Open Firmware Devicetree Unittest

Author: Gaurav Minocha <gaurav.minocha.os@gmail.com>

1. Introduction

This document explains how the test data required for executing OF unittest is attached to the live tree dynamically, independent of the machine's architecture.

It is recommended to read the following documents before moving ahead.

1. Documentation/devicetree/usage-model.rst
2. http://www.devicetree.org/Device_Tree_Usage

OF Selftest has been designed to test the interface (include/linux/of.h) provided to device driver developers to fetch the device information..etc. from the unflattened device tree data structure. This interface is used by most of the device drivers in various use cases.

2. Verbose Output (EXPECT)

If unittest detects a problem it will print a warning or error message to the console. Unittest also triggers warning and error messages from other kernel code as a result of intentionally bad unittest data. This has led to confusion as to whether the triggered messages are an expected result of a test or whether there is a real problem that is independent of unittest.

'EXPECT : text' (begin) and 'EXPECT / : text' (end) messages have been added to unittest to report that a warning or error is expected. The begin is printed before triggering the warning or error, and the end is printed after triggering the warning or error.

The EXPECT messages result in very noisy console messages that are difficult to read. The script scripts/dtc/of_unittest_expect was created to filter this verbosity and highlight mismatches between triggered warnings and errors vs expected warnings and errors. More information is available from 'scripts/dtc/of_unittest_expect --help'.

3. Test-data

The Device Tree Source file (drivers/of/unittest-data/testcases.dts) contains the test data required for executing the unit tests automated in drivers/of/unittest.c. Currently, following Device Tree Source Include files (.dtsi) are included in testcases.dts:

```
drivers/of/unittest-data/tests-interrupts.dtsi
drivers/of/unittest-data/tests-platform.dtsi
drivers/of/unittest-data/tests-phandle.dtsi
drivers/of/unittest-data/tests-match.dtsi
```

When the kernel is build with OF_SELFTEST enabled, then the following make rule:

```
$(obj)/%.dtb: $(src)/%.dts FORCE
        $(call if_changed_dep, dtc)
```

is used to compile the DT source file (testcases.dts) into a binary blob (testcases.dtb), also referred as flattened DT.

After that, using the following rule the binary blob above is wrapped as an assembly file (testcases.dtb.S):

```
$(obj)/%.dtb.S: $(obj)/%.dtb
        $(call cmd, dt_S_dtb)
```

The assembly file is compiled into an object file (testcases.dtb.o), and is linked into the kernel image.

3.1. Adding the test data

Un-flattened device tree structure:

Un-flattened device tree consists of connected device_node(s) in form of a tree structure described below:

```
// following struct members are used to construct the tree
struct device_node {
    ...
    struct  device_node *parent;
    struct  device_node *child;
    struct  device_node *sibling;
    ...
};
```

Figure 1, describes a generic structure of machine's un-flattened device tree considering only child and sibling pointers. There exists another pointer, `*parent`, that is used to traverse the tree in the reverse direction. So, at a particular level the child node and all the sibling nodes will have a parent pointer pointing to a common node (e.g. child1, sibling2, sibling3, sibling4's parent points to root

node):

```
root ('/')
|
child1 -> sibling2 -> sibling3 -> sibling4 -> null
|         |           |           |
|         |           |          null
|         |           |
|         |        child31 -> sibling32 -> null
|         |           |           |
|         |          null       null
|         |
|      child21 -> sibling22 -> sibling23 -> null
|         |          |            |
|        null       null         null
|
child11 -> sibling12 -> sibling13 -> sibling14 -> null
|           |           |            |
|           |           |           null
|           |           |
null        null       child131 -> null
                        |
                        null
```

Figure 1: Generic structure of un-flattened device tree

Before executing OF unittest, it is required to attach the test data to machine's device tree (if present). So, when selftest_data_add() is called, at first it reads the flattened device tree data linked into the kernel image via the following kernel symbols:

```
__dtb_testcases_begin - address marking the start of test data blob
__dtb_testcases_end   - address marking the end of test data blob
```

Secondly, it calls of_fdt_unflatten_tree() to unflatten the flattened blob. And finally, if the machine's device tree (i.e live tree) is present, then it attaches the unflattened test data tree to the live tree, else it attaches itself as a live device tree.

attach_node_and_children() uses of_attach_node() to attach the nodes into the live tree as explained below. To explain the same, the test data tree described in Figure 2 is attached to the live tree described in Figure 1:

```
root ('/')
    |
testcase-data
    |
test-child0 -> test-sibling1 -> test-sibling2 -> test-sibling3 -> null
    |              |               |                |
test-child01      null            null             null
```

Figure 2: Example test data tree to be attached to live tree.

According to the scenario above, the live tree is already present so it isn't required to attach the root('/') node. All other nodes are attached by calling of_attach_node() on each node.

In the function of_attach_node(), the new node is attached as the child of the given parent in live tree. But, if parent already has a child then the new node replaces the current child and turns it into its sibling. So, when the testcase data node is attached to the live tree above (Figure 1), the final structure is as shown in Figure 3:

```
root ('/')
|
testcase-data -> child1 -> sibling2 -> sibling3 -> sibling4 -> null
|                |          |           |           |
(...)              |          |           |          null
                 |          |         child31 -> sibling32 -> null
                 |          |           |           |
                 |          |          null        null
                 |          |
                 |        child21 -> sibling22 -> sibling23 -> null
                 |          |           |            |
                 |         null        null         null
                 |
                 child11 -> sibling12 -> sibling13 -> sibling14 -> null
                 |          |            |            |
                 null       null          |           null
                                          |
                                          child131 -> null
                                          |
                                          null
-------------------------------------------------------------------------

root ('/')
|
testcase-data -> child1 -> sibling2 -> sibling3 -> sibling4 -> null
|                |          |           |           |
```

```
|             (...)      (...)       (...)        null
|
test-sibling3 -> test-sibling2 -> test-sibling1 -> test-child0 -> null
|                |                |                |
null             null             null         test-child01
```

Figure 3: Live device tree structure after attaching the testcase-data.

Astute readers would have noticed that test-child0 node becomes the last sibling compared to the earlier structure (Figure 2). After attaching first test-child0 the test-sibling1 is attached that pushes the child node (i.e. test-child0) to become a sibling and makes itself a child node, as mentioned above.

If a duplicate node is found (i.e. if a node with same full_name property is already present in the live tree), then the node isn't attached rather its properties are updated to the live tree's node by calling the function update_node_properties().

3.2. Removing the test data

Once the test case execution is complete, selftest_data_remove is called in order to remove the device nodes attached initially (first the leaf nodes are detached and then moving up the parent nodes are removed, and eventually the whole tree). selftest_data_remove() calls detach_node_and_children() that uses of_detach_node() to detach the nodes from the live device tree.

To detach a node, of_detach_node() either updates the child pointer of given node's parent to its sibling or attaches the previous sibling to the given node's sibling, as appropriate. That is it :)