspace
.
├── appends
│   └── xxxxxxxxxxxx_1.0.bbappend
├── conf
│   └── layer.conf
├── README
└── recipes
    └── bballs
        └── bballs_1.0.bb

4 directories, 4 files
```

Kernel module with eSDKs – Build the module

- Build the new recipe

```
$ devtool build simplestmodule
```

*This will create the **simplestmodule.ko** kernel module*

Kernel module with eSDKs - build/deploy/run app

- **In a separate shell, source the eSDK environment and run qemu (this will be your target terminal)**

```
<open new clean shell>
$ cd /scratch/sdk/qemuarm
$ . environment-setup-armv5e-poky-linux-gnueabi
$ devtool runqemu nographic
```

- ***Get the target's IP address from the target serial console***

```
root@qemuarm:~#4:~# ifconfig
```

- **In the eSDK shell, deploy the output (*the target's ip address may change)***

```
$ devtool deploy-target -s simplestmodule root@<target_ip>
```

NOTE: the '-s' option will note any ssh keygen issues, allowing you to (for example) remove/add this IP address to the known hosts table

- **In the target shell, load the module, and observe the results**

```
# insmod /home/where????
```

Kernel module with eSDKs – Hooking a new module into the build

- **Run the devtool 'add recipe-name /path/to/source'**

```
$ devtool modify virtual/kernel
```

- This takes almost 1 hour fetching
- This adds EXTERNALSRC in an workspace/appends/simplestmodule_1.0.bbappend file that points to the source
- In other words, the source tree stays where it is, devtool just creates a wrapper recipe that points to it

Activity Ten

Analytics and the Event System

David Reyna

Introduction

- **Thesis:**
  - The bitbake event system, together with the event database that comes with Toaster, can be used to generate and provide access to analytical data and provide a new unique toolset to solve difficult problems
- **What we will cover today:**
  - The problem space for extracting and analyzing data
  - Introduce the bitbake event system, interfaces
  - Event Examples: Toaster, CLI tools, customized bitbake UI
  - Resources
- **The full presentation can be found here:**
  - http://events.linuxfoundation.org/sites/events/files/slides/BitbakeAnalytics_ELC_Portland.pdf
- **What that presentation additionally covers:**
  - Deep dive on the event system code and components
  - Event database, database schema, custom events, custom tools, use cases, gotchas

The Problem Space (as I see it)

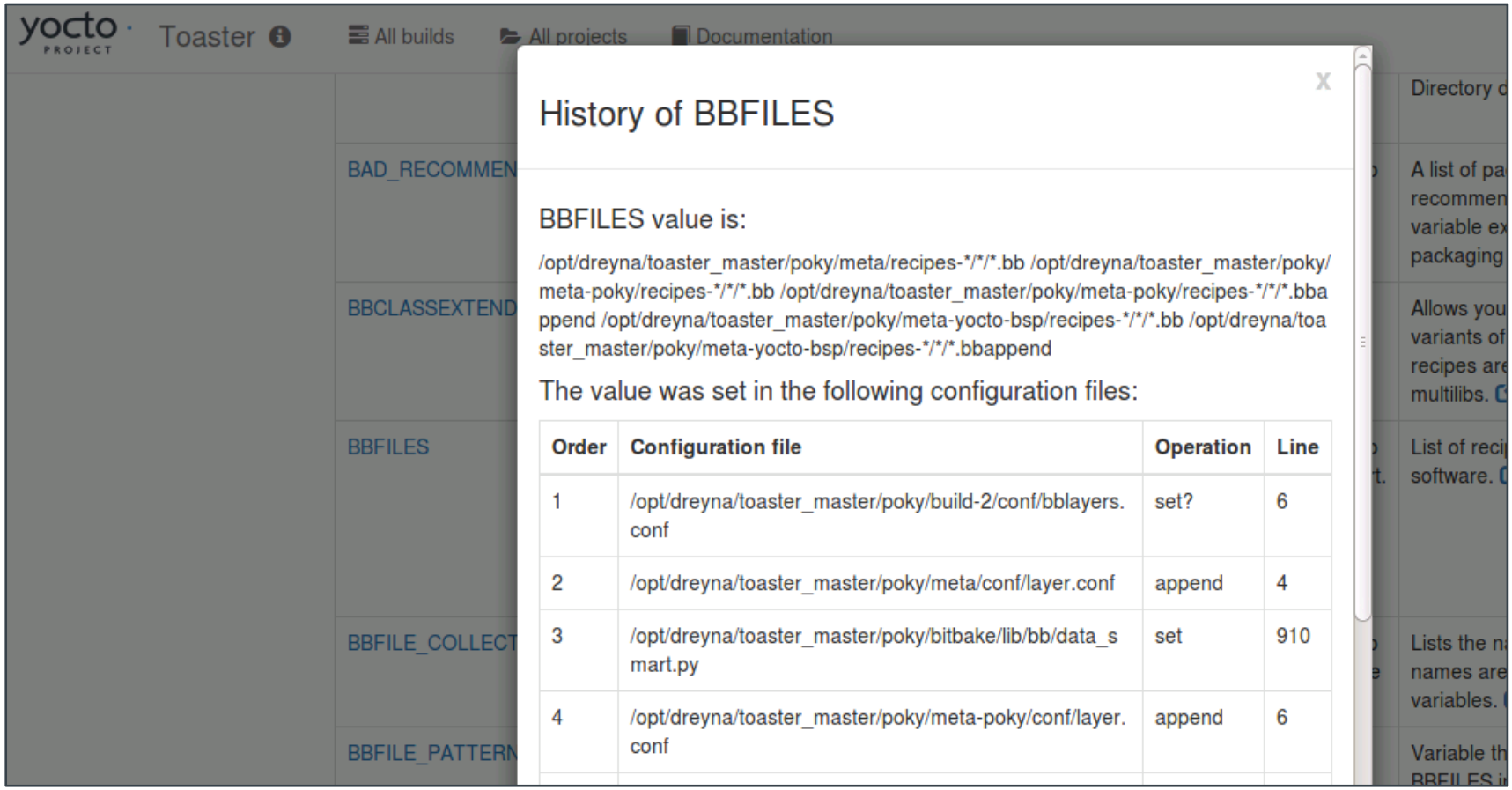
- **Types of addressable problems with analytics:**
  - Issues with time or coincident sensitivity
  - Issues with transient data values
  - Issues with transient UFOs (Unidentified Failing Objects)
  - Issues with trends (size, time, cache misses, scaling)
  - If the problem is a needle, where is the haystack to look in
- **We need:**
  - Easy access to deep data, time, and ordering
  - Reliable interaction with bitbake
  - Easy access to the data with tools, both provided and custom
  - Ability to acquire long term data, from a day to many months
  - Keep bitbake as pristine as possible
- **My builds are working, do I need this?**
  - Excellent, you are in good shape! However, if they stop working or when you do new work or try to scale, here are additional tools for your toolbox

The Problem Space (2)

- **Well known and documented data from bitbake builds:**
  - Logs (Build/Error logs)
  - Artifacts (Kernel, Images, SDKs)
  - Manifests (Image content, Licenses)
  - Variables (bitbake -e)
  - Dependencies
- **However…**
  - These only capture the final results of the build, not how the build progressed nor the intermediate or analytical information.
  - It is hard for example to correlate logs with other logs, let alone with other builds
- **The Answer!**
  - The bitbake event system
  - The bitbake event database
  - Events are easy to create, fire, listen to, and catch
  - There are more than 40 existing event types
  - Uses IPC over python xmlrpc sockets, with automatic data marshalling

Toaster Analytics – Intermediate Data Example

- **The Toaster database/GUI can for example display the intermediate values of bitbake variables, specifically each variable's modification history down to the file and line.**



Overview of Available Events

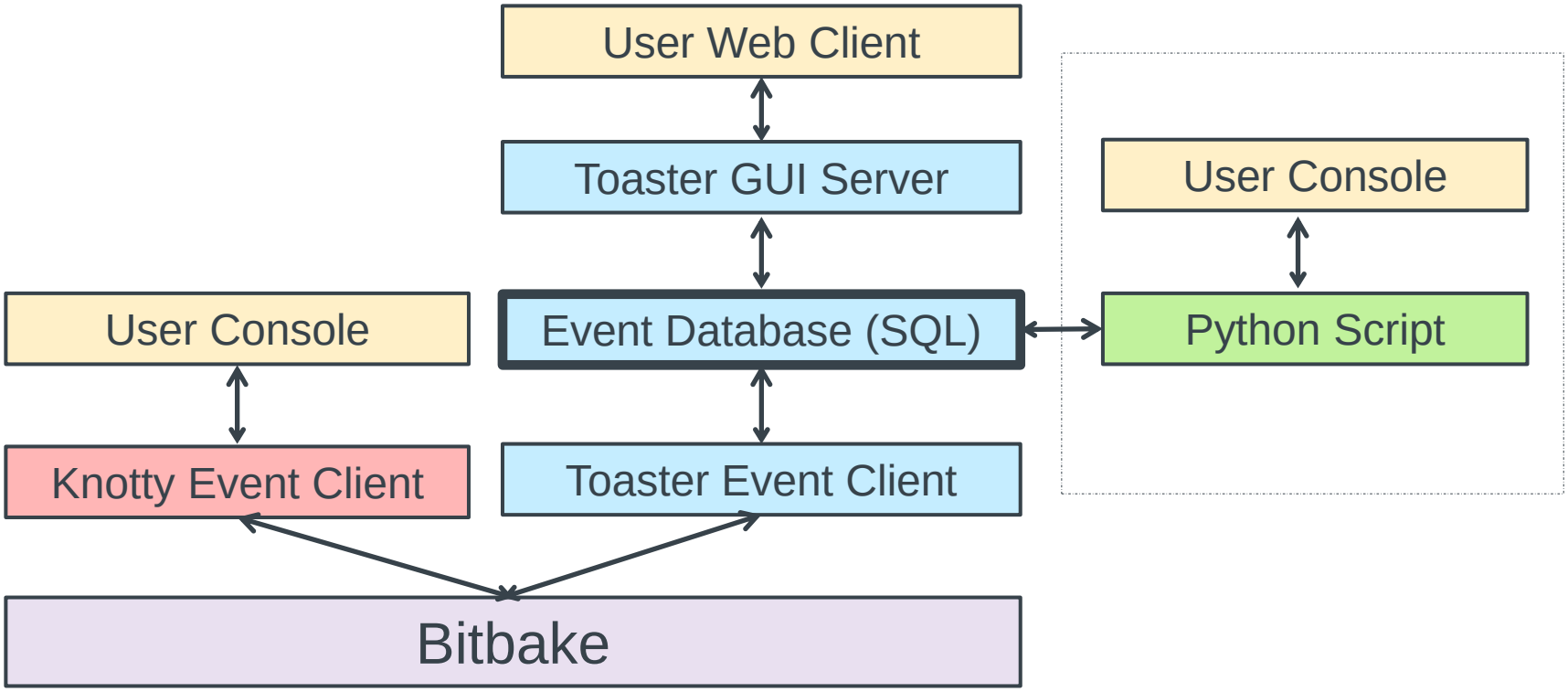
- BuildInit|BuildCompleted|BuildStarted
- ConfigParsed|RecipeParsed
- ParseCompleted|ParseProgress|ParseStarted
- MultipleProviders|NoProvider
- runQueueTaskCompleted|runQueueTaskFailed|runQueueTaskSkipped|
  runQueueTaskStarted
- TaskBase|TaskFailed|TaskFailedSilent|TaskStarted|
  TaskSucceeded
- sceneQueueTaskCompleted|sceneQueueTaskFailed|sceneQueueTaskStarted
- CacheLoadCompleted|CacheLoadProgress|CacheLoadStarted
- TreeDataPreparationStarted|TreeDataPreparationCompleted
- DepTreeGenerated|SanityCheck|SanityCheckPassed
- MetadataEvent
- LogExecTTY|LogRecord
- CommandCompleted|CommandExit|CommandFailed
- CookerExit

Event Clients (you are already an event user!)

- **Bitbake actually runs in a separate context, and expects an event client (called a “UI") to display bitbake's status and output**
- **Here are the existing bitbake event clients:**
  - **Knotty**: this is the default bitbake command line user interface that you know and love. It uses events to display the famous dynamic task list, and to show the various progress bars
  - **Toaster**: this is the bitbake GUI, which provides both a full event database and a full feature web interface. We will be using this as our primary example since it contains the most extensive implementation and support for events
  - **Depexp**: this executes a bitbake command to extract dependency data events, and then uses a GTK user interface to interact with it
  - **Ncurses**: this provides a simple ncurses-based terminal UI

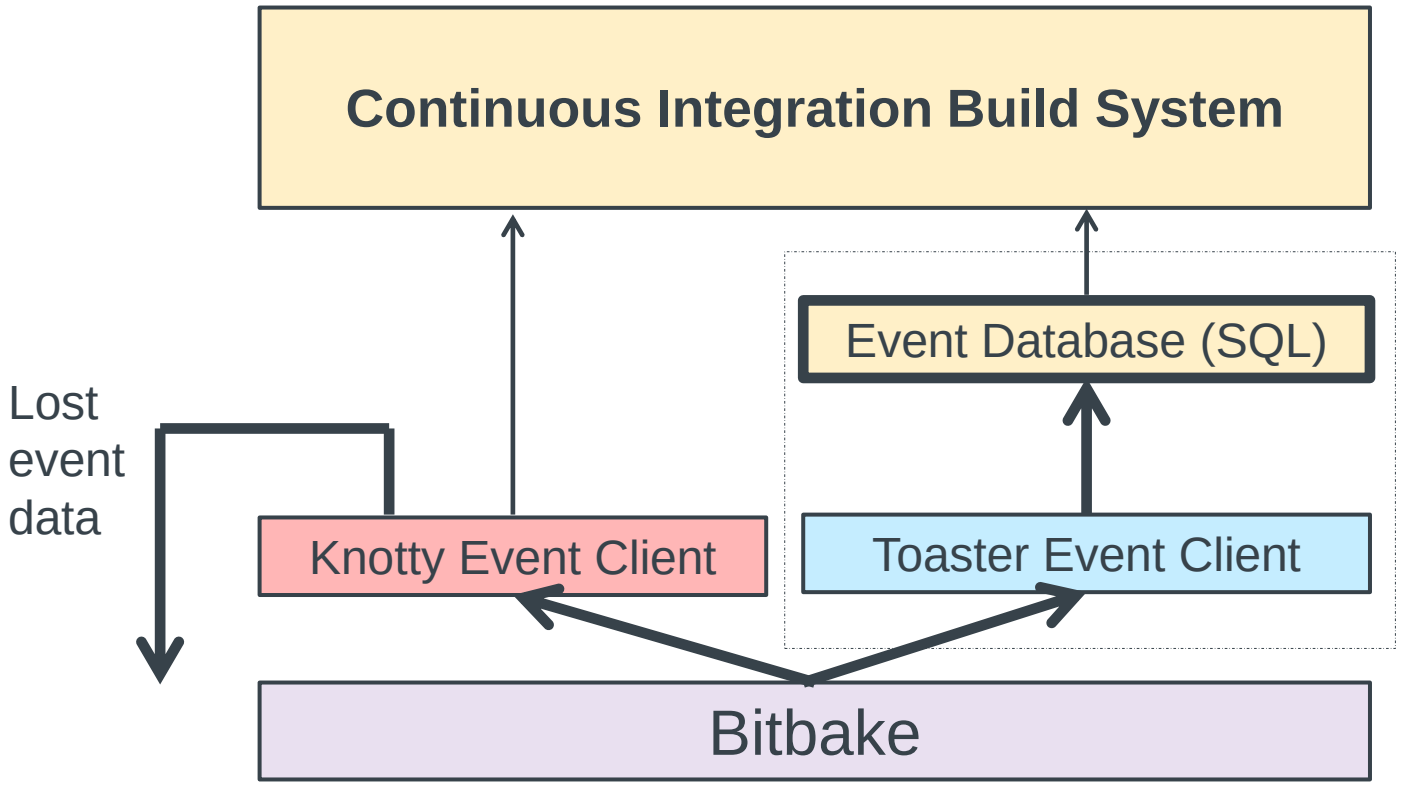
Event Database

- The event database is built into Toaster to maintain persistent build data
- It can however just as easily be used directly with command line scripts or other SQL compatible tools



Example Event Database with CI Builders

- **If you enable the Toaster UI in a CI system, you can additionally get the event artifacts together with your build artifacts (you will definitely need to select a production level database)**



Adding Build Data to the Event Database

- **There are two easy ways to get build data into the event database**
  - Create and execute your builds from within the Toaster GUI

```
$ cd /scratch
$ source poky/oe-init-build-env
$ source toaster start webport=0.0.0.0:8000    # local only: “127.0.0.1:8000”
$ firefox localhost:8000    # here, connect browser from your using Toaster URL
```

  - Start Toaster, and run your command line builds in that environment

```
$ . Poky/oe-init-build-env
$ source toaster start webport=0.0.0.0:8000
$ bitbake <whatever>
```

- **The ‘source toaster’ performs these tasks**
  - Creates the event database if not present, applies any schema updates
  - Starts the web client (this can be ignored for command line usage)
  - Sets the command line environment to use Toaster as the UI for bitbake (“BITBAKE_UI”)

Example 1: Custom command line analytic tool

Minimal Event Database Python Script

- Accessing the data in the event database is very simple. In this example we will print the data from the first-most Build record, and also look up and print the associated Target record

```
$ cat sample_toaster_db_read.py
#!/usr/bin/env python3

import sqlite3
conn = sqlite3.connect('toaster.sqlite')
c = conn.cursor()

c.execute("SELECT * FROM orm_build")
build=c.fetchone()
print('Build=%s' % str(build))

c.execute("SELECT * FROM orm_target where build_id = '%s'" % build[0])
print('Target=%s' % str(c.fetchone()))
$
$ ./sample_toaster_db_read.py
Build=(1, 'qemux86-64', 'poky', '2.2.1', '2017-02-12 23:55:52.137355', \
'2017-02-13 00:16:30.794711', 0, '/…/build_20170212_235552.805.log', \
'1.32.0', 1, 1478, 1478, '20170212235604')
Target=(1, 'core-image-base', '', 1, 0, '/…/license.manifest', 1, \
'/…/core-image-base-qemux86-64-20170212235604.rootfs.manifest')$
```

Full Feature Event Database Python Script

- **In this section we will work with an example python application that extracts and analyzes event data**
- **Specifically, we will attempt to investigate the question:**

  "*How exactly do the tasks of a build overlap execution with other tasks, and on a higher level how to recipes overlap execution with other recipes, plus what data can extract around this question*"
- **While this may not be a deep problem, and there are certainly OE tools that already provide similar information (e.g. pybootchart), the point is that (a) this was very easy and fast to write, and (b) you can now fully customize the analysis and output to your needs and desires.**

Task and Recipe Build Analysis Script

- Here is the list of available commands and features

```
$ more event_overlap.py # see db setup and schema info
$ ./event_overlap.py --help
Commands: ?
 ?                            : show help
 b,build   [build_id]         : show or select builds
 d,data                       : show histogram data
 t,task    [task]             : show task database
 r,recipe  [recipe]           : show recipes database
 e,events  [task]             : show task time events
 E,Events  [recipe]           : show recipe time events
 o,overlap [task|0|n]         : show task|zero|n_max execution overlaps
 O,Overlap [recipe|0|n]       : show recipe|zero|n_max execution overlaps
 g,graph   [task]   [> file]  : graph task execution overlap
 G,Graph   [recipe] [> file]  : graph recipe execution overlap
 h,html    [task]   [> file]  : HTML graph task execution overlap [to file]
 H,Html    [recipe] [> file]  : HTML graph recipe execution overlap [to file]
 q,quit                       : quit

Examples:
  * Recipe/task filters accept wild cards, like 'native-*, '*-lib*'
  * Recipe/task filters get an automatic wild card at the end
  * Task names are in the form 'recipe:task', so 'acl*patch'
    will specifically match the 'acl*:do_patch' task
  * Use 'o 2' for the tasks in the two highest overlap count sets
  * Use 'O 0' for the recipes with zero overlaps
```

Histogram of Parallel Task/Recipe Execution ('d')

```
Commands: d
Histogram:For each task, max number of tasks executing in parallel
       0   1   2   3   4   5   6   7   8   9
    ----------------------------------------
  0)    0 621  16  22  50  49  56  83  94  45
 10)   57  82  87  81  47  56  58  62  64  88
 20)  121 182 268 221 148

Histogram:For each recipe's task set, max number of recipes executing in parallel
       0   1   2   3   4   5   6   7   8   9
    ----------------------------------------
  0)    0   5   1   1   1   1   1   1   3   3
 10)    1   2   2   2   2   1   1   3   1   6
 20)    1   1   2   1   1   2   2   1   1   1
 30)    1   1   2   2   1   3   1   2   2   1
 40)    1   1   1   1   1   1   1   1   3   1
 50)    1   2   4   2   2   1   1   1   1   2
 60)    1   2   1   1   1   2   1   1   1   2
 70)    1   1   2   2   2   2   1   3   3   1
 80)    3   2   1   1   1  10   7   8   8   8
 90)    7   7   2   2   3   2   2   1   1   2
100)    2   1   1   1   2   2   1   3   2   3
110)    2   1   2   1   1   1   1   2   1   1
120)    2   1   1   2   1   1   1   2   1   2
130)    1   1   1   1
```

Histogram of Overlapping Task/Recipe Execution

```
…
Histogram:For each task, max number of tasks that overlap its build
       0   1   2   3   4   5   6   7   8   9
    ----------------------------------------
  0) 614   9  10  29  28  42  46  55  51  47
10)  56  52  48  59  28  33  63  29  43  60
20)  60  94 119 223 105  95  53  57  36  40
30)  20  26  15  17  13   8  11   9   9   2
40)   7  10   9   7   3   6   6   3   6   6
50)   6   6   6   2   2   5   3   1   3   1
60)   4   2   5   1   2   2   1   2   3   5
... (sparse) ...
980)   0   0   1

Histogram:For each recipe's task set, max number of recipes that overlap its build
       0   1   2   3   4   5   6   7   8   9
    ----------------------------------------
  0)  67   0   0   0   0   0   0   0   0   0
 10)   0   0   0   0   0   0   0   0   0   0
... (all zeros) ...
 80)   0   0   0   0   5   1   1   8   4   1
 90)   3   2   0   1   4   4   3   0   0   0
100)   0   0   0   0   0   0   2   1   0   0
110)   2   0   1   2   0   0   3   0   1   2
120)   0   0   0   0   0   0   0   0   2   0
130)   0  26   8   5   2   6   5   0   0   1
... (sparse) ...
170)   0   4   0   0   0   0   0   0   0   0
180)   0   0   0   0   0   0  69
```

Initial Results

- Here are some initial results when examining a “core-image-minimal” project with Task Count=2658 and Recipe Count=254
- We have as many as 148 tasks being able to run with all 24 available threads used
- There were 621 tasks that ran solo
- There were zero recipes that ran solo
- There was one task “linux-yocto:do_fetch” whose execution overlapped with 983 other tasks; the second most overlap was “python3-native:do_configure” with an overlap count of 798
- There were 69 recipes that overlaps with 186 other recipes, with the next highest overlap being 4 recipes that overlap with 171 other recipes
- The below sample HTML output page on task overlaps shows the amount of information available, with the recipe page too large to show in this context

Initial Results

- **Let us see the available builds:**

```
Command: b
List of available builds:
  BuildId=1) CompletedOn=2017-02-13 00:16:30.794711, Outcome=SUCCEEDED,
     Project=Command line builds, Target=core-image-base, Task=''
  BuildId=2) CompletedOn=2017-02-13 00:46:40.724932, Outcome=FAILED,
     Project=Command line builds, Target=core-image-base, Task=populate_sdk_ext
  BuildId=3) CompletedOn=2017-02-13 00:46:26.513568, Outcome=SUCCEEDED,
     Project=Command line builds, Target=core-image-base, Task=''
  BuildId=4) CompletedOn=2017-02-23 09:02:31.109727, Outcome=SUCCEEDED,
     Project=Command line builds, Target=quilt-native, Task=''
```

- **Select the minimal build #4**

```
Command: b 4
Fetching build #4
Build: CompletedOn=2017-02-23 09:02:31.109727, Outcome=SUCCEEDED,
   Project='Command line builds‘ Target='quilt-native', Task='', Machine='qemux86-64'
Success: build #4, Task Count=9, Recipe Count=1
```

- **Run the commands `d,t,r,o,O,g,G` to get a sense of the minimal outputs**

Initial Results

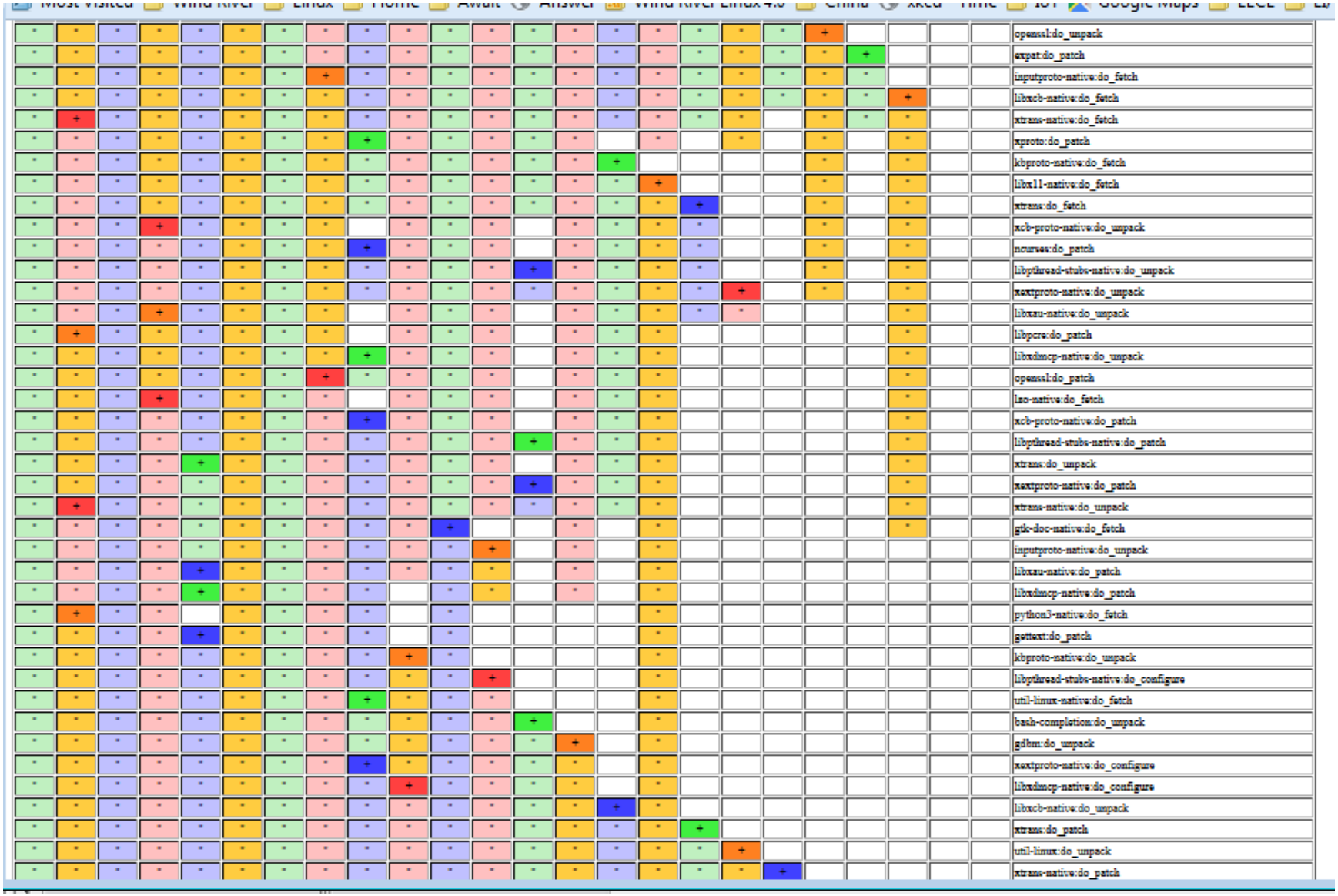
- **Now select the large build (#1) and explore. We shall use the recipe filter ‘zlib’ to limit the output:**

```
Command: b 1
Command: o zlib
Command: e zlib
Command: t zlib
Command: r zlib
Command: o zlib
Command: O 0
```

- **Make sure your window is very wide, and then run this command to see a graph of the task overlaps for zlib:**

```
Command: g zlib
```

Sample HTML Output of Task Overlap



Example 2: Custom Event Interface (knice)

Custom Event UI

- If the knotty UI is too simple (since it does not collect data) and the Toaster UI too large for your analytic needs, you can make your own bitbake UI and have it handle specific events as you need. Here is a simple tutorial on how to do that.
- What we will do is start with the “knotty” UI, and then customize it as the “knice” UI.

```
$ pushd ../bitbake/lib/bb/ui
$ cp knotty.py knice.py
$ sed -i -e "s/notty/nice/g" knice.py
$ vi knice.py
```

- We make a simple change:

```
-print("Nothing to do.  Use 'bitbake world' to build everything, \
    or run 'bitbake --help' for usage information.")
+print("NICE: Nothing to do.  Use 'bitbake world' to build everything, \
    or run 'bitbake --help' for usage information.")
```

- Now we run it:

```
$ popd
$ bitbake -u knice
NICE: Nothing to do.  Use 'bitbake world' to build everything, or run
'bitbake --help' for usage information.
```

Custom Event UI (2)

- Now let us instrument an event by updating “knice.py”.
- First, let us add "bb.event.DepTreeGenerated“ to the event list

```
$ vi ../bitbake/lib/bb/ui/knice.py

-     "bb.event.ProcessFinished"]
+     "bb.event.ProcessFinished","bb.event.DepTreeGenerated"]
```

- Now let us add a print statement to the otherwise empty "bb.event.DepTreeGenerated“ handler code

```
      if isinstance(event, bb.event.DepTreeGenerated):
+         logger.info("NICE: bb.event.DepTreeGenerated received!")
          continue
```

- Now we run it and see our code run!

```
[build]$ bitbake -u knice quilt-native [ | grep NICE ]
...
NOTE: NICE: bb.event.DepTreeGenerated received!          | ETA:  0:00:00
...
```

Resources

- **Source code and example event database**
  - This is available as part of the Yocto Project Developer Day Advanced Class (see https://www.yoctoproject.org/yocto-project-dev-day-north-america-2017, and https://wiki.yoctoproject.org/wiki/DevDay_US_2017)

- **Here is the Toaster documentation, and Youtube video!**
  - http://www.yoctoproject.org/docs/latest/toaster-manual/toaster-manual.html#toaster-manual-start
  - https://youtu.be/BlXdOYLgPxA

- **Basic information about bitbake UI's**
  - http://elinux.org/Bitbake_Cheat_Sheet

- **Here is design information on the event model for Toaster**
  - https://wiki.yoctoproject.org/wiki/Event_information_model_for_Toaster

- **Here is the original design information on Toaster and bitbake communication**
  - https://wiki.yoctoproject.org/wiki/Toaster_and_bitbake_communications

Bonus Example 3: Custom event types

Custom events

- Normally, for a custom event you merely sub-class the event class or some other existing class, and add your new content
- In this example, we show how we can easily extend "MetadataEvent" and use it on the fly, since the sub-event 'type' is an arbitrary string and the data load is a simple dictionary.
- Event creation:

```
my_event_data = {
  "TOOLCHAIN_OUTPUTNAME": d.getVar("TOOLCHAIN_OUTPUTNAME")
}
bb.event.fire(bb.event.MetadataEvent("MyMetaEvent", my_event_data), d)
```

- Event handler:

```
if isinstance(event, bb.event.MetadataEvent):
    if event.type == "MyMetaEvent":
        my_toochain = event.data["TOOLCHAIN_OUTPUTNAME"]
```

Bonus Example 4: Debugging coincident data in bitbake

Using Events for debugging bitbake

- **You can also use the event system in debugging bitbake or your classes.**
- **Example 1: The quintessential example is to use “logger.info()” to insert print statements into the code. This is implemented as an event, meaning that will it be passed to the correct external UI and not lost in some random log file.**
- **Example 2: The ESDK file used to be copied to the build’s “deploy/sdk” directory as part of the task “populate_sdk_ext”. However, it is somehow happening later, and it is hard reading the code to determine when and where that is now occurring. We can use the event stream to help narrow down the candidates.**
  - First, we add a log call into the event read loop in “bitbake/lib/bb/ui/toasterui.py”. This will provide a log of the received events as they go by, and also reveal when the ESDK file is created.

```
logger.info("FOO:"+str(event)+","+
    str(os.path.isfile('<path_to_esdk_file>')) )
```

- **I then run a build (in the Toaster context):**

Using Events for debugging bitbake (2)

- Second, we then run a build (in the Toaster context) and collect the events:

```
$ bitbake do_populate_sdk_ext > my_eventlog.txt
```

- Third, we examine the log to find when the file's state changed.

```
...
NOTE: FOO:<bb.event.DepTreeGenerated object at 0x7f94ec829710>,True
NOTE: FOO:<bb.event.MetadataEvent object at 0x7f94ec829358>,False
...
NOTE: FOO:<LogRecord: ... "Executing buildhistory_get_extra_sdkinfo ...">,False
...
NOTE: FOO:<LogRecord: BitBake.Main, ... sstate-build-populate_sdk_ext ...">,False
NOTE: FOO:<bb.build.TaskSucceeded object at 0x7f94e7f5f358>,True
…
```

- We see that the existing ESDK file was removed after "bb.event.DepTreeGenerated", and placed after "sstate-build-populate_sdk_ext". In other words it was moved out of the main "populate_sdk_ext" task and into its sstate task. QED.

Bonus Example 5: Toaster

Adding Build Data to the Event Database

- **There are many existing analytic views in Toaster**
- **Start the Toaster GUI in the build directory (with open ports)**

```
$ source toaster start webport=0.0.0.0:8000
```

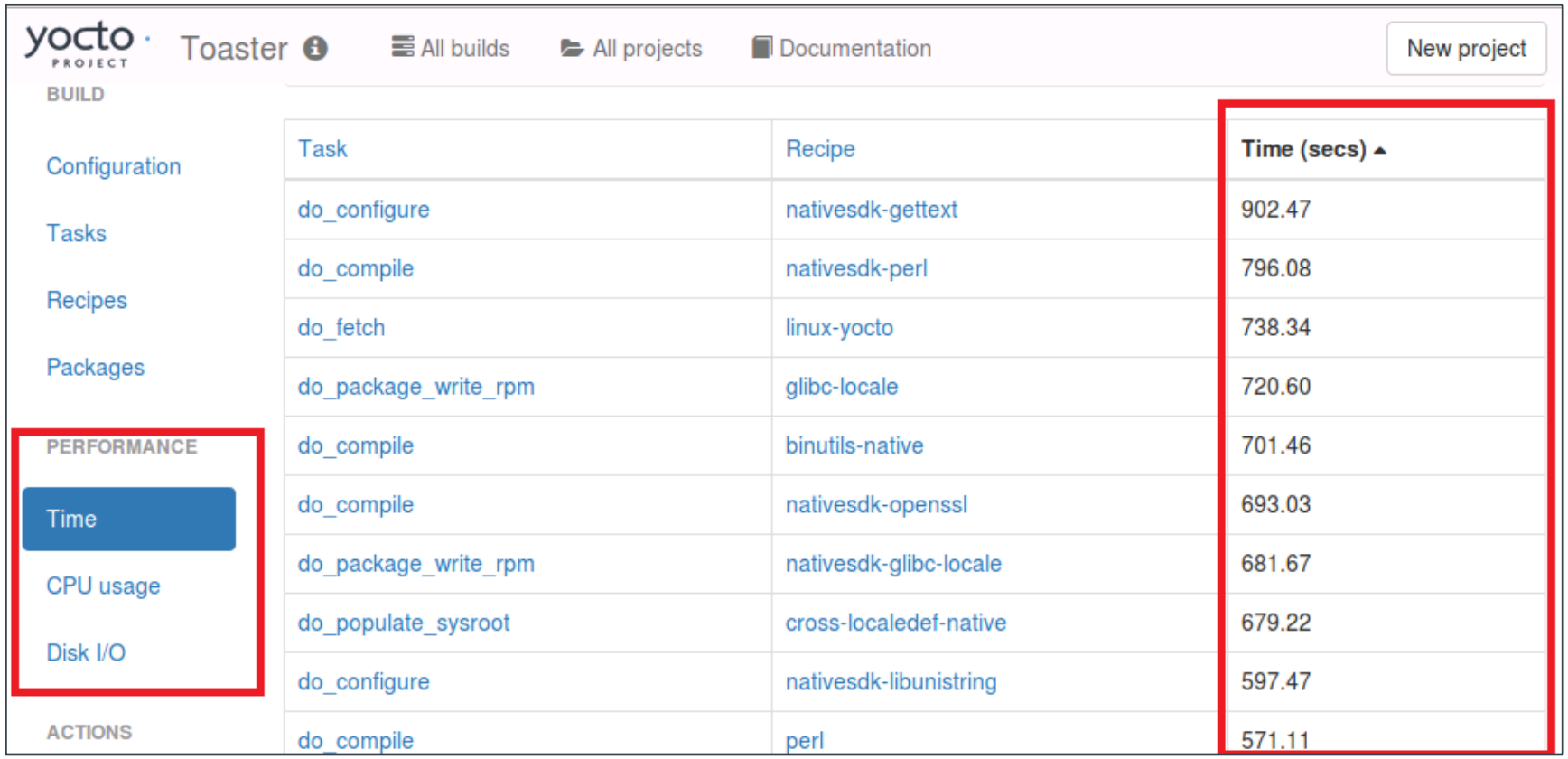
- **On your host, open your browser to:**

  http://devday-a.yocto.io:30000(+your session number)

- **Click on “All Builds”, and select a build**
- **Click on “Time”, “CPU Usage”, and “Disk I/O”**
- **Click on “Tasks”, and see the task order and cache usage**

Existing Toaster Analytics

- **The Toaster GUI already provides analytical data on builds, for example on sstate cache success rate, task execution time, CPU usage, and Disk I/O**



Activity Nine

Kernel And Security Open Forum Staff

Open Topics

- Kernel:
  - Is the YP-2.2 kernel already obsolete?
  - Kernel fragments for any kernel without explicit inherit?
  - Enhanced kernel audit details?
  - Distro and kernel feature integration?
- Security
  - The Yocto Project has a general policy for sustaining (released) branches. We tend to fix individual security issues (CVE) instead of upgrade.
  - There is a Yocto Project security mailing list: yocto-security@yoctoproject.org
  - Low volume. We are working on automatically mailing patches that include the CVE tag to the mailing list so it is searchable, but we have not yet done so.

Open Topics

- **This is tracked by including the relevant CVE tag, pointing to the CVE information in the patches themselves, such as:**

```
+From 7340f67b9860ea0531c1450e5aa261c50f67165d Mon Sep 17 00:00:00 2001
+From: Paul Eggert <eggert@Penguin.CS.UCLA.EDU>
+Date: Sat, 29 Oct 2016 21:04:40 -0700
+Subject: [PATCH] When extracting, skip ".." members
+
+* NEWS: Document this.
+* src/extract.c (extract_archive): Skip members whose names contain
+"..".
+
+CVE: CVE-2016-6321
+Upstream-Status: Backport
+
+Cherry picked from commit: 7340f67 When extracting, skip ".." members
+
+Signed-off-by: Sona Sarmadi <sona.sarmadi@enea.com>
```

- **The above version includes a fix, documents it (CVE: CVE-2016-6321), and then also documents where the fix came from (upstream commit 7340f67 of that project).**

Questions and Answers

Thank you for your participation!

Appendix: Board Bring-up

MinnowBoard Max Turbot SD Card Prep

- **Here is how to flash the microSD card for the MBM**
- **Insert the microSD card into your reader, and attach that to your host**
  1. Find the device number for the card (e.g. “/dev/sdc”). For example run “dmesg | tail” to find the last attached device
  2. Unmount any existing partitions from the SD card (for example “umount /media/<user>/boot”)
  3. Flash the image

     ```
     $ sudo dd if=tmp/deploy/images/intel-corei7-64/core-image-base-intel-corei7-64.hddimg of=<device_id> bs=1M
     ```

  4. On the host, right-click and eject the microSD card’s filesystem so that the image is clean

MinnowBoard Max Turbot SD Card Prep

- **Note: you can instead use the automatically generated WIC image**

1. Flash the image

```
$ sudo dd if=scratch/working/build-
mbm/tmp/deploy/images/intel-corei7-64/core-image-base-
intel-corei7-64.wic of=<device_id> bs=1M
```

2. Note that when the target boots, the WIC version of the image the kernel boot output does not appear on the serial console. This means that after 14 seconds of a blank screen you will then see the login prompt

MinnowBoard Max Turbot Board Bring-up

- Setting up the board connections
  1. Unpack the target
  2. Insert the provided micro-SD card (pin side up)
  3. Attach the ethernet cable from the target to the hub
  4. Attach the FTDI 6-pin connector. **The BLACK wire is on pin 1**, which has an arrow on the silk-mask and is on the center-side of the 6-pin inline connector near the microSD connector
  5. Connect the FTDI USB connector to your host
     (Note: the USB serial connection will appear on your host as soon as the FTDI cable is connected, regardless if the target is powered)

- Start your host's console for the USB serial console connection
  - On Linux, you can use the screen command, using your host's added serial device (for example "/dev/ttyUSB0):

    ```
    $ screen /dev/ttyUSB0 115200,cs8
    ```

    (FYI: "CTRL-A k" to kill/quit)
  - On Windows, you can use an application like "Teraterm", set the serial connection to the latest port (e.g. "COM23"), and set the baud rate to 115200 ("Setup > Serial Port… > Baud Rate…")

MinnowBoard Max Turbot Board Bring-up (2)

- Start the board
  1. Connect the +5 Volt power supply to the target
  2. You should see the board's EFI boot information appear in your host's serial console

- Run these commands to boot the kernel

```
Shell> connect -r
Shell> map -r
Shell> fs0:
Shell> bootx64
```

- You should now see the kernel boot

- At the login prompt, enter "**root**"

- *Note: see the appendix on instructions on how we create the microSD card images*

Dragon Board SD Card Prep

- TBD

Beagle Bone SD Card Prep

- TBD

Bonus Activity

Node.js

Henry Bruce

Introduction

- **Additional Project Setup**
  - tbd
- **Credits: Brendan Le Foll and Paul Eggleton**

What we'll be doing

- Understanding Node.js support in Open Embedded
- Using devtool to auto-generate Node.js recipes
- Building and deploying a package
- On-target Node.js application development
- Using devtool to package the application
- Discuss known issues and plans for future work

Node.js and Open Embedded

- **Layer index recipe search returns ~10 hits**
  - We'll be working with the meta-oe recipe
  - (4.x, oldest LTS version)
- **More versions are available in meta-nodejs**
- **Devtool support was introduced in krogoth**
  - Use pyro
- **There's still work to do. See bug #10653.**

Using devtool to generate recipe

- **Go to the build directory (with a clean shell)**

```
$ cd /scratch
$ . poky/oe-init-build-env
```

- **Create recipe from module in registry**

```
a) $ devtool add "npm://registry.npmjs.org;name=mraa;version=1.5.1"
   -- or --
b) $ devtool add /scratch/src/nodejs/mraa-1.5.1.tgz
```

- **Parses package.json for basis of recipe**
  - Package name and version
  - Description, homepage
  - Location of source
  - Licenses

- **Recursively goes through dependencies**
  - Creates shrinkwrap and lockdown files

```
$ devtool edit-recipe mraa
```

Under the hood

- NPM makes it hard to limit network access to fetch task
- Fetch task walks dependency tree fetching tarballs from NPM registry
- Build task uses ‘npm install’ with registry disabled (OE specific patch) to create node_modules
- Install tasks puts node_modules in correct place

Building and deploying

- **Build is really a pre-package task (apart from native gyp builds)**

```
$ devtool build mraa
```

- **Deploy as normal**

```
$ devtool deploy-target -s mraa
root@target_addr
```

- **Is module installed on the target?**

```
# npm -g ls mraa
```

Running on target

```
# export NODE_PATH=/usr/lib/node_modules
# node
> var mraa = require('mraa')
> console.log('mraa board: ' + mraa.getPlatformName())
> var gpio = new mraa.Gpio(360, true, true)
> gpio.dir(mraa.DIR_OUT)
> gpio.write(1)
```

Developing on target

- **Many ways of doing this. Let's keep it simple**
- **On your target**

```
# mkdir mmax-blinker
# cd mmax-blinker
```

- **Write some code and test it**

```
# cp $NODE_PATH/mraa/examples/javascript/Blink-IO.js .
# vi Blink-IO.js
```

change GPIO to "raw" id 360
add #!/usr/bin/node

```
# node Blink-IO.js
```

Create NPM module on target

- **Create package.json**

```
# cp $NODE_PATH/mraa/COPYING .
# npm init
# vi package.json
```

Add “bin” entry. Local dependency for mraa (or skip)

- **Install**

```
# npm -g install
```

- **Test**

```
# mmax-blinker
```

Create package for your application

- **Copy files to build host**

```
scp -r root@x.x.x.x:mmax-blinker mmax-blinker
```

- **Check dependencies**
  - Local dependencies are for development only
- **Now create package**

```
$ devtool add /path/to/mmax-blinker
$ devtool edit-recipe mmax-blinker
```

Build, deploy and run application

- **Build**

```
$ devtool build mmax-blinker
```

- **Deploy**

```
$ devtool deploy-target mmax-blinker root@x.x.x.x
# ln -s /usr/lib/node_modules/mmax-blinker/Blink-IO.js
/usr/bin/mmax-blinker
# chmod +x /usr/bin/mmax-blinker
```

- **Run**

```
# mmax-blinker
```

Keep in Touch

- https://wiki.yoctoproject.org/wiki/Nodejs_Workflow_Improvements