Extra Boot Configuration and Kernel Command Line

Masami Hiramatsu <mhiramat@kernel.org>
Senior Tech Lead, Linaro ltd.

Speaker

Masami Hiramatsu

- Work for Linaro
- Senior Tech Lead for a Linaro member's Landing Team
- Maintainer of Kprobes, bootconfig and related tracing features/tools

Agenda

- Boot time kernel parameters
  - Kernel command line
  - Limitations
- Extra Boot Configuration (Bootconfig)
  - Bootconfig Syntax
  - How to use Bootconfig
- Bootconfig API
  - Xbc node tree
  - APIs
- Combining bootconfig and kernel command line
  - Import kernel command line into bootconfig

Linaro

Kernel command line (cmdline)

- A way to pass boot options to kernel.
  - Pass a (long) string which contains the options.
  - The string is parsed in the early stage of the kernel boot.
- Different implementations for each architecture.
  - Defined in the boot protocol.
  - On x86, it passed over memory area.
  - Arm64 uses the devicetree (/chosen/bootargs)

Kernel command line

Size and coding limitation

- kernel cmdline size is 256 bytes in minimum
  - Minimum, most arch support a larger size (4kB).
- kernel cmdline is written in a single line
  - Bootloader wrapper will allow us to write in several lines

Fixed parameter API

- “PARAM” hook API (If PARAM is set then ...)
  - “PARAM” is a fixed key.
  - This is for parameter settings
- Nested parameter or pattern keys are not supported
  - E.g. ftrace.instance.FOO.event.BAR.BUZ.enable
    - FOO/BAR/BUZ are parameters in the key.

Extra Boot Configuration

A new kernel cmdline extension: Extra Boot Configuration (a.k.a. “bootconfig”)

- A plain ascii text file of structured key-value list (like sysctl.conf)

```
key.word = value
key.word2 {
     word3 = value
     nested-key {
          array-value-key = item1, item2, “arg1,arg2”
     }
}
```

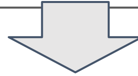
- Loaded with the initrd image when boot
- In-kernel API for flexible option handling
  - Search sub key words
  - Enumerate sub keys under given key

Example of kernel command line

```
root=UUID=12345678-beef-face-cafe-123456789abc ro quiet splash console=tty0
console=ttyS0,115200n8 video.brightness_switch_enabled=0 video.allow_duplicates=1
trace_options=”sym-addr” trace_clock=global trace_buf_size=1M
trace_event=”initcall:*,exceptions:*”
```

Example of kernel command line to bootconfig

```
root=UUID=12345678-beef-face-cafe-123456789abc ro quiet splash console=tty0
console=ttyS0,115200n8 video.brightness_switch_enabled=0 video.allow_duplicates=1
trace_options=”sym-addr” trace_clock=global trace_buf_size=1M
trace_event=”initcall:*,exceptions:*”
```



```
kernel {
      root = “UUID=12345678-beef-face-cafe-123456789abc”
      quiet                                     # Kernel doesn’t show log on console
      console = "tty0”, “ttyS0,115200n8"        # Kernel has 2 consoles
      video {                                   # Video module settings
            brightness_switch_enabled = 0
            allow_duplicates = 1
      }
      trace_options = sym-addr     # Symbol + address option to showing the address precisely
      trace_clock = global         # Use the global trace clock for synchronization
      trace_buf_size = 1MB         # Expand buffer size to 1MB because default size is too small
      trace_event = "initcall:*, exceptions:*"      # Trace initcall and exceptions events.
}
init {
    splash ; quiet ; ro      # Show splash, read only and be quiet.
}
```

Extra Boot Configuration Syntax

- Simple key-value set

```
KEY[.WORD[...]] [+|:]= VALUE[, VALUE2[...]][;][#COMMENT]
```

  - Single value or comma-separated values (Array)

```
key = value1, value2, value3
```

  - Same key-words can be merged, e.g.

```
key.word1 = value1
key.word2 = value2
```

  This can be written as;

```
key { word1 = value1; word2 = value2 }
```

Linaro

Syntax (cont.)

- Value assignment
  - ‘=’ defines the value of the key

    ```
    key = foo
    ```

  - ‘:=’ overwrites the previous assignment

    ```
    key = foo; key := bar
    ```

    Is

    ```
    key = bar
    ```

  - ‘+=’ appends a value as an array element

    ```
    key = foo; key += bar
    ```

    Is same as;

    ```
    key = foo, bar
    ```

Linaro

Syntax (cont.)

- Mixing key and value on the same key (5.14)
    - A parent key can have subkey and value

```
key = value
key.subkey = foo
key.subkey.subsubkey = bar
```

    - Note that you can **NOT** merge value and subkey by brace

```
key = value
key {
    ~~value~~
    subkey = foo
}
```

Linaro

Syntax (cont.)

- Comment
  - Shell script like comment is available

```
key = value # comment after key-value

# comment line1
# comment line2
key.subkey = bar
```

Linaro

Expand kernel command line

You can write kernel command line with bootconfig.

- The configuration keys starting with "kernel." are passed to kernel command line.

```
kernel {
    root = ”UUID=12345678-9abc-...
    console = tty0, “ttyS0,115200”
}
```

- Also "init." are passed to kernel command line but after "--". These are passed to init process (e.g. systemd)

```
init {
    splash
    quiet
}
```

How to setup the bootconfig

- Bootconfig text file will be appended to the initrd (initramfs)
- "`(ksrc)/tools/bootconfig/bootconfig`" command handles it.
  - Append bootconfig (-a)

    ```
    # bootconfig -a your-boot-config system-initrd-file
    ```
  - Check current applied bootconfig

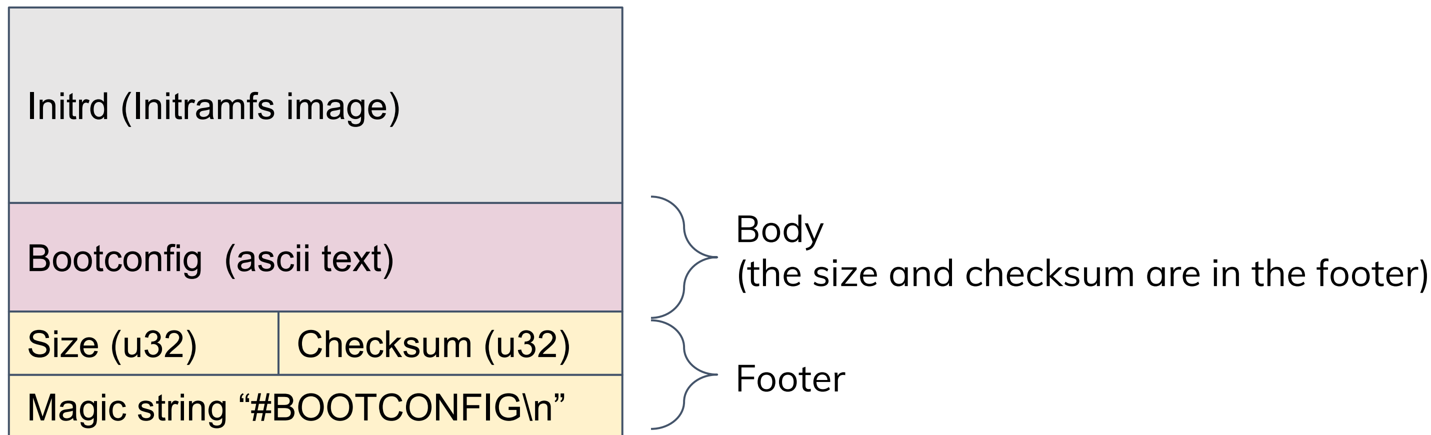
    ```
    # bootconfig system-initrd-file
    ```
  - Delete the bootconfig (-d)

    ```
    # bootconfig -d system-initrd-file
    ```
  - Reformat the bootconfig file to key-value list

    ```
    # bootconfig your-boot-config
    ```

Bootconfig and initrd

The “*bootconfig*” command appends a config file to the initrd image.



Note: you also need “**bootconfig**” kernel command line option to enable bootconfig.

Linaro

Procfs interface for extra boot configuration data

Linux kernel provides /proc/bootconfig, which shows bootconfig data as a key-value list

```
$ cat /proc/bootconfig
ftrace.event.kprobes.vfs_read.probes = "vfs_read $arg1 $arg2"
ftrace.event.kprobes.vfs_read.filter = "common_pid < 200"
ftrace.event.kprobes.vfs_read.enable = ""
ftrace.instance.foo.tracer = "function"
ftrace.instance.foo.ftrace.filters = "user_*"
ftrace.instance.foo.cpumask = "1"
ftrace.instance.foo.options = "nosym-addr"
ftrace.instance.foo.buffer_size = "512KB"
ftrace.instance.foo.trace_clock = "mono"
ftrace.instance.foo.event.signal.signal_deliver.actions = "snapshot"
kernel.trace_options = "sym-addr"
kernel.trace_buf_size = "1M"
```

Linaro

Native Bootconfig boot options

What is the benefit of using bootconfig options natively?

- You can write it in the structured key-value list with comments.
- You can use the flexible option syntax.
  E.g. boot-time tracing

  ftrace.[instance.INSTANCE.]event.GROUP.EVENT.hist.[N.]onmatch.[M.]trace = EVENT[, ARG1[…]]

  - The [blue] part is optional parameters in the **key**
  - The RED part is mandatory parameters in the **key**

To handle the bootconfig natively, you need to use Bootconfig API.

Native Bootconfig example: boot time tracing

You can use flexible boot-time options

```
ftrace.event {                                       # Possible to join key-words with “.”
      synthetic.initcall_latency {         # Add a synthetic event
            fields = "unsigned long func", "u64 lat"
            hist {                         # Make a histogram with function name and latency
                  keys = func.sym, lat;
                  values = lat; sort = lat
            }
      }
      initcall.initcall_start.hist {       # Save the initcall timestamp on @ts0 with function name key
            keys = func; var.ts0 = common_timestamp.usecs }
      initcall.initcall_finish.hist {      # Make a latency per function on @lat
            keys = func; var.lat = common_timestamp.usecs-$ts0;
            onmatch {                      # If the key (func name) matches call initcall_latency event
                  event = initcall.initcall_start;
                  trace = initcall_latency, func, $lat }
      }
}
```

Linaro

Bootconfig API

Bootconfig API is provided by “include/linux/bootconfig.h”.

- There are xbc_* functions and macros.
- “xbc_node” is the basement data.
    - Bootconfig is expressed as a tree of xbc_node

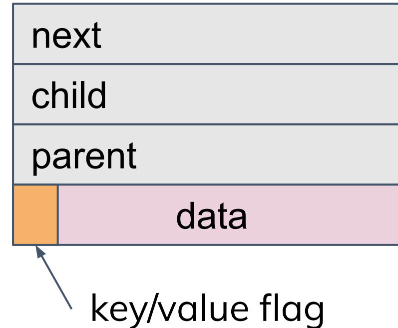
Some Notes:

- You need to enable CONFIG_BOOT_CONFIG
- All functions/data is released after boot (__init/__initdata)

Bootconfig API - xbc node

Xbc_node is the basic data structure for bootconfig API.

```
/* XBC tree node */
struct xbc_node {
        u16 next;
        u16 child;
        u16 parent;
        u16 data;
} __attribute__ ((__packed__));
```

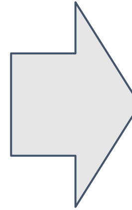


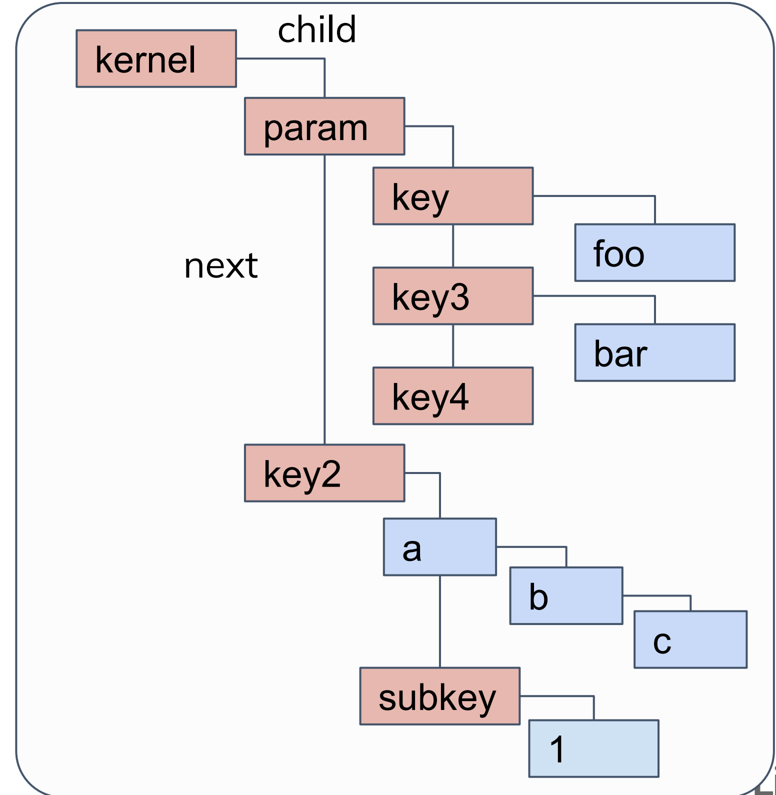
@next/@child/@parent are the index number to each node. (max # of node is 1024)

@data field is the offset to the data of the node. But the highest bit is used for key/value flag. (Thus, the max config size is 32k)

xbc_node tree creation

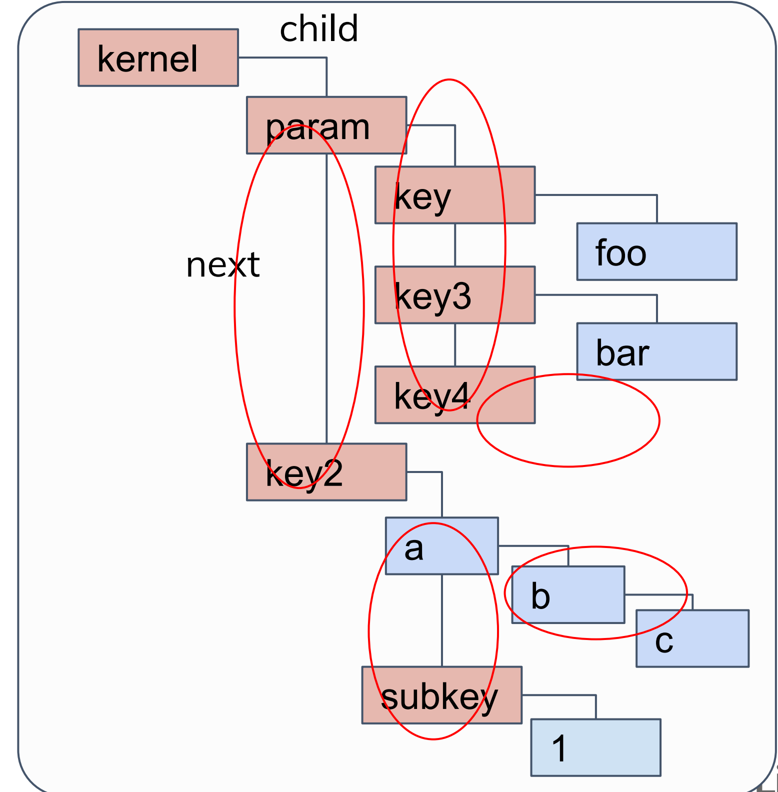
```
kernel {
    param.key = foo
    key2 = a, b ,c
    param {
        key3 = bar
        key4
    }
}
kernel.key2.subkey = 1
```





Characteristics of xbc_node tree

- The same subkey-trees are merged
- Keys are not sorted
- No-value key has no child node
- Array is a descendant list of child
- Value node must be the first child if mixed with subkeys



Linaro

Bootconfig API (key/value finder)

- `xbc_find_node("key")`
    - Find a **key**-node which matching the “key” under root node
- `xbc_node_find_subkey(node, "key")`
    - Find a **key**-node which matching the “key” under **@node** (partial tree search)
- `xbc_find_value("key", &vnode)`
    - Find a **value**-node and **value** which matching the key under root node
- `xbc_node_find_value(node, "key", &vnode)`
    - Find a **value**-node and **value** which matching the key under **@node** (partial tree search)

Bootconfig API (iteration)

Key-value pairs

- `xbc_for_each_key_value(knode, value) {}`
    - Loop on all key-value pairs.
- `xbc_node_for_each_key_value(node, knode, value) {}`
    - Loop on key-value pairs under @node.

Array

- `xbc_array_for_each_value(anode, value) {}`
    - Loop on the array values. @anode is the first “value” node of the array.

Subkeys

- `xbc_node_for_each_subkey(parent, child) {}`
    - Loop on subkey nodes under @parent
    - This skips value node.

Linaro

Bootconfig API (composing)

Compose key

- `xbc_node_compose_key(node, buf, size)`
  - Compose full key string of the @node.
  - E.g. @node is “key4”, you’ll get “kernel.param.key4” in @buf.
- `xbc_node_compose_key_after(root, node, buf, size)`
  - Compose a part of key string of the @node from @root.
  - E.g. @node is “key4” and @root is “kernel”, you’ll get “param.key4” in @buf.

Linaro

Bootconfig API (xbc_node)

Node Accessors

- `xbc_node_get_data()`
  - Get the data of the node
- `xbc_node_get_parent/xbc_node_get_child/xbc_node_get_next()`
  - Get the node pointed by the field (If no node pointed, return NULL)

Node validators

- `xbc_node_is_value/xbc_node_is_key/xbc_node_is_array/xbc_node_is_leaf()`
  - Check the node attributes
  - ‘Leaf’ means it is a key and has a value node as a child or no child. IOW, not an intermediate node.

Bootconfig API programming example

A simple example to get the parameter if set;

```
val = xbc_find_value("this.is.my.parameter", NULL); // Find the parameter
if (val) {
     // setup your parameter by @val
}
```

If the parameter is an array;

```
xbc_find_value("this.is.my.array", &anode);    // If the parameter is array
if (anode) {
   xbc_array_for_each_value(anode, val) {
         // setup your parameter by @val
   }
}
```

Bootconfig API programming example

Looping on the parameters under some key;

```
node = xbc_find_node("this.is"); // Find the parent parameter
if (node) {
    xbc_node_for_each_key_value(node, knode, val) {
        xbc_node_compose_key(knode, buf, size);    // Get the full name of @knode
        // Set @buf parameter with @val
    }
}
```

Note that @knode will only knows its subkey-word (e.g. “parameter”), but xbc_node_compose_key() makes the full name by tracking back the xbc_node tree. (e.g. “this.is.my.parameter”)

Bootconfig API programming example

Another good examples are Boot-time tracing (kernel/trace/trace_boot.c) and /proc/bootconfig code (fs/proc/bootconfig.c)

```
trace_node = xbc_find_node("ftrace"); // Find the parent node

// ftrace[.instance.INSTANCE_NAME].some-properties...
node = xbc_node_find_subkey(trace_node, “instance”);     // “ftrace.instance” exist?
if (node) {
     xbc_node_for_each_subkey(node, inst_node) {    // Then loop on instances
          inst_name = xbc_node_get_data(inst_node);
          tr = trace_array_get_by_name(inst_name);  // Make a new instance
          trace_boot_init_one_instance(tr, inst_node);        // setup the instance
     }
}
// If no instance, setup default instance.
trace_boot_init_one_instance(default_tr, trace_node);
```

Linaro

Bootconfig API and kernel cmdline API

Kernel cmdline APIs

- Set the fixed parameter key before probe.
- Kernel_param API extends parameters as value with types.
  - User can define fixed parameters with variables.
- No interface to search parameters.

Bootconfig APIs

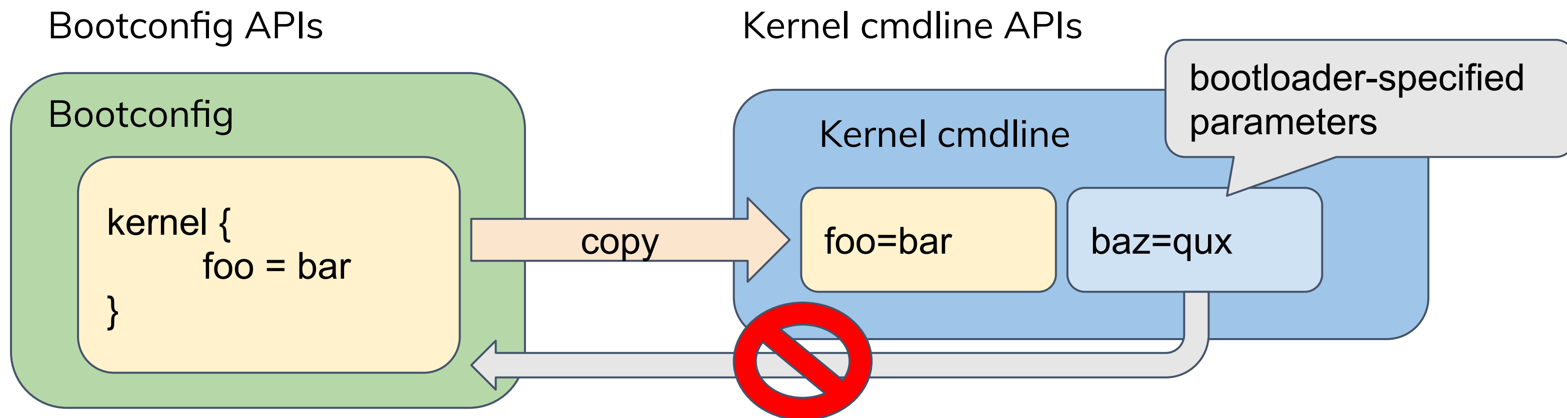
- Probe functions can search their parameters in tree.
- No hook interface.
- All values are treated as a string.
- Key tree structure is more flexible than fixed “key=value”.
  (e.g. boot-time tracer:
  ftrace.[instance.INSTANCE.]event.GROUP.EVENT.hist[.N].var.VAR=VALUE)
- **Only usable in boot time (__init function)**
- **Not able to handle parameters in the kernel command line**

Relationship between Bootconfig and cmdline

Bootconfig “kernel.” started key-values are copied into kernel cmdline

(and “init.” started one is also copied after “--”)

- Kernel cmdline API can handle the bootconfig given parameter.
- But bootconfig API only handles the parameter from bootconfig file.

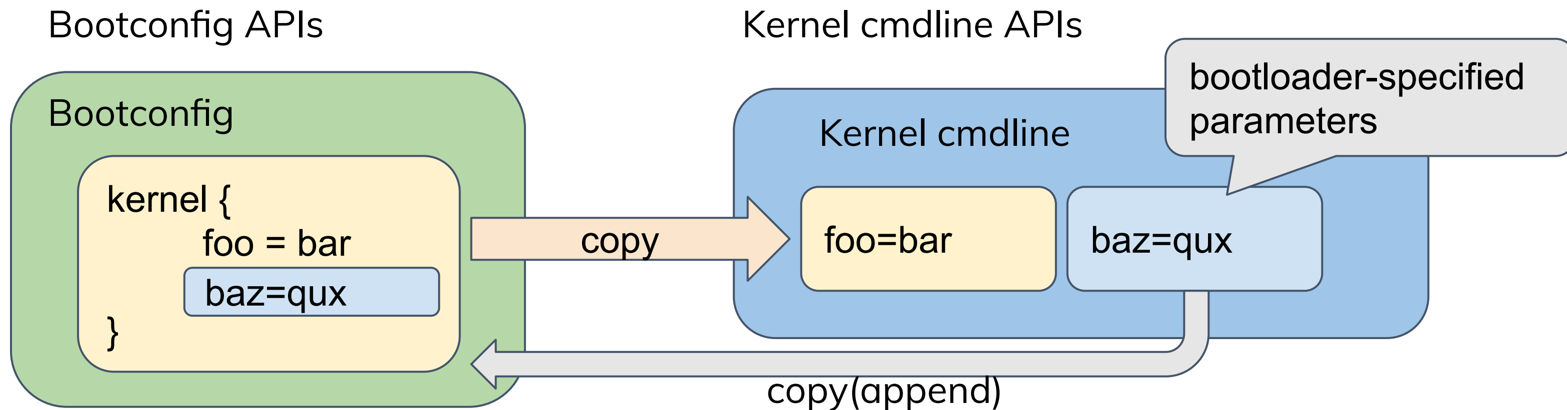


Idea: sync bootconfig with kernel cmdline

This will allow kernel subsystems to use bootconfig APIs for kernel cmdline options.

- Kernel/module can access parameters via bootconfig APIs.
- (Parameters which can not be parsed, should be ignored.)

But **who need it?**



Chicken and Egg problem

Bootconfig APIs are currently only available for kernel initialization

- All APIs are marked as __init. (freed after boot)
  - /proc/bootconfig is just a formatted snapshot.
- Boot-time tracing runs at the kernel boot time only.
  - **No other users**

If a module (driver) wants to use bootconfig, the API must not be __init.

- But currently there is no such user module.

So the first module user must remove **__init** from the bootconfig APIs.

If you have any usage, please share it on LKML.

Questions?

- Kernel documentation
  - Boot configuration
    https://www.kernel.org/doc/html/latest/admin-guide/bootconfig.html
  - Boot-time tracing
    https://www.kernel.org/doc/html/latest/trace/boottime-trace.html

Thank you