devicetree: kernel internals and practical troubleshooting

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.

Frank Rowand, Sony Mobile Communications

August 22, 2014

140821_1927

CAUTION

The material covered in this presentation is kernel version specific

Most information describes 3.16 or earlier

In cases where arch specific code is involved, there will be a bias to looking at arch/arm/

Chapter 1

Device tree

what is device tree?

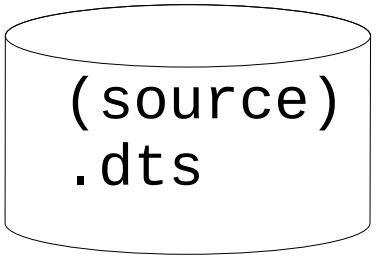
“A device tree is a tree data structure with nodes that describe the devices in a system. Each node has property/value pairs that describe the characteristics of the device being represented. Each node has exactly one parent except for the root node, which has no parent.”
(ePAPR v1.1)

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



.dts - device tree source file

```
/ {   /* 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>;

     console: serial@f991e000 {
        compatible = "qcom,msm-uartdm-v1.4", "qcom,msm-uartdm";
        reg = <0xf991e000 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!

http://elinux.org/images/f/f9/
Petazzoni-device-tree-dummies_0.pdf

.dts - device tree source file

```
/ {   /* incomplete .dts example */                <--- root node

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

   soc: soc {                                     <--- node
      ranges;                                     <--- property
      compatible = "simple-bus";                  <--- property

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

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

Key vocabulary

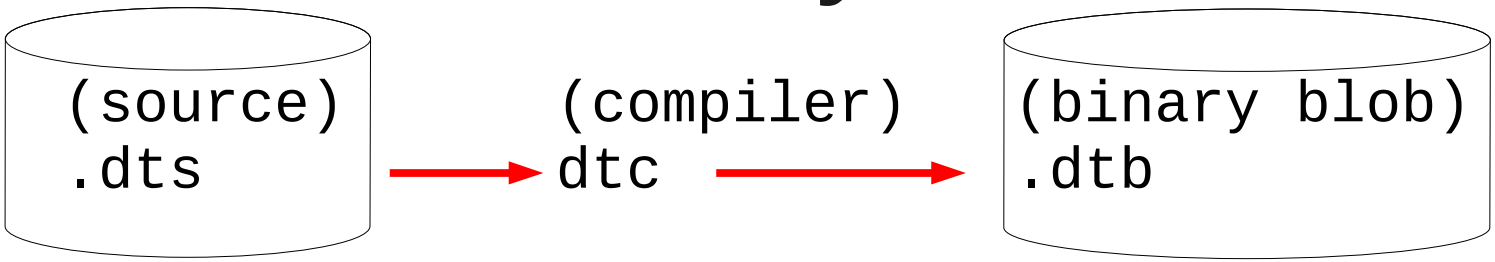
nodes
- the tree structure
- contain properties and other nodes

properties
- data values providing information about a node

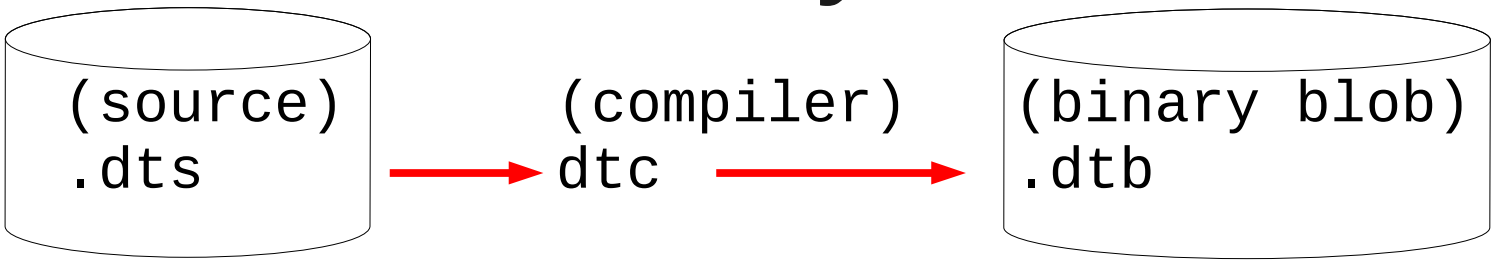
node '/': property 'compatible'
- will be used to match a machine_desc entry

other nodes: property 'compatible'
- will be used to match a driver

DT data life cycle



DT data life cycle

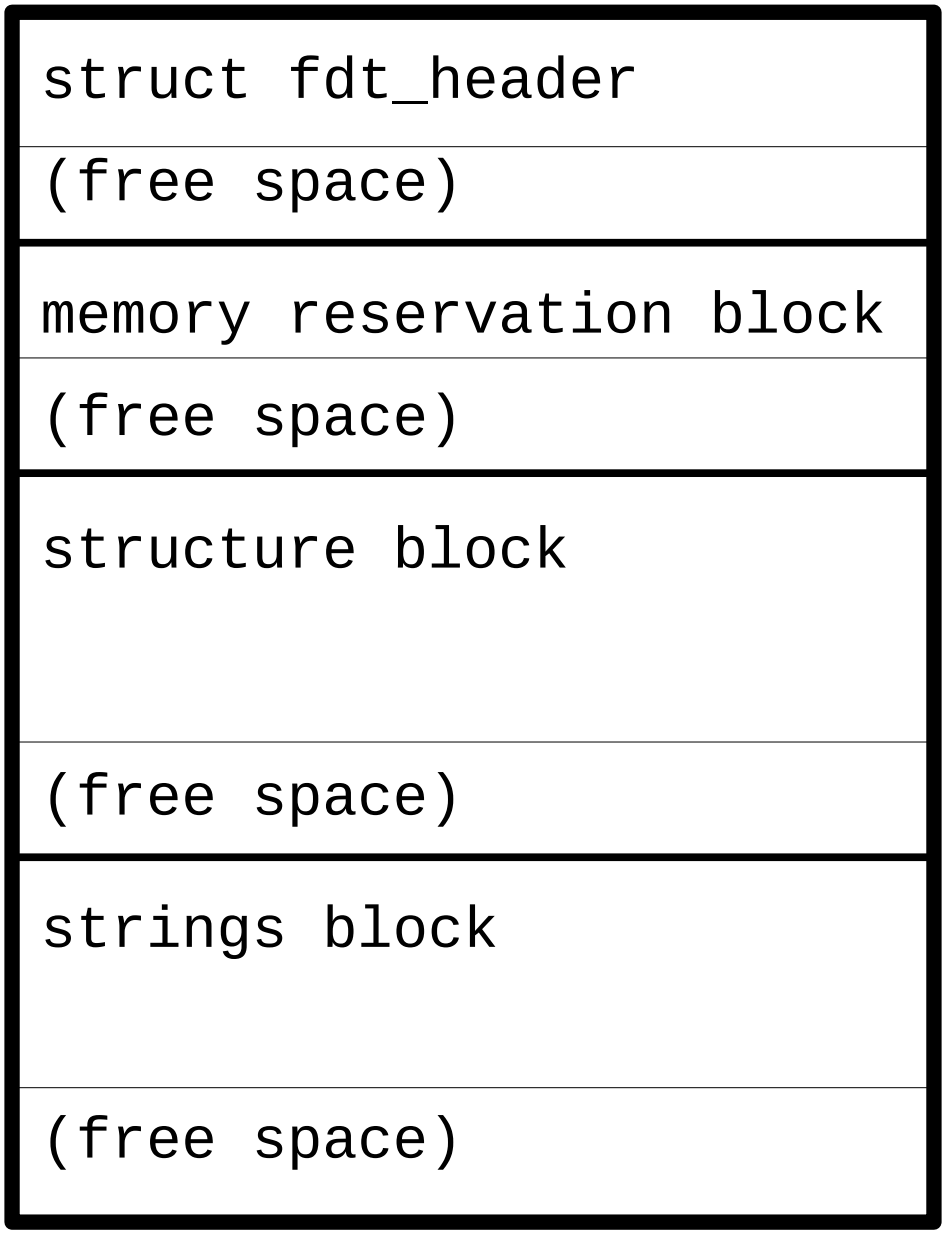


Binary Blob format

A “flat” format

Access via serial scan and offsets

Binary Blob format



info
offsets to blocks
section sizes

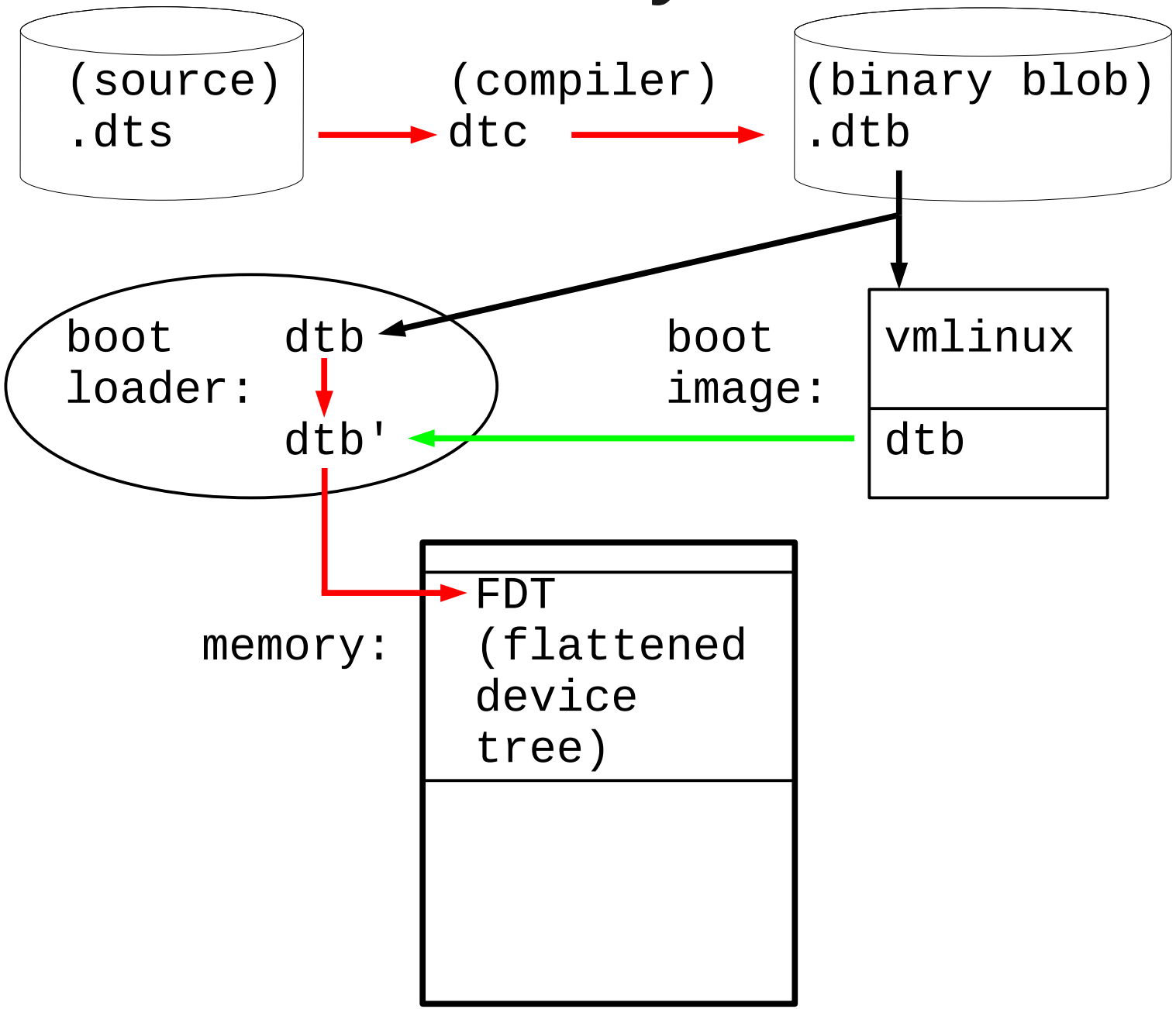
{address, size} tuples

nested nodes
- name embedded

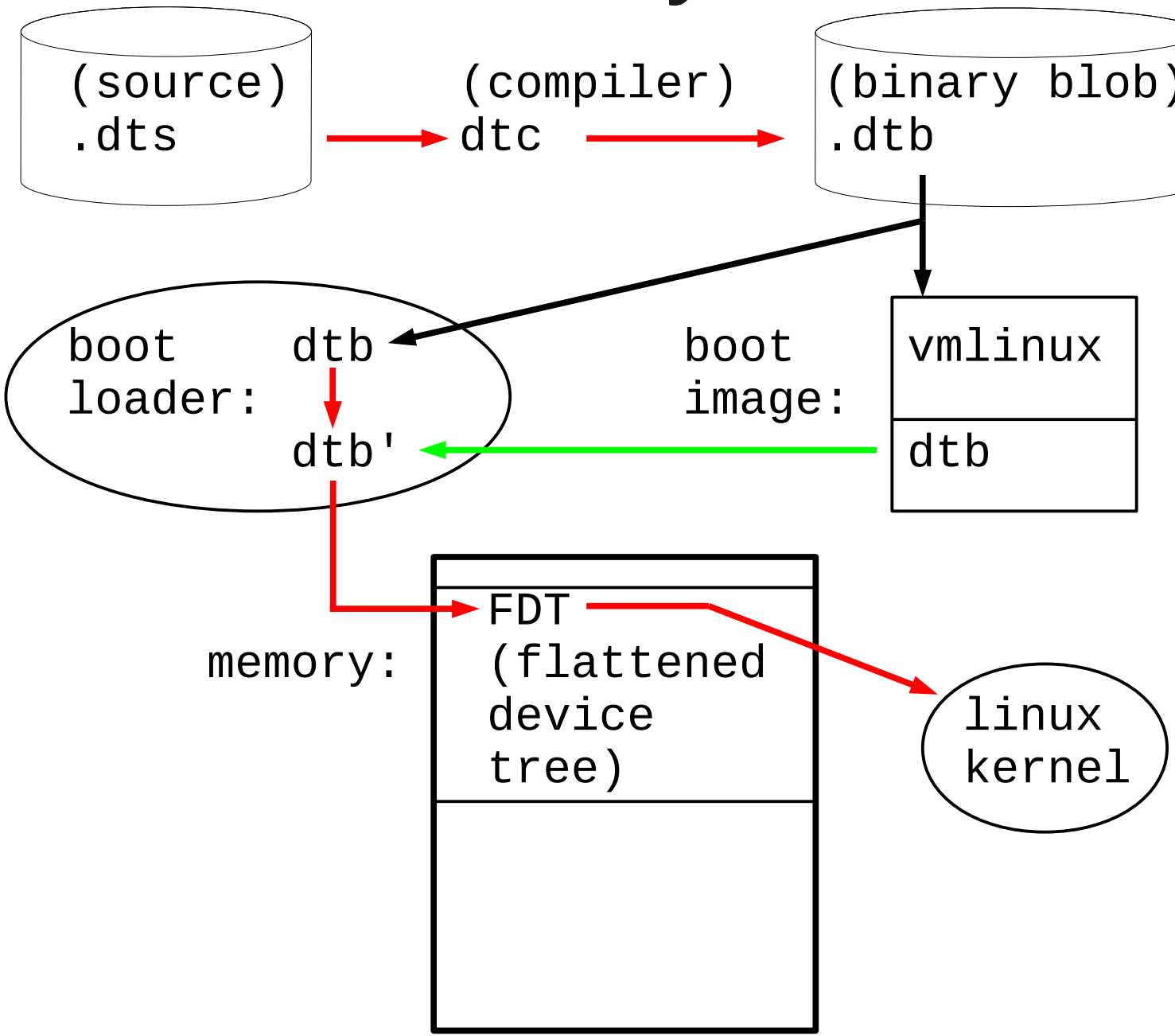
properties nested in nodes
- values embedded
- names are offsets in 'strings'

property names
- null terminated strings
- concatenated

DT data life cycle



DT data life cycle

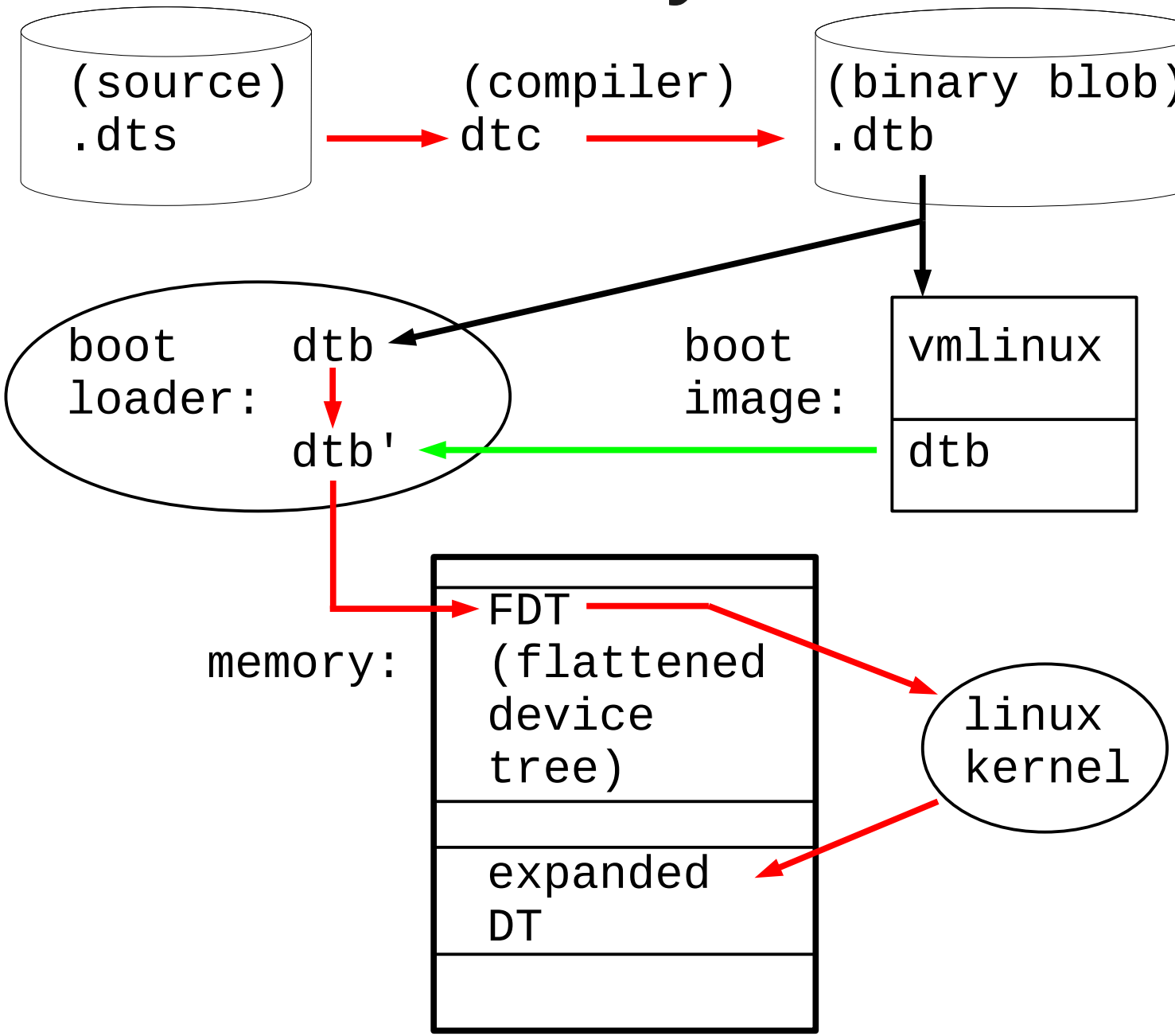


Flattened Device Tree format

A “flat” format.

Access via serial scan and offsets
using fdt_*() functions.

DT data life cycle



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

tree of struct device_node

```
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

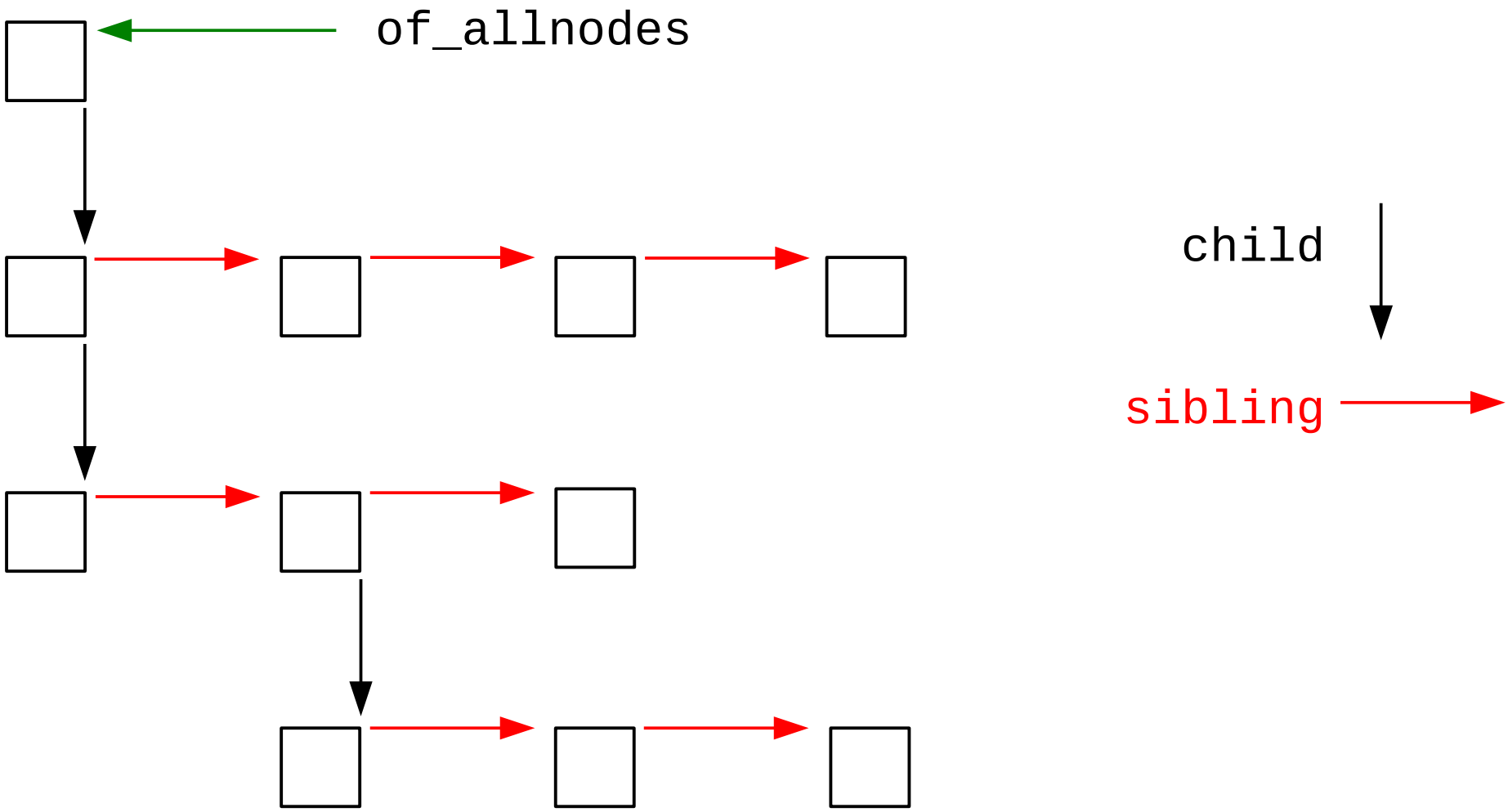
`tree of struct device_node`

Tree pointers

- child pointer
- sibling pointer

Expanded format

tree of struct device_node



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

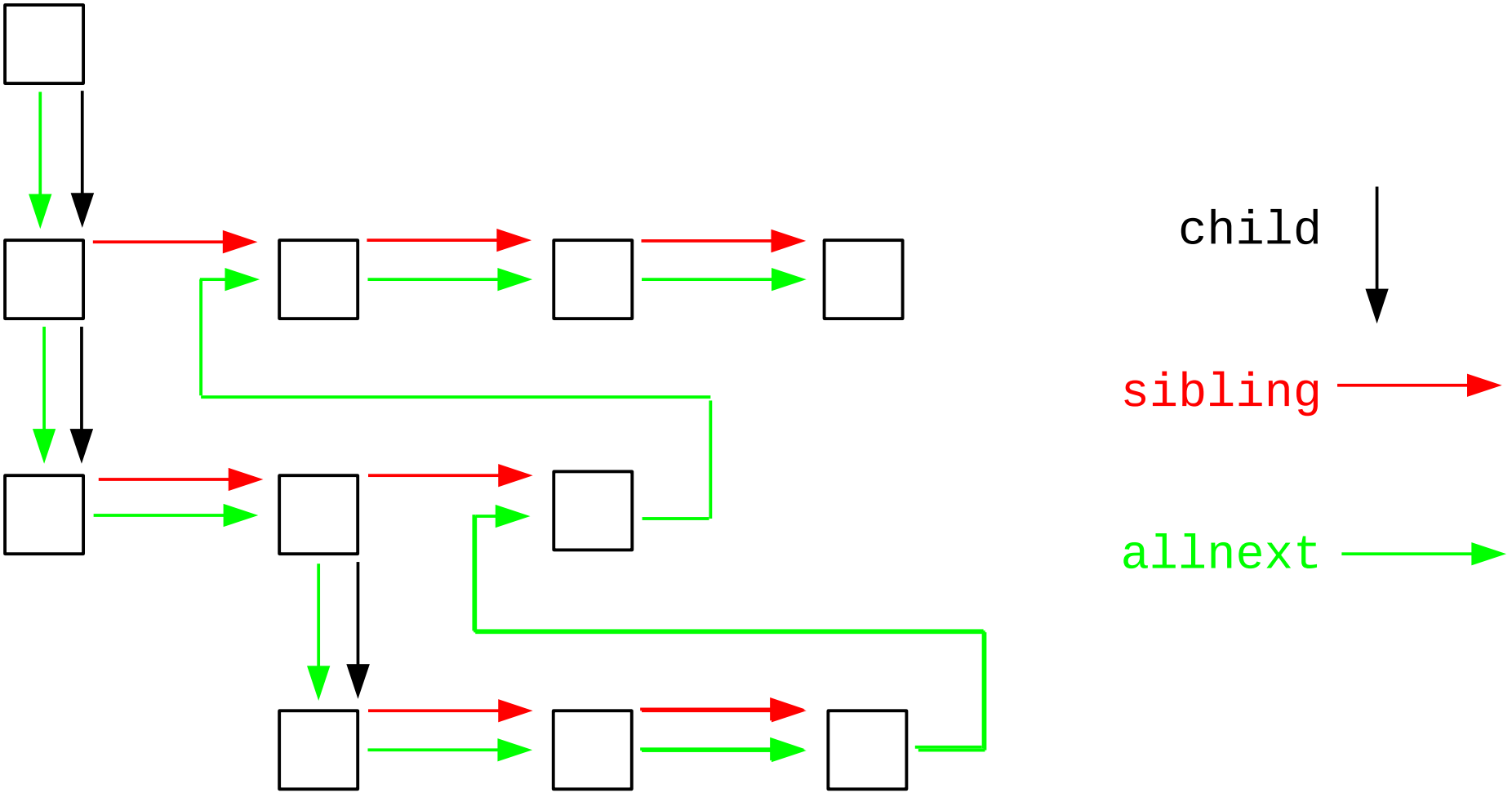
`tree of struct device_node`

Global linked list pointer

allnext pointer

Expanded format

tree of struct device_node



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:

```
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

```
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

Properties 'name' and 'device_type' are special.

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

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

device_node.name

device_node.type

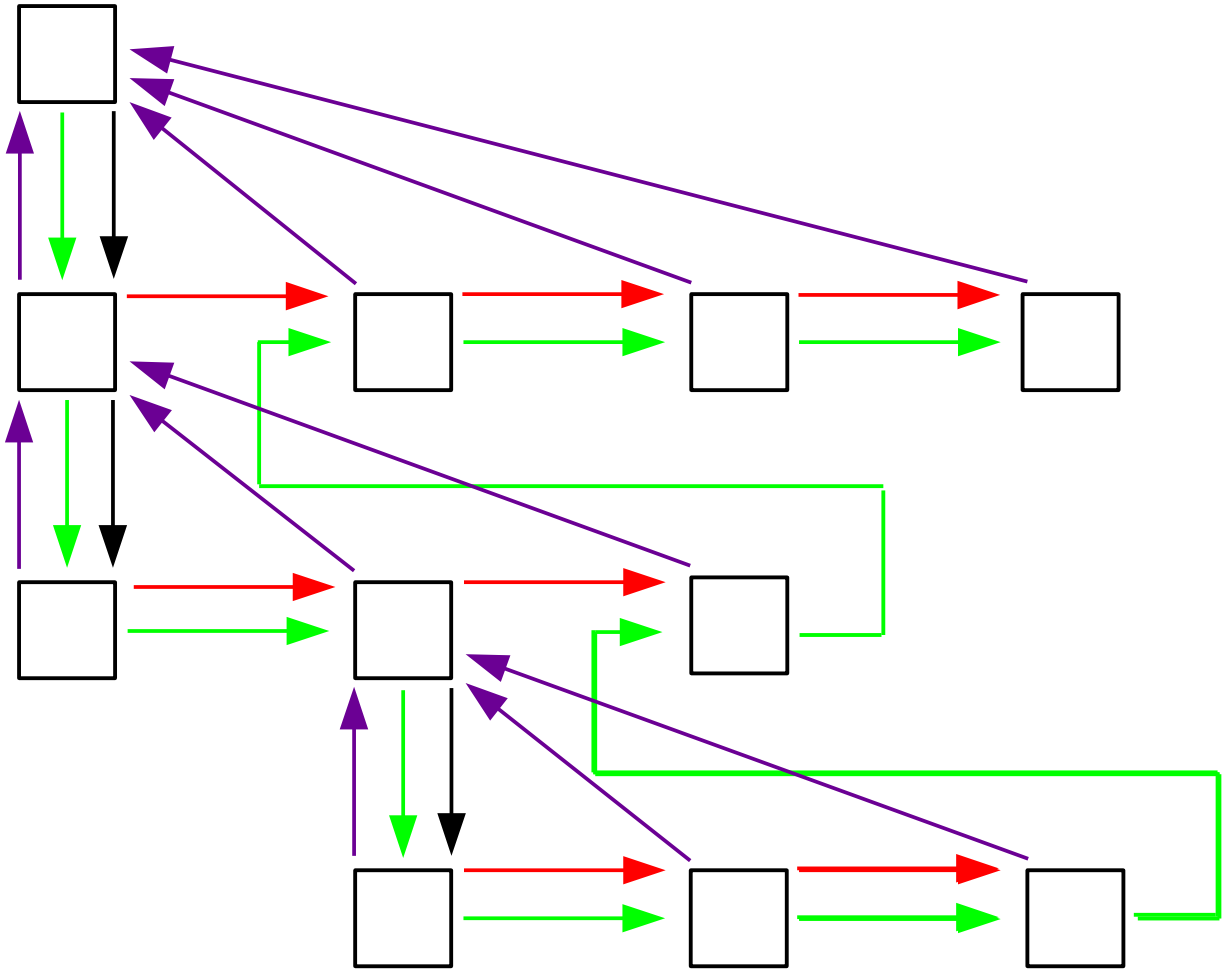
Expanded format

`tree of struct device_node`

parent pointer

Expanded format

tree of struct device_node



parent

child

sibling

allnext

Expanded format

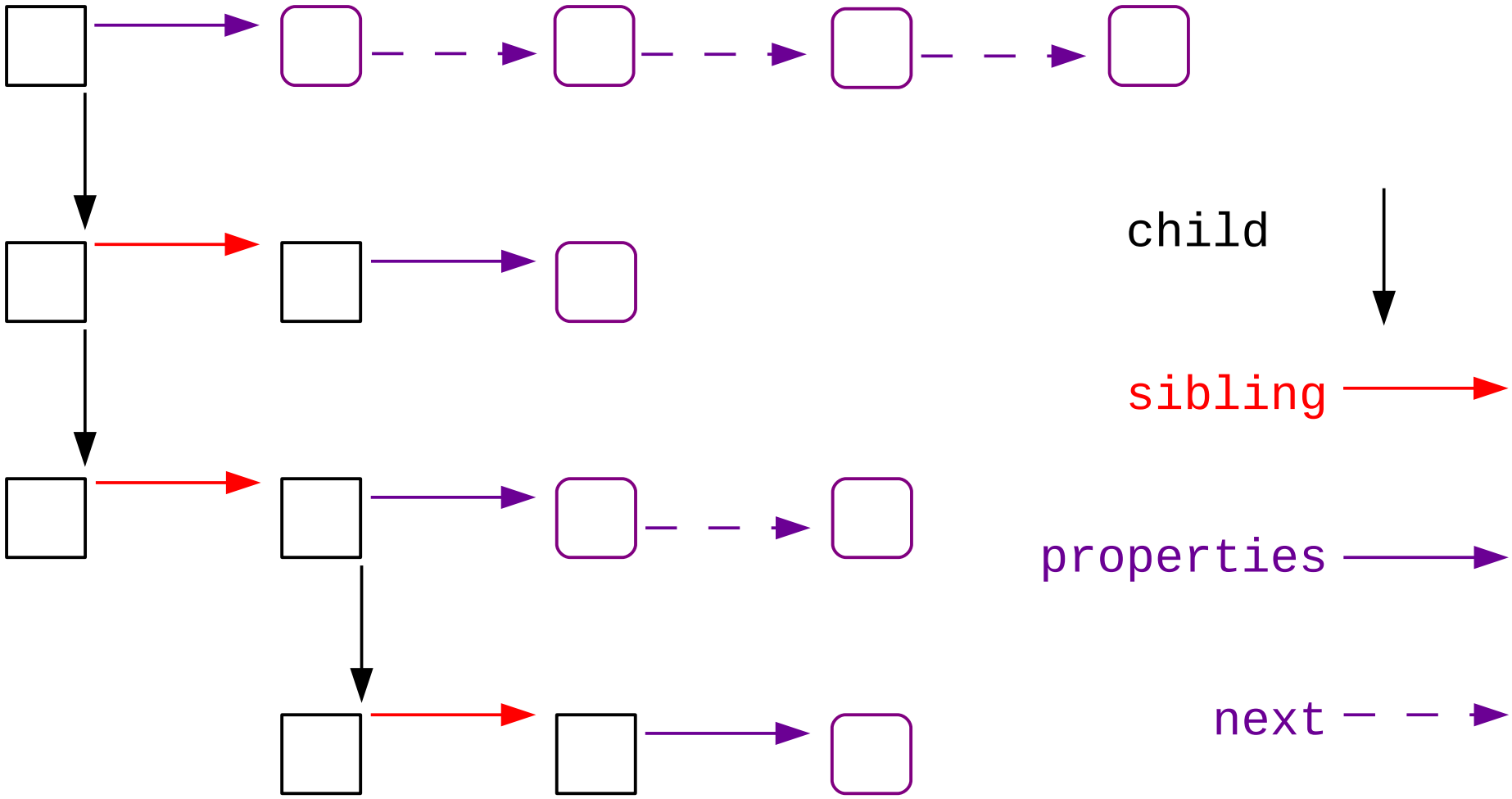
`tree of struct device_node`

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



Chapter 2

Matching boot customization options to the device tree

Kernel boot

Linux kernel - machine_desc

Boot customizations for different device trees

machine_desc

```
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;
   bool                   (*XXX_init)();
   bool                   (*YYY_init)();
   ...
```

machine_desc - populating

```
#define DT_MACHINE_START(_name, _namestr) \
struct machine_desc __mach_desc_##_name   \
__section__(".arch.info.init" = {         \
    .nr                 = ~0,             \
    .name               = _namestr,

#define MACHINE_END };
```

* Essential features extracted, actual code is on next slide

machine_desc - populating

```
#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                                               \
};
```

machine_desc - populating

```
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
```

machine_desc - populating

```
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


#ifdef CONFIG_ARCH_MULTIPLATFORM
     DT_MACHINE_START(GENERIC_DT, "Generic DT based system")
     MACHINE_END
#endif
```

machine_desc - populating

```
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:

```
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 - populating

```
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
   .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

```
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)();
```

My pseudocode conventions

- Will obviously fail to compile
- Will usually not show function arguments
- Each level of indention indicated 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

machine_desc hooks (all)

```
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.
```

'Best match' means found earliest in device tree root node 'compatible' property list

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()   <===============
```

machine_desc hooks (2 of 3)

```
    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()
```

Takeaway

Use fdt_*() functions before unflatten_device_tree()

Use of_*() functions after unflatten_device_tree()

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 is oversimplified, but we will continue with this deception for a few more slides:

```
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:

```
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()

auxdata may affect how the platform device was created

initcall - of_platform_populate()

Drivers may be bound to the devices during platform device creation if

- 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

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

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:

```
c0910edc T __initcall0_start
c0910edc t __initcall_ipc_ns_init0
c0910ee0 t __initcall_init_mmap_min_addr0
c0910ee4 t __initcall_net_ns_init0
c0910ee8 T __initcall1_start
...
c0910f50 T __initcall2_start
...
c0910f84 T __initcall3_start
...
...
c0911310 T __initcall7_start
...
c0911368 T __con_initcall_start
...
c0911368 T __initcall_end
```

Initcalls - order of execution

The order of functions within an init level is determined by:

- location in source file of initcall declaration
- compile order of source files
- link order of object files

A previous function within an init level may start asynchronous work and return while that work is occurring.

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

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:

- examining the order in System.map
- add 'initcall_debug' to the kernel command line to print each initcall to console as it is called

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\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.

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

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

what is likely to be accepted for new code:

```
“arch”      : customize machine()
                 of_platform_populate()
“device”    : platform_driver_register()
```

Chapter 4

Miscellaneous

Some unresolved DT issues

Circular dependencies on driver probe

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

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 nodes 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

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
- structure of expanded device tree
- customizing the boot process by machine_desc
- device creation
- driver binding

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