AVR32412: AVR32 AP7 TWI Driver

Features

- **Linux I2C driver model**
  - **How to access the AVR32 TWI in master mode under Linux**
- **I2C Linux kernel configuration**
- **Driver installation**
- **Linux I2C device interface**
  - **I2C communication**
  - **SMBus communication**
- **Example application**

1 Introduction

This application note covers the configuration, setup and usage of the I2C framework on Linux®. The application note gives also examples of how the Linux API can be used to form SMBus™ commands which can be used on I2C compatible hardware.

On the Atmel microcontrollers a Two Wire Interface (TWI) is available that is compatible with the Phillips I2C protocol but other hardware modules can also be used to emulate the I2C protocol. Currently are drivers available that support GPIOs as I2C interface. This is also covered in this application note.

32-bit AVR® Microcontrollers

Application Note

Rev. 32083A-AVR32-08/08

2 Linux I2C introduction

For more details on the specification of the bus and the protocol take a look at the NXP web-site http://nxp.com.

A variant of I2C, called SMBus, is now frequently used in systems for system management and monitoring. It is possible to mix and match I2C and SMBus devices, although there are some differences. Linux offers an API to form SMBus commands by using the I2C hardware. The user may check if the underlying hardware and software supports this by dedicated API calls. However, the user needs also to make sure that the electrical constraints are met because these can be not compliant on some systems. More information about the SMBus specification can be found on http://smbus.org/.

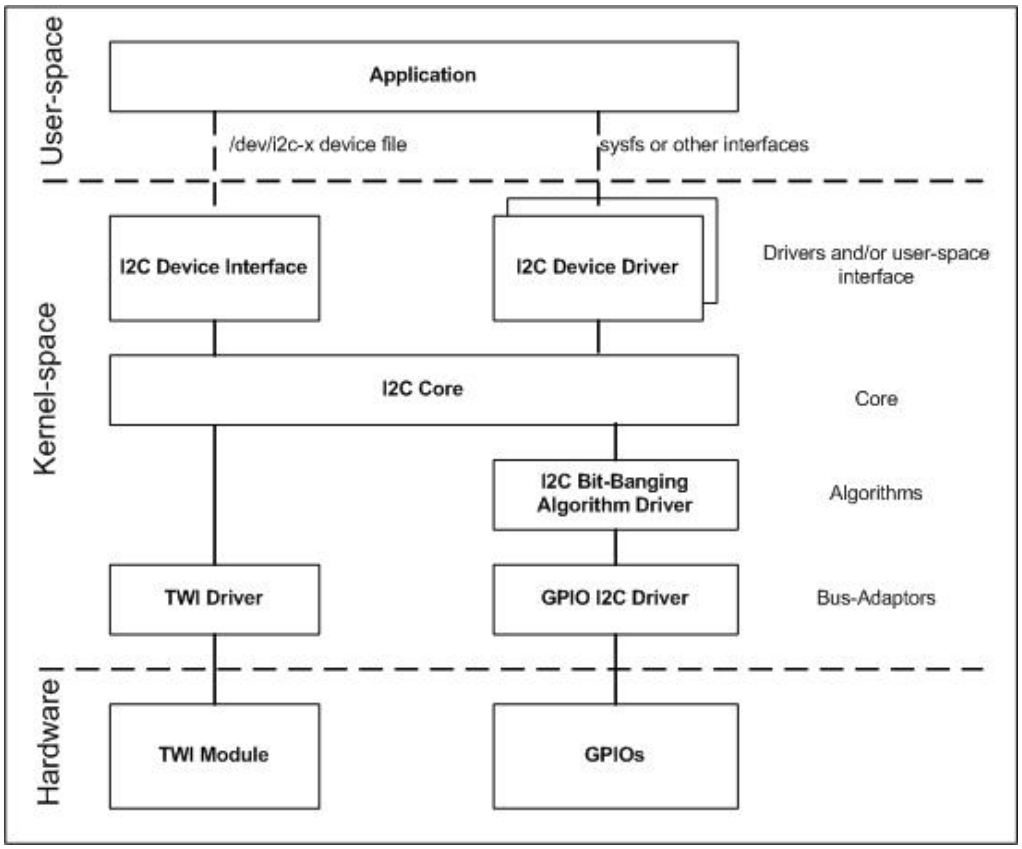
2.1 Linux I2C driver model

The Linux I2C driver model consists of several parts. Following list describes the terminology that is used in order to describe the whole model:

- Hardware bus: A low level bus that has a specific interface to handle transmissions. A hardware bus on the AVR®32 is for example the TWI module or a set of GPIOs. Other peripherals can also be used but currently no driver is available.
- Adapter: An adapter is a part specific implementation that works with a specific hardware bus. An example is the I2C adapter that works with the TWI module.
- Algorithm: An algorithm driver may work together with the adapter in order to handle the low level communication. An algorithm driver is needed when most of the protocol handling is not implemented in hardware. This is for instance the case when GPIOs are used to “bit-bang” the data. The algorithm driver could also be implemented into the adapter but this makes it impossible for other adapters to use the same algorithm.
- I2C driver: An I2C driver is a kernel driver which controls a particular type of device. Many drivers for various devices are available in the LM-sensors project (http://www.lm-sensors.org). Each instance of a device (for example the same type of device on different buses) is handled as a separate client. The driver can offer several interfaces in order to interact with user-space applications. A kernel I2C driver may not be needed if the device is controlled through the I2C device interface from a user-space application.
- I2C device interface: The I2C device interface driver provides an easy access for a user-space application to the available I2C busses. Because of this interface a kernel device driver may not be necessary.

The block diagram in Figure 2-1 shows the dependencies of the above mentioned parts on the AVR32.

Figure 2-1 Linux I2C driver model on the AVR32



On the AVR32 Linux operating system are currently two hardware interfaces supported by available drivers. These are:

- The TWI module
- GPIOs (General Purpose Input Output), several general purpose pins that can be used to emulate the I2C protocol (this is also called bit-banging).

Most common is to use the dedicated TWI (Two Wire Interface) module. This module integrates all relevant parts of the protocol in hardware and reduces therefore the needed software part.

More information about the Linux I2C driver model can be found in the Linux kernel source code directory under Documentation/i2c/.

Also a good article series about the I2C driver model is available on the Linux Journal web-site:

The Driver Model Core: http://www.linuxjournal.com/article/6717

I2C Drivers, Part I: http://www.linuxjournal.com/article/7136

I2C Drivers, Part II: http://www.linuxjournal.com/article/7252

3 Linux kernel configuration

The TWI driver needs to be selected in the kernel configuration before usage. AVR32 patches for the Linux kernel or the git repository are available from

http://avr32linux.org, The Linux kernel comes also with the Buildroot build system which is most likely the way many users prefer. More information about Buildroot, Linux kernel sources and so forth is available in the application note “AVR32737: AVR32 AP7 Linux Getting Started”. This application and more Linux related documentation is available from www.atmel.com/avr32. To configure the Linux kernel in a Buildroot environment run in the source directory following command:

```
make linux26-menuconfig
```

It is also possible to start the configuration from the Linux kernel source directory. To do so run in the Linux kernel source directory:

```
make ARCH=avr32 menuconfig
```

This will put you in the Linux configuration menu. A module selection can be done by pressing the space bar on the keyboard. There are two select options:

- built-in: Module is built into the Linux kernel.
- module: Module is not part of the kernel by default but can be loaded into the kernel during run-time.

To enable general I2C support, activate “Device Drivers -> I2C support” in the configuration menu.

3.1 GPIO-I2C setup

In order to build the adapter module and algorithm module needed for the GPIO configuration following selections in the kernel configuration are needed:

- Device Drivers -> I2C support ->I2C Algorithms -> I2C bit-banging interfaces
- Device Drivers -> I2C support -> I2C Hardware bus support -> GPIO-based bit-banging I2C (by selecting this option the above one will be selected automatically)

After the selection of the driver modules the device must be added to the system in the board code. Because every free GPIO pin can be used to emulate the I2C bus a selection must be done.

On the NGW100 the same pins that are used for the TWI module can be configured for a GPIO I2C bus in the kernel configuration. The selection is available in “System Type and features -> Use GPIO for i2c instead of built-in TWI module” in the kernel configuration menu. If other pins should be used or this configuration option is not available for the selected board the board setup code has to be edited.

The settings for the GPIO I2C setup are passed to the adapter by the platform data. The data is defined as a structure in include/linux/i2c-gpio.h and looks as follows:

```
struct i2c_gpio_platform_data {
        unsigned int    sda_pin;
        unsigned int    scl_pin;
        int             udelay;
        int             timeout;
        unsigned int    sda_is_open_drain:1;
        unsigned int    scl_is_open_drain:1;
        unsigned int    scl_is_output_only:1;
};
```

The description of the structure members are as listed in Table 3-1.

Table 3-1 i2c_gpio_platform_data structure members

Structure member	Function
`sda_pin`	GPIO pin ID to use for SDA
`scl_pin`	GPIO pin ID to use for SCL
`udelay`	Signal toggle delay. SCL frequency is (500 / `udelay`) kHz
`timeout`	Clock stretching timeout in jiffies. If the slave keeps SCL low for longer than this, the transfer will time out.
`sda_is_open_drain`	SDA is configured as open drain, i.e. the pin isn't actively driven high when setting the output value high. `gpio_get_value()` must return the actual pin state even if the pin is configured as an output.
`scl_is_open_drain`	SCL is set up as open drain. Same requirements as for `sda_is_open_drain` apply.
`scl_is_output_only`	SCL output drivers cannot be turned off.

Following code is derived from the NGW100 board code and shall serve as an example configuration for the needed data structures:

```
static struct i2c_gpio_platform_data i2c_gpio_data = {
        .sda_pin                = GPIO_PIN_PA(6),
        .scl_pin                = GPIO_PIN_PA(7),
        .sda_is_open_drain      = 1,
        .scl_is_open_drain      = 1,
        .udelay                 = 2,    /* close to 100 kHz */
};
static struct platform_device i2c_gpio_device = {
        .name           = "i2c-gpio",
        .id             = 0,
        .dev            = {
                .platform_data  = &i2c_gpio_data,
        },
};
```

Next step is to set up the needed GPIO pins for the I2C. All needed functions and definitions for setting up GPIOs in the board setup code are listed in the header file portmux.h. This file can be found in the Linux sources in the directory include/asm/arch/.

The function

```
at32_select_gpio(unsigned int pin, unsigned long flags)
```

is all we need to configure a GPIO. The pin number can be calculated by using the macros in the file at32ap7000.h or in a similar file if another derivate is used. To select I/O line 0 on the peripheral I/O (PIO) controller B for instance, use the following code to get the pin number for the at32_select_gpio function:

```
GPIO_PIN_PB(0)
```

For the I/O lines on the other PIO controllers A, C and so forth, use GPIO_PIN_PA(), GPIO_PIN_PC() … Available flags for the `at32_select_gpio` function are listed in Table 3-2.

Table 3-2 Possible flags for the GPIO pin selection function `at32_select_gpio`

Flag	Function
AT32_GPIOF_PULLUP	Enables pull-up if GPIO is configured as input
AT32_GPIOF_OUTPUT	Configures the GPIO as output
AT32_GPIOF_HIGH	Sets output pin high
AT32_GPIOF_DEGLITCH	Enables glitch-filter when GPIO is configured as input
AT32_GPIOF_MULTIDRV	Enables output drivers to be configured as open drain to support external drivers on the same pin.

Following code sets up the GPIO pins specified in the above structure and registers the I2C GPIO device with the system. The code is taken from the NGW100 board setup code.

```
at32_select_gpio(i2c_gpio_data.sda_pin, AT32_GPIOF_MULTIDRV |
    AT32_GPIOF_OUTPUT | AT32_GPIOF_HIGH);
at32_select_gpio(i2c_gpio_data.scl_pin, AT32_GPIOF_MULTIDRV |
    AT32_GPIOF_OUTPUT | AT32_GPIOF_HIGH);
platform_device_register(&i2c_gpio_device);
```

More information about the board setup code is available in the document “AVR32744: AVR32 AP7 Linux Custom Board Support”.

3.2 TWI module setup

To enable the TWI driver, “Device Drivers -> I2C support -> I2C Hardware bus support -> Atmel Two-Wire Interface (TWI)” must be selected in the kernel configuration.

After the selection of the driver modules the device must be added to the system in the board code. On the Atmel boards this is done by default by a call to `at32_add_device_twi(unsigned int id)` in the board setup code.

3.3 I2C device interface

The device interface provides easy access to the I2C busses for a user-space application as already described in the previous chapters. In order to add this module to the kernel, activate “Device Drivers -> I2C support ->I2C device interface” in the kernel configuration system.

4 Setting up the I2C modules on the target

After the system has booted the needed modules may have to be loaded when they have been built as modules. If the functionality has already been built into the kernel no further actions have to be taken. Loaded modules can be listed by running “lsmod” on the target. Load modules with the “modprobe” or “insmod” tool.

4.1 TWI module

If not already loaded insert the TWI module driver by running:

```
modprobe i2c-atmeltwi
```

(If the modules are not installed properly e.g. just copied by the user to the system they can be inserted with the insmod tool). The driver may also be remove by calling

```
rmmod i2c-atmeltwi
```

but this is not possible if other modules are loaded that depend on this module. In order to remove the module all other dependencies have to be removed first.

Another module that may be needed to load if not build into the kernel is i2c-core. This can also be done with the modprobe tool.

4.2 GPIO I2C setup

Modules needed for the GPIO I2C set up are `i2c-gpio` and `i2c-algo-bit`. These modules can also be loaded with the “modprobe” tool.

4.3 User-space I2C device interface

The Linux kernel provides an "I2C device interface" which is an I2C driver where the clients are associated with all devices on a particular I2C bus. The interface to the bus is basically a standard character device that lets a user-space application select a particular device on the bus, set its properties and read/write to the bus.

If this “I2C device interface” should be used to gain access from a user-space program rather then from a kernel driver to the I2C core an additional module is needed. Load the user-space I2C device interface with following command (if not built in the kernel):

```
modprobe i2c-dev
```

Each I2C bus is assigned a device node of /dev/i2c-n (where n is the number of the bus). This is done automatically if the mdev (or similar) application is installed on the system. If this is not the case the node can be created by hand with the mknod command. The device nodes are character devices with permission, device name, and major and minor numbers. If the module was loaded successfully and a valid bus is available the /dev directory should contain a device node. Type

```
ls -l /dev/i2c-0
```

to list the properties of the bus 0 device. This could look like this:

```
crwxr-xr-x    1 65534    65534     89,   0 Sep 25  2007 i2c-0
```

The first set of characters indicates that it's a character device and has read, write and execute permissions for owner, group and others. In the output above, the major number for the /dev entry is 89 and the minor number is 0. Make sure you have the same setting.

More information can be found in the Linux kernel source code directory under Documentation/i2c/dev-interface. Example code for using this interface is described in later chapters in this document.

5 Using the I2C device interface

In the documentation supplied with the Linux kernel, the usage of the Linux I2C interface is described in detail. It is a good reference. It is available in the folder Documentation/i2c/. Another source of examples is the LM-sensors package.

There are two interfaces available to do the communication, the read/write interface or the ioctl interface. But independent of the one we choose we will first have to open the device file like this:

```
int device_file;
char filename[10] = "/dev/i2c-0";

if ((device_file = open(filename,O_RDWR)) < 0) {
    /* Error */
    return -1;
}
```

5.1 read / write interface

Before using the read/write interface the slave address needs to be set. This can be done with an ioctl call. Following example shows how a slave address can be set:

```
int slave_address = 0x12; /* The I2C address */
if (ioctl(device_file,I2C_SLAVE,slave_address) < 0) {
    /* Error */
    return -1;
}
```

After setting the slave address it is now possible read from and write to the slave. The data that should be sent to read from the slave is passed in with a write/read call like in the example below:

```
buffer[0] = i2c_slave_register;
buffer[1] = 0x11;
buffer[2] = 0x48;
```

To write:

```
if ( write(device_file,buffer,3) != 3) {
    /* Error */
    return -1;
}
```

To read:

```
if (read(device_file,buffer,1) != 1) {
      /* Error */
      } else {
      /* buffer[0] contains the byte read*/
}
```

5.2 ioctl interface

The ioctl interface provides an advanced interface for a communication on the I2C bus. It provides two methods in order to use the I2C protocol or the SMBus protocol. The adapter should be tested if it provides the needed functionality before starting a communication.

5.2.1 I2C protocol

A standard message structure is available in the header file <linux/i2c-dev.h>.

```
struct i2c_rdwr_ioctl_data {
```

```
        struct i2c_msg *msgs;   /* pointers to i2c_msgs */
        __u32 nmsgs;            /* number of i2c_msgs */
};
```

An I2C message structure is defined in <linux/i2c.h>.

```
struct i2c_msg {
        __u16 addr;     /* slave address*/
        __u16 flags;
        __u16 len;      /* msg length */
        __u8 *buf;     /* pointer to msg data */
};
```

Where:

- `addr` is the slave address.
- `flags` contains whether it is a read or write operation. Other flags are also available such as 10-bit addressing. Flags are listed in the i2c.h header file.
- `len` is the number of bytes to write or read.
- `buf` points to the storage buffer for data to be received or data to be written.

The initialized structure is used as argument for an ioctl call. To do a combined read or write transfer the macro I2C_RDWR can be used. Following example transfers 3 data bytes to the slave. The 3 bytes can be used to transfer a register address and two data bytes.

```
struct i2c_rdwr_ioctl_data work_queue;
uint8_t msg_data[2] = {0,0};

work_queue.nmsgs = 1;
work_queue.msgs = (struct i2c_msg*) malloc(work_queue.nmsgs *
sizeof(struct i2c_msg));
work_queue.msgs[0].len   = 2;
work_queue.msgs[0].flags = 0;
work_queue.msgs[0].addr = 0x1; /* slave address */
msg_data[0] = 0x1; /* register address*/
msg_data[1] = 0xA; /* data */
msg_data[2] = 0xB; /* data */
work_queue.msgs[0].buf = msg_data;
if( ioctl(message.device_handle, I2C_RDWR,
    (unsigned long) &work_queue) < 0)
{
    ;/* Error*/
}
```

5.2.2 SMBus protocol

In order to use the SMBus protocol another structure is needed.

```
struct i2c_smbus_ioctl_data {
        __u8 read_write;
        __u8 command;
```

```
        __u32 size;
        union i2c_smbus_data *data;
};
```

Where:

- `read_write` specifies the a read or a write access. Macros are available in the header file.
- `command` is the first byte that will be transmitted in a read/write transmission. In most cases this is the register address.
- `size` specifies the transfer data size in bytes.
- `data` points to the data storage location.

To ease byte and word access to the data a union is used.

```
union i2c_smbus_data {
        __u8 byte;
        __u16 word;
        __u8 block[I2C_SMBUS_BLOCK_MAX + 2];
        /* block[0] is used for length and one more for PEC */
};
```

A one byte write transfer can be set up like this:

```
        struct i2c_smbus_ioctl_data args;

        args.read_write = I2C_SMBUS_WRITE;
        args.command = 0x01;
        args.size = I2C_SMBUS_BYTE;
        args.data = NULL;
        ioctl(device_file,I2C_SMBUS,&args);
```

The LM-sensors project has developed a custom i2c-dev.h header file that replaces the i2c.h and i2c-dev.h header files from the standard library. This file offers some nice macros in order to ease the SMBus protocol handling. The API is described in the kernel documentation. The header file is available in the source code package of this application note.

6 Custom I2C Linux driver

IC2 device handling can also be done in the kernel by adding an I2C client. Before writing a new driver from scratch a quick search at the LM-sensors project should be considered. Maybe a driver for the device exists already there.

If a new driver has to be written following documents are very useful. Documentation on how to write a Linux driver is available in the application note “AVR32743: AVR32 AP7 Linux Kernel Module Application Example”. In the Linux kernel documentation in the Documentation/i2c/ directory is a document available that describes all needed steps in order to build an I2C client. With these two documents all information should be available to build a custom I2C kernel driver.

Good examples of other IC2 device drivers as said before in the LM-project.

7 Source package information

Included with the application note is an example source code that shows some basic I2C/SMBus usage. The source code is taken from the LM-sensors project with minor changes. The tool can be used to print out the available I2C busses, the functionality of the bus-adapters and connected devices on the busses.

8 References

Linux kernel I2C documentation: Documentation/i2c/ in the Linux kernel sources.

LM-Sensors project: http://www.lm-sensors.org

SMBus specification: http://smbus.org/.

I2C specification: http://nxp.com

Related Atmel application notes available from http://www.atmel.com/products/avr32

- AVR32744: AVR32 AP7 Linux Custom Board Support
- AVR32408: AVR32 AP7 Linux GPIO driver

Linux Journal I2C articles:

- The Driver Model Core: http://www.linuxjournal.com/article/6717
- I2C Drivers, Part I: http://www.linuxjournal.com/article/7136
- I2C Drivers, Part II: http://www.linuxjournal.com/article/7252

Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-
Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com

Technical Support
Avr32@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Request
www.atmel.com/literature

Disclaimer: The information in this document is provided in connection with Atmel products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Atmel products. **EXCEPT AS SET FORTH IN ATMEL'S TERMS AND CONDITIONS OF SALE LOCATED ON ATMEL'S WEB SITE, ATMEL ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF ATMEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.** Atmel makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and product descriptions at any time without notice. Atmel does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in, automotive applications. Atmel's products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.

© 2008 Atmel Corporation. All rights reserved. Atmel®, Atmel logo and combinations thereof, AVR® and others, are the registered trademarks or trademarks of Atmel Corporation or its subsidiaries. Other terms and product names may be trademarks of others.