cpu-drivers.txt
CPU frequency and voltage scaling code in the Linux(TM) kernel

L i n u x C P U F r e q

C P U D r i v e r s

- information for developers -

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.

Contents:
---

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?

cpufreq_driver.name - The name of this driver.

cpufreq_driver.owner - THIS_MODULE;

cpufreq_driver.init - A pointer to the per-CPU initialization function.

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

cpufreq_driver.setpolicy _or_
cpufreq_driver.target - See below on the differences.

And optionally

cpufreq_driver.exit - A pointer to a per-CPU cleanup function.

cpufreq_driver.resume - A pointer to a per-CPU resume function which is called with interrupts disabled and _before_ the pre-suspend frequency and/or policy is restored by a call to ->target or ->setpolicy.

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

1.2 Per-CPU Initialization

Whenever a new CPU is registered with the device model, or after the cpufreq driver registers itself, the per-CPU initialization function cpufreq_driver.init is called. 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 is called with these values.

For setting some of these values, 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, a frequency table helper and/or the
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 setpolicy?

Most cpufreq drivers or even most cpu frequency scaling algorithms
only allow the CPU to be set to one frequency. For these, you use the
->target call.

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

1.4. target

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.

1.5 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 arch/i386/kernel/cpu/cpufreq/longrun.c

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_freq_table entries, with any value in
"index" you want to use, and the corresponding frequency in
"frequency". At the end of the table, you need to add a
cpufreq_freq_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 ascending
order.

By calling cpufreq_frequency_table_cpuinfo(struct cpufreq_policy *policy,
                                           struct cpufreq_frequency_table *table);
the cpuinfo.min_freq and cpuinfo.max_freq values are detected, and
policy->min and policy->max are set to the same values. This is
helpful for the per-CPU initialization stage.

int cpufreq_frequency_table_verify(struct cpufreq_policy *policy,
                                   struct cpufreq_frequency_table *table);
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.

int cpufreq_frequency_table_target(struct cpufreq_policy *policy,
                                   struct cpufreq_frequency_table *table,
                                   unsigned int target_freq,
                                   unsigned int relation,
                                   unsigned int *index);

is the corresponding frequency table helper for the ->target
stage. Just pass the values to this function, and the unsigned int
index returns the number of the frequency table entry which contains
the frequency the CPU shall be set to. PLEASE NOTE: This is not the
"index" which is in this cpufreq_table_entry.index, but instead
cpufreq_table[index]. So, the new frequency is
cpufreq_table[index].frequency, and the value you stored into the
frequency table "index" field is
cpufreq_table[index].index.