NAME

sensors.conf − libsensors configuration file

DESCRIPTION

sensors.conf describes how libsensors, and so all programs using it, should translate the raw readings from the kernel modules to real−world values.

SEMANTICS

On a given system, there may be one or more hardware monitoring chips. Each chip may have several features. For example, the LM78 monitors 7 voltage inputs, 3 fans and one temperature. Feature names are standardized. Typical feature names are in0, in1, in2... for voltage inputs, fan1, fan2, fan3... for fans and temp1, temp2, temp3... for temperature inputs.

Each feature may in turn have one or more sub−features, each representing an attribute of the feature: input value, low limit, high limit, alarm, etc. Sub−feature names are standardized as well. For example, the first voltage input (in0) would typically have sub−features in0_input (measured value), in0_min (low limit), in0_max (high limit) and in0_alarm (alarm flag). Which sub−features are actually present depend on the exact chip type.

The *sensors.conf* configuration file will let you configure each chip, feature and sub−feature in a way that makes sense for your system.

The rest of this section describes the meaning of each configuration statement.

CHIP STATEMENT

A *chip* statement selects for which chips all following *compute*, *label*, *ignore* and *set* statements are meant. A chip selection remains valid until the next *chip* statement. Example:

```
chip "lm78−*" "lm79−*"
```

If a chip matches at least one of the chip descriptions, the following configuration lines are examined for it, otherwise they are ignored.

A chip description is built from several elements, separated by dashes. The first element is the chip type, the second element is the name of the bus, and the third element is the hexadecimal address of the chip. Such chip descriptions are printed by sensors(1) as the first line for every chip.

The name of the bus is either *isa*, *pci*, *virtual*, *spi-**, *i2c-N* or *mdio* with *N* being a bus number as bound with a *bus* statement. This list isn't necessarily exhaustive as support for other bus types may be added in the future.

You may substitute the wildcard operator* for e very element. Note however that it wouldn't make any sense to specify the address without the bus type, so the address part is plain omitted when the bus type isn't specified. Here is how you would express the following matches:

LM78 chip at address 0x2d on I2C bus 1	lm78−i2c−1−2d
LM78 chip at address 0x2d on any I2C bus	lm78−i2c−*−2d
LM78 chip at address 0x290 on the ISA bus	lm78−isa−0290
Any LM78 chip on I2C bus 1	lm78−i2c−1−*
Any LM78 on any I2C bus	lm78−i2c−*−*
Any LM78 chip on the ISA bus	lm78−isa−*
Any LM78 chip	lm78−*

Any chip at address 0x2d on I2C bus 1	*−i2c−1−2d
Any chip at address 0x290 on the ISA bus	*−isa−0290

If several chip statements match a specific chip, they are all considered.

LABEL STATEMENT

A *label* statement describes how a feature should be called. Features without a *label* statement are just called by their feature name. Applications can use this to label the readings they present. Example:

label in3 "+5V"

The first argument is the feature name. The second argument is the feature description.

Note that you must use the raw feature name, which is not necessarily the one displayed by "sensors" by default. Use "sensors −u" to see the raw feature names. Same applies to all other statement types below.

IGNORE STATEMENT

An *ignore* statement is a hint that a specific feature should be ignored - probably because it returns bogus values (for example, because a fan or temperature sensor is not connected). Example:

ignore fan1

The only argument is the feature name. Please note that this does not disable anything in the actual sensor chip; it simply hides the feature in question from libsensors users.

COMPUTE STATEMENT

A *compute* statement describes how a feature's raw value should be translated to a real−world value, and how a real−world value should be translated back to a raw value again. This is most useful for voltage sensors, because in general sensor chips have a limited range and voltages outside this range must be divided (using resistors) before they can be monitored. Example:

compute in3 ((6.8/10)+1)*@, @/((6.8/10)+1)

The example above expresses the fact that the voltage input is divided using two resistors of values 6.8 Ohm and 10 Ohm, respectively. See the **VOLTAGE COMPUTATION DETAILS** section below for details.

The first argument is the feature name. The second argument is an expression which specifies how a raw value must be translated to a real−world value; '@' stands here for the raw value. This is the formula which will be applied when reading values from the chip. The third argument is an expression that specifies how a real−world value should be translated back to a raw value; '@' stands here for the real−world value. This is the formula which will be applied when writing values to the chip. The two formulas are obviously related, and are separated by a comma.

A *compute* statement applies to all sub−features of the target feature for which it makes sense. For example, the above example would affect sub−features in3_min and in3_max (which are voltage values) but not in3_alarm (which is a boolean flag.)

The following operators are supported in *compute* statements:

+ − * / ( ) ^ '

^x means exp(x) and 'x means ln(x).

You may use the name of sub−features in these expressions; current readings are substituted. You should be

careful though to avoid circular references.

If at any moment a translation between a raw and a real−world value is called for, but no *compute* statement applies, a one−on−one translation is used instead.

SET STATEMENT

A *set* statement is used to write a sub−feature value to the chip. Of course not all sub−feature values can be set that way, in particular input values and alarm flags can not. Valid sub−features are usually min/max limits. Example:

```
set in3_min  5 * 0.95
set in3_max  5 * 1.05
```

The example above basically configures the chip to allow a 5% deviance for the +5V power input.

The first argument is the feature name. The second argument is an expression which determines the written value. If there is an applying *compute* statement, this value is fed to its third argument to translate it to a raw value.

You may use the name of sub−features in these expressions; current readings are substituted. You should be careful though to avoid circular references.

Please note that *set* statements are only executed by sensors(1) when you use the **−s** option. Typical graphical sensors applications do not care about these statements at all.

BUS STATEMENT

A *bus* statement binds the description of an I2C or SMBus adapter to a bus number. This makes it possible to refer to an adapter in the configuration file, independent of the actual correspondence of bus numbers and actual adapters (which may change from moment to moment). Example:

```
bus "i2c−0" "SMBus PIIX4 adapter at e800"
```

The first argument is the bus number. It is the literal text *i2c−*, followed by a number. As there is a dash in this argument, it must always be quoted.

The second argument is the adapter name, it must match exactly the adapter name as it appears in */sys/class/i2c−adapter/i2c−*/name*. It should always be quoted as well as it will most certainly contain spaces or dashes.

The *bus* statements may be scattered randomly throughout the configuration file; there is no need to place the bus line before the place where its binding is referred to. Still, as a matter of good style, we suggest you place all *bus* statements together at the top of your configuration file.

Running **sensors --bus-list** will generate these lines for you.

In the case where multiple configuration files are used, the scope of each *bus* statement is the configuration file it was defined in. This makes it possible to have bus statements in all configuration files which will not unexpectedly interfere with each other.

STATEMENT ORDER

Statements can go in any order, however it is recommended to put 'set fanX_div' statements before 'set fanX_min' statements, in case a driver doesn't preserve the fanX_min setting when the fanX_div value is changed. Even if the driver does, it's still better to put the statements in this order to avoid accuracy loss.

VOLTAGE COMPUTATION DETAILS

Most voltage sensors in sensor chips have a range of 0 to 4.08 V. This is generally sufficient for the +3.3V and CPU supply voltages, so the sensor chip reading is the actual voltage.

Other supply voltages must be scaled with an external resistor network. The driver reports the value at the chip's pin (0 − 4.08 V), and the userspace application must convert this raw value to an actual voltage. The *compute* statements provide this facility.

Unfortunately the resistor values vary among motherboard types. Therefore you have to figure out the correct resistor values for your own motherboard.

For positive voltages (typically +5V and +12V), two resistors are used, with the following formula:

R1 = R2 * (Vs/Vin − 1)

where:

R1 and R2 are the resistor values
Vs is the actual voltage being monitored
Vin is the voltage at the pin

This leads to the following compute formula:

compute inX @*((R1/R2)+1), @/(((R1/R2)+1)

Real−world formula for +5V and +12V would look like:

compute in3 @*((6.8/10)+1), @/((6.8/10)+1)
compute in4 @*((28/10)+1), @/((28/10)+1)

For negative voltages (typically −5V and −12V), two resistors are used as well, but different boards use different strategies to bring the voltage value into the 0 − 4.08 V range. Some use an inverting amplifier, others use a positive reference voltage. This leads to different computation formulas. Note that most users won't have to care because most modern motherboards make little use of −12V and no use of −5V so they do not bother monitoring these voltage inputs.

Real−world examples for the inverting amplifier case:

compute in5 −@*(240/60), −@/(240/60)
compute in6 −@*(100/60), −@/(100/60)

Real−world examples for the positive voltage reference case:

compute in5 @*(1+232/56) − 4.096*232/56, (@ + 4.096*232/56)/(1+232/56)
compute in6 @*(1+120/56) − 4.096*120/56, (@ + 4.096*120/56)/(1+120/56)

Many recent monitoring chips have a 0 − 2.04 V range, so scaling resistors are even more needed, and resistor values are different.

There are also a few chips out there which have internal scaling resistors, meaning that their value is known and doesn't change from one motherboard to the next. For these chips, the driver usually handles the scaling so it is transparent to the user and no *compute* statements are needed.

TEMPERATURE CONFIGURATION

On top of the usual features, temperatures can have two specific sub−features: temperature sensor type (tempX_type) and hysteresis values (tempX_max_hyst, tempX_crit_hyst etc.).

THERMAL SENSOR TYPES

Available thermal sensor types:

1 PII/Celeron Diode
2 3904 transistor
3 thermal diode
4 thermistor
5 AMD AMDSI
6 Intel PECI

For example, to set temp1 to thermistor type, use:

```
set temp1_type 4
```

Only certain chips support thermal sensor type change, and even these usually only support some of the types above. Please refer to the specific driver documentation to find out which types are supported by your chip.

In theory, the BIOS should have configured the sensor types correctly, so you shouldn't have to touch them, but sometimes it isn't the case.

THERMAL HYSTERESIS MECHANISM

Many monitoring chips do not handle the high and critical temperature limits as simple limits. Instead, they have two values for each limit, one which triggers an alarm when the temperature rises and another one which clears the alarm when the temperature falls. The latter is typically a few degrees below the former. This mechanism is known as hysteresis.

The reason for implementing things that way is that high temperature alarms typically trigger an action to attempt to cool the system down, either by scaling down the CPU frequency, or by kicking in an extra fan. This should normally let the temperature fall in a timely manner. If this was clearing the alarm immediately, then the system would be back to its original state where the temperature rises and the alarm would immediately trigger again, causing an undesirable tight fan on, fan off loop. The hysteresis mechanism ensures that the system is really cool before the fan stops, so that it will not have to kick in again immediately.

So, in addition to tempX_max, many chips have a tempX_max_hyst sub-feature. Likewise, tempX_crit often comes with tempX_crit_hyst. tempX_emerg_hyst, tempX_min_hyst and tempX_lcrit_hyst exist too but aren't as common. Example:

```
set temp1_max     60
set temp1_max_hyst 56
```

The hysteresis mechanism can be disabled by giving both limits the same value.

Note that it is strongly recommended to set the hysteresis value after the limit value it relates to in the configuration file. Implementation details on the hardware or driver side may cause unexpected results if the hysteresis value is set first.

BEEPS

Some chips support alarms with beep warnings. When an alarm is triggered you can be warned by a beeping signal through your computer speaker. On top of per−feature beep flags, there is usually a master beep control switch to enable or disable beeping globally. Enable beeping using:

```
set beep_enable 1
```

or disable it using:

```
set beep_enable 0
```

WHICH STATEMENT APPLIES

If more than one statement of the same kind applies at a certain moment, the last one in the configuration file is used. So usually, you should put more general *chip* statements at the top, so you can overrule them below.

SYNTAX

Comments are introduced by hash marks. A comment continues to the end of the line. Empty lines, and lines containing only whitespace or comments are ignored. Other lines have one of the below forms. There must be whitespace between each element, but the amount of whitespace is unimportant. A line may be continued on the next line by ending it with a backslash; this does not work within a comment, **NAME** or **NUMBER**.

bus **NAME NAME NAME**
chip **NAME−LIST**
label **NAME NAME**
compute **NAME EXPR** , **EXPR**
ignore **NAME**
set **NAME EXPR**

A **NAME** is a string. If it only contains letters, digits and underscores, it does not have to be quoted; in all other cases, you must use double quotes around it. Within quotes, you can use the normal escape−codes from C.

A **NAME−LIST** is one or more **NAME** items behind each other, separated by whitespace.

A **EXPR** is of one of the below forms:

NUMBER
NAME
@
EXPR + **EXPR**
EXPR − **EXPR**
EXPR * **EXPR**
EXPR / **EXPR**
− **EXPR**
ˆ **EXPR**
‘ **EXPR**
(**EXPR**)

A **NUMBER** is a floating−point number. ‘10’, ‘10.4’ and ‘.4’ are examples of valid floating−point numbers; ‘10.’ or ‘10E4’ are not valid.

FILES

/etc/sensors3.conf
/etc/sensors.conf

The system-wide **libsensors**(3) configuration file. /etc/sensors3.conf is tried first, and if it doesn't exist, /etc/sensors.conf is used instead.

/etc/sensors.d

A directory where you can put additional libsensors configuration files. Files found in this directory will be processed in alphabetical order after the default configuration file. Files with names that start with a dot are ignored.

SEE ALSO

libsensors(3)

AUTHOR

Frodo Looijaard and the lm_sensors group https://hwmon.wiki.kernel.org/lm_sensors