PMBus core driver and internal API

Introduction

[from pmbus.org] The Power Management Bus (PMBus) is an open standard power-management protocol with a fully defined command language that facilitates communication with power converters and other devices in a power system. The protocol is implemented over the industry-standard SMBus serial interface and enables programming, control, and real-time monitoring of compliant power conversion products. This flexible and highly versatile standard allows for communication between devices based on both analog and digital technologies, and provides true interoperability which will reduce design complexity and shorten time to market for power system designers. Pioneered by leading power supply and semiconductor companies, this open power system standard is maintained and promoted by the PMBus Implementers Forum (PMBus-IF), comprising 30+ adopters with the objective to provide support to, and facilitate adoption among, users.

Unfortunately, while PMBus commands are standardized, there are no mandatory commands, and manufacturers can add as many non-standard commands as they like. Also, different PMBUs devices act differently if non-supported commands are executed. Some devices return an error, some devices return 0xff or 0xffff and set a status error flag, and some devices may simply hang up.

Despite all those difficulties, a generic PMBus device driver is still useful and supported since kernel version 2.6.39. However, it was necessary to support device specific extensions in addition to the core PMBus driver, since it is simply unknown what new device specific functionality PMBus device developers come up with next.

To make device specific extensions as scalable as possible, and to avoid having to modify the core PMBus driver repeatedly for new devices, the PMBus driver was split into core, generic, and device specific code. The core code (in pmbus_core.c) provides generic functionality. The generic code (in pmbus.c) provides support for generic PMBus devices. Device specific code is responsible for device specific initialization and, if needed, maps device specific functionality into generic functionality. This is to some degree comparable to PCI code, where generic code is augmented as needed with quirks for all kinds of devices.

PMBus device capabilities auto-detection

For generic PMBus devices, code in pmbus.c attempts to auto-detect all supported PMBus commands. Auto-detection is somewhat limited, since there are simply too many variables to consider. For example, it is almost impossible to autodetect which PMBus commands are paged and which commands are replicated across all pages (see the PMBus specification for details on multi-page PMBus devices).

For this reason, it often makes sense to provide a device specific driver if not all commands can be auto-detected. The data structures in this driver can be used to inform the core driver about functionality supported by individual chips.

Some commands are always auto-detected. This applies to all limit commands (lcrit, min, max, and crit attributes) as well as associated alarm attributes. Limits and alarm attributes are auto-detected because there are simply too many possible combinations to provide a manual configuration interface.

PMBus internal API

The API between core and device specific PMBus code is defined in drivers/hwmon/pmbus/pmbus.h. In addition to the internal API, pmbus.h defines standard PMBus commands and virtual PMBus commands.

Standard PMBus commands

Standard PMBus commands (commands values 0x00 to 0xff) are defined in the PMBUs specification.

Virtual PMBus commands

Virtual PMBus commands are provided to enable support for non-standard functionality which has been implemented by several chip vendors and is thus desirable to support.

Virtual PMBus commands start with command value 0x100 and can thus easily be distinguished from standard PMBus commands (which can not have values larger than 0xff). Support for virtual PMBus commands is device specific and thus has to be implemented in device specific code.

Virtual commands are named PMBUS_VIRT_xxx and start with PMBUS_VIRT_BASE. All virtual commands are word sized.

There are currently two types of virtual commands.

- READ commands are read-only; writes are either ignored or return an error.
- RESET commands are read/write. Reading reset registers returns zero (used for detection), writing any value causes the associated history to be reset.

Virtual commands have to be handled in device specific driver code. Chip driver code returns non-negative values if a virtual command is supported, or a negative error code if not. The chip driver may return -ENODATA or any other Linux error code in this case, though an error code other than -ENODATA is handled more efficiently and thus preferred. Either case, the calling PMBus core code will abort if the chip driver returns an error code when reading or writing virtual registers (in other words, the PMBus core code will never send a virtual command to a chip).

PMBus driver information

PMBus driver information, defined in struct pmbus_driver_info, is the main means for device specific drivers to pass information to the core PMBus driver. Specifically, it provides the following information.

- For devices supporting its data in Direct Data Format, it provides coefficients for converting register values into normalized data. This data is usually provided by chip manufacturers in device datasheets.
- Supported chip functionality can be provided to the core driver. This may be necessary for chips which react badly if non-supported commands are executed, and/or to speed up device detection and initialization.
- Several function entry points are provided to support overriding and/or augmenting generic command execution. This functionality can be used to map non-standard PMBus commands to standard commands, or to augment standard command return values with device specific information.

API functions

Functions provided by chip driver

All functions return the command return value (read) or zero (write) if successful. A return value of -ENODATA indicates that there is no manufacturer specific command, but that a standard PMBus command may exist. Any other negative return value indicates that the commands does not exist for this chip, and that no attempt should be made to read or write the standard command.

As mentioned above, an exception to this rule applies to virtual commands, which *must* be handled in driver specific code. See "Virtual PMBus Commands" above for more details.

Command execution in the core PMBus driver code is as follows:

```
if (chip_access_function) {
        status = chip_access_function();
        if (status != -ENODATA)
                return status;
}
if (command >= PMBUS_VIRT_BASE) /* For word commands/registers only */
        return -EINVAL;
return generic_access();
```

Chip drivers may provide pointers to the following functions in struct pmbus_driver_info. All functions are optional.

```
int (*read_byte_data)(struct i2c_client *client, int page, int reg);
```

Read byte from page <page>, register <reg>. <page> may be -1, which means "current page".

```
int (*read_word_data)(struct i2c_client *client, int page, int phase,
                      int reg);
```

Read word from page <page>, phase <pase>, register <reg>. If the chip does not support multiple phases, the phase parameter can be ignored. If the chip supports multiple phases, a phase value of 0xff indicates all phases.

```
int (*write_word_data)(struct i2c_client *client, int page, int reg,
                       u16 word);
```

Write word to page <page>, register <reg>.

```
int (*write_byte)(struct i2c_client *client, int page, u8 value);
```

Write byte to page <page>, register <reg>. <page> may be -1, which means "current page".

```
int (*identify)(struct i2c_client *client, struct pmbus_driver_info *info);
```

Determine supported PMBus functionality. This function is only necessary if a chip driver supports multiple chips, and the chip functionality is not pre-determined. It is currently only used by the generic pmbus driver (pmbus.c).

Functions exported by core driver

Chip drivers are expected to use the following functions to read or write PMBus registers. Chip drivers may also use direct I2C commands. If direct I2C commands are used, the chip driver code must not directly modify the current page, since the selected page is cached in the core driver and the core driver will assume that it is selected. Using pmbus_set_page() to select a new page is mandatory.

```
int pmbus_set_page(struct i2c_client *client, u8 page, u8 phase);
```

Set PMBus page register to <page> and <phase> for subsequent commands. If the chip does not support multiple phases, the phase parameter is ignored. Otherwise, a phase value of 0xff selects all phases.

```
int pmbus_read_word_data(struct i2c_client *client, u8 page, u8 phase,
                         u8 reg);
```

Read word data from <page>, <phase>, <reg>. Similar to i2c_smbus_read_word_data(), but selects page and phase first. If the chip does not support multiple phases, the phase parameter is ignored. Otherwise, a phase value of 0xff selects all phases.

```
int pmbus_write_word_data(struct i2c_client *client, u8 page, u8 reg,
                          u16 word);
```

Write word data to <page>, <reg>. Similar to i2c_smbus_write_word_data(), but selects page first.

```
int pmbus_read_byte_data(struct i2c_client *client, int page, u8 reg);
```

Read byte data from <page>, <reg>. Similar to i2c_smbus_read_byte_data(), but selects page first. <page> may be -1, which means "current page".

```
int pmbus_write_byte(struct i2c_client *client, int page, u8 value);
```

Write byte data to <page>, <reg>. Similar to i2c_smbus_write_byte(), but selects page first. <page> may be -1, which means "current page".

```
void pmbus_clear_faults(struct i2c_client *client);
```

Execute PMBus "Clear Fault" command on all chip pages. This function calls the device specific write_byte function if defined. Therefore, it must _not_ be called from that function.

```
bool pmbus_check_byte_register(struct i2c_client *client, int page, int reg);
```

Check if byte register exists. Return true if the register exists, false otherwise. This function calls the device specific write_byte function if defined to obtain the chip status. Therefore, it must _not_ be called from that function.

```
bool pmbus_check_word_register(struct i2c_client *client, int page, int reg);
```

Check if word register exists. Return true if the register exists, false otherwise. This function calls the device specific write_byte function if defined to obtain the chip status. Therefore, it must _not_ be called from that function.

```
int pmbus_do_probe(struct i2c_client *client, struct pmbus_driver_info *info);
```

Execute probe function. Similar to standard probe function for other drivers, with the pointer to struct pmbus_driver_info as additional argument. Calls identify function if supported. Must only be called from device probe function.

```
const struct pmbus_driver_info
      *pmbus_get_driver_info(struct i2c_client *client);
```

Return pointer to struct pmbus_driver_info as passed to pmbus_do_probe().

PMBus driver platform data

PMBus platform data is defined in include/linux/pmbus.h. Platform data currently provides a flags field with four bits used:

```
#define PMBUS_SKIP_STATUS_CHECK                 BIT(0)

#define PMBUS_WRITE_PROTECTED                   BIT(1)

#define PMBUS_NO_CAPABILITY                     BIT(2)

#define PMBUS_READ_STATUS_AFTER_FAILED_CHECK    BIT(3)

struct pmbus_platform_data {
        u32 flags;              /* Device specific flags */

        /* regulator support */
        int num_regulators;
        struct regulator_init_data *reg_init_data;
};
```

Flags

PMBUS_SKIP_STATUS_CHECK

During register detection, skip checking the status register for communication or command errors.

Some PMBus chips respond with valid data when trying to read an unsupported register. For such chips, checking the status register is mandatory when trying to determine if a chip register exists or not. Other PMBus chips don't support the STATUS_CML register, or report communication errors for no explicable reason. For such chips, checking the status register must be disabled.

Some i2c controllers do not support single-byte commands (write commands with no data, i2c_smbus_write_byte()). With such controllers, clearing the status register is impossible, and the PMBUS_SKIP_STATUS_CHECK flag must be set.

PMBUS_WRITE_PROTECTED

Set if the chip is write protected and write protection is not determined by the standard WRITE_PROTECT command.

PMBUS_NO_CAPABILITY

Some PMBus chips don't respond with valid data when reading the CAPABILITY register. For such chips, this flag should be set so that the PMBus core driver doesn't use CAPABILITY to determine it's behavior.

PMBUS_READ_STATUS_AFTER_FAILED_CHECK

Read the STATUS register after each failed register check.

Some PMBus chips end up in an undefined state when trying to read an unsupported register. For such chips, it is necessary to reset the chip pmbus controller to a known state after a failed register check. This can be done by reading a known register. By setting this flag the driver will try to read the STATUS register after each failed register check. This read may fail, but it will put the chip into a known state.