How to Implement a new CPUFreq Processor Driver

Authors:

- Dominik Brodowski <linux@brodo.de>
- Rafael J. Wysocki <rafael.j.wysocki@intel.com>
- Viresh Kumar <viresh.kumar@linaro.org>

1. What To Do?

So, you just got a brand-new CPU / chipset with datasheets and want to add cpufreq support for this CPU / chipset? Great. Here are some hints on what is necessary:

1.1 Initialization

First of all, in an __initcall level 7 (module_init()) or later function check whether this kernel runs on the right CPU and the right chipset. If so, register a struct cpufreq_driver with the CPUfreq core using cpufreq_register_driver()

What shall this struct cpufreq_driver contain?

.name - The name of this driver.

.init - A pointer to the per-policy initialization function.

.verify - A pointer to a "verification" function.

.setpolicy _or_ .fast_switch _or_ .target _or_ .target_index - See below on the differences.

And optionally

.flags - Hints for the cpufreq core.

.driver_data - cpufreq driver specific data.

.get_intermediate and target_intermediate - Used to switch to stable frequency while changing CPU frequency.

.get - Returns current frequency of the CPU.

.bios_limit - Returns HW/BIOS max frequency limitations for the CPU.

.exit - A pointer to a per-policy cleanup function called during CPU_POST_DEAD phase of cpu hotplug process.

.suspend - A pointer to a per-policy suspend function which is called with interrupts disabled and _after_ the governor is stopped for the policy.

.resume - A pointer to a per-policy resume function which is called with interrupts disabled and _before_ the governor is started again.

.ready - A pointer to a per-policy ready function which is called after the policy is fully initialized.

.attr - A pointer to a NULL-terminated list of "struct freq_attr" which allow to export values to sysfs.

.boost_enabled - If set, boost frequencies are enabled.

.set_boost - A pointer to a per-policy function to enable/disable boost frequencies.

1.2 Per-CPU Initialization

Whenever a new CPU is registered with the device model, or after the cpufreq driver registers itself, the per-policy initialization function cpufreq_driver.init is called if no cpufreq policy existed for the CPU. Note that the .init() and .exit() routines are called only once for the policy and not for each CPU managed by the policy. It takes a `struct cpufreq_policy *policy` as argument. What to do now?

If necessary, activate the CPUfreq support on your CPU.

Then, the driver must fill in the following values:

policy->cpuinfo.min_freq _and_ policy->cpuinfo.max_freq	the minimum and maximum frequency (in kHz) which is supported by this CPU
policy->cpuinfo.transition_latency	the time it takes on this CPU to switch between two frequencies in nanoseconds (if appropriate, else specify CPUFREQ_ETERNAL)
policy->cur	The current operating frequency of this CPU (if appropriate)
policy->min, policy->max, policy->policy and, if necessary, policy->governor	must contain the "default policy" for this CPU. A few moments later, cpufreq_driver.verify and either cpufreq_driver.setpolicy or cpufreq_driver.target/target_index is called with these values.

policy->cpus	Update this with the masks of the (online + offline) CPUs that do DVFS along with this CPU (i.e. that share clock/voltage rails with it).

For setting some of these values (cpuinfo.min[max]_freq, policy->min[max]), the frequency table helpers might be helpful. See the section 2 for more information on them.

1.3 verify

When the user decides a new policy (consisting of "policy,governor,min,max") shall be set, this policy must be validated so that incompatible values can be corrected. For verifying these values cpufreq_verify_within_limits(`struct cpufreq_policy *policy`, `unsigned int min_freq`, `unsigned int max_freq`) function might be helpful. See section 2 for details on frequency table helpers.

You need to make sure that at least one valid frequency (or operating range) is within policy->min and policy->max. If necessary, increase policy->max first, and only if this is no solution, decrease policy->min.

1.4 target or target_index or setpolicy or fast_switch?

Most cpufreq drivers or even most cpu frequency scaling algorithms only allow the CPU frequency to be set to predefined fixed values. For these, you use the ->target(), ->target_index() or ->fast_switch() callbacks.

Some cpufreq capable processors switch the frequency between certain limits on their own. These shall use the ->setpolicy() callback.

1.5. target/target_index

The target_index call has two arguments: `struct cpufreq_policy *policy`, and `unsigned int` index (into the exposed frequency table).

The CPUfreq driver must set the new frequency when called here. The actual frequency must be determined by freq_table[index].frequency.

It should always restore to earlier frequency (i.e. policy->restore_freq) in case of errors, even if we switched to intermediate frequency earlier.

Deprecated

The target call has three arguments: `struct cpufreq_policy *policy`, unsigned int target_frequency, unsigned int relation.

The CPUfreq driver must set the new frequency when called here. The actual frequency must be determined using the following rules:

- keep close to "target_freq"
- policy->min <= new_freq <= policy->max (THIS MUST BE VALID!!!)
- if relation==CPUFREQ_REL_L, try to select a new_freq higher than or equal target_freq. ("L for lowest, but no lower than")
- if relation==CPUFREQ_REL_H, try to select a new_freq lower than or equal target_freq. ("H for highest, but no higher than")

Here again the frequency table helper might assist you - see section 2 for details.

1.6. fast_switch

This function is used for frequency switching from scheduler's context. Not all drivers are expected to implement it, as sleeping from within this callback isn't allowed. This callback must be highly optimized to do switching as fast as possible.

This function has two arguments: `struct cpufreq_policy *policy` and `unsigned int target_frequency`.

1.7 setpolicy

The setpolicy call only takes a `struct cpufreq_policy *policy` as argument. You need to set the lower limit of the in-processor or in-chipset dynamic frequency switching to policy->min, the upper limit to policy->max, and -if supported- select a performance-oriented setting when policy->policy is CPUFREQ_POLICY_PERFORMANCE, and a powersaving-oriented setting when CPUFREQ_POLICY_POWERSAVE. Also check the reference implementation in drivers/cpufreq/longrun.c

1.8 get_intermediate and target_intermediate

Only for drivers with target_index() and CPUFREQ_ASYNC_NOTIFICATION unset.

get_intermediate should return a stable intermediate frequency platform wants to switch to, and target_intermediate() should set CPU to that frequency, before jumping to the frequency corresponding to 'index'. Core will take care of sending notifications and driver doesn't have to handle them in target_intermediate() or target_index().

Drivers can return '0' from get_intermediate() in case they don't wish to switch to intermediate frequency for some target frequency. In that case core will directly call ->target_index().

NOTE: ->target_index() should restore to policy->restore_freq in case of failures as core would send notifications for that.

2. Frequency Table Helpers

As most cpufreq processors only allow for being set to a few specific frequencies, a "frequency table" with some functions might assist in some work of the processor driver. Such a "frequency table" consists of an array of struct cpufreq_frequency_table entries, with driver specific values in "driver_data", the corresponding frequency in "frequency" and flags set. At the end of the table, you need to add a cpufreq_frequency_table entry with frequency set to CPUFREQ_TABLE_END. And if you want to skip one entry in the table, set the frequency to CPUFREQ_ENTRY_INVALID. The entries don't need to be in sorted in any particular order, but if they are cpufreq core will do DVFS a bit quickly for them as search for best match is faster.

The cpufreq table is verified automatically by the core if the policy contains a valid pointer in its policy->freq_table field.

cpufreq_frequency_table_verify() assures that at least one valid frequency is within policy->min and policy->max, and all other criteria are met. This is helpful for the ->verify call.

cpufreq_frequency_table_target() is the corresponding frequency table helper for the ->target stage. Just pass the values to this function, and this function returns the of the frequency table entry which contains the frequency the CPU shall be set to.

The following macros can be used as iterators over cpufreq_frequency_table:

cpufreq_for_each_entry(pos, table) - iterates over all entries of frequency table.

cpufreq_for_each_valid_entry(pos, table) - iterates over all entries, excluding CPUFREQ_ENTRY_INVALID frequencies. Use arguments "pos" - a `cpufreq_frequency_table *` as a loop cursor and "table" - the `cpufreq_frequency_table *` you want to iterate over.

For example:

```
struct cpufreq_frequency_table *pos, *driver_freq_table;

cpufreq_for_each_entry(pos, driver_freq_table) {
        /* Do something with pos */
        pos->frequency = ...
}
```

If you need to work with the position of pos within driver_freq_table, do not subtract the pointers, as it is quite costly. Instead, use the macros cpufreq_for_each_entry_idx() and cpufreq_for_each_valid_entry_idx().