第 8 章 I2 C 协议

I2 C （ The Inter-Integrated Circuit ）及其 子集 SMBus （ System Management Bus ）均为同步串行接口，普遍存在于桌面电脑和嵌入式设备中。 本章通过实现访问 I2 C EEPROM 和 I2 C RTC 的驱动实例，让我们理解内核如何支持 I2 C /SMBus 主机适配器和客户设备。在结束本章之前，让我们也浏览一下内核支持的两种其它的串行接口：串行外围接口（ SPI ）总线和 1-wire 总线。

所有这些串行接口（ I2 C ， SMBus ， SPI 和 1-wire ），都有两个共同的特性：

· 交换的数据总量较少。

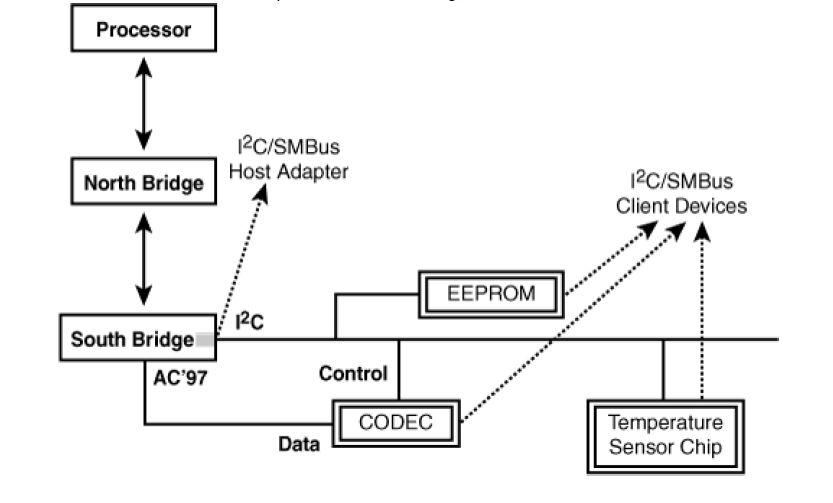
· 数据传输率较低。

I2 C/SMBus 是什么？

I2 C 是广泛用于桌面和笔记本电脑中的串行总线，用于处理器和一些外设之间的接口，这些外设包括 EEPROM 、音频编解码器 以及监控温度、供电电压等参数的专用芯片。此外， I2 C 也在嵌入式设备中大行其道，用于和 RTC ，智能电池电路、多路复用器，端口扩展卡，光收发器，以及其它类似设备之间的通信。由于 I2 C 被大量的微控制器所支持，在当前的市场上可找到大量便宜的 I2 C 设备。

I2 C 和 SMBus 为主－从协议，其通信双方为主机适配器（主控制器）和客户设备（从设备）。主机控制器在桌面电脑上通常为南桥芯片组的一部分；而在嵌入式设备上，通常为微控制器的一部分。图 8.1 显示了在 PC 兼容硬件上 I2 C 总线的例子。

图 8.1. PC 兼容硬件上 的 I2 C/SMBus.



I2 C 及其子集 SMBus 最初分别为 Philips 和 Intel 所开发，均为 2 线接口 。这 2 根线为时钟线和双向数据线，分别被称为串行时钟（ Serial Clock ， SCL ）和串行数据（ Serial Data ， SDA ）。由于 I2 C 总线仅需要一对总线，因此在电路板上占用了更少的空间。因此带来的问题是带宽较窄。 I2 C 在标准模式下支持最高 100Kbps 的传输率，在快速模式下最高可达 400Kbps （然而， SMBus 最高仅支持 100Kbps ）。因此它们仅适用于慢速设备。即使 I2 C 支持双向数据交换，因为仅有一根数据线，故通信是半双工的。

I2 C 和 SMBus 设备使用 7 位地址。协议也支持 10 位地址，但很多设备仅响应 7 位地址，故在总线上最多有 127 个设备。源于协议的主－从特性，设备地址也称为从地址。

I2 C 核心

I2 C 核心由主机适配器驱动和客户驱动可利用的函数和数据结构组成。核心中的公共代码简化了驱动开发者的工作。核心也间接使客户驱动独立于主机适配器，以使客户设备即使用于采用不同 I2 C 主机适配器的电路板上，亦可保持客户驱动运行如常。核心层的此机制及其好处也可在内核中其它的很多设备驱动类中发现，如 PCMCIA ， PCI 和 USB 等。

除了核心，内核的 I2 C 底层由如下组成：

· I2 C 主机适配器的设备驱动。属于总线驱动，通常由适配器驱动和算法（ algorithm ）驱动组成。前者利用后者和 I2 C 总线交互。

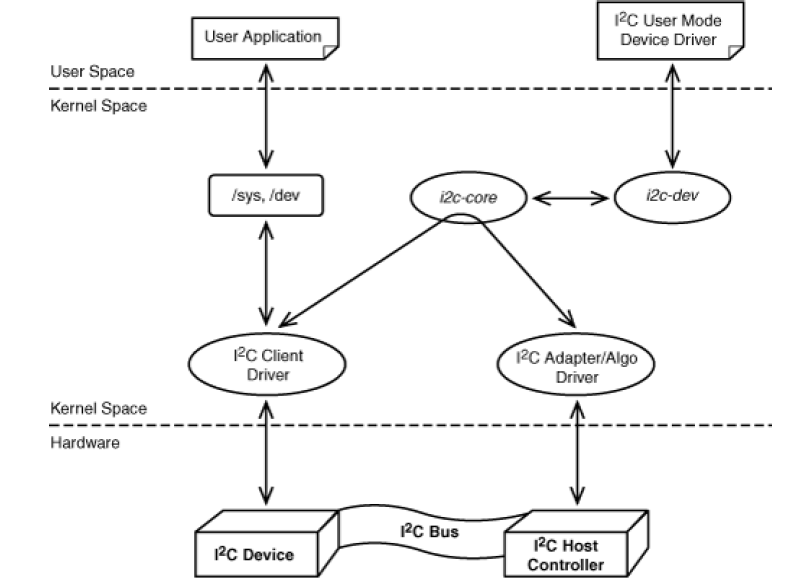
· I2C 客户设备的设备驱动。

· i2c-dev ，允许在用户模式下实现 I2 C 客户驱动。

你更可能的是实现客户驱动，而不是适配器或 algorithm 驱动，因为相比于 I2 C 主机适配器，有多得多的 I2C 设备。因此在本章中，我们将主要讨论客户驱动。

图 8.2 展示了 Linux 的 I2 C 子系统。它显示了 I2 C 内核模块和 I2 C 总线上的主机适配器和客户设备的交互。

图 8.2. Linux I2 C 子系统



由于 SMBus 是 I2 C 的子集，因此仅使用 SMBus 指令和你的设备交互的驱动可工作于 SMBus 和 I2 C 适配器。表 8.1 列出了 I2 C 核心提供的和 SMBus 兼容的数据传输流程。

表 8.1. I2C 核心提供的和 SMBus 兼容的数据 访问函数

函数

作用

i2c\_smbus\_read\_byte()

从设备读取一个字节（不定义位置偏移，使用以前发起的命令的偏移）

i2c\_smbus\_write\_byte()

从设备写入一个字节（使用以前发起的命令的偏移）

i2c\_smbus\_write\_quick()

向设备发送一个比特 ( 取代清单 8.1 中的 Rd/Wr 位 ).

i2c\_smbus\_read\_byte\_data()

从设备指定偏移处读取一个字节

i2c\_smbus\_write\_byte\_data()

向设备指定偏移处写入一个字节

i2c\_smbus\_read\_word\_data()

从设备指定偏移处读取二个字节

i2c\_smbus\_write\_word\_data()

向设备指定偏移处写入二个字节

i2c\_smbus\_read\_block\_data()

从设备指定偏移处读取一块数据 .

i2c\_smbus\_write\_block\_data()

向设备指定偏移处写入一块数据 . (<= 32 字节 )

总线事务

在实现驱动例子之前，通过放大镜观察导线 ，让我们来更好的理解 I2 C 协议。清单 8.1 展示了和 I2 C EEPROM 交互的代码片断，以及在总线上发生的相应的事务。这些事务是在运行代码片断时通过相连的 I2 C 总线分析仪捕获的。这些代码使用的时用户模式的 I2 C 函数（在第 19 章 “ 用户空间的驱动 ” 中，我们将讨论更多的用户模式的 I2 C 编程）。

清单 8.1. I2 C 总线上的事务

Code View:

/\* ... \*/

/\*

\* Connect to the EEPROM. 0x50 is the device address.

\* smbus\_fp is a file pointer into the SMBus device.

\*/

ioctl(smbus\_fp, 0x50, slave);

/\* Write a byte (0xAB) at memory offset 0 on the EEPROM \*/

i2c\_smbus\_write\_byte\_data(smbus\_fp, 0, 0xAB);

/\*

\* This is the corresponding transaction observed

\* on the bus after the write:

\* S 0x50 Wr [A] 0 [A] 0xAB [A] P

\*

\* S is the start bit, 0x50 is the 7-bit slave address (0101000b),

\* Wr is the write command (0b), A is the Accept bit (or

\* acknowledgment) received by the host from the slave, 0 is the

\* address offset on the slave device where the byte is to be

\* written, 0xAB is the data to be written, and P is the stop bit.

\* The data enclosed within [] is sent from the slave to the

\* host, while the rest of the bits are sent by the host to the

\* slave.

\*/

/\* Read a byte from offset 0 on the EEPROM \*/

res = i2c\_smbus\_read\_byte\_data(smbus\_fp, 0);

/\*

\* This is the corresponding transaction observed

\* on the bus after the read:

\* S 0x50 Wr [A] 0 [A] S 0x50 Rd [A] [0xAB] NA P

\*

\* The explanation of the bits is the same as before, except that

\* Rd stands for the Read command (1b), 0xAB is the data received

\* from the slave, and NA is the Reverse Accept bit (or the

\* acknowledgment sent by the host to the slave).

\*/

设备例子： EEPROM

我们的第一个客户驱动例子时 I2 C 总线上的 EEPROM 设置，如图 8.1 所示。几乎所有的笔记本和桌面电脑都有类似的 EEPROM ，用于存储 BIOS 配置信息。例子中的 EEPROM 有两个 memory bank 。相对应于每个 memory bank ，驱动提供 /dev 接口： /dev/eep/0 和 /dev/eep/1 。应用程序在这些节点上操作，和 EEPROM 交换数据。

每个 I2 C/SMBus 客户设备都分配有一个从地址，作为设备标识。例子中的 EEPROM 有两个从地址， SLAVE\_ADDR1 和 SLAVE\_ADDR2 ，分别对应于两个 memory bank 。

例子中驱动所使用的 I2 C 指令和 SMBus 兼容，因此它可以工作于 I2 C 和 SMBus EEPROM 。

初始化

正如所有的驱动类那样， I2 C 客户驱动也有 init() 入口点。初始化用于分配数据结构，向 I2 C 核心注册驱动，将 sysfs 和 Linux 设备模型联系在一起。这些在清单 8.2 中完成。

清单 8.2. 初始化 EEPROM 驱动

Code View:

/\* Driver entry points \*/

static struct file\_operations eep\_fops = {

.owner = THIS\_MODULE,

.llseek = eep\_llseek,

.read = eep\_read,

.ioctl = eep\_ioctl,

.open = eep\_open,

.release = eep\_release,

.write = eep\_write,

};

static dev\_t dev\_number; /\* Allotted Device Number \*/

static struct class \*eep\_class; /\* Device class \*/

/\* Per-device client data structure for each

\* memory bank supported by the driver

\*/

struct eep\_bank {

struct i2c\_client \*client; /\* I2C client for this bank \*/

unsigned int addr; /\* Slave address of this bank \*/

unsigned short current\_pointer; /\* File pointer \*/

int bank\_number; /\* Actual memory bank number \*/

/\* ... \*/ /\* Spinlocks, data cache for

slow devices,.. \*/

};

#define NUM\_BANKS 2 /\* Two supported banks \*/

#define BANK\_SIZE 2048 /\* Size of each bank \*/

struct ee\_bank \*ee\_bank\_list; /\* List of private data

structures, one per bank \*/

/\*

\* Device Initialization

\*/

int \_\_init

eep\_init(void)

{

int err, i;

/\* Allocate the per-device data structure, ee\_bank \*/

ee\_bank\_list = kmalloc(sizeof(struct ee\_bank)\*NUM\_BANKS,

GFP\_KERNEL);

memset(ee\_bank\_list, 0, sizeof(struct ee\_bank)\*NUM\_BANKS);

/\* Register and create the /dev interfaces to access the EEPROM

banks. Refer back to Chapter 5, "Character Drivers" for

more details \*/

if (alloc\_chrdev\_region(&dev\_number, 0,

NUM\_BANKS, "eep") < 0) {

printk(KERN\_DEBUG "Can't register device\n");

return -1;

}

eep\_class = class\_create(THIS\_MODULE, DEVICE\_NAME);

for (i=0; i < NUM\_BANKS;i++) {

/\* Connect the file operations with cdev \*/

cdev\_init(&ee\_bank[i].cdev, &ee\_fops);

/\* Connect the major/minor number to the cdev \*/

if (cdev\_add(&ee\_bank[i].cdev, (dev\_number + i), 1)) {

printk("Bad kmalloc\n");

return 1;

}

class\_device\_create(eep\_class, NULL, (dev\_number + i),

NULL, "eeprom%d", i);

}

/\* Inform the I2C core about our existence. See the section

"Probing the Device" for the definition of eep\_driver \*/

err = i2c\_add\_driver(&eep\_driver);

if (err) {

printk("Registering I2C driver failed, errno is %d\n", err);

return err;

}

printk("EEPROM Driver Initialized.\n");

return 0;

}

清单 8.2 发起了设备节点的创建，但为了完成此过程，需要添加如下内容至 /etc/udev/rules.d/ 目录下合适的规则文件中：

KERNEL:"eeprom[0-1]\*", NAME="eep/%n"

作为从内核收到的 uevent 的响应，将创建 /dev/eep/0 和 /dev/eep/1 。需要从第 n 个 memory bank 读取数据的用户模式程序可以操作 /dev/eep/n 来达到其目的。

清单 8.3 实现了 EEPROM 驱动的 open() 函数。当应用程序打开 /dev/eep/X 时，内核将调用 eep\_open() 。 eep\_open() 在私有区域中存储了每个设备相关的数据结构，因此可以从驱动中的其它函数中直接访问。

清单 8.3. 打开 EEPROM 驱动

int

eep\_open(struct inode \*inode, struct file \*file)

{

/\* The EEPROM bank to be opened \*/

n = MINOR(file->f\_dentry->d\_inode->i\_rdev);

file->private\_data = (struct ee\_bank \*)ee\_bank\_list[n];

/\* Initialize the fields in ee\_bank\_list[n] such as

size, slave address, and the current file pointer \*/

/\* ... \*/

}

探测设备

I2 C 客户驱动，在主机控制器驱动和 I2 C 核心的合作下，使其自身成为从设备，其过程如下：

1. 在初始化过程中，注册 probe() 方法。当相连的主机控制器被检测出， I2 C 核心将调用此方法。在清单 8.2 中， eep\_init() 通过调用 i2c\_add\_driver() 注册 eep\_probe() 。

static struct i2c\_driver eep\_driver =

{

.driver = {

.name = "EEP", /\* Name \*/

},

.id = I2C\_DRIVERID\_EEP, /\* ID \*/

.attach\_adapter = eep\_probe, /\* Probe Method \*/

.detach\_client = eep\_detach, /\* Detach Method \*/

};

i2c\_add\_driver(&eep\_driver); `

设备标识符 I2C\_DRIVERID\_EEP ，对于每个设备应该是唯一的，并在 include/linux/i2c-id.h 中定义之。

2. 当 I2 C 核心调用客户驱动的 probe() 方法时，表明主机适配器已经存在。在 probe() 里将调用 i2c\_probe() ，其参数为驱动所关联的从设备的地址，以及具体的探测函数 attach() 。

清单 8.4 实现了 EEPROM 驱动的 probe() 方法： eep\_probe() 。 normal\_i2c 指明了 EEPROM bank 的地址，它是 i2c\_client\_address\_data 结构体的成员。此结构体中其它的成员能被用于更多的地址控制。你可以通过设置 ignore 字段要求 I2 C 核心忽略一段地址范围。 如果你想绑定一个从地址到一个特殊的主机适配器上，你也可以使用 probe 成员指定（适配器、从地址）对。对于某些场合，这样做有其用处。例如，你的处理器支持两个 I2 C 主机适配器，在总线 1 上 有一个 EEPROM ，总线 2 上有一个温度传感器，两个设备从地址相同。

3. 主机控制器在总线上搜索步骤 2 中指定的从设备。为此，它产生一个总线事务，例如 S SLAVE\_ADDR Wr ， S 是起始位， SLAVE\_ADDR 是设备的数据手册中指定的 7bit 的从地址， Wr 是 “ 总线事务 ” 一节中所描述过的写命令。如果某个运行中的从设备存在于总线上，它将发送确认比特 ([A]) 加以回应。

4. 在步骤 3 中如果主机适配器检测到从设备， I2 C 核心会调用步骤 2 中在 i2c\_probe() 的第三个参数中指定的 attach() 。对于 EEPROM 驱动，此例程为 eep\_attach() ，它将注册和设备关联的客户数据结构，如清单 8.5 所示。如果你的设备需要初始的编程序列（例如，在数字视频接口（ Digital Visual Interface ， DVI ）传输芯片开始工作之前，必须对它的寄存器进行初始化 ），可在此例程中完成这些操作。

清单 8.4. 探测 EEPROM Bank 的存在

#include <linux/i2c.h>

/\* The EEPROM has two memory banks having addresses SLAVE\_ADDR1

\* and SLAVE\_ADDR2, respectively

\*/

static unsigned short normal\_i2c[] = {

SLAVE\_ADDR1, SLAVE\_ADDR2, I2C\_CLIENT\_END

};

static struct i2c\_client\_address\_data addr\_data = {

.normal\_i2c = normal\_i2c,

.probe = ignore,

.ignore = ignore,

.forces = ignore,

};

static int

eep\_probe(struct i2c\_adapter \*adapter)

{

/\* The callback function eep\_attach(), is shown

\* in Listing 8.5

\*/

return i2c\_probe(adapter, &addr\_data, eep\_attach);

}

清单 8.5. Attaching a Client

int

eep\_attach(struct i2c\_adapter \*adapter, int address, int kind)

{

static struct i2c\_client \*eep\_client;

eep\_client = kmalloc(sizeof(\*eep\_client), GFP\_KERNEL);

eep\_client->driver = &eep\_driver; /\* Registered in Listing 8.2 \*/

eep\_client->addr = address; /\* Detected Address \*/

eep\_client->adapter = adapter; /\* Host Adapter \*/

eep\_client->flags = 0;

strlcpy(eep\_client->name, "eep", I2C\_NAME\_SIZE);

/\* Populate fields in the associated per-device data structure \*/

/\* ... \*/

/\* Attach \*/

i2c\_attach\_client(new\_client);

}

检查适配器的功能

每个主机适配器的功能都有限。一个适配器可能不支持表 8.1 中包含的所有命令。例如，它可能支持 SMBus read\_word 命令，但不支持 read\_block 命令。客户驱动在使用这些命令前必须检查适配器是否对其提供支持。

I2 C 核心提供两个函数以完成此功能：

1. i2c\_check\_functionality() 检查某个特定的功能 是否被支持。

2. i2c\_get\_functionality() 返回包含所有被支持功能 的掩码。

在 include/linux/i2c.h 可看到所有可能支持 功能 的列表。

访问设备

为了从 EEPROM 读取数据，首先需要从与此设备节点关联的私有数据域中收集调用线程的信息。其次，使用 I2 C 核心提供的 SMBus 兼容的数据访问例程（表 8.1 显示了可用的函数）读取数据。最后，发送数据至用户空间，并增加内部文件指针，以便下一次的 read()/write() 操作可以从上一次结束处开始。这些步骤在清单 8.6 中完成，此清单忽略了通常的完整性和错误检查。

清单 8.6. 从 EEPROM 读取数据

Code View:

ssize\_t

eep\_read(struct file \*file, char \*buf,

size\_t count, loff\_t \*ppos)

{

int i, transferred, ret, my\_buf[BANK\_SIZE];

/\* Get the private client data structure for this bank \*/

struct ee\_bank \*my\_bank =

(struct ee\_bank \*)file->private\_data;

/\* Check whether the smbus\_read\_word() functionality is

supported \*/

if (i2c\_check\_functionality(my\_bank->client,

I2C\_FUNC\_SMBUS\_READ\_WORD\_DATA)) {

/\* Read the data \*/

while (transferred < count) {

ret = i2c\_smbus\_read\_word\_data(my\_bank->client,

my\_bank->current\_pointer+i);

my\_buf[i++] = (u8)(ret & 0xFF);

my\_buf[i++] = (u8)(ret >> 8);

transferred += 2;

}

/\* Copy data to user space and increment the internal

file pointer. Sanity checks are omitted for simplicity \*/

copy\_to\_user(buffer, (void \*)my\_buf, transferred);

my\_bank->current\_pointer += transferred;

}

return transferred;

}

写数据至设备与此类似，使用的是 i2c\_smbus\_write\_XXX() 函数。

一些 EEPROM 芯片有 RFID （ Radio Frequency Identification ）发送器，用于无线发送存储的信息。用于自动化供应链处理，例如货物监控和资产跟踪。这些 EEPROM 通常通过一个访问保护 bank 来控制对数据 bank 的安全访问。对于此类情况，为了能够操作数据 bank ，驱动还不得不对访问保护 bank 中的相应位进行处理。

为了从用户空间访问 EEPROM 块，需要开发应用程序以操作 /dev/eep/n 。为了读出 EEPROM 块的内容，需要做如下操作：

bash> od –a /dev/eep/0

0000000 S E R # dc4 ff soh R P nul nul nul nul nul nul nul

0000020 @ 1 3 R 1 1 5 3 Z J 1 V 1 L 4 6

0000040 5 1 0 H sp 1 S 2 8 8 8 7 J U 9 9

0000060 H 0 0 6 6 nul nul nul bs 3 8 L 5 0 0 3

0000100 Z J 1 N U B 4 6 8 6 V 7 nul nul nul nul

0000120 nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul

\*

0000400

作为练习，试着去修改 EEPROM 驱动，创建 EEPROM 块的 /sys 接口，而不是 /dev 接口。你可以重用第 5 章清单 5.7“ 使用 Sysfs 控制并口 LED 电路板 ” 中的代码，帮助完成此工作。

其它函数

为了获得全功能的驱动，你需要添加剩余的入口点。这些和第 5 章所讨论的普通字符驱动差别不大，因此未提供代码清单：

· 为了支持 lseek() 系统调用，用于给内部文件指针赋以新值，需要实现 llseek() 函数。内部文件指针存储了有关 EEPROM 访问的状态信息。

· 为了验证数据的完整性， EEPROM 驱动可实现 ioctl() 函数，用于校准并验证存储的数据的校验和。

· EEPROM 中不需要 poll() 和 fsync() 方法。

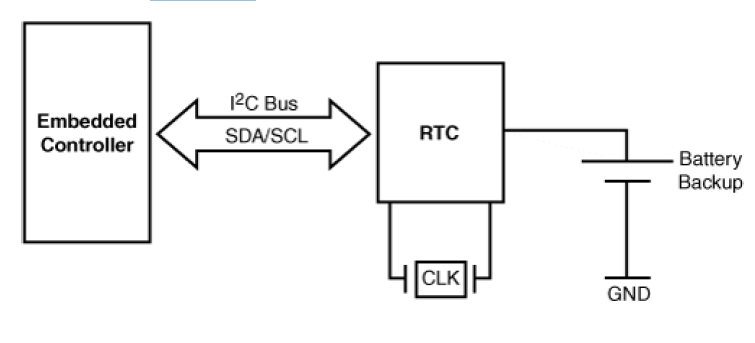
· 如果你选择将驱动编译为一个模块，需要提供 exit() 方法，以注销 设备，并清理客户设备特定的数据结构。从 i2c 核心卸载驱动只需执行如下操作：

i2c\_del\_driver(&eep\_driver);

设备例子：实时时钟

让我们选取通过 I2 C 总线和嵌入式控制器相连的 RTC 芯片的例子。连接框图见图 8.3 。

图 8.3. 嵌入式系统上的 I2 C RTC



设定 RTC 的 I2 C 从地址为 0x60 ，其寄存器空间组织如表 8.2 。

表 8.2. I2 C RTC 的寄存器分布

寄存器名

描述

偏移

RTC\_HOUR\_REG

小时计数

0x0

RTC\_MINUTE\_REG

分钟计数

0x1

RTC\_SECOND\_REG

秒钟计数

0x2

RTC\_STATUS\_REG

标志记中断状态

0x3

RTC\_CONTROL\_REG

使能／禁止 RTC

0x4

让我们的驱动基于前面讨论过的 EEPROM 驱动。我们将假设 I2 C 客户驱动结构、从设备注册和 II2 C 核心函数已经完成 ，仅实现和 RTC 通信的代码。

当 I2 C 核心检测到从地址为 0x60 的设备在 I2 C 总线上时，将调用 myrtc\_attach() 。其调用序列类似于清单 8.5 中的 eep\_attach() 。假定在 myrtc\_attach() 中你必须完成如下的芯片初始化：

1. 将 RTC 状态寄存器（ RTC\_STATUS\_REG ）清零。

2. 通过设置 RTC 控制寄存器（ RTC\_CONTROL\_REG ）中的相应位，启动 RTC （如果它还未开始运行）。

为了完成以上功能，让我们构建一个 i2c\_msg 结构体，使用 i2c\_transfer() 在总线上产生 I2 C 事务。此传输机制为 I2 C 所独有，和 SMBus 并不兼容 。为了向前面讨论过的两个 RTC 寄存器写入数据，你必须构建两个 i2c\_msg 消息。第一个消息设置寄存器偏移。在我们的例子中， RTC\_STATUS\_REG 的值为 3 。第二个消息携带希望写入指定偏移的字节数。在此例中，共两个字节，一个字节写入 RTC\_STATUS\_REG ，另一个写入 RTC\_CONTROL\_REG 。

Code View:

#include <linux/i2c.h> /\* For struct i2c\_msg \*/

int

myrtc\_attach(struct i2c\_adapter \*adapter, int addr, int kind)

{

u8 buf[2];

int offset = RTC\_STATUS\_REG; /\* Status register lives here \*/

struct i2c\_msg rtc\_msg[2];

/\* Write 1 byte of offset information to the RTC \*/

rtc\_msg[0].addr = addr; /\* Slave address. In our case,

this is 0x60 \*/

rtc\_msg[0].flags = I2C\_M\_WR; /\* Write Command \*/

rtc\_msg[0].buf = &offset; /\* Register offset for

the next transaction \*/

rtc\_msg[0].len = 1; /\* Offset is 1 byte long \*/

/\* Write 2 bytes of data (the contents of the status and

control registers) at the offset programmed by the previous

i2c\_msg \*/

rtc\_msg[1].addr = addr; /\* Slave address \*/

rtc\_msg[1].flags = I2C\_M\_WR; /\* Write command \*/

rtc\_msg[1].buf = &buf[0]; /\* Data to be written to control

and status registers \*/

rtc\_msg[1].len = 2; /\* Two register values \*/

buf[0] = 0; /\* Zero out the status register \*/

buf[1] |= ENABLE\_RTC; /\* Turn on control register bits

that start the RTC \*/

/\* Generate bus transactions corresponding to the two messages \*/

i2c\_transfer(adapter, rtc\_msg, 2);

/\* ... \*/

printk("My RTC Initialized\n");

}

因为 RTC 已经被初始化，并开始计时了，你可以通过读取 RTC\_HOUR\_REG ， RTC\_MINUTE\_REG 和 RTC\_SECOND\_REG 来获取当前的时间。其操作如下：

Code View:

#include <linux/rtc.h> /\* For struct rtc\_time \*/

int

myrtc\_gettime(struct i2c\_client \*client, struct rtc\_time \*r\_t)

{

u8 buf[3]; /\* Space to carry hour/minute/second \*/

int offset = 0; /\* Time-keeping registers start at offset 0 \*/

struct i2c\_msg rtc\_msg[2];

/\* Write 1 byte of offset information to the RTC \*/

rtc\_msg[0].addr = addr; /\* Slave address \*/

rtc\_msg[0].flags = 0; /\* Write Command \*/

rtc\_msg[0].buf = &offset; /\* Register offset for

the next transaction \*/

rtc\_msg[0].len = 1; /\* Offset is 1 byte long \*/

/\* Read current time by getting 3 bytes of data from offset 0

(i.e., from RTC\_HOUR\_REG, RTC\_MINUTE\_REG, and RTC\_SECOND\_REG) \*/

rtc\_msg[1].addr = addr; /\* Slave address \*/

rtc\_msg[1].flags = I2C\_M\_RD; /\* Read command \*/

rtc\_msg[1].buf = &buf[0]; /\* Data to be read from hour, minute

and second registers \*/

rtc\_msg[1].len = 3; /\* Three registers to read \*/

/\* Generate bus transactions corresponding to the above

two messages \*/

i2c\_transfer(adapter, rtc\_msg, 2);

/\* Read the time \*/

r\_t->tm\_hour = BCD2BIN(buf[0]); /\* Hour \*/

r\_t->tm\_min = BCD2BIN(buf[1]); /\* Minute \*/

r\_t->tm\_sec = BCD2BIN(buf[2]); /\* Second \*/

return(0);

}

myrtc\_gettime() 实现了总线相关的 RTC 驱动的底层部分。 RTC 驱动的顶层部分应该和内核的 RTC API 保持一致，如第 5 章的 “RTC 子系统 ” 一节中所讨论的。此机制的好处是不管你的 RTC 是位于 PC 的南桥内部，还是如本例一样，位于嵌入式控制器的外部，应用程序可以不加改变而运行。

RTC 通常用 BCD （ Binary Coded Decimal ）格式存储时间，每组位元（ 4 位）表示 0 ～ 9 之间的数，而不是 0 ～ 15 。内核提供了宏 BCD2BIN() 用于将 BCD 码变换成十进制数，以及宏 BIN2BCD() 用于相反的操作。当从 RTC 寄存器读取数据时， myrtc\_gettime() 利用了宏 BCD2BIN() 用于转换。

drivers/rtc/rtc-ds1307.c 提供了 RTC 驱动的实例，用于处理 Dallas/Maxim DS13XX 系列 I2 C 芯片。

作为 2 线总线， I2 C 总线没有从设备用于中断请求的信号线，但一些 I2 C 主机适配器可以中断 CPU ，触发数据传输请求。然而，此中断驱动操作对于 I2C 客户驱动 是透明的，隐藏于 I2 C 核心提供的服务例程里。假设图 8.3 中 I2 C 主机控制器是嵌入式 SoC 的一部分，并有中断 CPU 的能力， myrtc\_attach() 里对 i2c\_transfer() 的调用 将完成如下操作：

· 构建对应于 rtc\_msg[0] 的事务，并使用主机控制器驱动提供的服务例程写入总线。

· 等待直到主机控制器触发发送结束中断，表明 rtc\_msg[0] 已经在信号线上。

· 在中断处理例程里，查看 I2 C 主机控制器状态寄存器，判断是否从 RTC 从设备里接收到确认信号。

· 如果主机控制器的状态和控制寄存器并非全部正确，则返回错误。

· 对于 rtc\_msg[1] 重复同样过程。

I2C-dev

有时，当你需要支持大量慢速的 I2 C 设备时，从用户空间对所有这些设备进行驱动就很有必要了。 I2 C 层支持 i2c-dev 驱动以达到此目的。第 19 章的 “ 用户模式 I2C ” 中，有使用 i2c-dev 实现用户模式 I2 C 驱动的例子。

使用 LM-Sensors 监控硬件

LM-Sensors 项目，主页为 www.lm-sensors.org ，使 Linux 具有硬件监控能力。很多计算机系统使用传感器芯片来监控诸如温度、供电电压以及风扇转速等参数。周期性的检查这些参数是非常重要的。损坏了的 CPU 风扇可能会导致随机、异常的软件问题。如果是医疗设备系统出现故障，其后果将难以想象！

LM-Sensors 利用传感器芯片的设备驱动来排除故障。它利用 sensors 程序产生状态报告， sensors-detect 脚本检查你的系统，并帮助你产生相应的配置文件。

大多数芯片利用 I2 C/SMBus 总线方式向 CPU 提供硬件监控接口。这些设备驱动是 I2 C 客户驱动，但位于 drivers/hwmon/ 目录，而不是 drivers/i2c/chips/ 。具体例子可见 National Semiconductor 公司的 LM87 芯片，它能监控电压、温度和风扇。 drivers/hwmon/lm87.c 为其驱动的具体实现。 I2 C 驱动 ID 号从 1000 到 1999 都保留给了传感器芯片（参见 include/linux/i2c-id.h ）。

也有几个传感器芯片和 CPU 之间的接口采用 ISA/LPC 总线，而不是 I2 C/SMBus 。其它输出模拟量的通过模数转换器（ ADC ）传送给 CPU 。这些芯片的驱动和 I2 C 总线传感器驱动一起都位于 drivers/hwmon/ 目录。非 I2 C 总线传感器驱动的例子是 drivers/hwmon/hdaps.c ，它是加速度传感器驱动，出现在某些 IBM/ 联想的笔记本电脑里，我们在第 7 章 “ 输入驱动 ” 中讨论过。另一个非 I2 C 总线的传感器的例子是 Winbond 83627HF 超级 I/O 芯片，由 drivers/hwmon/w83627hf.c 驱动。

串行外设接口总线（ SPI ）

串行外设接口（ Serial Peripheral Interface ， SPI ）总线和 I2 C 类似，也是串行的主－从接口，集成于很多微控制器内部。和 I2 C 使用 2 线相比，它使用 4 线：串行时钟（ Serial CLocK ， SCLK ），片选（ Chip Select ， CS ） , 主设备输出从设备输入（ Master Out Slave In ， MOSI ），主设备输入从设备输出（ Master In Slave Out ， MISO ）。 MOSI 用于传送数据至从设备， MISO 用于从从设备读出数据。和 I2 C 不同，由于 SPI 总线有专用的数据线用于数据的发送和接收，因此可以工作于全双工。 SPI 的典型速度为几 MHz ，不像 I2C 为几十～几百 KHz ，因此 SPI 吞吐量大得多。

当前市面上可找到的 SPI 外设包括 RF 芯片、智能卡接口、 EEPROM 、 RTC 、触摸传感器、以及 ACD 。

内核提供了一个核心 API 用于通过 SPI 总线交换信息。典型的 SPI 客户驱动如下：

向 SPI 核心注册 probe() 、 remove() 方法。 suspend() 和 resume() 方法可选。

#include <linux/spi/spi.h>

static struct spi\_driver myspi\_driver = {

.driver = {

.name = "myspi",

.bus = &spi\_bus\_type,

.owner = THIS\_MODULE,

},

.probe = myspidevice\_probe,

.remove = \_\_devexit\_p(myspidevice\_remove),

}

spi\_register\_driver(&myspi\_driver);

SPI 核心创建对应于此设备的 spi\_device 结构体，当调用注册的驱动方法时，用作调用参数。

使用函数如 spi\_sync() 和 spi\_async() 和 SPI 设备交换数据。前者等待操作完成，后者当数据传输完成时，异步触发对注册的回调程序的调用。这些数据访问例程被从适当的地方调用，如 SPI 中断处理程序， sysfs 方法，或者定时器处理程序。下面的代码片断演示了 SPI 数据的传输：

#include <linux/spi/spi.h>

struct spi\_device \*spi; /\* Representation of a

SPI device \*/

struct spi\_transfer xfer; /\* Contains transfer buffer

details \*/

struct spi\_message sm; /\* Sequence of spi\_transfer

segments \*/

u8 \*command\_buffer; /\* Data to be transferred \*/

int len; /\* Length of data to be

transferred \*/

spi\_message\_init(&sm); /\* Initialize spi\_message \*/

xfer.tx\_buf = command\_buffer; /\* Device-specific data \*/

xfer.len = len; /\* Data length \*/

spi\_message\_add\_tail(&xfer, &sm); /\* Add the message \*/

spi\_sync(spi, &sm); /\* Blocking transfer request \*/

作为 SPI 设备的例子，我们可参考第 7 章简单讨论过的触摸屏控制器 ADS7846 。其驱动完成如下操作：

使用 spi\_register\_driver() 向 SPI 核心注册 probe() ， remove() ， suspend() 和 resume() 方法。

probe() 方法使用 input\_register\_device() 向输入子系统注册驱动，并使用 request\_irq() 请求中断 。

驱动从其中断服务程序中使用 spi\_async() 收集触摸坐标。此函数当数据传输完成时，触发对注册的回调程序的调用。

如第 7 章所讨论的，回调函数通过输入事件接口 /dev/input/eventX ，使用 input\_report\_abs() 和 input\_report\_key() ，依次报告触摸坐标和点击。诸如 X Windows 和 gpm 这些程序和事件接口紧密合作，响应触摸输入。

通过软件的方式 控制 I/O 引脚，使其符合某种协议进行交互的驱动称为 bit-banging 驱动。 SPI bit-banging 驱动的例子，可参考 drivers/spi/spi\_butterfly.c ，它是用于和 Atmel 公司 AVR 处理器系列 Butterfly 板上的 DataFlash 芯片交互的驱动。将你的主机的并口和 AVR Butterfly 连接在一起，使用专用的 dongle 和 spi\_butterfly 可以进行 bit-banging 操作。 Documentation/spi/butterfly 提供了关于此驱动更详细的描述。

当前没有类似于 i2c-dev 的、针对用户空间的 SPI 驱动。你只能编写内核驱动和 SPI 设备交互。

在嵌入式系统中，你可能会碰到处理器和集成各种功能的协处理器一起工作的解决方案。譬如，飞思卡尔的电源管理和音频组件（ Power Management and Audio Component ， PMAC ）芯片 MC13783 和基于 ARM9 的 i.MX27 控制器协同工作就是这样的一个例子。 PMAC 集成了 RTC ，电池充电器，触摸屏接口， ADC 模块和音频编码。处理器和 PMAC 之间通过 SPI 通信。 SPI 总线不含中断线，通过配置 GPIO 管脚， PMAC 可以从外部中断处理器。

1-Wire 总线

由 Dallas/Maxim 开发的 1-wire 协议使用 1-wire （或 w1 ）总线传送电源和信号。地回路通过其它途径解决。它提供了和慢速设备之间接口的简单途径，减少了空间、费用以及复杂性。使用此协议的设备实例是 ibutton （ www.ibutton.com ），用于感知温度，传送数据，或保存独特的 ID 号。

另一通过单一的引脚提供接口的 w1 芯片是 Dallas/Maxim 的 DS2433 ，它是容量为 4kb 的 1-wire EEPROM 。此芯片的驱动位于 drivers/w1/slaves/w1\_ds2433.c ，通过 sysfs 节点提供对 EEPROM 的访问。

和 w1 设备驱动相关的主要数据结构是 w1\_family 和 w1\_family\_ops ，都定义于 w1\_family.h 中。

调试

为了收集 I2 C 的调试信息，在内核的配置菜单的 Device Drivers-> I2C Support 下，选中 I2C Core debugging messages ， I2C Algorithm debugging messages ， I2C Bus debugging messages 和 I2C Chip debugging messages 。类似的，为了调试 SPI ，需要在 Device Drivers->SPI Support 下选中 Debug Support for SPI drivers 。

为了理解总线上 I2 C 包的数据流， 可在运行清单 8.1 时， 将 I2 C 总线分析仪和你的 电路板板连接在一起。 lm-snesor 包包括 i2cdump 工具，用于输出 I2 C 总线上设备的寄存器中的内容。

Linux I2 C 的邮件列表位于： http://lists.lm-sensors.org/mailman/listinfo/i2c .

查看源代码

在 2.4 版本的源码树中，所有 I2 C/SMBus 相关的源码包含在一个单独的目录 drivers/i2c/ 中， 2.6 版本内核中， I2 C 代码被分层次的组织： drivers/i2c/busses/ 目录包括适配器驱动， drivers/i2c/algos/ 目录包含 algorithm 驱动， drivers/i2c/chips/ 目录包含 客户 驱动。你也可以在源码树中其它的地方发现客户驱动。例如， drivers/sound/ 目录包含使用 I2 C 接口的音频芯片组的驱动。在 Documentation/i2c/ 目录下可找到提示以及更多的例子。

内核的 SPI 服务函数位于 drivers/spi/spi.c 。 ADS7846 触摸控制器的 SPI 驱动由 drivers/input/touchscreen/ads7846.c 实现。第 17 章 “ 内存技术设备 ” 讨论的 MTD 子系统实现了 SPI flash 芯片驱动。其例子为 drivers/mtd/devices/mtd\_dataflash.c ，它实现了访问 Atmel 的 DataFlash SPI 芯片的驱动。

drivers/w1/ 目录包含了内核对 w1 协议的支持。 w1 接口主机控制器的驱动位于 drivers/w1/masters/ ， w1 从设备驱动在 drivers/w1/slaves/ 。

表 8.3 概括了本章使用的主要数据结构，及其在内核源码树中的位置。表 8.4 列出了本章所用到的主要内核编程接口，以及它定义的位置。

表 8.3. 数据机构概述

数据结构

位置

描述

i2c\_driver

include/linux/i2c.h

代表一个 I2 C 驱动

i2c\_client\_address\_data

include/linux/i2c.h

I2 C 客户驱动所负责 的从地址

i2c\_client

include/linux/i2c.h

用于标识一个连接到 I2 C 总线上的芯片

i2c\_msg

include/linux/i2c.h

描述在 I2 C 总线上欲产生的一次传输事务

spi\_driver

include/linux/spi/spi.h

代表一个 SPI 驱动

spi\_device

include/linux/spi/spi.h

代表一个 SPI 设备

spi\_transfer

include/linux/spi/spi.h

SPI 传输缓冲区的细节

spi\_message

include/linux/spi/spi.h

spi\_transfer 分段序列

w1\_family

drivers/w1/w1\_family.h

代表 w1 从驱动

w1\_family\_ops

drivers/w1/w1\_family.h

w1 从驱动入口点

表 8.4. 内核编程接口概述

内核接口

位置

描述

i2c\_add\_driver()

include/linux/i2c.h drivers/i2c/i2c-core.c

向 I2C 核心注册驱动入口点

i2c\_del\_driver()

drivers/i2c/i2c-core.c

从 I2C 核心移除驱动

i2c\_probe()

drivers/i2c/i2c-core.c

定义驱动所负责的从设备地址，如果 i2c 核心探测到某一地址，对应的 attach() 函数将调用

i2c\_attach\_client()

drivers/i2c/i2c-core.c

向相应主机适配器所服务的客户列表增加一个客户

i2c\_detach\_client()

drivers/i2c/i2c-core.c

Detach 一个活动的客户 . 通常在客户驱动或关联的主机适配器注销时进行

i2c\_check\_functionality()

include/linux/i2c.h

验证主机适配器是否支持某功能

i2c\_get\_functionality()

include/linux/i2c.h

获得主机适配器所支持的所有功能的掩码

i2c\_add\_adapter()

drivers/i2c/i2c-core.c

注册主机适配器。

i2c\_del\_adapter()

drivers/i2c/i2c-core.c

注销主机适配器。

SMBus-compatible I2C data access routines

drivers/i2c/i2c-core.c

见表 8.1

i2c\_transfer()

drivers/i2c/i2c-core.c

通过 I2C 总线发送 i2c\_msg 。此函数和 SMBus 不兼容。

spi\_register\_driver()

drivers/spi/spi.c

向 SPI 核心注册驱动入口点。

spi\_unregister\_driver()

include/linux/spi/spi.h

注销 SPI 驱动。

spi\_message\_init()

include/linux/spi/spi.h

初始化 SPI message.

spi\_message\_add\_tail()

include/linux/spi/spi.h

添加一条 SPI 消息到传输列表。

spi\_sync()

drivers/spi/spi.c

通过 SPI 总线同步传输数据。此函数阻塞直至完成。

spi\_async()

include/linux/spi/spi.h

使用完成回调机制，通过 SPI 总线异步传输数据。