Qualcomm

November 13th, 2023

Driver Development Kit (DDK) and Vendor Workflow

John Moon <quic_johmoo@quicinc.com>
Software Engineer
Qualcomm Innovation Center, Inc.

Agenda

1. What is the DDK?
2. How We Built External Modules Before DDK
3. How We Build External Modules Now
4. Pros and Cons

What is the DDK?

What is DDK?

- Google migrated the AOSP kernel build to Bazel.
  - Website: bazel.build
- To build the Android kernel and other kernel artifacts (modules, boot images, etc.), they provide a framework called "Kleaf".
- One part of Kleaf is the Driver Development Kit (DDK) which is used to build external modules.
- For Android 14+, Kleaf is strongly recommended.

How We Built External Modules Before DDK

Kernel Team

- The kernel team manages the internal `kernel_platform` tree.
  - This includes vendor modules, configurations, KMI compliance, etc.
- Common build definitions are under the kernel team's purview.

Tech Teams

- Specialized teams are split out into "tech teams" or "tech packs."
- These teams are focused on one specific aspect of the system (e.g., display, audio, Bluetooth, etc.).
- Tech teams have their own Git repository and their own location in the vendor tree.
- Tech teams that include kernel modules build those modules out-of-tree.
- We have around 50 tech pack kernel modules that we build this way.

Qualcomm Technologies' Repo Project Layout (pre-migration)

A quick tour of our tree

- **`build.sh`** builds the kernel and in-tree* vendor modules

```
vendor
├── device/qcom/common/dlkm
│   └── Build_external_kernelmodule.mk
├── kernel_platform
│   ├── build/kernel
│   │   ├── build_module.sh
│   │   └── build.sh
│   └── msm-kernel
│       └── <Qualcomm kernel sources>
└── vendor/qcom/opensource/techpack
    ├── Android.mk
    ├── my_module.c
    ├── Kbuild
    └── Makefile
```

* “In-tree” means in the downstream msm-kernel tree.

Qualcomm Technologies' Repo Project Layout (pre-migration)

A quick tour of our tree

- Tech pack modules contain what you might expect: source code, a `Kbuild` file, and a `Makefile`.

```
vendor
├── device/qcom/common/dlkm
│   └── Build_external_kernelmodule.mk
├── kernel_platform
│   ├── build/kernel
│   │   ├── build_module.sh
│   │   └── build.sh
│   └── msm-kernel
│       └── <Qualcomm kernel sources>
└── vendor/qcom/opensource/techpack
    ├── Android.mk
    ├── my_module.c
    ├── Kbuild
    └── Makefile
```

Qualcomm Technologies' Repo Project Layout (pre-migration)

A quick tour of our tree

- The tech pack **`Android.mk`** file defines a target for the top-level AOSP build. For example:

```
include $(CLEAR_VARS)
LOCAL_SRC_FILES           := my_module.c
LOCAL_MODULE              := my_module.ko
LOCAL_MODULE_KBUILD_NAME  := my_module.ko
LOCAL_MODULE_TAGS         := optional
LOCAL_MODULE_DEBUG_ENABLE := true
LOCAL_MODULE_PATH         := $(KERNEL_MODULES_OUT)
include $(DLKM_DIR)/Build_external_kernelmodule.mk
```

```
vendor
├── device/qcom/common/dlkm
│   └── Build_external_kernelmodule.mk
├── kernel_platform
│   ├── build/kernel
│   │   ├── build_module.sh
│   │   └── build.sh
│   └── msm-kernel
│       └── <Qualcomm kernel sources>
└── vendor/qcom/opensource/techpack
    ├── Android.mk
    ├── my_module.c
    ├── Kbuild
    └── Makefile
```

Qualcomm Technologies' Repo Project Layout (pre-migration)

A quick tour of our tree

- The tech pack **`Android.mk`** file defines a target for the top-level AOSP build. For example:

```
include $(CLEAR_VARS)
LOCAL_SRC_FILES            := my_module.c
LOCAL_MODULE               := my_module.ko
LOCAL_MODULE_KBUILD_NAME   := my_module.ko
LOCAL_MODULE_TAGS          := optional
LOCAL_MODULE_DEBUG_ENABLE  := true
LOCAL_MODULE_PATH          := $(KERNEL_MODULES_OUT)
include $(DLKM_DIR)/Build_external_kernelmodule.mk
```

- This calls out to **`Build_external_kernelmodule.mk`**.

```
vendor
├── device/qcom/common/dlkm
│   └── Build_external_kernelmodule.mk
├── kernel_platform
│   ├── build/kernel
│   │   ├── build_module.sh
│   │   └── build.sh
│   └── msm-kernel
│       └── <Qualcomm kernel sources>
└── vendor/qcom/opensource/techpack
    ├── Android.mk
    ├── my_module.c
    ├── Kbuild
    └── Makefile
```

Qualcomm Technologies' Repo Project Layout (pre-migration)

A quick tour of our tree

- **`Build_external_kernelmodule.mk`** is used by every techpack.
- Internally, it ends up calling out to **`build_module.sh`**.
- **`build_module.sh`** is the one who *finally* calls:

```
make -C ${EXT_MOD} \
     M=${EXT_MOD_REL} \
     KERNEL_SRC=${ROOT_DIR}/${KERNEL_DIR} \
     O=${OUT_DIR} \
     modules
```

```
vendor
├── device/qcom/common/dlkm
│   └── Build_external_kernelmodule.mk
├── kernel_platform
│   ├── build/kernel
│   │   ├── build_module.sh
│   │   └── build.sh
│   └── msm-kernel
│       └── <Qualcomm kernel sources>
└── vendor/qcom/opensource/techpack
    ├── Android.mk
    ├── my_module.c
    ├── Kbuild
    └── Makefile
```

How We Build External Modules with DDK Now

Qualcomm Technologies' Repo Project Layout (post-migration)

A quick tour of our tree

- **`build.sh`** is gone.
- Instead, we have Bazel build definitions (**`BUILD.bazel`**).
- We define a **`kernel_build()`** target using Kleaf.
- So, to build our kernel and in-tree* modules, we do:

```
bazel build //msm-kernel:${target}_${variant}
```

```
vendor
├── device/qcom/common/dlkm
│   └── Build_external_kernelmodule.mk
├── kernel_platform
│   ├── build/kernel
│   │   ├── build_module.sh
│   │   └── kleaf
│   │       └── <Kleaf framework>
│   ├── msm-kernel
│   │   ├── BUILD.bazel
│   │   └── <Qualcomm kernel sources>
│   ├── vendor -> ../vendor
│   └── WORKSPACE
├── vendor/qcom/opensource/techpack
│   ├── Android.mk
│   ├── BUILD.bazel
│   └── my_module.c
└── WORKSPACE
```

* “In-tree” means in our msm-kernel tree.

Qualcomm Technologies' Repo Project Layout (post-migration)

A quick tour of our tree

- Tech packs now define a **`ddk_module()`** in their Bazel build definitions.
- For example:

```
load("//build/kernel/kleaf:kernel.bzl", "ddk_module")

ddk_module(
    name = "my_module",
    srcs = ["my_module.c"],
    out = "my_module.ko",
    deps = ["//msm-kernel:all_headers"],
    kernel_build = "//msm-kernel:${target}_${variant}",
)
```

- Note: no **`Kbuild`** or **`Makefile`**!

```
vendor
├── device/qcom/common/dlkm
│   └── Build_external_kernelmodule.mk
├── kernel_platform
│   ├── build/kernel
│   │   ├── build_module.sh
│   │   └── kleaf
│   │       └── <Kleaf framework>
│   ├── msm-kernel
│   │   ├── BUILD.bazel
│   │   └── <Qualcomm kernel sources>
│   ├── vendor -> ../vendor
│   └── WORKSPACE
├── vendor/qcom/opensource/techpack
│   ├── Android.mk
│   ├── BUILD.bazel
│   └── my_module.c
└── WORKSPACE
```

Qualcomm Technologies' Repo Project Layout (post-migration)

A quick tour of our tree

- Top-level AOSP build isn't using Bazel yet, so we still need to bridge that gap.
- So, the **`Android.mk`** file is doing the same thing:

```
include $(CLEAR_VARS)
LOCAL_SRC_FILES           := my_module.c
LOCAL_MODULE              := my_module.ko
LOCAL_MODULE_KBUILD_NAME  := my_module.ko
LOCAL_MODULE_TAGS         := optional
LOCAL_MODULE_DEBUG_ENABLE := true
LOCAL_MODULE_PATH         := $(KERNEL_MODULES_OUT)
include $(DLKM_DIR)/Build_external_kernelmodule.mk
```

```
vendor
├── device/qcom/common/dlkm
│   └── Build_external_kernelmodule.mk
├── kernel_platform
│   ├── build/kernel
│   │   ├── build_module.sh
│   │   └── kleaf
│   │       └── <Kleaf framework>
│   ├── common
│   │   └── <ACK sources>
│   ├── msm-kernel
│   │   ├── BUILD.bazel
│   │   └── <Qualcomm kernel sources>
│   ├── vendor -> ../vendor
│   └── WORKSPACE
├── vendor/qcom/opensource/techpack
│   ├── Android.mk
│   ├── BUILD.bazel
│   └── my_module.c
└── WORKSPACE
```

Qualcomm Technologies' Repo Project Layout (post-migration)

A quick tour of our tree

- **`Build_external_kernelmodule.mk`** is used the same way and still ends up calling out to **`build_module.sh`**.
- **`build_module.sh`** is the one who *finally* calls out to build the module with Bazel:

```
bazel build \
  //vendor/qcom/opensource/techpack:my_module
```

- The AOSP build is none the wiser that tech packs are using Bazel.

- Note: This method requires a symlink to the vendor tree because Kleaf only operates properly inside the **`kernel_platform`** tree.

```
vendor
├── device/qcom/common/dlkm
│   └── Build_external_kernelmodule.mk
├── kernel_platform
│   ├── build/kernel
│   │   ├── build_module.sh
│   │   └── kleaf
│   │       └── <Kleaf framework>
│   ├── common
│   │   └── <ACK sources>
│   ├── msm-kernel
│   │   ├── BUILD.bazel
│   │   └── <Qualcomm kernel sources>
│   ├── vendor -> ../vendor
│   └── WORKSPACE
├── vendor/qcom/opensource/techpack
│   ├── Android.mk
│   ├── BUILD.bazel
│   └── my_module.c
└── WORKSPACE
```

Pros and Cons

Pro: Limiting Kernel Header Visibility

Making sure your modules aren't using unsafe headers

- This is the biggest benefit our kernel team has seen.
- Now, if a tech team tries to use a private kernel header, they get an error message:

```
: fatal error: 'mm/slab.h' file not found
#include <mm/slab.h>
         ^~~~~~~~~~~
1 error generated.
```

- Because Kleaf only copies public kernel headers (e.g., `include/linux`) into the DDK build sandbox, the private kernel headers are not available!
- This revealed several places where tech teams were using private headers that we weren't even aware of.

Pro: Upstream Support and Collaboration

kleaf: Add super_image rule	John Moon	Ramji, Ulises M… , +2	kernel/build	main	Sep 08	M	Merged
kleaf: Prevent symlink dereferenc…	John Moon	Yifan	kernel/build	main	Jul 11	XS	Merged
dist: add support for configurable …	John Moon	Jingwen, Matthias … , +3	…/build/bazel_common_rules	master	Jun 22	M	Merged
kleaf: Add --no-group to all rsync c…	John Moon	Matthias … , Ulises M… , +1	kernel/build	master	Jun 21	M	Merged
kleaf: Add label_flag for extra sym…	John Moon	Guru Das… , Matthias … , +2	kernel/build	master (ack_extra_…	Jun 16	S	Merged
kleaf: Add flag to ignore missing r…	John Moon	Matthias … , Yifan	kernel/build	master	Jun 16	S	Merged

Google + QuIC

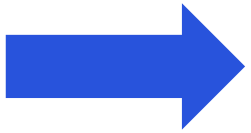
- Google has been a great tech partner during this migration.
- At the time of this slide being written, I personally have filed 59 bugs against Kleaf (50 of which have been resolved).
- We've been able to mold Kleaf to our use cases.

Pro: Less Boilerplate

Obviate the Kbuild and Makefile

- The DDK auto-generates your `Makefile` and `Kbuild` files.
- This is nice for **new** drivers but can be somewhat annoying for an **existing** driver.
- In some cases, we dropped about 50% of the lines of code needed to express module builds.
  - This is largely due to Starlark macros allowing us to loop build logic over multiple platforms and architectures.

```
vendor/qcom/opensource/techpack
├── Android.mk
├── my_module.c
├── Kbuild
└── Makefile
```



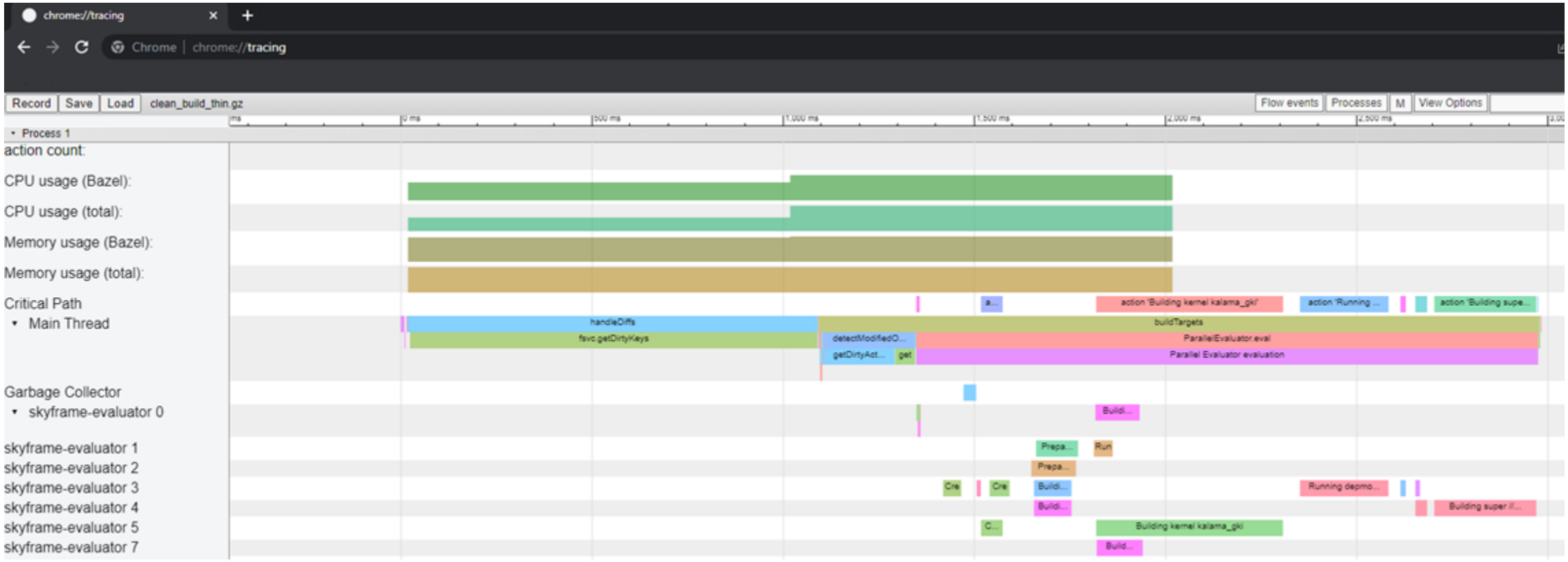
```
vendor/qcom/opensource/techpack
├── Android.mk
├── my_module.c
└── BUILD.bazel
```

Pro: Query Your DAG

- What headers does this module use?
  - `bazel query 'kind(ddk_headers, deps(//vendor/qcom/opensource/techpack:target))'`
- What kernel modules does this module depend on?
  - `bazel query 'kind(kernel_module, deps((//vendor/qcom/opensource/techpack:target))’`
- What modules depend on this module?
  - `bazel query '//vendor/qcom/opensource/... intersect allpaths(//vendor/qcom/opensource/..., //vendor/qcom/opensource/techpack:target)'`

Pro: Build Performance Profiling

https://bazel.build/rules/performance#performance-profiling



Pro: Easier Inter-Dependencies and API Separation

- A module that provides dependencies can define:
  - which header files are “public” (**`ddk_headers`** / **`ddk_module.hdrs`**).
  - which Bazel packages are allowed to depend on them (package visibility).
- A module that depends on other modules can do so:
  - by adding a module to their **`deps`**.

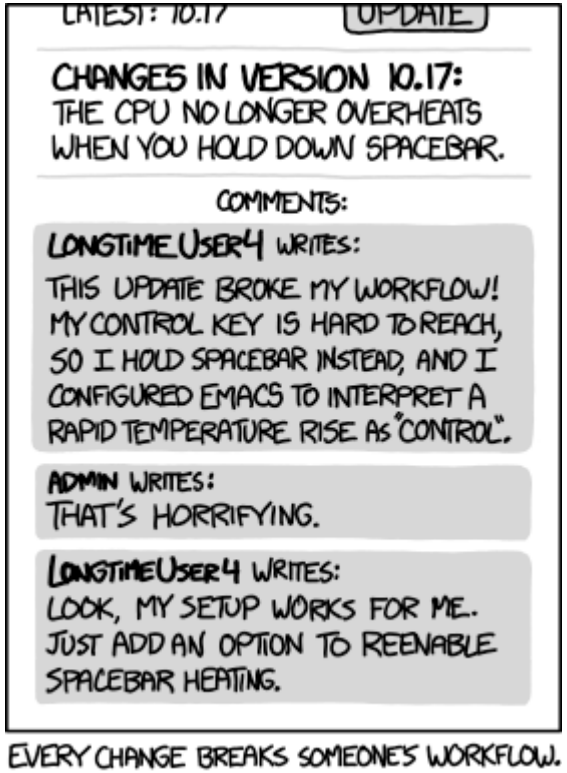
```
ddk_module(
    # Module name
    name = "mod_with_foo",
    ...
    # Headers listed here will be exposed to any module that
    # depends on this module
    hdrs = ["include/foo_public_header.h"],
    # Directories to include for header search (passed to -I)
    includes = ["include"],
    # Other Bazel rules which are allowed to depend on
    # this module
    visibility = ["//vendor/qcom/opensource/techpack_b:__pkg__"]
)
```

```
ddk_module(
    # Module name
    name = "mod_using_foo",
    ...
    # Header dependencies from kernel and any other
    # ddk_modules which define symbols this module needs
    deps = [
        "//msm-kernel:all_headers",
        "//vendor/qcom/opensource/techpack_a:mod_with_foo",
    ],
)
```

Con: Needing to Learn a New Build System

What's wrong with Makefiles and shell scripts?

- Bazel uses Starlark (Python-like) configuration language for build files.
  - Many kernel developers are not familiar with Python and even less so with Starlark.
- There was a lot of confusion around Bazel's build phases and when build logic would be executed.
- Seemingly simple things turn out to be very complicated.
  - Examples: environment variable handling, using non-Bazel files to influence build, outputting an arbitrary set of files
  - This is for good reason, but it's hard to tell kernel developers they simply can't do something they've been doing with Makefiles for their whole career.
- Our tech teams had tons of hacks in place to get things working for specific use cases in the AOSP build.
  - Bazel unified the strategy, but it was difficult to get there.



Credit: https://xkcd.com/1172/

Con: Debugging

Sandboxing and new types of errors

- Bazel builds things in a sandbox to make sure you have all dependencies declared properly.
  - This makes it more difficult for users to diagnose errors.
  - Long sandbox path names pollute output.
- Things that may have worked before break because of sandboxing.
  - Great for finding host dependencies you weren't aware of, but annoying if you don't understand.

```
 C 18:23:44 Value of CONFIG_VIRT_DRIVERS is redundant by fragment msm-kernel/arch/arm64/configs/vendor/<target>_GKI.config:
C 18:23:44 #
C 18:23:44 # merged configuration written to /workdir/work.refs/heads/<branch>/src/linux/kernel_platform/out/bazel/output_user_root/7b7139b3c45fad3cb3fcca945a9db469
sandbox/processwrapper-sandbox/74/execroot/__main__/outdcf70a65/android14-6.1/msm-kernel/arch/arm64/configs/vendor/<target>-gki_defconfig (needs make)
C 18:23:44 #
C 18:23:44 ===========================================================
C 18:23:44 ERROR! Defconfig fragment did not apply as expected
C 18:23:44 -PM_SILENT_MODE m
C 18:23:44 [736 / 956] Preparing for module build (btf_debug_info=default;lto=thin;notrim) @//msm-kernel:<target>_<variant>_modules_prepare; 5s processwrapper-sandb
x
C 18:23:44 [737 / 956] checking cached actions
C 18:23:44 [737 / 956] [Prepa] Building kernel (btf_debug_info=default;lto=thin;notrim) @//msm-kernel:<target>_<variant>; 9s
C 18:23:44 ERROR: /workdir/work.refs/heads/qcom-6.1/src/linux/kernel_platform/msm-kernel/BUILD.bazel:275:21 Middleman _middlemen/msm-kernel_S<target>_Ugki_Udist-run
iles failed: (Exit 1): bash failed: error executing command (from target //msm-kernel:<target>_gki_config) /bin/bash -c ... (remaining 1 argument skipped)
C 18:23:44
 C 18:23:44 Use --sandbox_debug to see verbose messages from the sandbox and retain the sandbox build root for debugging
C 18:23:44 INFO: Elapsed time: 244.688s, Critical Path: 39.30s
C 18:23:44 INFO: 102 processes: 29 internal, 73 processwrapper-sandbox.
C 18:23:44 ERROR: Build did NOT complete successfully
C 18:23:44 Build exited with non-zero exit code 1
```

Questions?

`ddk_module()` example

ddk_module() example

```
# Load the ddk_module macro (similar to Python's import)
load("//build/kernel/kleaf:kernel.bzl", "ddk_module")

ddk_module(
    # Module name
    name = "mod_with_foo",
    # Source files to include in this build
    srcs = glob([
        "*.c",
        "include/foo_private_header.h"
    ]),
    # Output module name
    out = "mod_with_foo.ko",
    # Headers listed here will be exposed to any module that
    # depends on this module
    hdrs = ["include/foo_public_header.h"],
    # Directories to include for header search (passed to -I)
    includes = ["include"],
    # The kernel to build against
    kernel_build = "//common:kernel_aarch64",
    # Header dependencies from kernel
    deps = ["//common:all_headers"],
    # Other Bazel rules which are allowed to depend on
    # this module
    visibility = ["//ddk_module_example/mod_using_foo:__pkg__"]
)
```

```
ddk_module_example
├── mod_using_foo
│   ├── BUILD.bazel
│   └── mod_using_foo.c
└── mod_with_foo
    ├── BUILD.bazel
    ├── include
    │   ├── foo_private_header.h
    │   └── foo_public_header.h
    └── mod_with_foo.c
```

ddk_module() example

```
# Load the ddk_module macro (similar to Python's import)
load("//build/kernel/kleaf:kernel.bzl", "ddk_module")

ddk_module(
    # Module name
    name = "mod_using_foo",
    # Source files to include in this build
    srcs = glob(["*.c"]),
    # Output module name
    out = "mod_using_foo.ko",
    # The kernel to build against
    kernel_build = "//common:kernel_aarch64",
    # Header dependencies from kernel and any other
    # ddk_modules which define symbols this module needs
    deps = [
        "//common:all_headers",
        "//ddk_module_example/mod_with_foo",
    ],
)
```

```
ddk_module_example
├── mod_using_foo
│   ├── BUILD.bazel
│   └── mod_using_foo.c
└── mod_with_foo
    ├── BUILD.bazel
    ├── include
    │   ├── foo_private_header.h
    │   └── foo_public_header.h
    └── mod_with_foo.c
```

`ddk_module()` example

- **`bazel build //ddk_module_example/mod_using_foo:mod_using_foo`**

File: bazel-bin/ddk_module_example/mod_using_foo/mod_using_foo_makefiles/makefiles/Makefile

```
# Include symbol: ddk_module_example/mod_with_foo/mod_with_foo_Module.symvers
EXTRA_SYMBOLS += $(COMMON_OUT_DIR)/ddk_module_example/mod_with_foo/mod_with_foo_Module.symvers
modules modules_install clean:
    $(MAKE) -C $(KERNEL_SRC) M=$(M) $(KBUILD_OPTIONS) KBUILD_EXTRA_SYMBOLS="$(EXTRA_SYMBOLS)" $(@)
```

File: bazel-bin/ddk_module_example/mod_using_foo/mod_using_foo_makefiles/makefiles/Kbuild

```
# Build ddk_module_example/mod_using_foo/mod_using_foo.ko
obj-m += mod_using_foo.o

# The module mod_using_foo.ko has a source file mod_using_foo.c

# Common flags for mod_using_foo.o

LINUXINCLUDE := \
  -I$(ROOT_DIR)/common/arch/arm64/include \
  -I$(ROOT_DIR)/common/arch/arm64/include/uapi \
  -I$(ROOT_DIR)/common/include \
  -I$(ROOT_DIR)/common/include/uapi \
  $(LINUXINCLUDE)

CFLAGS_mod_using_foo.o += @$(ROOT_DIR)/ddk_module_example/mod_using_foo/mod_using_foo.cflags
```

File: bazel-bin/ddk_module_example/mod_using_foo/mod_using_foo_makefiles/makefiles/mod_using_foo.cflags

```
-I$(ROOT_DIR)/ddk_module_example/mod_with_foo/include
```

`ddk_module()` example

- **`bazel build //ddk_module_example/mod_using_foo:mod_using_foo`**

File: bazel-bin/ddk_module_example/mod_using_foo/mod_using_foo_makefiles/makefiles/Makefile

```
# Include symbol: ddk_module_example/mod_with_foo/mod_with_foo_Module.symvers
EXTRA_SYMBOLS += $(COMMON_OUT_DIR)/ddk_module_example/mod_with_foo/mod_with_foo_Module.symvers
modules modules_install clean:
    $(MAKE) -C $(KERNEL_SRC) M=$(M) $(KBUILD_OPTIONS) KBUILD_EXTRA_SYMBOLS="$(EXTRA_SYMBOLS)" $(@)
```

File: bazel-bin/ddk_module_example/mod_using_foo/mod_using_foo_makefiles/makefiles/Kbuild

```
# Build ddk_module_example/mod_using_foo/mod_using_foo.ko
obj-m += mod_using_foo.o

# The module mod_using_foo.ko has a source file mod_using_foo.c

# Common flags for mod_using_foo.o

LINUXINCLUDE := \
  -I$(ROOT_DIR)/common/arch/arm64/include \
  -I$(ROOT_DIR)/common/arch/arm64/include/uapi \
  -I$(ROOT_DIR)/common/include \
  -I$(ROOT_DIR)/common/include/uapi \
  $(LINUXINCLUDE)

CFLAGS_mod_using_foo.o += @$(ROOT_DIR)/ddk_module_example/mod_using_foo/mod_using_foo.cflags
```

File: bazel-bin/ddk_module_example/mod_using_foo/mod_using_foo_makefiles/makefiles/mod_using_foo.cflags

```
-I$(ROOT_DIR)/ddk_module_example/mod_with_foo/include
```

Symbols from `mod_with_foo` included automatically

`ddk_module()` example

- **`bazel build //ddk_module_example/mod_using_foo:mod_using_foo`**

File: bazel-bin/ddk_module_example/mod_using_foo/mod_using_foo_makefiles/makefiles/Makefile

```
# Include symbol: ddk_module_example/mod_with_foo/mod_with_foo_Module.symvers
EXTRA_SYMBOLS += $(COMMON_OUT_DIR)/ddk_module_example/mod_with_foo/mod_with_foo_Module.symvers
modules modules_install clean:
    $(MAKE) -C $(KERNEL_SRC) M=$(M) $(KBUILD_OPTIONS) KBUILD_EXTRA_SYMBOLS="$(EXTRA_SYMBOLS)" $(@)
```

File: bazel-bin/ddk_module_example/mod_using_foo/mod_using_foo_makefiles/makefiles/Kbuild

```
# Build ddk_module_example/mod_using_foo/mod_using_foo.ko
obj-m += mod_using_foo.o

# The module mod_using_foo.ko has a source file mod_using_foo.c

# Common flags for mod_using_foo.o

LINUXINCLUDE := \
  -I$(ROOT_DIR)/common/arch/arm64/include \
  -I$(ROOT_DIR)/common/arch/arm64/include/uapi \
  -I$(ROOT_DIR)/common/include \
  -I$(ROOT_DIR)/common/include/uapi \
  $(LINUXINCLUDE)

CFLAGS_mod_using_foo.o += @$(ROOT_DIR)/ddk_module_example/mod_using_foo/mod_using_foo.cflags
```

File: bazel-bin/ddk_module_example/mod_using_foo/mod_using_foo_makefiles/makefiles/mod_using_foo.cflags

```
-I$(ROOT_DIR)/ddk_module_example/mod_with_foo/include
```

Symbols from `mod_with_foo` included automatically

Common headers from the kernel included here

`ddk_module()` example

- **`bazel build //ddk_module_example/mod_using_foo:mod_using_foo`**

File: bazel-bin/ddk_module_example/mod_using_foo/mod_using_foo_makefiles/makefiles/Makefile

```
# Include symbol: ddk_module_example/mod_with_foo/mod_with_foo_Module.symvers
EXTRA_SYMBOLS += $(COMMON_OUT_DIR)/ddk_module_example/mod_with_foo/mod_with_foo_Module.symvers
modules modules_install clean:
    $(MAKE) -C $(KERNEL_SRC) M=$(M) $(KBUILD_OPTIONS) KBUILD_EXTRA_SYMBOLS="$(EXTRA_SYMBOLS)" $(@)
```

File: bazel-bin/ddk_module_example/mod_using_foo/mod_using_foo_makefiles/makefiles/Kbuild

```
# Build ddk_module_example/mod_using_foo/mod_using_foo.ko
obj-m += mod_using_foo.o

# The module mod_using_foo.ko has a source file mod_using_foo.c

# Common flags for mod_using_foo.o

LINUXINCLUDE := \
  -I$(ROOT_DIR)/common/arch/arm64/include \
  -I$(ROOT_DIR)/common/arch/arm64/include/uapi \
  -I$(ROOT_DIR)/common/include \
  -I$(ROOT_DIR)/common/include/uapi \
  $(LINUXINCLUDE)

CFLAGS_mod_using_foo.o += @$(ROOT_DIR)/ddk_module_example/mod_using_foo/mod_using_foo.cflags
```

File: bazel-bin/ddk_module_example/mod_using_foo/mod_using_foo_makefiles/makefi[illegible]_using_foo.cflags

```
-I$(ROOT_DIR)/ddk_module_example/mod_with_foo/include
```

Symbols from `mod_with_foo` included automatically

Common headers from the kernel included here

Public headers from `mod_with_foo` included

`ddk_module()` example

- **`bazel build //ddk_module_example/mod_using_foo:mod_using_foo`**

```
INFO: Analyzed target //ddk_module_example/mod_using_foo:mod_using_foo (0 packages loaded, 9
INFO: Found 1 target...
INFO: From Building external kernel module (lto=default;trim) @//ddk_module_example/mod_with
Target //ddk_module_example/mod_using_foo:mod_using_foo up-to-date:
  bazel-bin/ddk_module_example/mod_using_foo/mod_using_foo/mod_using_foo.ko
  bazel-bin/ddk_module_example/mod_using_foo/mod_using_foo/mod_using_foo.check_no_remaining
  bazel-bin/ddk_module_example/mod_using_foo/mod_using_foo/mod_using_foo_Module.symvers
INFO: Elapsed time: 430.135s, Critical Path: 424.23s
INFO: 16 processes: 2 internal, 14 linux-sandbox.
INFO: Build completed successfully, 16 total actions
```

- Build completes, and output **`.ko`** is placed in **`bazel-bin`**.

Thank you

Qualcomm

Follow us on:

For more information, visit us at:

qualcomm.com & qualcomm.com/blog