

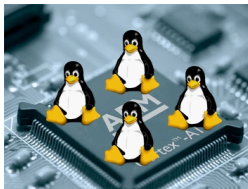


SMP bring up on ARM SoCs

Gregory CLEMENT

Free Electrons

gregory.clement@free-electrons.com





- ▶ Embedded Linux engineer and trainer at Free Electrons since 2010
 - ▶ Embedded Linux **development**: kernel and driver development, system integration, boot time and power consumption optimization, consulting, etc.
 - ▶ Embedded Linux **training**, Linux driver development training and Android system development training, with materials freely available under a Creative Commons license.
 - ▶ <http://free-electrons.com>
- ▶ Contributing to **kernel support for the Armada 370, 375, 38x and Armada XP** ARM SoCs from Marvell.
- ▶ Co-maintainer of mvebu sub-architecture (SoCs from Marvell Engineering Business Unit)
- ▶ Living near **Lyon**, France



Motivation and Overview

- ▶ Motivation
 - ▶ All the new ARM CPUs are now SMP capable.
 - ▶ Most of the new ARM SoCs actually uses multiple CPUs.
 - ▶ Did SMP bring up for two kinds of ARM SoCs during past year.
 - ▶ Documentation I would have like to find.
- ▶ Overview
 - ▶ SMP operations and functions.
 - ▶ Implementation of these functions.
 - ▶ Issues encountered during SMP bring up.



SMP on ARM SoC?

- ▶ A symmetric multiprocessor system (SMP) is a multiprocessor system with centralized shared memory called main memory (MM) operating under a single operating system with two or more homogeneous processors. (Wikipedia)
- ▶ Most of the SMP code is not architecture dependent (in kernel directory).
- ▶ Most of the hardware support is related to the ARM specification.
- ▶ SoC level: all that is beyond the CPU, mainly initialization and power down the CPUs.



Assumptions

- ▶ SoC support is working on one CPU core.
- ▶ Support is following the last requirements
(see “Your new ARM SoC Linux support check-list” from Thomas Petazzoni:
<http://free-electrons.com/pub/conferences/2013/elc/arm-soc-checklist/>)
- ▶ Datasheet is available.
- ▶ No hardware bug (or at least they are documented).



- ▶ Few SMP functions related to the SoC:

```
struct smp_operations {  
void (*smp_init_cpus)(void);  
void (*smp_prepare_cpus)(unsigned int max_cpus);  
void (*smp_secondary_init)(unsigned int cpu);  
int (*smp_boot_secondary)(unsigned int cpu,  
                           struct task_struct *idle);  
[...]  
};
```

- ▶ Only `smp_boot_secondary()` is mandatory, others are optional and depend on the need of the SoCs.



Role of each SMP operations 1/2

- ▶ `smp_init_cpus()`:
 - ▶ Setup the set of possible CPUs (via `cpu_possible()`).
 - ▶ Can be removed if the CPU topology is up to date in the device tree.
 - ▶ Called **very early** during the boot process (from `setup_arch()`).
- ▶ `smp_prepare_cpus()`:
 - ▶ Enables coherency.
 - ▶ Initializes `cpu_possible` map.
 - ▶ Prepares the resources (power, ram, clock...).
 - ▶ Called early during the boot process (before the initcalls but after `setup_arch()`).

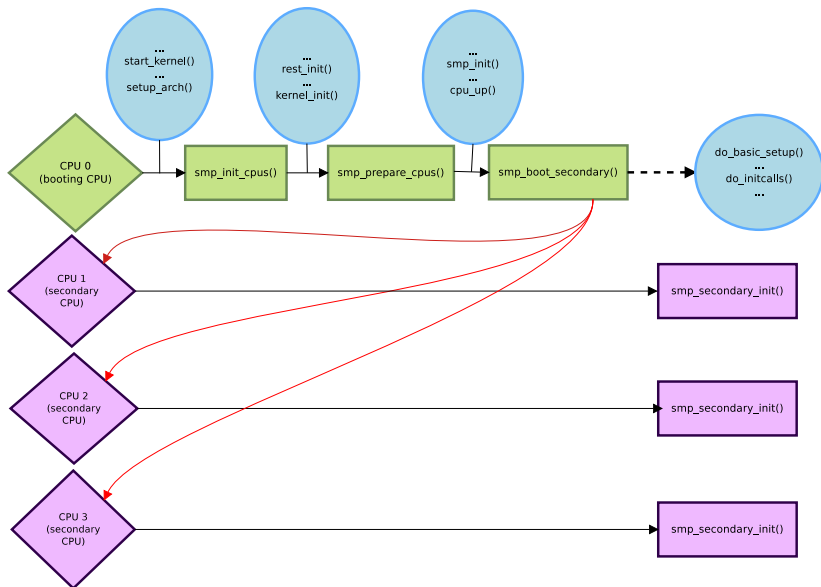


Role of each SMP operations 2/2

- ▶ `smp_secondary_init()`:
 - ▶ Supposed to “perform platform specific initialization of the specified CPU”.
 - ▶ Mostly dealing with pen release stuff which seems to be wrong. Russell King: *The pen_release stuff is only there for platforms where there's no proper way of controlling the secondary CPUs except by using a software method.*
 - ▶ Called from `secondary_start_kernel()` on the CPU which has just been started.
- ▶ `smp_boot_secondary()`:
 - ▶ Actually boots a secondary CPU identified by the CPU number given in parameter.
 - ▶ Called from `cpu_up()` on the booting CPU.



Booting sequence of the SMP operations





Hotplug SMP operations

- ▶ When `CONFIG_HOTPLUG_CPU` is selected three more functions are added to `smp_operations`:

```
struct smp_operations {  
    [...]  
    int (*cpu_kill)(unsigned int cpu);  
    void (*cpu_die)(unsigned int cpu);  
    int (*cpu_disable)(unsigned int cpu);  
};
```

- ▶ To support the hotplug features `cpu_kill()` and `cpu_die()` must be provided.



Role of each SMP hotplug operations 1/2

- ▶ `cpu_kill()`:
 - ▶ Supposed to do the powering off and/or cutting off clocks to the dying CPU.
 - ▶ Need to be synchronized with `cpu_die()`.
 - ▶ Needed for kexec (only user of `platform_can_cpu_hotplug()`, which checks the presence of this function).
 - ▶ Called on the thread which is asking for a CPU to be shutdown.
- ▶ `cpu_die()`:
 - ▶ Supposed to ensure death of the CPU.
 - ▶ Not supposed to return from this function, returns **only** if the SoC can't power down the CPU.
 - ▶ Either power down the CPU or at least call `cpu_do_idle()`.
 - ▶ Called from the idle thread for the CPU which has been shutdown.



Role of each SMP hotplug operations 2/2

- ▶ `cpu_disable()`:
 - ▶ Common ARM code takes care of the shutdown of a CPU: no more interrupts can be handled by the kernel after the routine returns, including local timers which are shutdown.
 - ▶ By default, the common code allows disabling all CPUs except the first one (the boot CPU). This function can be used to change this behavior.
 - ▶ Runs on the CPU to be shutdown.



SMP related features: nice to have

Not mandatory features, but they are usually added in the same time that SMP support.

- ▶ IRQ affinity: allows to dedicate one or several CPUs to an interrupt.
 - ▶ Part of the IRQ chip driver.
 - ▶ Implemented in the following handler:

```
int (*irq_set_affinity)(struct irq_data *data,  
const struct cpumask *dest, bool force)
```
- ▶ Local timer: timer dedicated to a CPU, thanks to this, no more need to broadcast a tick to all the CPUs.
 - ▶ No more a specific API to register it since 3.12.
 - ▶ Still needs to use the `percpu` API and the notifiers to implement it.



Use of the PSCI for SMP

PSCI stands for Power State Coordination Interface.

- ▶ Needed for virtualization.
- ▶ Used to coordinate OSes and hypervisors.
- ▶ Only solution to deal with secure mode.
- ▶ Provides functions used for SMP bring-up such as `CPU_ON` or `CPU_OFF`.
- ▶ Function provided by the firmware, the SMP operations are no more used.
- ▶ ID of these functions provided through the device tree.



Implementing the SMP operations 1/2

- ▶ `smp_init_cpus()`
 - ▶ Typically if an SCU unit is available the number of CPUs is retrieved using `scu_get_core_count()`.
 - ▶ And then the CPU possible map is built using `set_cpu_possible()`.
 - ▶ However as the cpu topology is supposed to be described in the device tree, the whole map is already built in the function `arm_dt_init_cpu_maps`.
- ▶ `smp_prepare_cpus()`
 - ▶ If SCU is present then the coherency is enabled by using `scu_enable()`.
 - ▶ Most of the time resources are allocated and the registers are mapped using `ioremap()` and its friends.
 - ▶ The way to enable (if needed) the power and the clocks is highly platform specific and does not involve common code.



Implementing the SMP operations 2/2

- ▶ `smp_secondary_init()`
 - ▶ No common code used here but a lot of copy/paste of the `pen_release` mechanism.
- ▶ `smp_boot_secondary()`
 - ▶ Releases the CPU from reset by writing on a power management register.
 - ▶ Some time writes “magic” pattern read by the *bootrom* which will release the CPU from reset.
 - ▶ Uses the `pen_release` if the secondary CPU has already been released from reset.



Implementing the SMP hotplug operations 1/2

- ▶ `cpu_kill()`
 - ▶ Most of the platforms check that the CPU is going to die, by reading a register of the SoC wrote by the dying CPU.
 - ▶ Some platforms really check that the dying CPU went in the reset state.
 - ▶ Then it actually powers down the CPU if the dying CPU has not done it by itself.
- ▶ `cpu_die()`
 - ▶ On most of the platforms, the dying CPU writes a register of the SoC, usually by setting to 0 the jump address of the CPU.
 - ▶ Some platforms switch in low power mode.
 - ▶ The others just call `cpu_do_idle()`.



Implementing the SMP hotplug operations 2/2

- ▶ `cpu_disable()`
 - ▶ Only implemented if the behavior of the platforms is different of the default (ie all CPUs except the first one can be killed).
 - ▶ Currently only used by shmobile where any CPU can be killed.



Registering the SMP operations

- ▶ Used to be part of the machine description. The `smp_operations` structure of the SoC was associated to the `.smp` field using the `smp_ops()` helper.
- ▶ From 3.16, they should be directly registered using the `CPU_METHOD_OF_DECLARE()` helper.
- ▶ Then the operation will be fetched from the device tree.



Implementing a local timer

- ▶ Should be part of a clock source driver in `drivers/clocksource/`.
- ▶ Each local timer will be called during the CPU bring up (`CPU_STARTING`), so the setup of the timer must be associated to a notifier using `register_cpu_notifier()`.
- ▶ The timer must be stopped when the CPU is shutting down (`CPU_DYING`), so this function will be also associated to the notifier.
- ▶ The interrupt has to be registered with `request_percpu_irq()`.
- ▶ And the `clock_event_device` has to be allocated with `alloc_percpu()`.



Implementing the IRQ affinity

- ▶ Should be part of an IRQ chip driver in `drivers/irqchip/`.
- ▶ The interrupts associated to `(struct irq_data *data` received should be treated only by the CPU represented by `struct cpumask *dest`.
- ▶ The `cpumask` can be manipulated using the helpers located in `include/linux/cpumask.h`.
- ▶ It is valid, depending of the hardware capabilities, to set the IRQ affinity of only of a subset of the group of the CPUs.



SMP and device tree: CPU topology - "cpus"

From `Documentation/devicetree/bindings/arm/cpus.txt`:
The device tree allows to describe the layout of CPUs in a system through the "cpus" node, which in turn contains a number of subnodes (ie "cpu") defining properties for every cpu.

The `cpus` nodes are composed of:

- ▶ `#address-cells`: 1 for 32-bit and 2 for 64-bit
- ▶ `#size-cells`: Always 0
- ▶ `enable-method`:
 - ▶ Optional for ARM 32-bit, but highly recommended for new implementations from 3.16.
 - ▶ For ARM 32 bits, the string is the one used for registering the SMP operations with `CPU_METHOD_OF_DECLARE()`.
 - ▶ Required for ARM v8 and must be "spin-table" or "psci"
 - ▶ Possible to use "psci" for ARM 32-bit too.



SMP and device tree: CPU bindings - "cpu"

The `cpu` subnodes are composed of:

- ▶ `device_type`: Always "cpu".
- ▶ `reg`: Related to the MPIDR, more or less the ID of the CPU.
- ▶ `compatible`: The CPU type for example: "arm,cortex-a9" or "marvell,pj4b".
- ▶ `enable-method`: Supposed to be defined at the CPU level but can be inherit from the CPUs level.



SMP and device tree: CPU bindings - Example

```
cpus {  
    #address-cells = <1>;  
    #size-cells = <0>;  
    enable-method = "marvell,armada-380-smp";  
  
    cpu@0 {  
        device_type = "cpu";  
        compatible = "arm,cortex-a9";  
        reg = <0>;  
    };  
    cpu@1 {  
        device_type = "cpu";  
        compatible = "arm,cortex-a9";  
        reg = <1>;  
    };  
};
```




SMP and device tree: PSCI

Fully defined in

`Documentation/devicetree/bindings/arm/psci.txt`.

It is more than just supporting SMP.

- ▶ `compatible`: Always `"arm,psci"`
- ▶ `device_type`: Always `"cpu"`.
- ▶ `method`: `"smc"` or `"hvc"` depending on the method used to call the PSCI firmware.
- ▶ `cpu_off`: Function ID for `CPU_OFF` operation.
- ▶ `cpu_on`: Function ID for `CPU_ON` operation.
- ▶ There are other functions defined but they are not used as SMP operations.



SMP and device tree: PSCI - Example

```
psci {
    compatible = "arm,psci";
    method = "smc";
    cpu_suspend = <0x84000002>;
    cpu_off = <0x84000004>;
    cpu_on = <0x84000006>;
};

cpus {
    #address-cells = <1>;
    #size-cells = <0>;

    cpu@0 {
        compatible = "arm,cortex-a15";
        device_type = "cpu";
        reg = <0>;
    };

    [...]

    cpu@3 {
        compatible = "arm,cortex-a15";
        device_type = "cpu";
        reg = <3>;
    };
};
```



Issues encountered during SMP Bring-up: coherency

- ▶ *Symptom*: The kernel booted on the CPU 0 but the other CPUs failed to boot.
 - ▶ On the secondary CPUs the L1 cache were corrupted.
 - ▶ The L1 cache needed to be invalidated before starting a secondary CPU.



Issues encountered during SMP Bring-up: coherency

- ▶ *Symptom*: The kernel crashed pretty early during the SMP initialization.
 - ▶ The registers controlling the coherency were not mapped yet.
 - ▶ The initialization of the resources associated to the coherency were done in an initcall, however the coherency for SMP is called before the initcalls.
 - ▶ The solution was to move this initialization in the `.init_time` handler called directly from `start_kernel`.



Issues encountered during SMP Bring-up: timers

- ▶ *Symptom*: The kernel booted but then nothing happened in userspace
 - ▶ No timer ticked at all so no schedule happened.
 - ▶ Here the problem was that the field `irq` of the timer was not filled.



Issues encountered during SMP Bring-up: timers

- ▶ *Symptom*: Sometimes the kernel hanged, just after booting, sometimes later when we have already switched to userspace.
 - ▶ There was no local timer neither broadcast timer: no tick on the secondary CPUs. As soon as the task ran on a secondary CPU, then it hanged, as no schedule happened.
 - ▶ The reason in our case was that `CONFIG_HAVE_TWD` was not selected.

Questions?

Gregory CLEMENT

`gregory.clement@free-electrons.com`

Slides under CC-BY-SA 3.0

<http://free-electrons.com/pub/conferences/2014/elc/clement-smp-bring-up-on-arm-soc>