NAME

cpufreq - CPU frequency control framework

SYNOPSIS

device cpufreq

#include <sys/cpu.h>

int
cpufreq_levels(*device_t dev, struct cf_level *levels, int *count*);

int
cpufreq_set(*device_t dev, const struct cf_level *level, int priority*);

int
cpufreq_get(*device_t dev, struct cf_level *level*);

int
cpufreq_drv_settings(*device_t dev, struct cf_setting *sets, int *count*);

int
cpufreq_drv_type(*device_t dev, int *type*);

int
cpufreq_drv_set(*device_t dev, const struct cf_setting *set*);

int
cpufreq_drv_get(*device_t dev, struct cf_setting *set*);

DESCRIPTION

The **cpufreq** driver provides a unified kernel and user interface to CPU frequency control drivers. It combines multiple drivers offering different settings into a single interface of all possible levels. Users can access this interface directly via sysctl(8) or by indicating to */etc/rc.d/power_profile* that it should switch settings when the AC line state changes via rc.conf(5).

SYSCTL VARIABLES

These settings may be overridden by kernel drivers requesting alternate settings. If this occurs, the original values will be restored once the condition has passed (e.g., the system has cooled sufficiently). If a sysctl cannot be set due to an override condition, it will return EPERM.

The frequency cannot be changed if TSC is in use as the timecounter and the hardware does not support invariant TSC. This is because the timecounter system needs to use a source that has a constant rate. (On invariant TSC hardware, the TSC runs at the P0 rate regardless of the configured P-state.) Modern hardware mostly has invariant TSC. The timecounter source can be changed with the *kern.timecounter.hardware* sysctl. Available modes are in *kern.timecounter.choice* sysctl entry.

dev.cpu.%d.freq
: Current active CPU frequency in MHz.

dev.cpu.%d.freq_driver
: The specific **cpufreq** driver used by this cpu.

dev.cpu.%d.freq_levels
: Currently available levels for the CPU (frequency/power usage). Values are in units of MHz and milliwatts.

dev.DEVICE.%d.freq_settings
: Currently available settings for the driver (frequency/power usage). Values are in units of MHz and milliwatts. This is helpful for understanding which settings are offered by which driver for debugging purposes.

debug.cpufreq.lowest
: Lowest CPU frequency in MHz to offer to users. This setting is also accessible via a tunable with the same name. This can be used to disable very low levels that may be unusable on some systems.

debug.cpufreq.verbose
: Print verbose messages. This setting is also accessible via a tunable with the same name.

debug.hwpstate_pstate_limit
: If enabled, the AMD hwpstate driver limits administrative control of P-states (including by powerd(8)) to the value in the 0xc0010061 MSR, known as "PStateCurLim[CurPstateLimit]." It is disabled (0) by default. On some hardware, the limit register seems to simply follow the configured P-state, which results in the inability to ever raise the P-state back to P0 from a reduced frequency state.

SUPPORTED DRIVERS

The following device drivers offer absolute frequency control via the **cpufreq** interface. Usually, only one of these can be active at a time.

acpi_perf	ACPI CPU performance states
est(4)	Intel Enhanced SpeedStep
hwpstate	AMD Cool’n’Quiet2 used in K10 through Family 17h
hwpstate_intel(4)	Intel SpeedShift driver
ichss	Intel SpeedStep for ICH
powernow	AMD PowerNow! and Cool’n’Quiet for K7 and K8
smist	Intel SMI-based SpeedStep for PIIX4

The following device drivers offer relative frequency control and have an additive effect:

acpi_throttle	ACPI CPU throttling
p4tcc	Pentium 4 Thermal Control Circuitry

KERNEL INTERFACE

Kernel components can query and set CPU frequencies through the **cpufreq** kernel interface. This involves obtaining a **cpufreq** device, calling **cpufreq_levels**() to get the currently available frequency levels, checking the current level with **cpufreq_get**(), and setting a new one from the list with **cpufreq_set**(). Each level may actually reference more than one **cpufreq** driver but kernel components do not need to be aware of this. The *total_set* element of *struct cf_level* provides a summary of the frequency and power for this level. Unknown or irrelevant values are set to CPUFREQ_VAL_UNKNOWN.

The **cpufreq_levels**() method takes a **cpufreq** device and an empty array of *levels*. The *count* value should be set to the number of levels available and after the function completes, will be set to the actual number of levels returned. If there are more levels than *count* will allow, it should return E2BIG.

The **cpufreq_get**() method takes a pointer to space to store a *level*. After successful completion, the output will be the current active level and is equal to one of the levels returned by **cpufreq_levels**().

The **cpufreq_set**() method takes a pointer a *level* and attempts to activate it. The *priority* (i.e., CPUFREQ_PRIO_KERN) tells **cpufreq** whether to override previous settings while activating this level. If *priority* is higher than the current active level, that level will be saved and overridden with the new level. If a level is already saved, the new level is set without overwriting the older saved level. If **cpufreq_set**() is called with a NULL *level*, the saved level will be restored. If there is no saved level, **cpufreq_set**() will return ENXIO. If *priority* is lower than the current active level’s priority, this method returns EPERM.

DRIVER INTERFACE

Kernel drivers offering hardware-specific CPU frequency control export their individual settings through the **cpufreq** driver interface. This involves implementing these methods: **cpufreq_drv_settings**(),

cpufreq_drv_type(), **cpufreq_drv_set**(), and **cpufreq_drv_get**(). Additionally, the driver must attach a device as a child of a CPU device so that these methods can be called by the **cpufreq** framework.

The **cpufreq_drv_settings**() method returns an array of currently available settings, each of type *struct cf_setting*. The driver should set unknown or irrelevant values to CPUFREQ_VAL_UNKNOWN. All the following elements for each setting should be returned:

```
struct cf_setting {
        int         freq;     /* CPU clock in MHz or 100ths of a percent. */
        int         volts;    /* Voltage in mV. */
        int         power;    /* Power consumed in mW. */
        int         lat;      /* Transition latency in us. */
        device_t dev;         /* Driver providing this setting. */
};
```

On entry to this method, *count* contains the number of settings that can be returned. On successful completion, the driver sets it to the actual number of settings returned. If the driver offers more settings than *count* will allow, it should return E2BIG.

The **cpufreq_drv_type**() method indicates the type of settings it offers, either CPUFREQ_TYPE_ABSOLUTE or CPUFREQ_TYPE_RELATIVE. Additionally, the driver may set the CPUFREQ_FLAG_INFO_ONLY flag if the settings it provides are information for other drivers only and cannot be passed to **cpufreq_drv_set**() to activate them.

The **cpufreq_drv_set**() method takes a driver setting and makes it active. If the setting is invalid or not currently available, it should return EINVAL.

The **cpufreq_drv_get**() method returns the currently-active driver setting. The *struct cf_setting* returned must be valid for passing to **cpufreq_drv_set**(), including all elements being filled out correctly. If the driver cannot infer the current setting (even by estimating it with **cpu_est_clockrate**()) then it should set all elements to CPUFREQ_VAL_UNKNOWN.

SEE ALSO

acpi(4), est(4), timecounters(4), powerd(8), sysctl(8)

AUTHORS

Nate Lawson
Bruno Ducrot contributed the *powernow* driver.

BUGS

The following drivers have not yet been converted to the **cpufreq** interface: longrun(4).

Notification of CPU and bus frequency changes is not implemented yet.

When multiple CPUs offer frequency control, they cannot be set to different levels and must all offer the same frequency settings.