Kernel Course: Lecture 18

Working with Embedded Buses

Sam Protsenko

April 15, 2019

GlobalLogic

Agenda

1. I2C Overview
2. RTC Module
3. Kernel Driver
4. Assignments

I2C Overview

Serial vs Parallel

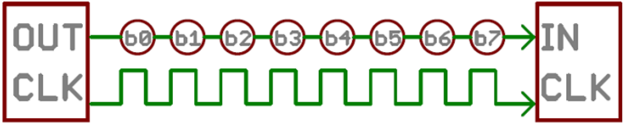


Figure 1: Serial Bus

Serial buses have won:

- (+) Easier to implement the HW
- (+) Can be used on higher frequencies

Parallel buses are used for in-chip communications

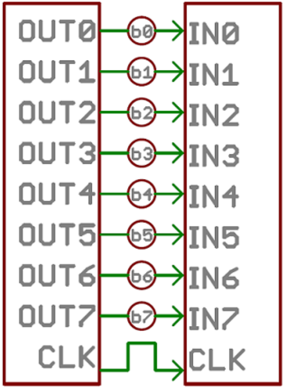


Figure 2: Parallel Bus

Serial Bus: Async vs Sync

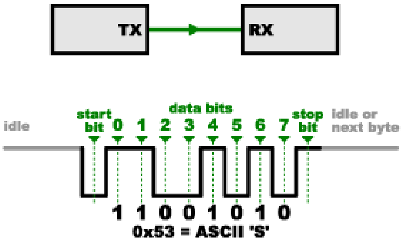


Figure 3: Asynchronous

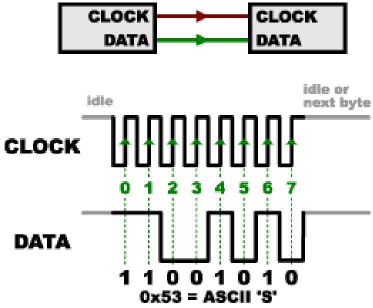


Figure 4: Synchronous

(-) Low-speed transmission

(+) Long distance

(+) High-speed transmission

(-) Short distance (due to clock skew)

Buses on Embedded Boards

- Serial buses:
  - SPI
  - I2C / SMBus
  - UART
  - USB
  - 1-wire
  - CAN
  - PCI-e
- Parallel buses (rarely used in Embedded):
  - ISA
  - PCI
  - Parallel port

Discoverable vs Non-discoverable

Discoverability

Discoverable bus is able to find connected devices during *enumeration*, so that no device tree entry is needed.

- Discoverable buses:
  - USB
  - PCI
- Non-/semi-discoverable buses:
  - SPI
  - **I2C**
  - Platform devices (pseudo-bus)

I2C Basic Facts

- Low-speed bus to connect on-board and external devices to SoC
- Only two wires (clock, data)
- Allows several devices on one bus
- Devices have I2C addresses
- Master/slave
- Master initiates communication and provides clock signal
- Multi-master scheme is allowed
- Keep wires short (up to $0.5\,\mathrm{m}$ @ $400\,\mathrm{kHz}$)

Usual I2C Devices

- Sensors (temp, accel, magn, gyro, …)
- Real-Time Clock (RTC)
- EEPROM
- I/O Expanders
- LCD Displays
- Touch screens
- …a lot more, wherever low-speed bus is OK

I2C Bus Example

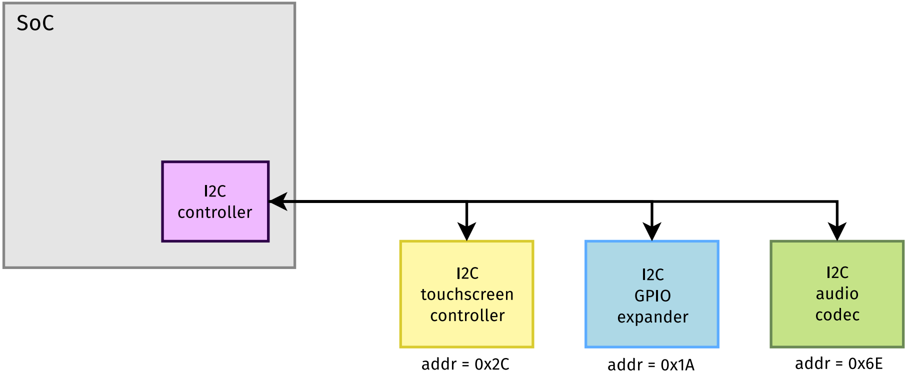


Figure 5: Client Devices on I2C Bus

I2C Connection

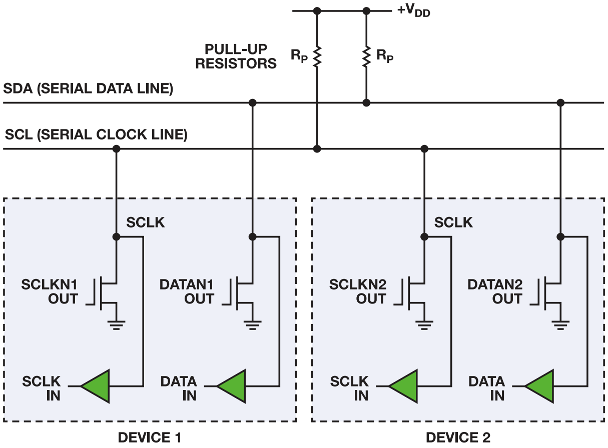


Figure 6: Scheme of Regular I2C Connection

Open Drain

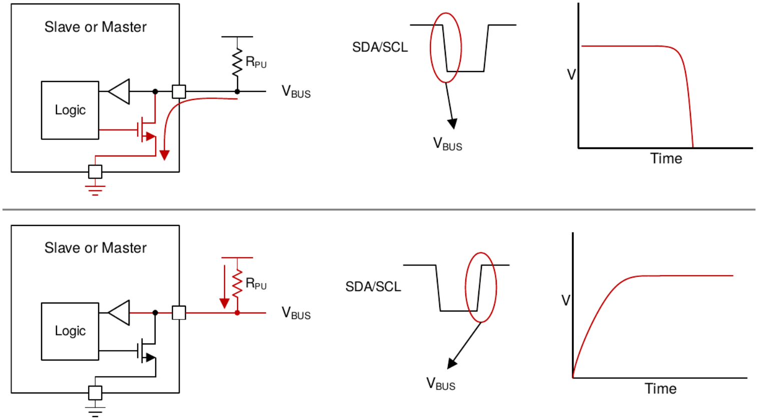


Figure 7: Open Drain Function in I2C Connection

I2C Packet

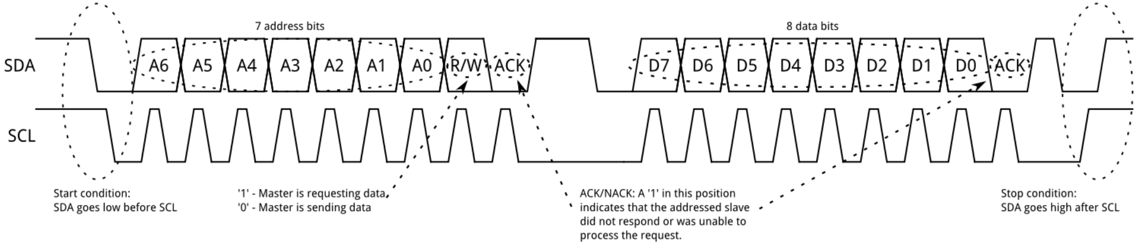


Figure 8: I2C Packet Format

I2C Packet: Write Register

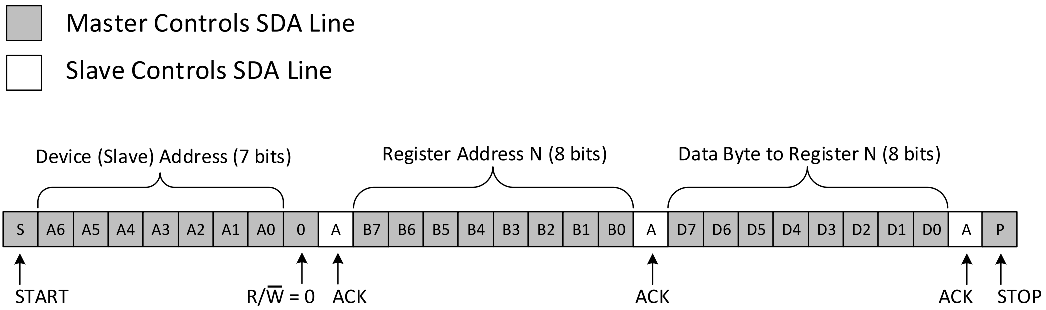


Figure 9: I2C Write to Slave Device's Register

I2C Packet: Read Register

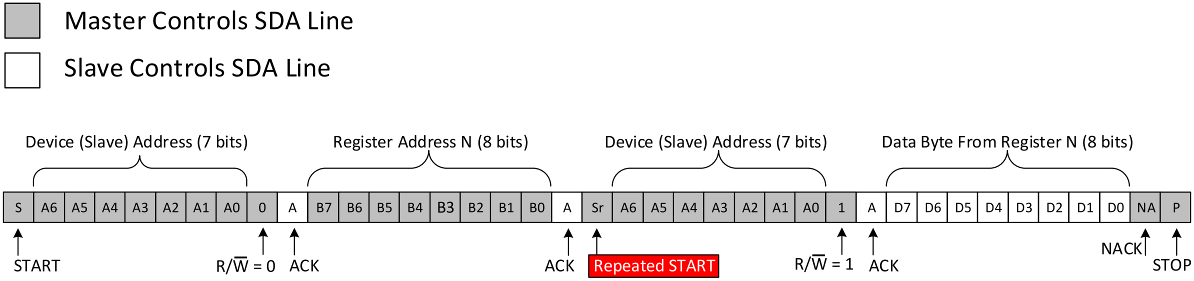
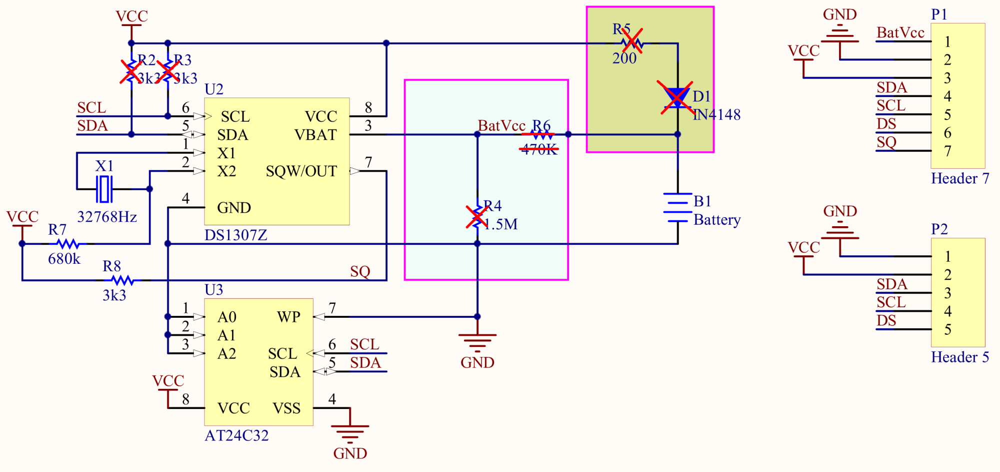


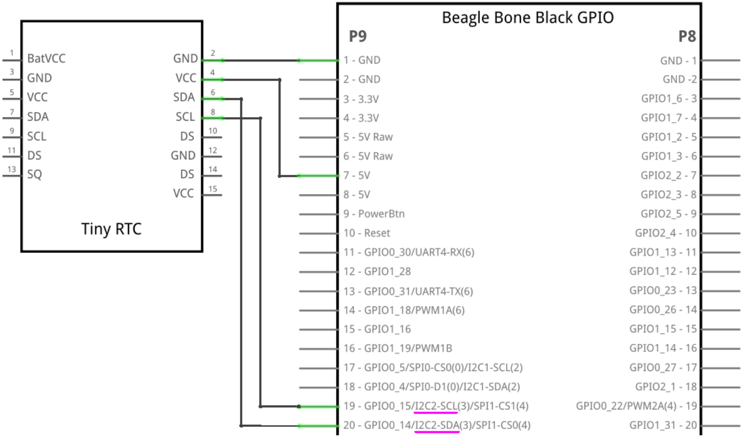
Figure 10: I2C Read from Slave Device's Register

RTC Module

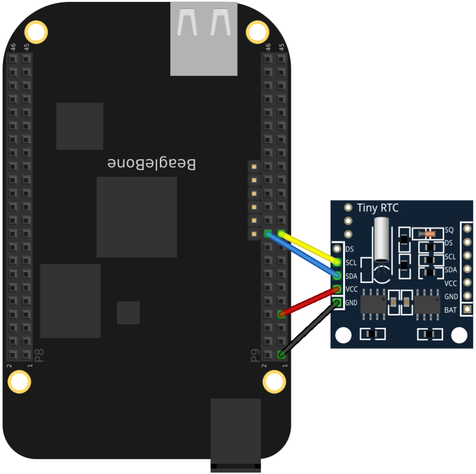
TinyRTC Module Schematic



TinyRTC Module Connection



TinyRTC Module Connection (cont'd)



RTC Read/Write

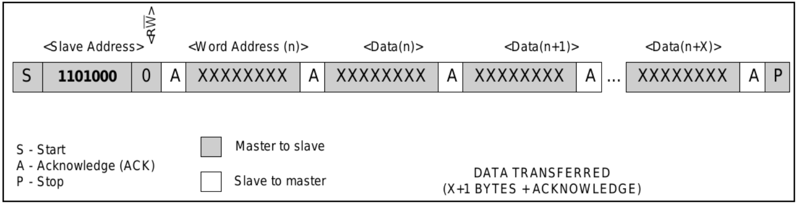


Figure 11: DS1307 Data Write

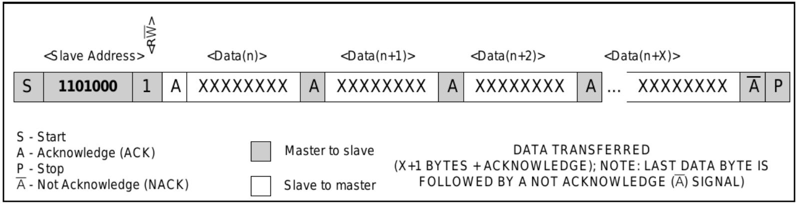


Figure 12: DS1307 Data Read

RTC Read From Register

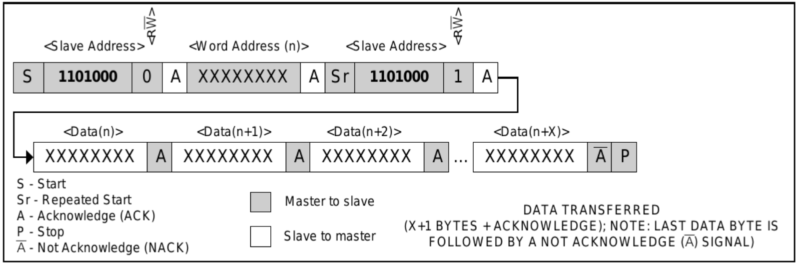


Figure 13: DS1307 Read Starting From Specified Register

- If the register pointer is not written to before the initiation of a read mode, the first address that is read is the last one stored in the register pointer
- The register pointer automatically increments after each byte are read

RTC Registers

ADDRESS	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	FUNCTION	RANGE
00h	CH	10 Seconds			Seconds				Seconds	00–59
01h	0	10 Minutes			Minutes				Minutes	00–59
02h	0	12	10 Hour	10 Hour	Hours				Hours	1–12 +AM/PM
		24	PM/ AM							00–23
03h	0	0	0	0	0	DAY			Day	01–07
04h	0	0	10 Date		Date				Date	01–31
05h	0	0	0	10 Month	Month				Month	01–12
06h	10 Year				Year				Year	00–99
07h	OUT	0	0	SQWE	0	0	RS1	RS0	Control	—
08h–3Fh									RAM 56 x 8	00h–FFh

0 = Always reads back as 0.

Figure 14: DS1307 I2C Registers

RTC Read From Register (cont'd)

From DS1307 datasheet:

- "The divider chain is reset whenever the seconds register is written. Write transfers occur on the I2C acknowledge from the DS1307. Once the divider chain is reset, to avoid rollover issues, the remaining time and date registers must be written within one second"
- ...So it's recommended to write all registers (time/date) in one transfer, starting from seconds register write

Kernel Driver

I2C IP-Core in SoC

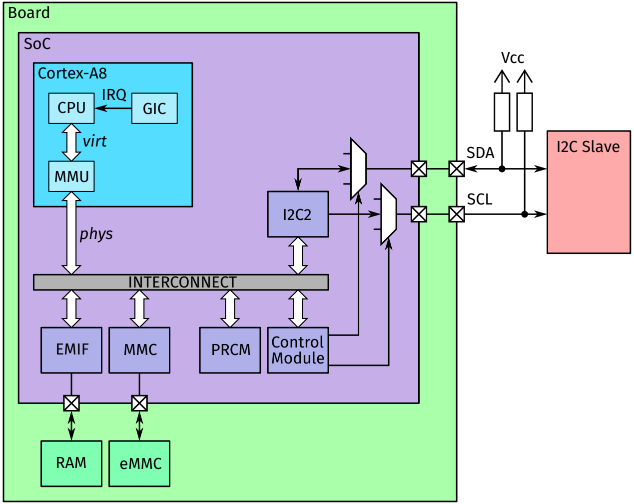
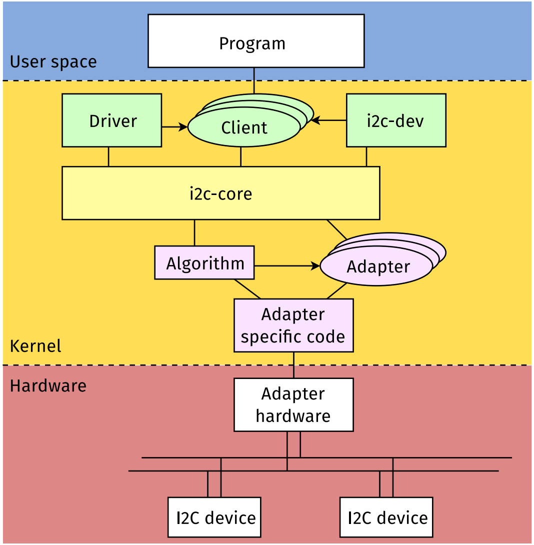


Figure 15: Integration of I2C Hardware Module in AM335x SoC

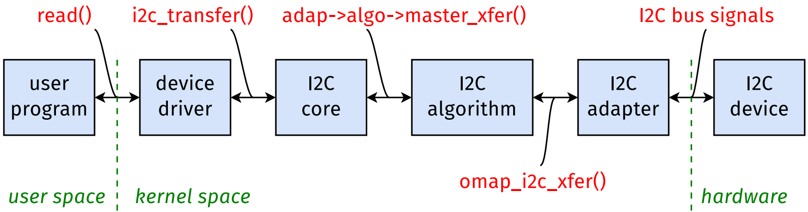
I2C Subsystem Driver Architecture

There are 3 driver layers:

- **Adapter**: I2C controller driver; usually incudes algorithm
- **i2c-core**: I2C/SMBus protocols implementation; device/driver match
- **Client**: Driver for particular device on I2C



I2C Call Chain



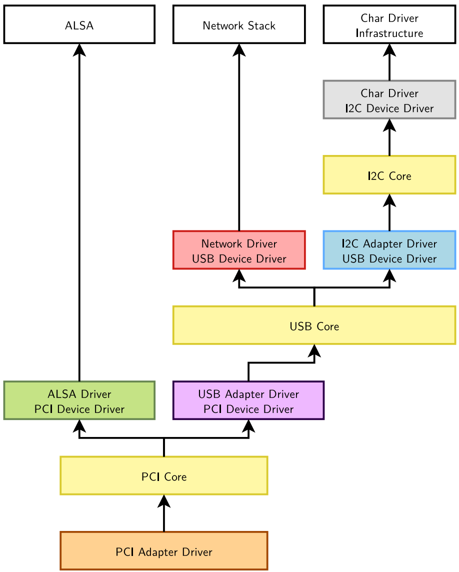
1. User space application reads from character device file
2. Device driver calls I2C API function (from I2C core)
3. I2C core calls transfer callback from registered algorithm
4. Callback leads to adapter driver transfer function
5. Adapter driver initiates physical transfer on I2C bus
6. Data is obtained from device and passed back to user

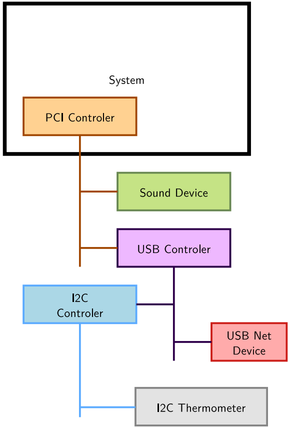
I2C Adapter Driver

- Makes it possible to write platform-independent drivers
- Driver: `drivers/i2c/busses/i2c-omap.c`
- Bindings: `Documentation/devicetree/bindings/i2c/i2c-omap.txt`
- Enabled in: `arch/arm/boot/dts/am33xx.dtsi`:

```
/ {
        ocp {
                i2c2: i2c@4819c000 {
                        compatible = "ti,omap4-i2c";
                        #address-cells = <1>;
                        #size-cells = <0>;
                        ti,hwmods = "i2c3";
                        reg = <0x4819c000 0x1000>;
                        interrupts = <30>;
                        status = "disabled";
                };
        };
};
```

Driver Model is Recursive (PC use-case)





Take Five

Kernel Frameworks

- Buses-centric API (for clients):
  - I2C: `linux/i2c.h`
  - SPI: `linux/spi/spi.h`
  - USB: `linux/usb.h`
- Device function centric frameworks:
  - IIO (sensors, ADC, DAC): `linux/iio/iio.h`
  - RTC: `linux/rtc.h`
  - input_dev: `linux/input.h`
- Helpers:
  - regmap: `linux/regmap.h`
  - MFD: `linux/mfd/core.h`

Plain I2C Communication

```c
int i2c_master_send(struct i2c_client *client, const char *buf, int count);
int i2c_master_recv(struct i2c_client *client, char *buf, int count);
```

- These routines read and write some bytes from/to a client
- The client contains the i2c address, so you do not have to include it
- The second parameter contains the bytes to read/write
- The third the number of bytes to read/write (must be less than the length of the buffer, also should be less than 64k since msg.len is `u16`)
- Returned is the actual number of bytes read/written

Plain I2C Communication (cont'd)

```c
int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msg, int num);
```

- This sends a series of messages
- Each message can be a read or write, and they can be mixed in any way
- The transactions are combined: no stop bit is sent between transaction
- The `i2c_msg` structure contains for each message:
    - the client address
    - the number of bytes of the message
    - and the message data itself

SMBus Communication

- SMBus = System Management Bus
- SMBus protocol is a subset from the I2C protocol
- Many devices use only the same subset
- If you write a driver for some I2C device, please try to use the SMBus commands if at all possible
- This makes it possible to use the device driver on both SMBus adapters and I2C adapters
- (the SMBus command set is automatically translated to I2C on I2C adapters, but plain I2C commands can not be handled at all on most pure SMBus adapters)

SMBus Communication (cont'd)

```
s32 i2c_smbus_read_byte(struct i2c_client *client);
s32 i2c_smbus_write_byte(struct i2c_client *client, u8 value);
s32 i2c_smbus_read_byte_data(struct i2c_client *client, u8 command);
s32 i2c_smbus_write_byte_data(struct i2c_client *client, u8 command, u8 value);
s32 i2c_smbus_read_word_data(struct i2c_client *client, u8 command);
s32 i2c_smbus_write_word_data(struct i2c_client *client, u8 command, u16 value);
s32 i2c_smbus_read_block_data(struct i2c_client *client, u8 command, u8 *values);
s32 i2c_smbus_write_block_data(struct i2c_client *client, u8 command, u8 length, const u8 *values);
s32 i2c_smbus_read_i2c_block_data(struct i2c_client *client, u8 command, u8 length, u8 *values);
s32 i2c_smbus_write_i2c_block_data(struct i2c_client *client, u8 command, u8 length,
                                   const u8 *values);
```

- All these transactions return a negative errno value on failure
- The 'write' transactions return 0 on success
- The 'read' transactions return the read value, except for block transactions, which return the number of values read
- The block buffers need not be longer than 32 bytes

regmap

- regmap = Register Map
- Register I/O for I2C and SPI
- We can use it instead of discussed I2C API
- Eliminates redundancy between drivers
- Can cache registers
- Can handle locking
- Can handle endianness conversion

regmap API

```c
#include <linux/regmap.h>

struct regmap_config {
        int reg_bits;               /* number of bits in register addr */
        int val_bits;               /* number of bits in register value */
        unsigned int max_register;  /* maximum valid register address */
        ...
};

struct regmap *devm_regmap_init_i2c(struct i2c_client *i2c,
                                    struct regmap_config *config);
int regmap_read(struct regmap *map, unsigned int reg, unsigned int *val);
int regmap_write(struct regmap *map, unsigned int reg, unsigned int val);
int regmap_bulk_read(struct regmap *map, unsigned int reg, void *val,
                     size_t val_count);
int regmap_bulk_write(struct regmap *map, unsigned int reg, const void *val,
int regmap_update_bits(struct regmap *map, unsigned int reg,
                       unsigned int mask, unsigned int val);
```

RTC in kernel

- RTC framework creates character device:
  `/dev/rtc0`, `/dev/rtc1`, etc;
  We can `read()` and `ioctl()` that file
- RTC framework also creates sysfs nodes in:
  `/sys/class/rtc/rtcX/`
  We can use `wakealarm` node to set alarm
- There are existing user-space tools in Linux:
  - `hwclock`
  - `rtcwake`
  - `date`
- Our device doesn't have alarm interrupt

RTC API

```c
#include <linux/rtc.h>

struct rtc_class_ops {
        int (*read_time)(struct device *, struct rtc_time *);
        int (*set_time)(struct device *, struct rtc_time *);
        int (*read_alarm)(struct device *, struct rtc_wkalrm *);
        int (*set_alarm)(struct device *, struct rtc_wkalrm *);
        int (*alarm_irq_enable)(struct device *, unsigned int enabled);
        ...
};

struct rtc_device *devm_rtc_allocate_device(struct device *dev);

rtc->uie_unsupported = 1; /* no IRQ line = no Update Interrupt Enable */
rtc->ops = (struct rtc_class_ops)...;

int rtc_register_device(struct rtc_device *rtc);
```

RTC API (cont'd)

```
/* If device has alarm interrupt line (DS1307 doesn't) */
void rtc_update_irq(struct rtc_device *rtc, unsigned long num,
                    unsigned long events);

Helper functions:
int rtc_year_days(unsigned int day, unsigned int month, unsigned int year);
int rtc_valid_tm(struct rtc_time *tm);
time64_t rtc_tm_to_time64(struct rtc_time *tm);
...
```

There is also **nvmem** API, but we won't cover it here.

BCD Format

- BCD = binary-coded decimal
- Each of the two nibbles of each byte represent a decimal digit

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0	0	1	0	0	0	1	1

$$\underbrace{0\ 0\ 1\ 0}_{2}\ \underbrace{0\ 0\ 1\ 1}_{3}$$

$$\boxed{2*10 + 3 = 23}$$

BCD helper functions:

```c
#include <linux/bcd.h>

unsigned bcd2bin(unsigned char val); /* decode BCD value to dec value */
unsigned char bin2bcd(unsigned val); /* encode dec value to BCD value */
```

Bit-banged I2C

“Bit-banging” is GPIO emulation of some protocol.

- Can be useful on cheap boards without I2C controllers
- …or if there is no unused I2C addresses left on bus
- There is already implemented bit-banged I2C driver:
  `Documentation/devicetree/bindings/i2c/i2c-gpio.txt`
- There are more ready to use GPIO drivers; for details see:
  `Documentation/driver-api/gpio/drivers-on-gpio.rst`

Linux Turns 27!

- A lot of frameworks already exist
- Check out **`Documentation/`**
- ~~Doxygen~~ kernel-doc comments
- Linux code base is a good source of examples
- `git grep` is useful to find references:

```
$ git grep -l --all-match -e devm_regmap \
            -e devm_rtc -- drivers/
$ git grep -n -e _i2c_init --and \( -e module \
            -e initcall \) -- '*.[ch]'
```

RTC Driver Implementation

Device Tree Definition

Listing 1: Device Tree definition for our driver

```
&i2c2 {
        ds1307x: ds1307x@68 {
                compatible = "dallas,ds1307x";
                reg = <0x68>;
        };
};
```

Note that:

- `i2c2_pins` are already muxed (`PIN_INPUT_PULLUP | MUX_MODE3`)
- `i2c2.status` is `okay`
- Our driver will be bound on `insmod`

I2C Driver: Init

Listing 2: Initialization of I2C driver

```c
static struct i2c_driver ds1307x_driver = {
	.driver = {
		.name		= "ds1307x",
		.of_match_table = ds1307x_of_match,
	},
	.probe		= ds1307x_probe,
	.remove		= ds1307x_remove,
	.id_table	= ds1307x_id,
};

module_i2c_driver(ds1307x_driver);
```

Listing 3: Device Tables used in our driver

```
static const struct i2c_device_id ds1307x_id[] = {
        { "ds1307x" },
        { },
};
MODULE_DEVICE_TABLE(i2c, ds1307x_id);

static const struct of_device_id ds1307x_of_match[] = {
        { .compatible = "dallas,ds1307x" }, /* ds1307 already exists */
        { }
};
MODULE_DEVICE_TABLE(of, ds1307x_of_match);
```

I2C Driver: Probe and remove

Listing 4: Probe and remove functions in our driver

```c
#include <linux/module.h>
#include <linux/i2c.h>

static int ds1307x_probe(struct i2c_client *client,
                         const struct i2c_device_id *id)
{
        /* TODO */
        return 0;
}

static int ds1307x_remove(struct i2c_client *client)
{
        /* TODO */
        return 0;
}
```

Listing 5: Example of raw I2C API usage

```
	int ret;
	u8 reg = 0x01;	/* I2C register */
	u8 buf;		/* where to read */
	u8 len = 1;	/* bytes to read */
	struct i2c_msg msg[2] = {
		{
			.addr = client->addr,
			.len = 1,
			.buf = &reg,
		},
		{
			.addr = client->addr,
			.flags = I2C_M_RD,	/* read */
			.len = len,
			.buf = &buf,
		}
	};

	ret = i2c_transfer(client->adapter, msg, 2);
	if (ret < 0)
		return ret;

	pr_info("### read data = %#x\n", buf);
```

SMBus API Example

Listing 6: Example of SMBus API usage

```
        s32 data;

        data = i2c_smbus_read_byte_data(client, 0x01); /* minutes */
        pr_info("### read data = %#x\n", data);
```

Address Conflicts

- Be aware of possible I2C address conflicts on the same bus!
- On `insmod` you can see something like this:

Listing 7: Example of conflict on bus

```
i2c i2c-2: Failed to register i2c client ds1307 at 0x68 (-16)
i2c i2c-2: of_i2c: Failure registering /ocp/i2c@4819c000/ds1307@68
i2c i2c-2: Failed to create I2C device for /ocp/i2c@4819c000/ds1307@68
```

Where 16 is `EBUSY`.

Listing 8: No conflict

```
rtc-ds1307 2-0068: SET TIME!
rtc-ds1307 2-0068: registered as rtc1
```

Let's check if our device is visible on bus:

```
# i2cdetect -r 2 -y

     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: 50 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- 68 -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

After `insmod` we'll see `UU` instead `68` (means it's used).
We can also use `i2cdump`, `i2cget` and `i2cset` tools:

```
# i2cget -y 2 0x68 0x01

0x49
```

Userspace Tools (cont'd)

Once driver is implemented, we can use it for keeping the system time:

```
Set system time:
# date -s "2018-08-28 11:30:00"

Write system time to our RTC:
# hwclock -w -f /dev/rtc1

Read time from our RTC:
# hwclock -r -f /dev/rtc1

Set system time from our RTC:
# hwclock -s -f /dev/rtc1

Show system time:
# date
```

Assignments

Assignment 1 (basic)

- Connect RTC module to your BBB
- Check with `i2cdetect` tool it's visible on bus
- Write Device Tree definition for your driver (use `"ds1307x"` as compatible string)
- Write I2C driver boilerplate; make sure that `probe()` is called
- Try to read from some I2C register; make sure it's the same value as `i2cget` tool reports

Assignment 2 (advanced)

- Implement reading/writing registers via regmap
- Implement RTC using RTC kernel API
- Set your RTC using `hwclock` tool; make sure it works
- Test if it’s consistent between power-off / power-on
- **Hint**: If in troubles, use existing drivers as examples/templates

References

John Madieu.
Linux Device Drivers Development.

P. Raghavan, Amol Lad, Sriram Neelakandan.
Embedded Linux System Design and Development.

`Documentation/i2c/writing-clients`

Thank you!