Linux Cpu-freq Documentation

The kernel development community

Jan 15, 2023

CONTENTS

Author: Dominik Brodowski <linux@brodo.de>

Clock scaling allows you to change the clock speed of the CPUs on the fly. This is a nice method to save battery power, because the lower the clock speed, the less power the CPU consumes.

CHAPTER

ONE

GENERAL DESCRIPTION OF THE CPUFREQ CORE AND CPUFREQ NOTIFIERS

Authors:

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

1.1 1. General Information

The CPUFreq core code is located in drivers/cpufreq/cpufreq.c. This cpufreq code offers a standardized interface for the CPUFreq architecture drivers (those pieces of code that do actual frequency transitions), as well as to "notifiers". These are device drivers or other part of the kernel that need to be informed of policy changes (ex. thermal modules like ACPI) or of all frequency changes (ex. timing code) or even need to force certain speed limits (like LCD drivers on ARM architecture). Additionally, the kernel "constant" loops_per_jiffy is updated on frequency changes here.

Reference counting of the cpufreq policies is done by cpufreq_cpu_get and cpufreq_cpu_put, which make sure that the cpufreq driver is correctly registered with the core, and will not be unloaded until cpufreq_put_cpu is called. That also ensures that the respective cpufreq policy doesn't get freed while being used.

1.2 2. CPUFreq notifiers

CPUFreq notifiers conform to the standard kernel notifier interface. See linux/include/linux/notifier.h for details on notifiers.

There are two different CPUFreq notifiers - policy notifiers and transition notifiers.

1.2.1 2.1 CPUFreq policy notifiers

These are notified when a new policy is created or removed.

The phase is specified in the second argument to the notifier. The phase is CPUFREQ_CREATE_POLICY when the policy is first created and it is CPUFREQ_REMOVE_POLICY when the policy is removed.

The third argument, a `void *pointer`, points to a struct cpufreq_policy consisting of several values, including min, max (the lower and upper frequencies (in kHz) of the new policy).

1.2.2 2.2 CPUFreq transition notifiers

These are notified twice for each online CPU in the policy, when the CPUfreq driver switches the CPU core frequency and this change has no any external implications.

The second argument specifies the phase - CPUFREQ_PRECHANGE or CPUFREQ_POSTCHANGE.

The third argument is a struct cpufreq_freqs with the following values:

policy	a pointer to the struct cpufreq_policy
old	old frequency
new	new frequency
flags	flags of the cpufreq driver

1.3 3. CPUFreq Table Generation with Operating Performance Point (OPP)

For details about OPP, see Documentation/power/opp.rst

dev_pm_opp_init_cpufreq_table - This function provides a ready to use conversion routine to translate the OPP layer's internal information about the available frequencies into a format readily providable to cpufreq.

> **Warning:** Do not use this function in interrupt context.

Example:

```
soc_pm_init()
{
        /* Do things */
        r = dev_pm_opp_init_cpufreq_table(dev, &freq_table);
        if (!r)
                policy->freq_table = freq_table;
        /* Do other things */
}
```

Note: This function is available only if CONFIG_CPU_FREQ is enabled in addition to CONFIG_PM_OPP.

dev_pm_opp_free_cpufreq_table Free up the table allocated by dev_pm_opp_init_cpufreq_table

CHAPTER

TWO

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>

2.1 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:

2.1.1 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.

2.1.2 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.

2.1.3 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.

2.1.4 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.

2.1.5 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.

2.1.6 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.

2.1.7 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`.

2.1.8 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

2.1.9 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.

2.2 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().

CHAPTER

THREE

GENERAL DESCRIPTION OF SYSFS CPUFREQ STATS

information for users

Author: Venkatesh Pallipadi <venkatesh.pallipadi@intel.com>

3.1 1. Introduction

cpufreq-stats is a driver that provides CPU frequency statistics for each CPU. These statistics are provided in /sysfs as a bunch of read_only interfaces. This interface (when configured) will appear in a separate directory under cpufreq in /sysfs (<sysfs root>/devices/system/cpu/cpuX/cpufreq/stats/) for each CPU. Various statistics will form read_only files under this directory.

This driver is designed to be independent of any particular cpufreq_driver that may be running on your CPU. So, it will work with any cpufreq_driver.

3.2 2. Statistics Provided (with example)

cpufreq stats provides following statistics (explained in detail below).

- time_in_state
- total_trans
- trans_table

All the statistics will be from the time the stats driver has been inserted (or the time the stats were reset) to the time when a read of a particular statistic is done. Obviously, stats driver will not have any information about the frequency transitions before the stats driver insertion.

```
<mysystem>:/sys/devices/system/cpu/cpu0/cpufreq/stats # ls -l
total 0
drwxr-xr-x  2 root root    0 May 14 16:06 .
drwxr-xr-x  3 root root    0 May 14 15:58 ..
--w-------  1 root root 4096 May 14 16:06 reset
-r--r--r--  1 root root 4096 May 14 16:06 time_in_state
-r--r--r--  1 root root 4096 May 14 16:06 total_trans
-r--r--r--  1 root root 4096 May 14 16:06 trans_table
```

- **reset**

Write-only attribute that can be used to reset the stat counters. This can be useful for evaluating system behaviour under different governors without the need for a reboot.

- **time_in_state**

This gives the amount of time spent in each of the frequencies supported by this CPU. The cat output will have "<frequency> <time>" pair in each line, which will mean this CPU spent <time> usertime units of time at <frequency>. Output will have one line for each of the supported frequencies. usertime units here is 10mS (similar to other time exported in /proc).

```
<mysystem>:/sys/devices/system/cpu/cpu0/cpufreq/stats # cat time_in_state
3600000 2089
3400000 136
3200000 34
3000000 67
2800000 172488
```

- **total_trans**

This gives the total number of frequency transitions on this CPU. The cat output will have a single count which is the total number of frequency transitions.

```
<mysystem>:/sys/devices/system/cpu/cpu0/cpufreq/stats # cat total_trans
20
```

- **trans_table**

This will give a fine grained information about all the CPU frequency transitions. The cat output here is a two dimensional matrix, where an entry <i,j> (row i, column j) represents the count of number of transitions from Freq_i to Freq_j. Freq_i rows and Freq_j columns follow the sorting order in which the driver has provided the frequency table initially to the cpufreq core and so can be sorted (ascending or descending) or unsorted. The output here also contains the actual freq values for each row and column for better readability.

If the transition table is bigger than PAGE_SIZE, reading this will return an -EFBIG error.

```
<mysystem>:/sys/devices/system/cpu/cpu0/cpufreq/stats # cat trans_table
From  :    To
        :   3600000   3400000   3200000   3000000   2800000
3600000:         0         5         0         0         0
3400000:         4         0         2         0         0
3200000:         0         1         0         2         0
3000000:         0         0         1         0         3
2800000:         0         0         0         2         0
```

3.3 3. Configuring cpufreq-stats

To configure cpufreq-stats in your kernel:

```
Config Main Menu
        Power management options (ACPI, APM)  --->
                CPU Frequency scaling  --->
                        [*] CPU Frequency scaling
                        [*]   CPU frequency translation statistics
```

“CPU Frequency scaling” (CONFIG_CPU_FREQ) should be enabled to configure cpufreq-stats.

“CPU frequency translation statistics” (CONFIG_CPU_FREQ_STAT) provides the statistics which includes time_in_state, total_trans and trans_table.

Once this option is enabled and your CPU supports cpufrequency, you will be able to see the CPU frequency statistics in /sysfs.

CHAPTER

FOUR

MAILING LIST

There is a CPU frequency changing CVS commit and general list where you can report bugs, problems or submit patches. To post a message, send an email to linux-pm@vger.kernel.org.

CHAPTER

FIVE

LINKS

the FTP archives: * ftp://ftp.linux.org.uk/pub/linux/cpufreq/

how to access the CVS repository: * http://cvs.arm.linux.org.uk/

the CPUFreq Mailing list: * http://vger.kernel.org/vger-lists.html#linux-pm

Clock and voltage scaling for the SA-1100: * http://www.lartmaker.nl/projects/scaling