Getting Started

Installing Dependencies

KUnit has the same dependencies as the Linux kernel. As long as you can build the kernel, you can run KUnit.

Running tests with kunit_tool

kunit_tool is a Python script, which configures and builds a kernel, runs tests, and formats the test results. From the kernel repository, you can run kunit_tool:

```
./tools/testing/kunit/kunit.py run
```

For more information on this wrapper, see: Documentation/dev-tools/kunit/run_wrapper.rst.

Creating a `.kunitconfig`

By default, kunit_tool runs a selection of tests. However, you can specify which unit tests to run by creating a `.kunitconfig` file with kernel config options that enable only a specific set of tests and their dependencies. The `.kunitconfig` file contains a list of kconfig options which are required to run the desired targets. The `.kunitconfig` also contains any other test specific config options, such as test dependencies. For example: the `FAT_FS` tests - `FAT_KUNIT_TEST`, depends on `FAT_FS`. `FAT_FS` can be enabled by selecting either `MSDOS_FS` or `VFAT_FS`. To run `FAT_KUNIT_TEST`, the `.kunitconfig` has:

> **System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\dev-tools\kunit\(linux-master)(Documentation)(dev-tools)(kunit)start.rst, line 38)**
>
> Cannot analyze code. No Pygments lexer found for "none".
>
> ```
> .. code-block:: none
>
>         CONFIG_KUNIT=y
>         CONFIG_MSDOS_FS=y
>         CONFIG_FAT_KUNIT_TEST=y
> ```

1. A good starting point for the `.kunitconfig`, is the KUnit default config. Run the command:

```
cd $PATH_TO_LINUX_REPO
cp tools/testing/kunit/configs/default.config .kunitconfig
```

> **Note**
>
> You may want to remove CONFIG_KUNIT_ALL_TESTS from the `.kunitconfig` as it will enable a number of additional tests that you may not want.

2. You can then add any other Kconfig options, for example:

> **System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\dev-tools\kunit\(linux-master)(Documentation)(dev-tools)(kunit)start.rst, line 58)**
>
> Cannot analyze code. No Pygments lexer found for "none".
>
> ```
> .. code-block:: none
>
>         CONFIG_LIST_KUNIT_TEST=y
> ```

Before running the tests, kunit_tool ensures that all config options set in `.kunitconfig` are set in the kernel `.config`. It will warn you if you have not included dependencies for the options used.

> **Note**
>
> If you change the `.kunitconfig`, kunit.py will trigger a rebuild of the `.config` file. But you can edit the `.config` file directly or with tools like `make menuconfig O=.kunit`. As long as its a superset of `.kunitconfig`, kunit.py won't overwrite your changes.

Running Tests (KUnit Wrapper)

1. To make sure that everything is set up correctly, invoke the Python wrapper from your kernel repository:

```
./tools/testing/kunit/kunit.py run
```

If everything worked correctly, you should see the following:

```
Generating .config ...
Building KUnit Kernel ...
Starting KUnit Kernel ...
```

The tests will pass or fail.

> **Note**
>
> Because it is building a lot of sources for the first time, the `Building KUnit kernel` may take a while.

Running Tests without the KUnit Wrapper

If you do not want to use the KUnit Wrapper (for example: you want code under test to integrate with other systems, or use a different/ unsupported architecture or configuration), KUnit can be included in any kernel, and the results are read out and parsed manually.

> **Note**
>
> `CONFIG_KUNIT` should not be enabled in a production environment. Enabling KUnit disables Kernel Address-Space Layout Randomization (KASLR), and tests may affect the state of the kernel in ways not suitable for production.

Configuring the Kernel

To enable KUnit itself, you need to enable the `CONFIG_KUNIT` Kconfig option (under Kernel Hacking/Kernel Testing and Coverage in `menuconfig`). From there, you can enable any KUnit tests. They usually have config options ending in `_KUNIT_TEST`.

KUnit and KUnit tests can be compiled as modules. The tests in a module will run when the module is loaded.

Running Tests (without KUnit Wrapper)

Build and run your kernel. In the kernel log, the test output is printed out in the TAP format. This will only happen by default if KUnit/tests are built-in. Otherwise the module will need to be loaded.

> **Note**
>
> Some lines and/or data may get interspersed in the TAP output.

Writing Your First Test

In your kernel repository, let's add some code that we can test.

1. Create a file `drivers/misc/example.h`, which includes:

```
int misc_example_add(int left, int right);
```

2. Create a file `drivers/misc/example.c`, which includes:

```
#include <linux/errno.h>

#include "example.h"

int misc_example_add(int left, int right)
{
        return left + right;
}
```

3. Add the following lines to `drivers/misc/Kconfig`:

```
config MISC_EXAMPLE
        bool "My example"
```

4. Add the following lines to `drivers/misc/Makefile`:

```
obj-$(CONFIG_MISC_EXAMPLE) += example.o
```

Now we are ready to write the test cases.

1. Add the below test case in `drivers/misc/example_test.c`:

```
#include <kunit/test.h>
#include "example.h"

/* Define the test cases. */

static void misc_example_add_test_basic(struct kunit *test)
{
        KUNIT_EXPECT_EQ(test, 1, misc_example_add(1, 0));
        KUNIT_EXPECT_EQ(test, 2, misc_example_add(1, 1));
        KUNIT_EXPECT_EQ(test, 0, misc_example_add(-1, 1));
        KUNIT_EXPECT_EQ(test, INT_MAX, misc_example_add(0, INT_MAX));
        KUNIT_EXPECT_EQ(test, -1, misc_example_add(INT_MAX, INT_MIN));
}

static void misc_example_test_failure(struct kunit *test)
{
        KUNIT_FAIL(test, "This test never passes.");
}

static struct kunit_case misc_example_test_cases[] = {
        KUNIT_CASE(misc_example_add_test_basic),
        KUNIT_CASE(misc_example_test_failure),
        {}
};

static struct kunit_suite misc_example_test_suite = {
        .name = "misc-example",
        .test_cases = misc_example_test_cases,
};
kunit_test_suite(misc_example_test_suite);
```

2. Add the following lines to `drivers/misc/Kconfig`:

```
config MISC_EXAMPLE_TEST
        tristate "Test for my example" if !KUNIT_ALL_TESTS
        depends on MISC_EXAMPLE && KUNIT=y
        default KUNIT_ALL_TESTS
```

3. Add the following lines to `drivers/misc/Makefile`:

```
obj-$(CONFIG_MISC_EXAMPLE_TEST) += example_test.o
```

4. Add the following lines to `.kunitconfig`:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\dev-tools\kunit\(linux-master)(Documentation)(dev-tools)(kunit)start.rst, line 217)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

        CONFIG_MISC_EXAMPLE=y
        CONFIG_MISC_EXAMPLE_TEST=y
```

5. Run the test:

```
./tools/testing/kunit/kunit.py run
```

You should see the following failure:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\dev-tools\kunit\(linux-master)(Documentation)(dev-tools)(kunit)start.rst, line 230)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

        ...
        [16:08:57] [PASSED] misc-example:misc_example_add_test_basic
        [16:08:57] [FAILED] misc-example:misc_example_test_failure
        [16:08:57] EXPECTATION FAILED at drivers/misc/example-test.c:17
```

```
[16:08:57]      This test never passes.
...
```

Congrats! You just wrote your first KUnit test.

Next Steps

- Documentation/dev-tools/kunit/architecture.rst - KUnit architecture.
- Documentation/dev-tools/kunit/run_wrapper.rst - run kunit_tool.
- Documentation/dev-tools/kunit/run_manual.rst - run tests without kunit_tool.
- Documentation/dev-tools/kunit/usage.rst - write tests.
- Documentation/dev-tools/kunit/tips.rst - best practices with examples.
- Documentation/dev-tools/kunit/api/index.rst - KUnit APIs used for testing.
- Documentation/dev-tools/kunit/kunit-tool.rst - kunit_tool helper script.
- Documentation/dev-tools/kunit/faq.rst - KUnit common questions and answers.