There have been many presentations on what a devicetree looks like and how to create a devicetree. This talk instead examines how the Linux kernel uses a devicetree. Topics include the kernel devicetree framework, device creation, resource allocation, driver binding, and connecting objects. Troubleshooting will consider initialization, allocation, and binding ordering; kernel configuration; and driver problems.
CAUTION

The material covered in this presentation is kernel version specific and is actively evolving.

Most information describes 3.15-rc1 – 3.16-rc7

In cases where arch specific code is involved, I will be looking at arch/arm/
Chapter 1

Device tree
what is device tree?

“A device tree is a tree data structure with nodes that describe the devices in a system. Each node has property/value pairs that describe the characteristics of the device being represented. Each node has exactly one parent except for the root node, which has no parent.”
(ePAPR v1.1)
what is device tree?

“A device tree is a **tree data structure** with nodes that **describe the devices** in a system. Each **node** has property/value pairs that **describe the characteristics of the device** being represented. Each node has exactly one parent except for the **root node**, which has no parent.”

(ePAPR v1.1)

A device tree describes hardware that can not be located by probing.
DT data life cycle

(source)
.dts
model = "Qualcomm APQ8074 Dragonboard";
compatible = "qcom,apq8074-dragonboard";
interrupt-parent = <&intc>;

soc: soc {
    ranges;
    compatible = "simple-bus";

    intc: interrupt-controller@f90000000 {
        compatible = "qcom,msm-qgic2";
        interrupt-controller;
        reg = <0xf90000000 0x1000>,
            <0xf90020000 0x1000>;

    }

    console: serial@f991e000 {
        compatible = "qcom,msm-uartdm-v1.4", "qcom,msm-uartdm";
        reg = <0xf991e0000 0x1000>;
        interrupts = <0 108 0x0>;
    };
}
.dts - device tree source file

Thomas Pettazzoni's ELC 2014 talk
“Device Tree For Dummies” is an excellent introduction to

- device tree source
- boot loader mechanisms
- much more!

  Petazzoni-device-tree-dummies_0.pdf
.dts - device tree source file

Thomas Pettazzoni's ELC 2014 talk
“Device Tree For Dummies” is an excellent introduction to

- device tree source
- boot loader mechanisms
- much more!

  Petazzoni-device-tree-dummies_0.pdf
/* incomplete .dts example */

model = "Qualcomm APQ8074 Dragonboard";
compatible = "qcom,apq8074-dragonboard";
interrupt-parent = <&intc>;

soc: soc {
    ranges;
    compatible = "simple-bus";
}

intc: interrupt-controller@f9000000 { 
    compatible = "qcom,msm-qgic2";
    interrupt-controller;
    reg = <0xf9000000 0x1000>,
         <0xf9002000 0x1000>;
}

serial@f991e000 { 
    compatible = "qcom,msm-uartdm-v1.4", "qcom,msm-uartdm";
    reg = <0xf991e000 0x1000>;
    interrupts = <0 108 0x0>;
};
Key vocabulary

nodes
- the tree structure
- contain properties and other nodes

properties
- data values providing information about a node

node '/': property 'compatible'
- will be used to match a machine_desc entry

other nodes: property 'compatible'
- will be used to match a driver
DT data life cycle

(source) .dts → (compiler) dtc → (binary blob) .dtb
DT data life cycle

(source) .dts → (compiler) dtc → (binary blob) .dtb
Binary Blob format

A “flat” format

Access via serial scan and offsets
## Binary Blob format

<table>
<thead>
<tr>
<th>struct fdt_header</th>
<th>info</th>
</tr>
</thead>
<tbody>
<tr>
<td>(free space)</td>
<td>offsets to blocks</td>
</tr>
<tr>
<td>memory reservation block</td>
<td>section sizes</td>
</tr>
<tr>
<td>(free space)</td>
<td>{address, size} tuples</td>
</tr>
<tr>
<td>structure block</td>
<td>nested nodes</td>
</tr>
<tr>
<td>(free space)</td>
<td>- name embedded</td>
</tr>
<tr>
<td>strings block</td>
<td>properties nested in nodes</td>
</tr>
<tr>
<td>(free space)</td>
<td>- values embedded</td>
</tr>
<tr>
<td></td>
<td>- names are offsets in 'strings'</td>
</tr>
<tr>
<td></td>
<td>property names</td>
</tr>
<tr>
<td></td>
<td>- null terminated strings</td>
</tr>
<tr>
<td></td>
<td>- concatenated</td>
</tr>
</tbody>
</table>
DT data life cycle

(source) .dts ➔ (compiler) dtc ➔ (binary blob) .dtb

boot loader: dtb ➔ dtb'

boot image:
  vmlinux
dtb

memory:
  FDT (flattened device tree)
DT data life cycle

(source) .dts → (compiler) dtc → (binary blob) .dtb

boot loader: dtb

boot image:

Vmlinux
dtb

memory:
FDT
(flattened device tree)

linux kernel
Flattened Device Tree format

A “flat” format.

Access via serial scan and offsets using fdt_*() functions.
Flattened Device Tree format

A “flat” format.

Access via serial scan and offsets using fdt_*() functions.
DT data life cycle

(source) .dts -> (compiler) dtc -> (binary blob) .dtb

boot loader: dtb -> dtb'

boot image: vmlinux

dtb

memory: FDT
(flattened device tree)
expanded DT

linux kernel
Expanded format

A “computer-sciency” data structure

Plan is to change implementation soon
- interfaces to access data should not change
- implications for DT usage should not change
Expanded format

A “computer-sciency” data structure

Exact details not too interesting unless you are maintaining the DT internals, so I will mostly not leave the slides up long enough for you to read the details. If you are interested, read the slides at home.

I will be trying to emphasize concepts instead of details.
Expanded format

A tree data structure

Access and modified via tree operations using of_*() functions

Access all nodes via a linked list

Created during boot

Nodes and properties can be added or deleted after boot
Expanded format

A tree data structure

Access and modified via tree operations using of_*() functions

Access all nodes via a linked list

Created during boot

Nodes and properties can be added or deleted after boot
Expanded format

struct device_node {
    const char *name;
    const char *type;
    phandle phandle;
    const char *full_name;

    struct property *properties;
    struct property *deadprops;
    struct device_node *parent;
    struct device_node *child;
    struct device_node *sibling;
    struct device_node *next;
    struct device_node *allnext;
    struct kobject kobj;
    unsigned long _flags;
    void    *data;

#if defined(CONFIG_SPARC)
    ...
#endif
};
Expanded format

tree of struct device_node

struct device_node {

    struct property *properties;
    struct device_node *parent;
    struct device_node *child;
    struct device_node *sibling;
    struct device_node *allnext;

};
Expanded format

tree of struct device_node

Tree pointers

child pointer

sibling pointer
Expanded format

```
  of_allnodes
```

```plaintext
tree of struct device_node
```

```
  child
```

```
  sibling
```

Expanded format

Tree of struct device_node

Tree pointers

  child pointer

  sibling pointer

Used to find node by tree search

  of_find_node_by_path()
Expanded format

tree of struct device_node

Global linked list pointer

allnext pointer

removal targeted for 3.19

- API: unchanged
- implementation: depth first traversal of tree
Expanded format

tree of struct device_node

child

sibling

allnext
allnext linked list

Follows a depth first traversal of the tree

After boot: YES

After dynamic node addition: NO

To be safe, think of allnext as a randomly ordered linked list of all nodes.
allnext linked list - internal usage

Common pattern:

```c
of_find_by_XXX(struct device node *from, ...) {
    np = from ? from->allnext : of_allnodes;
    for (; np; np = np->allnext)
        ...
}
```

If 'from' is NULL then search from of_allnodes, else search from a specific starting point
Common pattern:

```c
of_find_by_XXX(struct device node *from, ...) {
    np = from ? from->allnext : of_allnodes;
    for (; np; np = np->allnext)
        ...
```

If 'from' is NULL then search from of_allnodes, else search from a specific starting point
allnext linked list - internal usage

Find nodes by attribute

```c
struct device_node {
    const char *name;
    const char *type;
    phandle phandle;
    const char *full_name;
}
```
allnext linked list - internal usage

Find nodes by attribute

of_find_node_by_name     (*from, ...)  
of_find_node_by_type     (*from, ...)  
of_find_node_by_phandle  (handle)  
          handle is unique, so *from not needed  
of_find_node_with_property (*from, ...)  
         traverse allnext and properties
allnext linked list - internal usage

Find nodes by attribute

of_find_node_by_name       (*from, …)
of_find_node_by_type       (*from, …)
of_find_node_by_phandle    (handle)

handle is unique, so *from not needed

of_find_node_with_property (*from, …)

traverse allnext and properties
allnext linked list

Properties 'name' and 'device_type' are special.

```plaintext
memory { device_type = "memory"; reg = <0 0>; }
```

In addition to existing on the node's properties list, these properties are hoisted into:

```plaintext
device_node.name
device_node.type
```
Expanded format

parent pointer

tree of struct device_node
Expanded format  

`tree of struct device_node`

- **parent**
- **child**
- **sibling**
- **allnext**
Expanded format

properties pointer

struct property {
  char     *name;
  int      length;
  void     *value;
  struct property  *next;
  unsigned long     _flags;
  unsigned int     unique_id;
  struct bin_attribute attr;
};
Expanded format

tree of struct device_node

child

sibling

properties

next
Chapter 2

Matching boot customization options to the device tree

Kernel boot
A struct machine_desc describes boot customizations for a specific device tree.

A struct machine_desc may be used for several different device trees.
A struct machine_desc describes boot customizations for a specific device tree.
A struct machine_desc may be used for several different device trees.
struct machine_desc {
    unsigned int          nr;          /* architecture */
    const char            *name;       /* architecture */
    unsigned long         atag_offset;
    char  *dt_compat;                  /* 'compatible' strings */
    unsigned int          nr_irqs;
    phys_addr_t           dma_zone_size;
    ...
    enum reboot_mode      reboot_mode;
    unsigned              l2c_aux_val;  /* L2 cache */
    unsigned              l2c_aux_mask; /* L2 cache */
    ...
    struct smp_operations *smp;
    ...
    void                  (*init_XXX)();
    ...
}
struct machine_desc {
    ...
    void (*l2c_write_sec)();
    bool (*smp_init)();
    void (*fixup)();
    void (*dt_fixup)();
    void (*init_meminfo)();
    void (*reserve)();
    void (*map_io)();
    void (*init_early)();
    void (*init_irq)();
    void (*init_time)();
    void (*init_machine)();
    void (*init_late)();
    void (*handle_irq)();
    void (*restart)();
}
machine_desc runtime hooks

struct machine_desc {
    ...
    void   (*l2c_write_sec)();
    ...
    void   (*handle_irq)();
    void   (*restart)();
machine_desc - populating array

#define DT_MACHINE_START(_name, _namestr) \
static const struct machine_desc __mach_desc_##_name \
__used \
__attribute__(((__section__(".arch.info.init")))) = { \
    .nr = ~0, \
    .name = _namestr, \
};

#define MACHINE_END
Minimalist Example

```c
static const char * const qcom_dt_match[] __initconst = {
    "qcom,apq8074-dragonboard",
    "qcom,apq8084",
    "qcom,msm8660-surf",
    NULL
};

DT_MACHINE_START(QCOM_DT, "Qualcomm (Flattened Device Tree)")
    .dt_compat = qcom_dt_match,
MACHINE_END
```
#define DT_MACHINE_START(_name, _namestr)        \
static const struct machine_desc __mach_desc_##_name = {  \
        .name           = _namestr, \

DT_MACHINE_START(QCOM_DT, "Qualcomm (Flattened Device Tree)")
        .dt_compat = qcom_dt_match,
MACHINE_END

$ cat /proc/cpuinfo | grep Hardware
Hardware : Qualcomm (Flattened Device Tree)

System.map:
    c0905c5c T __arch_info_begin
    c0905c5c t __mach_desc_QCOM_DT
    c0905db4 T __arch_info_end
machine_desc - populating array

multiple machine_desc example

DT_MACHINE_START(QCOM_DT, "Qualcomm (Flattened Device Tree)"
            .dt_compat = qcom_dt_match,
MACHINE_END

#define CONFIG_ARCH_MULTIPLATFORM
   DT_MACHINE_START(GENERIC_DT, "Generic DT based system")
MACHINE_END
#endif
machine_desc - populating array

multiple machine_desc example

DT_MACHINE_START(QCOM_DT, "Qualcomm (Flattened Device Tree)")
   .dt_compat = qcom_dt_match,
MACHINE_END

DT_MACHINE_START(GENERIC_DT, "Generic DT based system")
MACHINE_END

Result in System.map from linker magic:

  c0905c5c T __arch_info_begin
  c0905c5c t __mach_desc_GENERIC_DT.18665
  c0905cb4 t __mach_desc_QCOM_DT
  c0905d0c T __arch_info_end
machine_desc - best match

Selecting the machine_desc that best matches the Device Tree loaded by the bootloader
The struct `machine_desc` with
- a matching compatible
- that is the leftmost in the Device Tree root node compatible list
best match - Device Tree

First DT source example was not complex enough so here is a more complex example

/ {
    model = "TI Zoom3";
    compatible = "ti,omap3-zoom3", "ti,omap36xx", "ti,omap3";
};
DT_MACHINE_START(OMAP3_DT, "Generic OMAP3 (Flattened Device Tree")
    .init_early     = omap3430_init_early,
    .dt_compat      = omap3_boards_compat,
MACHINE_END
DT_MACHINE_START(OMAP36XX_DT, "Generic OMAP36xx (Flattened Device Tree")
    .init_early     = omap3630_init_early,
    .dt_compat      = omap36xx_boards_compat,
MACHINE_END

static const char *omap3_boards_compat[] __initconst = {
    "ti,omap3430",
    "ti,omap3",
    NULL,
};
static const char *omap36xx_boards_compat[] __initconst = {
    "ti,omap36xx",
    NULL,
};
/ {
    model = "TI Zoom3";
    compatible = "ti,omap3-zoom3", "ti,omap36xx", "ti,omap3";
};

static const char *omap3_boards_compat[] __initconst = {
    "ti,omap3430",
    "ti,omap3",
    "ti,omap3",
    NULL,
};

static const char *omap36xx_boards_compat[] __initconst = {
    "ti,omap36xx",
    NULL,
};
/ {
    model = "TI Zoom3";
    compatible = "ti,omap3-zoom3", "ti,omap36xx", "ti,omap3";
};

DT_MACHINE_START(OMAP36XX_DT, "Generic OMAP36xx (Flattened Device Tree)"
    .init_early = omap3630_init_early,
    .dt_compat = omap36xx_boards_compat,
MACHINE_END

static const char *omap36xx_boards_compat[] __initconst = {
    "ti,omap36xx",
    NULL,
};
The struct `machine_desc` with
- a matching compatible
- that is the leftmost in the Device Tree
  root node compatible list
Why provide options for ranking?

/ {
    model = "TI Zoom3";
    compatible = "ti,omap3-zoom3", "ti,omap36xx", "ti,omap3";
};

Allows a newer kernel to customize the boot for specific classes or models of older hardware with a baked in device tree.
Why provide options for ranking?

Allows a newer kernel to customize the boot for specific classes or models of older hardware with a baked in device tree.

Not intended to allow shipping half baked device tree instances.
My pseudocode conventions

Will obviously fail to compile
Will usually not show function arguments
Each level of indentation indicates either
  body of control statement (if, while, etc)
  entry into function listed on previous line
Double indentation indicates an intervening
level of function call is not shown
Will often leave out many details or fabricate
specific details in the interest of simplicity
struct machine_desc {
    ...
    void (*l2c_write_sec)();
    bool (*smp_init)();
    void (*fixup)();
    void (*dt_fixup)();
    void (*init_meminfo)();
    void (*reserve)();
    void (*map_io)();
    void (*init_early)();
    void (*init_irq)();
    void (*init_time)();
    void (*init_machine)();
    void (*init_late)();
    void (*handle_irq)();
    void (*restart)();
}
machine_desc runtime hooks

struct machine_desc {
    ...
    void (*l2c_write_sec)();
    ...
    void (*handle_irq)();
    void (*restart)();
}
static const char * const tegra_dt_board_compat[] = {
    "nvidia,tegra124",
    "nvidia,tegra114",
    "nvidia,tegra30",
    "nvidia,tegra20",
    NULL
};

DT_MACHINE_START(TEGRA_DT, "NVIDIA Tegra SoC (Flattened Device Tree)
    .l2c_aux_val    = 0x3c400001,
    .l2c_aux_mask   = 0xc20fc3fe,
    .smp            = smp_ops(tegra_smp_ops),
    .map_io         = tegra_map_common_io,
    .init_early     = tegra_init_early,
    .init_irq       = tegra_dt_init_irq,
    .init_machine   = tegra_dt_init,
    .init_late      = tegra_dt_init_late,
    .restart        = tegra_pmc_restart,
    .dt_compat      = tegra_dt_board_compat,
MACHINE_END
machine_desc hooks (all)

```c
start_kernel()
    pr_notice("%s", linux_banner)
setup_arch()
    mdesc = setup_machine_fdt(__atags_pointer)
    mdesc = of_flat_dt_match_machine()
    /* sometimes firmware provides buggy data */
    mdesc->dt_fixup()
early_paging_init()
    mdesc->init_meminfo()
arm_memblock_init()
    mdesc->reserve()
paging_init()
    devicemaps_init()
    mdesc->map_io()
...
    arm_pm_restart = mdesc->restart
unflatten_device_tree() <================
    if (mdesc->smp_init())
...
    handle_arch_irq = mdesc->handle_irq
...
    mdesc->init_early()
pr_notice("Kernel command line: %s\n", ...)
init_IRQ()
    machine_desc->init_irq()
    outer_cache.write_sec = machine_desc->l2c_write_sec
time_init()
    machine_desc->init_time()
rest_init()
    kernel_thread(kernel_init, ...)
    kernel_init()
    do_initcalls()
    customize_machine()
    machine_desc->init_machine()
    // device probing, driver binding
    init_machine_late()
    machine_desc->init_late()
```
machine_desc hooks (0 of 3)

start_kernel()
    pr_notice("%s", linux_banner)
setup_arch()
    mdesc = setup_machine_fdt(__atags_pointer)
    mdesc = of_flat_dt_match_machine()

    /*
    * Iterate through machine match
    * tables to find the best match for
    * the machine compatible string in
    * the FDT.
machine_desc hooks (1 of 3)

start_kernel()
  pr_notice("%s", linux_banner)
setup_arch()
  mdesc = setup_machine_fdt(__atags_pointer)
  mdesc = of_flat_dt_match_machine()
  /* sometimes firmware provides buggy data */
  mdesc->dt_fixup()
early_paging_init()
  mdesc->init_meminfo()
arm_memblock_init()
  mdesc->reserve()
paging_init()
  devicemaps_init()
  mdesc->map_io()
...
  arm_pm_restart = mdesc->restart
unflatten_device_tree()  <==============
unflatten_device_tree()  
if (mdesc->smp_init())
...
handle_arch_irq = mdesc->handle_irq
...
mdesc->init_early()
/* end of setup_arch() */
pr_notice("Kernel command line: %s\n", ...)
init_IRQ()
    machine_desc->init_irq()
outer_cache.write_sec =
    machine_desc->l2c_write_sec

time_init()
    machine_desc->init_time()
machine_desc hooks (3 of 3)

rest_init()
    kernel_thread(kernel_init, ...)
    kernel_init()
    do_initcalls()
    customize_machine()
    machine_desc->init_machine()
    // device probing, driver binding
    init_machine_late()
    machine_desc->init_late()
machine_desc magic values

Bug or Feature?

Values that affect the Device Tree boot process
machine_desc magic values

Bug or Feature?

```c
struct machine_desc {
    ...
    unsigned    l2c_aux_val;    /* L2 cache */
    unsigned    l2c_aux_mask;   /* L2 cache */
```

A magic value is required to enable check for compatible L2 nodes in the device tree:

```c
.l2c_aux_val = 0,    // default value
.l2c_aux_mask = ~0,  // not default value
```
machine_desc magic values

struct machine_desc {
    ...
    unsigned l2c_aux_val; /* L2 cache */
    unsigned l2c_aux_mask; /* L2 cache */
}

Examples:

DT_MACHINE_START(ROCKCHIP_DT, "Rockchip Cortex-A9 (Device Tree)"
    .l2c_aux_val = 0,
    .l2c_aux_mask = ~0,
    .dt_compat = rockchip_board_dt_compat,

DT_MACHINE_START(TEGRA_DT, "NVIDIA Tegra SoC (Flattened Device Tree)"
    .l2c_aux_val = 0x3c400001,
    .l2c_aux_mask = 0xc20fc3fe,
    ...
    .dt_compat = tegra_dt_board_compat,
machine_desc magic values

init_irq()
    if (... && (l2c_aux_mask || l2c_aux_val))
        l2x0_of_init(l2c_aux_val, l2c_aux_mask)
        np = of_find_matching_node(NULL, l2x0_ids)
        data = of_match_node(l2x0_ids, )->data
        if (...)
            data->of_parse(np, &aux_val, &aux_mask)
            // example of_parse()
            val = AAA
            mask = BBB
            over_ride = of_property_read_bool(, "wt-override")
            if (over_ride)
                val |= CCC
                mask |= DDD
                *aux_val &= mask
                *aux_mask |= val
                *aux_mask &= ~mask
        __l2c_init(, aux_val, aux_mask, )
    /* aux_mask: bits we preserve, aux_val: bits we set */
init_irq()
    if (... && (l2c_aux_mask || l2c_aux_val))
    l2x0_of_init(l2c_aux_val, l2c_aux_mask)
    np = of_find_matching_node(NULL, l2x0_ids)
    data = of_match_node(l2x0_ids, )->data
    if (...)
        data->of_parse(np, &aux_val, &aux_mask)
        // example of_parse()
        val = AAA
        mask = BBB
        over_ride = of_property_read_bool(, "wt-override")
        if (over_ride)
            val |= CCC
            mask |= DDD
        *aux_val &= mask
        *aux_mask |= val
        *aux_mask &= ~mask
    __l2c_init(, aux_val, aux_mask, )
    /* aux_mask: bits we preserve, aux_val: bits we set
init_irq()
if (...) && (l2c_aux_mask || l2c_aux_val)
    l2x0_of_init(l2c_aux_val, l2c_aux_mask)
    np = of_find_matching_node(NULL, l2x0_ids)
    data = of_match_node(l2x0_ids, )->data
    if (...)
        data->of_parse(np, &aux_val, &aux_mask)
        // example of_parse()
        val  = AAA
        mask = BBB
        over_ride = of_property_read_bool(, "wt-override")
        if (over_ride)
            val  |= CCC
            mask |= DDD
            *aux_val  &= mask
            *aux_mask |= val
            *aux_mask &= ~mask
    __l2c_init(, aux_val, aux_mask, )
    /* aux_mask: bits we preserve, aux_val: bits we set */
Takeaway

The struct machine_desc with the best match for the Device Tree root node compatible string is chosen.

The values in the struct machine_desc can alter the boot process.

Minimize use of machine_desc hooks.
Takeaway

The struct `machine_desc` with the best match for the Device Tree root node compatible string is chosen

The values in the struct `machine_desc` can alter the boot process

Minimize use of `machine_desc` hooks
Chapter 3

More kernel boot

Creating devices

Matching devices and drivers
Chapter 3.1

More kernel boot

Creating devices

Matching devices and drivers
Initcalls

Previous pseudo-code of boot is oversimplified, but I will continue with this deception for a few more slides:

```c
do_initcalls()
customize_machine()
if (machine_desc->init_machine)
    machine_desc->init_machine()
else
    of_platform_populate()
// driver binding
init_machine_late()
machine_desc->init_late()
```
Initcalls

But one clue about the deception - initcalls occur in this order:

```c
char *initcall_level_names[] = {
  "early",
  "core",
  "postcore",
  "arch",
  "subsys",
  "fs",
  "device",
  "late",
};
```
Initcall - of_platform_populate()

if (machine_desc->init_machine)
    machine_desc->init_machine()
    /* this function will call
       * of_platform_populate() */
else
    of_platform_populate()

Watch out for board specific data passed in of_platform_populate(, lookup,,,)

See the struct of_dev_auxdata header comment in include/linux/of_platform.h regarding device names and providing platform data
initcall - of_platform_populate()

of_platform_populate(, NULL,,)
    for each child of DT root node
        rc = of_platform_bus_create(child, matches, lookup, parent, true)
        if (node has no 'compatible' property)
            return
        auxdata = lookup[X], where:
            # lookup[X]->compatible matches node compatible property
            # lookup[X]->phys_addr matches node resource 0 start
        if (auxdata)
            bus_id = auxdata->name
            platform_data = auxdata->platform_data
        dev = of_platform_device_create_pdata(, bus_id, platform_data, )
        dev = of_device_alloc(np, bus_id, parent)
        dev->dev.bus = &platform_bus_type
        dev->dev.platform_data = platform_data
        of_device_add(dev)
            bus_probe_device()
                ret = bus_for_each_drv(, __device_attach)
                error = __device_attach()
                if (!driver_match_device()) return 0
                return driver_probe_device()
        if (node 'compatible' property != "simple-bus")
            return 0
    for_each_child_of_node(bus, child)
        rc = of_platform_bus_create()
        if (rc) break
    if (rc) break
initcall - of_platform_populate()

of_platform_populate(, NULL,,,) /* lookup is NULL */
for each child of DT root node
rc = of_platform_bus_create(child, )
if (node has no 'compatible' property)
    return

<< create platform device for node >>
<< try to bind a driver to device >>

if (node 'compatible' property != "simple-bus")
    return 0
for_each_child_of_node(bus, child)
    rc = of_platform_bus_create(child, )
    if (rc) break
if (rc) break
<< create platform device for node >>
<< try to bind a driver to device >>

auxdata = lookup[X], with matches:
  lookup[X]->compatible == node 'compatible' property
  lookup[X]->phys_addr == node resource 0 start
if (auxdata)
  bus_id = auxdata->name
  platform_data = auxdata->platform_data
dev = of_platform_device_create_pdata(', bus_id,
                                      platform_data,)

dev = of_device_alloc(', bus_id,)
dev->dev.bus = &platform_bus_type
dev->dev.platform_data = platform_data
of_device_add(dev)
  bus_probe_device()
  ret = bus_for_each_drv(', __device_attach)
  error = __device_attach()
  if (!driver_match_device())
    return 0
  return driver_probe_device()
initcall - of_platform_populate()

platform device created for
- children of root node
- recursively for deeper nodes if 'compatible' property == “simple-bus”

platform device not created if
- node has no 'compatible' property
initcall - of_platform Populate()

platform device created for
- children of root node
- recursively for deeper nodes if 'compatible' property == "simple-bus"

platform device not created if
- node has no 'compatible' property
initcall - of_platform_populate()

auxdata may affect how the platform device was created
initcall - of_platform_populate()

auxdata may affect how the platform device was created
Drivers may be bound to the devices during platform device creation if driver registered at an earlier initcall level

- the driver called platform_driver_register() from a core_initcall() or a postcore_initcall()
- the driver called platform_driver_register() from an arch_initcall() that was called before of_platform_populate()
Drivers may be bound to the devices during platform device creation if driver registered at an earlier initcall level

- the driver called platform_driver_register() from a core_initcall() or a postcore_initcall()

- the driver called platform_driver_register() from an arch_initcall() that was called before of_platform_populate()
Creating other devices

Devices that are not platform devices were not created by `of_platform_populate()`.

These devices are typically non-discoverable devices sitting on more remote busses. For example:

- i2c
- SoC specific busses
Creating other devices

Devices that are not platform devices were not created by of_platform_populate().

These devices are typically created by the bus driver probe function
Creating other devices

Devices that are not platform devices were not created by of_platform_populate().

These devices are typically created by the bus driver probe function.
Chapter 3.2

More kernel boot

Creating devices

Matching devices and drivers
initcall - // driver binding

```
platform_driver_register()
    driver_register()
    while (dev = iterate over devices on the platform_bus)
        if (!driver_match_device()) return 0
        if (dev->driver) return 0
        driver_probe_device()
        really_probe(dev, drv)
            ret = pinctrl_bind_pins(dev)
            if (ret)
                goto probe_failed
            if (dev->bus->probe)
                ret = dev->bus->probe(dev)
                if (ret) goto probe_failed
            else if (drv->probe)
                ret = drv->probe(dev)
                if (ret) goto probe_failed
        driver_bound(dev)
        driver_deferred_probe_trigger()
        if (dev->bus)
            blocking_notifier_call_chain()
```
initcall - // driver binding

Reformatting the previous slide to make it more readable (see next slide)
initcall - // driver binding

platform_driver_register()
  while (dev = iterate over devices on platform_bus)
    if (!driver_match_device()) return 0
    if (dev->driver) return 0
  driver_probe_device()
    really_probe(dev, drv)
      ret = pinctrl_bind_pins(dev)
      if (ret)
        goto probe_failed
      if (dev->bus->probe)
        ret = dev->bus->probe(dev)
        if (ret) goto probe_failed
      else if (drv->probe)
        ret = drv->probe(dev)
        if (ret) goto probe_failed
  driver_bound(dev)
  driver_deferred_probe_trigger()
  if (...) blocking_notifier_call_chain()
Non-platform devices

When a bus controller driver probe function creates the devices on its bus, the device creation will result in the device probe function being called if the device driver has already been registered.

Note the potential interleaving between device creation and driver binding.
Non-platform devices

When a bus controller driver probe function creates the devices on its bus, the device creation will result in the device probe function being called if the device driver has already been registered.

Note the potential interleaving between device creation and driver binding.
Getting side-tracked

Some deeper understanding of initcalls will be required to be able to explain driver_deferred_probe_trigger()
Initcalls

Previous pseudo-code is oversimplified:

do_initcalls()
  customize_machine()
  if (machine_desc->init_machine)
    machine_desc->init_machine()
  else
    of_platform_populate()
    // device probing, driver binding
  init_machine_late()
  machine_desc->init_late()
Initcalls - actual implementation

do_initcalls()
    for (level = 0; level < ...; level++)
        do_initcall_level(level)
            for (fn = ...; fn < ...; fn++)
                do_one_initcall(*fn)
                    ret = rn()
Initcalls

static initcall_t *initcall_levels[] = {
    __initcall0_start,
    __initcall1_start,
    __initcall2_start,
    __initcall3_start,
    __initcall4_start,
    __initcall5_start,
    __initcall6_start,
    __initcall7_start,
    __initcall_end,
};
Initcalls - order of execution

Pointers to functions for each init level are grouped together by linker scripts
Example \${KBUILD_OUTPUT}/System.map:

\texttt{c0910edc} T \texttt{__initcall0_start}

\texttt{c0910edc} t \texttt{__initcall_ipc_ns_init0}

\texttt{c0910ee0} t \texttt{__initcall_init_mmap_min_addr0}

\texttt{c0910ee4} t \texttt{__initcall_net_ns_init0}

\texttt{c0910ee8} T \texttt{__initcall1_start}

... 

\texttt{c0910f50} T \texttt{__initcall2_start}

...

\texttt{c0910f84} T \texttt{__initcall3_start}

...

...

\texttt{c0911310} T \texttt{__initcall7_start}

...

\texttt{c0911368} T \texttt{__con_initcall_start}

...

\texttt{c0911368} T \texttt{__initcall_end}
Initcalls - order of execution

The order of initcall functions within an init level should be considered to be non-deterministic.
Initcalls - order of execution

The order of initcall functions within an init level should be considered to be non-deterministic

- Whether a probe will defer may be based on when hardware becomes available
- Initcalls are allowed to start asynchronous activity
- Option to scramble the order of device_initcalls has been proposed to increase test coverage of handling probe defer dependencies
Initcalls - order of execution

If you suspect that an initcall ordering is resulting in interdependent drivers failing to probe, then ordering can be determined by:

- add 'initcall_debug' to the kernel command line to print each initcall to console as it is called
- examining the order in System.map (does not account for deferred probes)
Initcalls - order of execution

If you suspect that an initcall ordering is resulting in interdependent drivers failing to probe, then the solution is **NOT** to play games to re-order them.

The solution is to use deferred probe.
Deferred Probe - driver example

```
serial_omap_probe()
    uartirq = irq_of_parse_and_map()
    if (!uartirq)
        return -EPROBE_DEFER
```

A required resource is not yet available, so the driver needs to tell the probe framework to defer the probe until the resource is available
Deferred Probe - probe framework

really_probe()
    if (dev->bus->probe)
        ret = dev->bus->probe()
        if (ret) goto probe_failed
    driver_deferred_probe_trigger()
    goto done

probe_failed:
    if (ret == -EPROBE_DEFER)
        dev_info("Driver %s requests probe" 
                "deferral
", drv->name)
        driver_deferred_probe_add(dev);
    /* trigger occur while probing? */
    if (local_trigger_count != ...)
        driver_deferred_probe_trigger()
Deferred Probe - probe framework

really_probe()
    if (dev->bus->probe)
        ret = dev->bus->probe()
        if (ret) goto probe_failed
    driver_deferred_probe_trigger()
    goto done

probe.failed:
    if (ret == -EPROBE_DEFER)
        dev_info("Driver %s requests probe\n  "deferral\n", drv->name)
        driver_deferred_probe_add(dev);
    /* trigger occur while probing? */
    if (local_trigger_count != ...)
        driver_deferred_probe_trigger()
Deferred Probe - probe framework

```c
really_probe()
    if (dev->bus->probe)
        ret = dev->bus->probe()
        if (ret) goto probe_failed
    driver_deferred_probe_trigger()
    goto done
probe_failed:
    if (ret == -EPROBE_DEFER)
        dev_info("Driver %s requests probe" 
            "deferral\n", drv->name)
        driver_deferred_probe_add(dev);
    /* trigger occur while probing? */
    if (local_trigger_count != ...)
        driver_deferred_probe_trigger()
```
Deferred Probe - probe framework

driver_deferred_probe_trigger()

/*
 * A successful probe means that all the devices in the pending list should be triggered to be reprobed. Move all the deferred devices into the active list so they can be retried by the workqueue
 */
Deferred Probe - probe framework

driver_deferred_probe_trigger()

Called when:
- a driver is bound
- a new device is created
- as a late_initcall: deferred_probe_initcall()

The framework does not know if the resource(s) required by a driver are now available. It just blindly retries all of the deferred probes.
Deferred Probe - probe framework

driver_deferred_probe_trigger()

Called when:
- a driver is bound
- a new device is created
- as a late_initcall: deferred_probe_initcall()

The framework does not know if the resource(s) required by a driver are now available. It just blindly retries all of the deferred probes.
Initcalls - parallelism support

Additional *_sync level added after each other level to allow asynchronous activity to complete before beginning next level

For details:
https://lkml.org/lkml/2006/10/27/157
Initcalls - initcall level #defines

core_initcall(fn)          __define_initcall(fn, 1)
core_initcall_sync(fn)     __define_initcall(fn, 1s)
postcore_initcall(fn)      __define_initcall(fn, 2)
postcore_initcall_sync(fn) __define_initcall(fn, 2s)
arch_initcall(fn)          __define_initcall(fn, 3)
arch_initcall_sync(fn)     __define_initcall(fn, 3s)
subsys_initcall(fn)        __define_initcall(fn, 4)
subsys_initcall_sync(fn)   __define_initcall(fn, 4s)
fs_initcall(fn)            __define_initcall(fn, 5)
fs_initcall_sync(fn)       __define_initcall(fn, 5s)
rootfs_initcall(fn)        __define_initcall(fn, rootfs)
device_initcall(fn)        __define_initcall(fn, 6)
device_initcall_sync(fn)   __define_initcall(fn, 6s)
late_initcall(fn)          __define_initcall(fn, 7)
late_initcall_sync(fn)     __define_initcall(fn, 7s)
Initcalls - driver register, device create

what is in the kernel now:

"core" : platform_driver_register()
"postcore" : platform_driver_register()
"subsys" : platform_driver_register()
"arch" : customize_machine()
of_platform_populate()
"device" : platform_driver_register()
Initcalls - driver register, device create

what is likely to be accepted for new code:

“arch” : customize machine()
of_platform Populate()

“device” : platform_driver_register()
Chapter 4

Miscellaneous
Some DT issues

Circular dependencies on driver probe
   Use a third driver to resolve the conflict

Devices with multiple compatible strings
   If multiple drivers match one of the values then
   the first one to probe is bound. The winner is
   based on the arbitrary initcall order.

A better result would be for the driver with the
most specific compatible to be bound.
The Near Future and Near Past
GIT PULL request for v3.17

Preparation for device tree overlay support
- notifiers moved from before to after change
- batch changes
- notifiers emitted after entire batch applied
- unlocked versions of node and property
  add / remove functions (caller ensures locks)

Enable console on serial ports specified by
/chosen/stdout-path

Data for DT unit tests no longer in the booted dtb
- dynamically loaded before tests
- dynamically unloaded after tests
partial TODO (v3.17 pull request)

=== General structure ===
- Switch from custom lists to (h)list_head for nodes and properties structure
- Remove of_allnodes list and iterate using list of child node alone

=== CONFIG_OF_DYNAMIC ===
- Switch to RCU for tree updates and get rid of global spinlock
- Always set ->full_name at of_attach_node() time
v3.18 and beyond

Preparation for device tree overlay support
- Device Tree Dynamic Resolver (at v2)
- more...

Internal data structure implementation changes
Some things not covered

- Memory, IRQs, clocks
- pinctrl
- Devices and busses other than platform devices
- sysfs
- locking and reference counting
- '/chosen' node
- details of matching machine desc to device tree
- dynamic node and property addition / deletion
- smp operations
- device tree self test
Review

- life cycle of device tree data, data structures access functions change after blob expansion
- customizing the boot process by machine_desc uses best machine_desc match to device tree
- device creation
- driver binding
- ordering dance between device creation and various drivers binding, deferred probes

You should now be able to better understand Documentation/devicetree/usage-model.txt
THE END

Thank you for your attention...
Questions?
How to get a copy of the slides

1) leave a business card with me

2) frank.rowand@sonymobile.com