linaro Connect 2025

Cuttlefish, Kernels, and Bootloaders

Linaro Connect 2025

rammuthiah@google.com

What is Cuttlefish

- Android Virtual Device used by kernel, systems, and BSP devs across the Android Ecosystem to help develop pre-silicon hardware, kernel software, or test various different android configurations

Why should you use it?

- Virtio compliant
    - GPU, SND, Input, Net, Wifi, Block, pmem
    - QEMU, CrosVM, QNX, OpenSynergy
- ADB, WebRTC, serial
- Used to test upstream Linux
    - Android Common Kernel's CI/CD pipeline
- AArch64, x86_64, riscv64
    - Google Cloud, AWS, w/ or w/o GPU, ARM Bare Metal, Emulation
- Bootloader support (U-Boot)
    - UEFI compatibility
    - Bootconfig + AVB support
    - Fastboot
- CTS / VTS Coverage (~95% pass rate)

aosp-main migration

- Cloud Android's launcher is moving as a whole to our github and being removed from AOSP
    - Will take place over the next ~6 months
- The device implementation will be released on the AOSP schedule going forward (as part of the quarterly Android release)
    - Please post any patches to AOSP, the team will review and post internally on your behalf
- Compatibility between the launcher and device will be maintained across all supported releases (Android 12 is currently earliest supported)

Current Events

- Software Defined Vehicle development on Cuttlefish
- Enabling your own CI
  - Cuttlefish Orchestration (Host / Cloud)
- aosp-main Migration
- Docker Container Strategy (x86 / ARM)

Getting Started

- Install our host packages
  - cuttlefish-base and cuttlefish-user - https://github.com/google/android-cuttlefish
  - Both x86_64 and arm64 Hosts are supported
    - Prebuilt debian packages can be found here
    - Also present in Debian Experimental thanks to Paul Liu
  - Docker Container
- For Orchestration (CI or local developer usage)
  - Cloud Android Orchestrator - https://github.com/google/cloud-android-orchestration

Android Build

```
$ mkdir android && cd android
$ repo init -u https://android.googlesource.com/platform/manifest-b main
$ repo sync -j
$ source build/envsetup.sh
$ lunch aosp_cf_x86_64_only_phone-trunk_staging-userdebug
$ m -j
```

Kernel + Modules + GBL Build

```
$ mkdir kernel && cd kernel
$ repo init -u https://android.googlesource.com/kernel/manifest-b \
    common-android-mainline # or common-android16-6.12
$ repo sync -j
$ tools/bazel run //common:kernel_x86_64_dist
$ tools/bazel run //common-modules/virtual-device:virtual_device_x86_64_dist
$ tools/bazel run --config=gbl //bootable/libbootloader:gbl_efi_dist
```

Bootloader Build

```
$ mkdir u-boot && cd u-boot
$ repo init -u https://android.googlesource.com/kernel/manifest -b \
    u-boot-mainline
$ repo sync -j
$ tools/bazel run //u-boot:crosvm_x86_64
```

Launch / Interact w/ the device

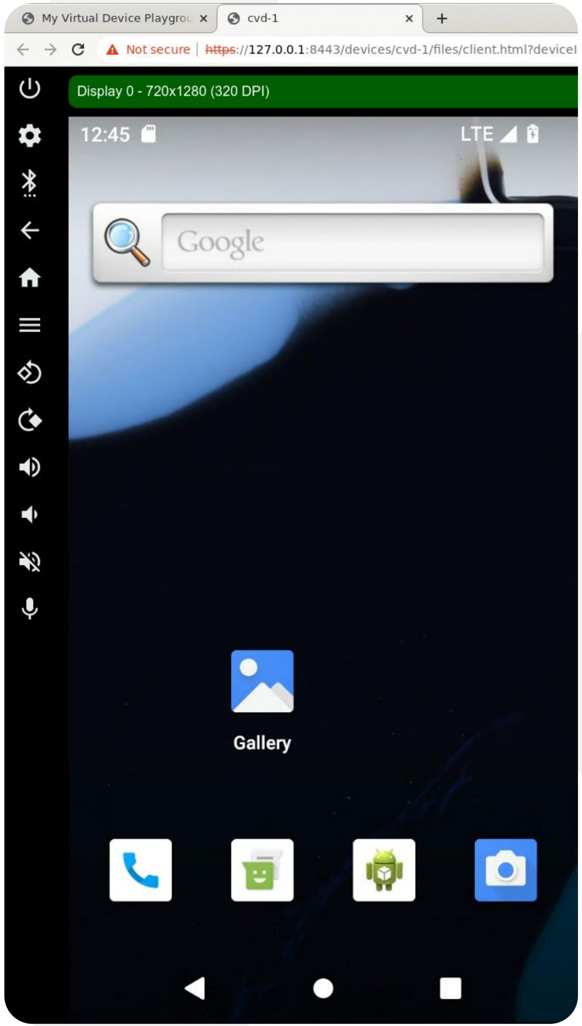
```
$ cvd create -kernel_path /path/to/bzImage \
    -initramfs_path /path/to/kernel/module/ramdisk
$ adb shell
$ tail -f ~/cuttlefish_runtime/kernel.log // dmesg
> Go to https://127.0.0.1:8443/
```

Launch w/ Bootloader

```
$ cvd create -bootloader /path/to/u-boot.rom \
    -pause_in_bootloader -console=true
```

Launch w/ GBL

```
$ cvd create -android_efi_loader \
    /path/to/gbl_x86_64_prod.efi
```



What's next

- EFI Boot by default
- Media Acceleration (Video Encode/Decode, Camera)
- aosp-main migration

References

cloud-android-ext@google.com - Feature requests are welcome!

https://source.android.com/docs/setup/create/cuttlefish - for more information

linaro Connect 2025

Generic Android Bootloaders

Linaro Connect 2025

rammuthiah@google.com

What is a Bootloader?

- Single or multi stage firmware that
  - Initializes hardware
  - Loads (and verifies) the OS (kernel, rootfs) from the available boot devices
  - Collects all boot parameters (i.e. kernel commandline, device tree, bootconfig, ACPI, etc.)
  - Assembles these into memory
  - Jumps into the kernel

What is an Android Bootloader?

- A bootloader with some Android specific functions
  - Android Boot Image Parsing (boot, init_boot, vendor_boot, dtb, dtbo)
  - Fastboot
  - Android Verified Boot
  - Keymint integration
  - Protected Virtual Machine Firmware Loading
  - Etc.
- And device specific functions
  - TEE Support (Trustonic, QTEE, …)
  - Block Drivers (UFS, eMMC, SDCard)
  - Crypto
  - Graphics (Boot Splash, fastboot UX) & Buttons
  - Measured Boot
  - Hypervisors

Pain Points

- Duplication
  - fastboot, libavb, PVM FW Load, Boot image support are all re-implemented by each Android manufacturer
- Updatability
  - Bootloader trees are forked with the device and stop receiving updates within a few years of launch
- Annual Android boot flow modifications
  - PQC in the coming years
  - Boot Images over the last few years (Android 11 - 14)
    - Bootconfig (Android 12)
  - PVM FW Load (Android 15)

Existing Solutions

- Upstream Uboot
    - Android Things
- Coreboot
- UEFI
    - EFIDroid
- nmbl

Requirements

- Backwards compatibility
- Discoverable Calls
- Closed-Source bootloader support

UEFI

- Widely used (EDK2, UBoot, LK)
  - Interface is stable, revisions are now released every ~10 years
- Discoverable Calls
- SystemReady
- Drivers (Protocols) are already defined for all common functionalities (i.e. block, memory allocation, etc)

Generic Bootloader (GBL)

linaro Connect

- Annually released EFI boot application as part of the Android Release
- Security patches provided for lifetime of the Android release
- Supports all new Android boot requirements
- Developed in AOSP
- Technical details
  - no_std rust UEFI Application
  - arm64, x86_64, riscv_64 targets available
  - Libavb, boringssl, open-dice, libufdt, libfdt built in
  - Part of the Android Common kernel tree
  - Built w/ bazel
  - Cuttlefish + devboard support

GBL Protocols (Required)

- Block IO
- Hash IO
- RNG
- Memory Allocation
- Android specific
  - OS Configuration
  - AB Slot
  - Android Verified Boot

GBL Protocols (Recommended)

- Simple Text Input / Output
- Debug - next presentation :)
- Android specific
  - Image Loading
  - Fastboot USB
  - Fastboot
  - Android Virtualiztion Framework

Links + Documentation

- GBL Readme - https://android.googlesource.com/platform/bootable/libbootloader/+/refs/heads/gbl-mainline/gbl/
- GBL Docs - https://android.googlesource.com/platform/bootable/libbootloader/+/refs/heads/gbl-mainline/gbl/docs
- Email rammuthiah@google.com, paul.liu@linaro.org, android-gbl@google.com
- Source can be found at android-mainline and android16-6.12

What’s Next

linaro Connect

- GBL for Android 16 release in June 2025
- Upstreaming of Android UEFI Protocols to EDK2, UBoot, and LK over next 2 quarters
- Continuing to flesh out LittleKernel UEFI support
- GBL Interface Freeze for Android 17 - October 2025
- GBL as a requirement for Android 17

linaro
Connect
2025

Android Bootloader Development & Debug

Linaro Connect 2025
paul.liu@linaro.org

Goal

- To enhance the developer debug experience with GBL

This will

- Enable partners to bring up GBL on their hardware more efficiently
- Make developers more productive when adding new features

Loading debug symbols

- In gdb, use command “add-symbol-file HelloWorld.debug <TextAddress> -s .data <DataAddress>”
- TextAddress is the address of text section.
- DataAddress is the address of data section.
- The problem is how to get these address?
- EFI applications are load dynamically by EDK2 or U-boot.

TextAddress and DataAddress

- TextAddress = LoadAddress + Text section offset.
- DataAddress = LoadAddres + Data section offset.
- EFI applications are in PE format. So we can use readpe (or python pefile library) to get the offset of the sections.
- The problem is how to get the LoadAddress.

LoadAddress - by debug log

- EDK2 will output debug log to I/O port 0x402 on x86.
  - “Loading driver at <LoadAddress> EntryPoint=<EntryAddress> HelloWorld.efi”
- U-boot will output debug log by enabling U-boot’s log command.

LoadAddress - by EFI Debug Support Table

- UEFI Spec. Section 18.4
- The LoadAddress is stored in EFI_LOADED_IMAGE_PROTOCOL table. If we can locate this table. Then we can get the LoadAddress of the EFI application.
- So the first step is to locate the EFI System Table.

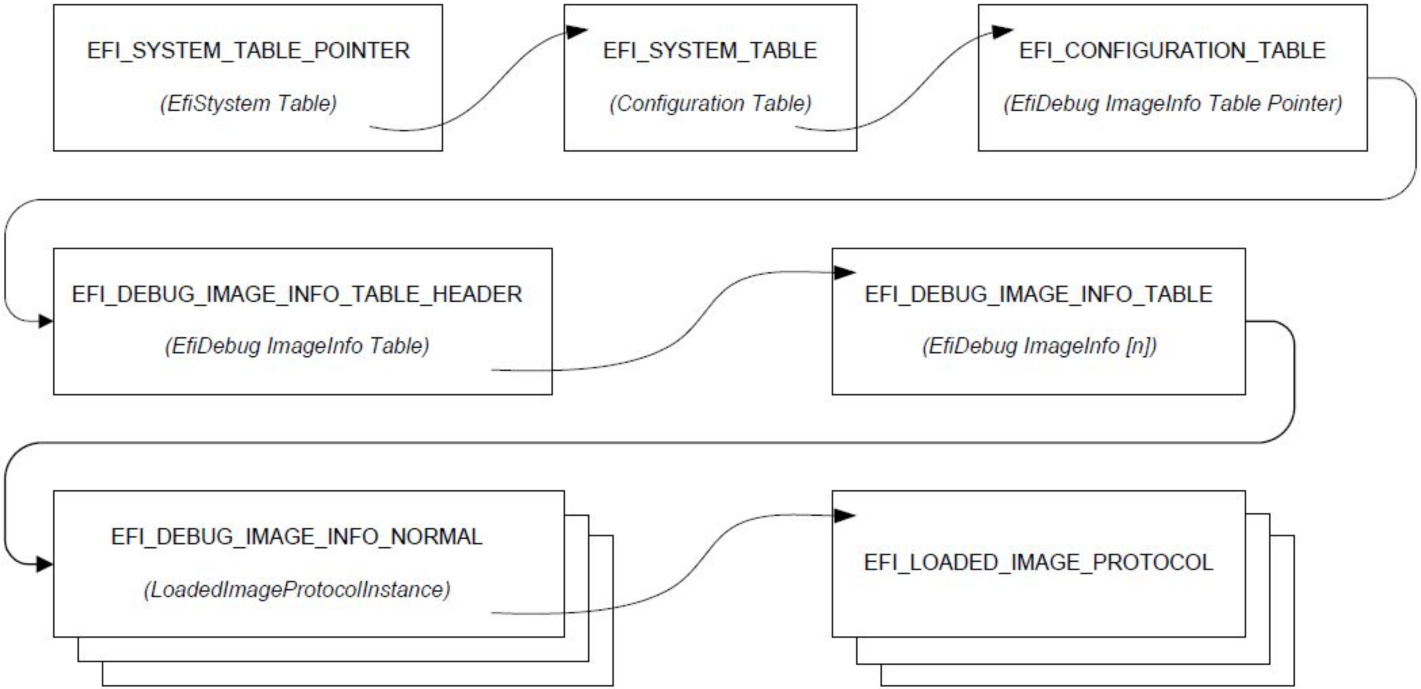
Locate the EFI System Table.

- An external debugger to determine loaded image information in a quiescent manner.
- The EFI system table can be located by an off-target hardware debugger by searching for the EFI_SYSTEM_TABLE_POINTER structure.
- The structure is located on a 4M boundary as close to the top physical memory as feasible.

EFI_SYSTEM_TABLE_POINTER
Signature: UINT64 EfiSystemTableBase: EFI_PHYSICAL_ADDRESS Crc32: UINT32

EFI_DEBUG_IMAGE_INFO_TABLE

- We publish an EFI_CONFIGURATION_TABLE that leads to a database of pointers to all instances of the loaded image protocol.



EFI_DEBUG_IMAGE_INFO_TABLE_HEADER

EFI_DEBUG_IMAGE_INFO_TABLE_HEADER
UpdateStatus: volatile UINT32 TableSize: UINT32 EfiDebugImageInfoTable: EFI_DEBUG_IMAGE_INFO

EFI_DEBUG_IMAGE_INFO_NORMAL
ImageInfoType: UINT32 LoadedImageProtocolInstance: EFI_LOADED_IMAGE_PROTOCOL ImageHandle: EFI_HANDLE

Links

- Spec: https://uefi.org/specs/UEFI/2.10/18_Protocols_Debugger_Support.html
- gdb extension for locating SYSTEM_TABLE: https://github.com/tianocore/edk2/commit/d985bd4b973327a3a79dfd258c17b256d7fa1e7d

We will be demoing GBL and the debugger support at Demo Friday!

linaro Connect 2025

Thank You!
Questions?