SusaNET

Kevin Sangeelee (kevin :at: susa.net)

Raspberry Pi PCF8563 Real Time Clock (RTC)

June 15, 2012 Technical Stuff Electronics, hardware, raspberry pi, raspi

Having recently received my Raspberry Pi, one of the first things I wanted to do was hook up a real-time clock chip I had lying around (a NXP PCF8563) and learn how to drive I2C from the BCM2835 hardware registers. Turns out it's quite easy to do, and I think makes a useful project to learn with.

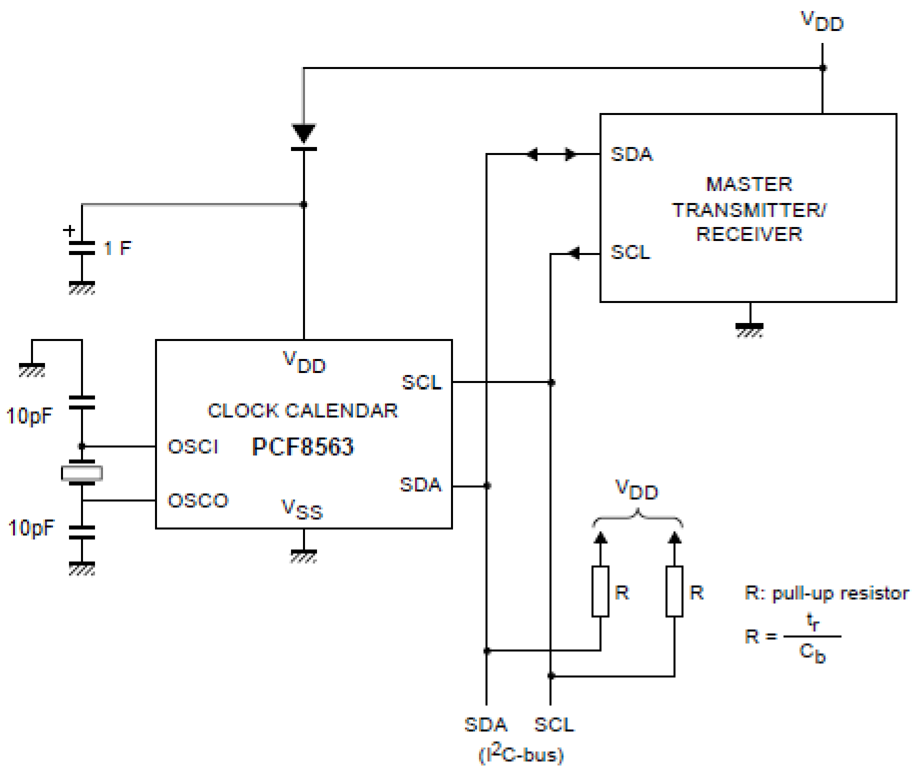
So, here are some notes I made getting it to work, initially with Chris Boot's forked kernel that incorporates some I2C handling code created by Frank Buss into the kernel's I2C bus driver framework.

After getting it to work with the kernel drivers, I created some C code to drive the RTC chip directly using the BCM2835 I2C registers, using mmap() to expose Peripheral IO space in the user's (virtual) memory map, the technique I learned from Gert's Gertboard demo software, though my code's simpler (hopefully without limiting functionality!).

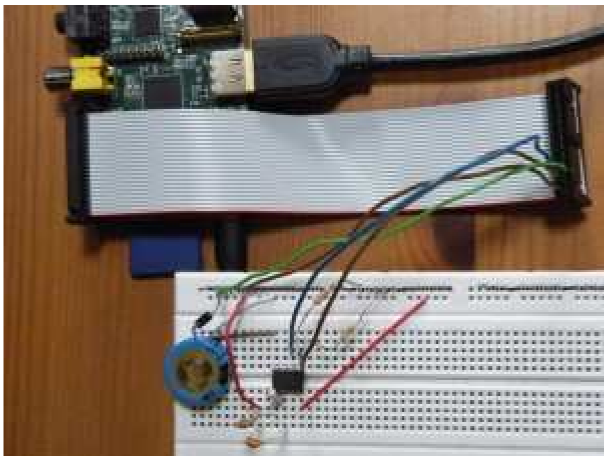
Note: Revision 2 boards require the code to access BSC1 (I2C1) rather than BSC0 (I2C0), so changes to the peripheral base address may be required, or in the case if the Linux I2C driver, a reference to i2c-1 rather than i2c-0. It should be simple enough, but I don't want to write about things I haven't done or tested, so a bit of extra work by the reader may be required.

The Circuit

The PCF8563 datasheet's reference schematic is all I used for the Raspi's clock, though for the crystal I ditched the the trimmer capacitor from the datasheet's schematic and used two 10pF caps as per datasheet – good enough for the stability that I'm likely to need.



This translates to the following circuit on a breadboard. The large double-layer capacitor can be found easily at Maplin or Farnell, a 1F 5.5V, and while physically quite a bit larger than a CR2032 (or similar) battery, it should give months of power between charges. An electrolytic capacitor would also work for testing, but use a 150-300 ohm resistor in series to limit the current draw when charging, or your Raspi will likely reboot (since we're only allowed to draw 50mA from the 3V3 pin).



The four wires hooked up to the Raspberry Pi's expansion header are simply 3V3, GND, I2C0_SCL and I2C0_SDA (pins P1-01, P1-06, P1-05 and P1-03 respectively based on the Wiki reference).

Take care when wiring up the expansion header, since it's easy to make mistakes when translating pins from diagram to board. It's always worth at least checking voltage as reference points (e.g. measure +3.3V across what you think are pins 3V3 and GND) to make sure you're not top to bottom on your translation, for example, before plugging up.

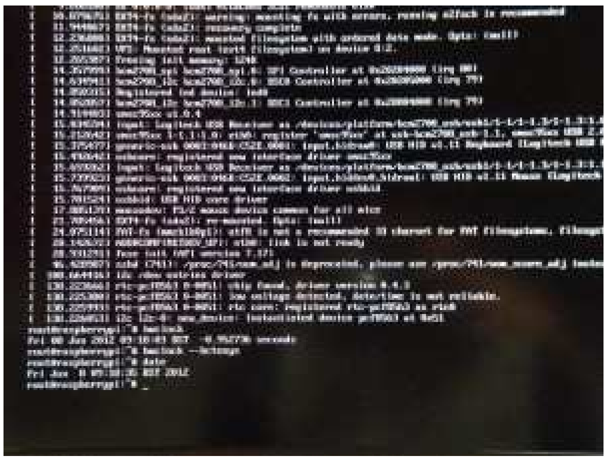
The circuit is so simple that it could just be wired together onto a 26-pin header connector, given a little care and forethought, and perhaps a bit of epoxy putty or similar to fix it all together.

The Kernel Driver

Next I had to think about software – it turned out that between Chris Boot and Frank Buss, there was already a kernel with support for the Raspi's I2C bus in a way that the existing pcf8563 driver can use.

```
root@raspberrypi:~# modprobe i2c-dev
root@raspberrypi:~# modprobe rtc-pcf8563
root@raspberrypi:~# echo pcf8563 0x51 > /sys/class/i2c-adapter/i2c-0
/new_device
```

When running Chris's kernel, the above commands are all that are required to get the PCF8563 working with the hwclock command. The value 0×51 above is the hardwired I2C device address of the RTC chip.



```
[illegible]
```

And that's it! Chris's 3.2.18 kernel was screwing up my SD card access periodically (when it boots, it runs more or less flawlessly, but it often fails to boot) and ended up corrupting an otherwise good SD card image. I think this issue has been fixed since his 3.2.19 r3 build, but I haven't tested it.

The C Code Alternative

My original intention had been to learn how to drive the Raspberry Pi's IO Peripheral devices at a register level, so having seen that the circuit was wired and working using the kernel drivers, I went back to the Foundation's official kernel, which has no inbuilt I2C support, to find out how to drive the chip in user-space.

After a quick attempt to access the IO peripheral space directly using pointers (ok, daft to even try, but you never know…), I had a look at the source code of Gert's demo, which accesses GPIO among others, to see how it can be done.

Essentially, he accesses /dev/mem, which is a virtual file that makes available the physical address space (ARM Physical Addresses in the BCM2835 datasheet), via a chunk of user-space memory that's judiciously allocated using malloc(), and that's mapped to /dev/mem using the mmap().

I simplified this by allowing the kernel supply me with the virtual address of the physical address space, on the basis that it will automatically allocate on a page boundary. All the BCM2835 peripherals I've looked at in the datasheet are based at a 4K boundary.

If you're used to driving I2C from a microcontroller, either bit-banging (driving the SCL and SDA pins directly with software) or with an MCU peripheral (through hardware registers), then you may find that the BCM2835 hardware does a little more for you than you're used to. In particular, you use hardware registers to define the I2C device-address and the number of data-bytes (e.g. BSC0_A & BSC0_DLEN), and the BCM2835 handles the writing of the address, the bytes, and the corresponding acknowledge bits for you automatically.

It also provides a 16 byte FIFO buffer. You fill the buffer and it writes what you put into it. If you're reading from a device, it buffers it in the FIFO for you to read. There's very little low level stuff to do other than check if the buffer's full/empty, and read/write the FIFO accordingly. In the example below, I don't even need to monitor the buffer's status, since no transaction exceeds the FIFOs 16 bytes.

Below is the C source of my code used to drive the RTC directly from user-space using the BMC peripheral registers. It allows reading and displaying of the RTC chip time, setting the RTC from the system clock, and setting the system clock from the RTC. All times are handled as UTC. The code also configures GPIO pins 0 and 1 as ALT0 (SDA and SCL for BMC0), since they're not routed to the I2C peripheral by default.

Everything is written for clarity rather than elegance, and it's all in a single file for the same reason (it's not a big program).

```
/*
  * pcf8563_i2c_rtc.c - example of accessing a PCF8563 via the
BSC0 (I2C) peripheral on a BCM2835 (Raspberry Pi)
  *
  * Copyright 2012 Kevin Sangeelee.
  * Released as GPLv2, see <http://www.gnu.org/licenses/>
  *
  * This is intended as an example of using Raspberry Pi hardware
registers to drive an RTC chip. Use at your own risk or
  * not at all. As far as possible, I've omitted anything that
doesn't relate to the RTC or the Raspi registers. There are more
  * conventional ways of doing this using kernel drivers, though
they're harder to follow what's happening in hardware.
  */
#include <stdio.h>
#include <time.h>
#include <fcntl.h>
#include <sys/mman.h>

#define IOBASE   0x20000000

#define BSC0_BASE (IOBASE + 0x205000)
#define GPIO_BASE (IOBASE + 0x200000)
```

```
#define BSC0_C        *(bsc0.addr + 0x00)
#define BSC0_S        *(bsc0.addr + 0x01)
#define BSC0_DLEN    *(bsc0.addr + 0x02)
#define BSC0_A        *(bsc0.addr + 0x03)
#define BSC0_FIFO    *(bsc0.addr + 0x04)

#define BSC_C_I2CEN    (1 << 15)
#define BSC_C_INTR    (1 << 10)
#define BSC_C_INTT    (1 << 9)
#define BSC_C_INTD    (1 << 8)
#define BSC_C_ST    (1 << 7)
#define BSC_C_CLEAR    (1 << 4)
#define BSC_C_READ    1

#define START_READ    BSC_C_I2CEN|BSC_C_ST|BSC_C_CLEAR|BSC_C_READ
#define START_WRITE    BSC_C_I2CEN|BSC_C_ST

#define BSC_S_CLKT    (1 << 9)
#define BSC_S_ERR    (1 << 8)
#define BSC_S_RXF    (1 << 7)
#define BSC_S_TXE    (1 << 6)
#define BSC_S_RXD    (1 << 5)
#define BSC_S_TXD    (1 << 4)
#define BSC_S_RXR    (1 << 3)
#define BSC_S_TXW    (1 << 2)
#define BSC_S_DONE    (1 << 1)
#define BSC_S_TA    1

#define CLEAR_STATUS    BSC_S_CLKT|BSC_S_ERR|BSC_S_DONE

#define PAGESIZE 4096
#define BLOCK_SIZE 4096

struct bcm2835_peripheral {
    unsigned long addr_p;
    int mem_fd;
    void *map;
    volatile unsigned int *addr;
};

struct bcm2835_peripheral gpio = {GPIO_BASE};
struct bcm2835_peripheral bsc0 = {BSC0_BASE};

// Some forward declarations...
void dump_bsc_status();
int map_peripheral(struct bcm2835_peripheral *p);
void unmap_peripheral(struct bcm2835_peripheral *p);
```

```
int systohc = 0;
int hctosys = 0;
struct tm t;
time_t now;

// BCD helper functions only apply to BCD 0-99 (one byte) values
unsigned int bcdtod(unsigned int bcd) {
    return ((bcd & 0xf0) >> 4) * 10 + (bcd & 0x0f);
}
unsigned int dtobcd(unsigned int d) {
    return ((d / 10) << 4) + (d % 10);
}

// Function to wait for the I2C transaction to complete
void wait_i2c_done() {
        //Wait till done, let's use a timeout just in case
        int timeout = 50;
        while((!((BSC0_S) & BSC_S_DONE)) && --timeout) {
            usleep(1000);
        }
        if(timeout == 0)
            printf("wait_i2c_done() timeout. Something went
wrong.\n");
}

////////////////
//  main()
////////////////
int main(int argc, char *argv[]) {

    if(argc == 2) {
        if(!strcmp(argv[1], "-w"))
            systohc = 1;
        if(!strcmp(argv[1], "-s"))
            hctosys = 1;
    }

    if(map_peripheral(&gpio) == -1) {
        printf("Failed to map the physical GPIO registers into the
virtual memory space.\n");
        return -1;
    }
    if(map_peripheral(&bsc0) == -1) {
        printf("Failed to map the physical BSC0 (I2C) registers
into the virtual memory space.\n");
        return -1;
    }

    /* BSC0 is on GPIO 0 & 1 */
    *gpio.addr &= ~0x3f; // Mask out bits 0-5 of FSEL0 (i.e. force
to zero)
    *gpio.addr |= 0x24;  // Set bits 0-5 of FSEL0 to binary
'100100'

    // I2C Device Address 0x51 (hardwired into the RTC chip)
    BSC0_A = 0x51;

    if(systohc) {

        printf("Setting RTC from system clock\n");

        now = time(NULL);
        gmtime_r(&now, &t);    // explode time_t (now) into an
struct tm

        //////
        // Write Operation to set the time (writing 15 of the 16
RTC registers to
        // also reset status, alarm, and timer settings).
        //////
        BSC0_DLEN = 16;
        BSC0_FIFO = 0;        // Addr 0
        BSC0_FIFO = 0;        // control1
        BSC0_FIFO = 0;        // control2
        BSC0_FIFO = dtobcd(t.tm_sec);        // seconds
        BSC0_FIFO = dtobcd(t.tm_min);    // mins
        BSC0_FIFO = dtobcd(t.tm_hour);    // hours
        BSC0_FIFO = dtobcd(t.tm_mday);    // days
        BSC0_FIFO = dtobcd(t.tm_wday);    // weekdays (sun 0)
        BSC0_FIFO = dtobcd(t.tm_mon + 1);    // months 0-11 -->
1-12
        BSC0_FIFO = dtobcd(t.tm_year - 100);    // years
        BSC0_FIFO = 0x0;    // alarm min
        BSC0_FIFO = 0x0;    // alarm hour
        BSC0_FIFO = 0x0;    // alarm day
        BSC0_FIFO = 0x0;    // alarm weekday
        BSC0_FIFO = 0x0;    // CLKOUT control
        BSC0_FIFO = 0x0;    // timer control

        BSC0_S = CLEAR_STATUS; // Reset status bits (see #define)
        BSC0_C = START_WRITE;    // Start Write (see #define)

        wait_i2c_done();
    }

    //////
    // Write operation to restart the PCF8563 register at index 2
('secs' field)
    //////
    BSC0_DLEN = 1;    // one byte
    BSC0_FIFO = 2;    // value 2
    BSC0_S = CLEAR_STATUS; // Reset status bits (see #define)
    BSC0_C = START_WRITE;    // Start Write (see #define)

    wait_i2c_done();

    //////
    // Start Read of RTC chip's time
    //////
    BSC0_DLEN = 7;
    BSC0_S = CLEAR_STATUS; // Reset status bits (see #define)
    BSC0_C = START_READ;    // Start Read after clearing FIFO (see
#define)

    wait_i2c_done();

    // Store the values read in the tm structure, after masking
unimplemented bits.
    t.tm_sec = bcdtod(BSC0_FIFO & 0x7f);
    t.tm_min = bcdtod(BSC0_FIFO & 0x7f);
    t.tm_hour = bcdtod(BSC0_FIFO & 0x3f);
    t.tm_mday = bcdtod(BSC0_FIFO & 0x3f);
    t.tm_wday = bcdtod(BSC0_FIFO & 0x07);
    t.tm_mon = bcdtod(BSC0_FIFO & 0x1f) - 1; // 1-12 --> 0-11
    t.tm_year = bcdtod(BSC0_FIFO) + 100;

    printf("%02d:%02d:%02d %02d/%02d/%02d (UTC on PCF8563)\n",
        t.tm_hour,t.tm_min,t.tm_sec,
        t.tm_mday,t.tm_mon + 1,t.tm_year - 100);

    if(hctosys) {
        printf("Setting system clock from RTC\n");
        now = timegm(&t);
        stime(&now);
    }

    unmap_peripheral(&gpio);
    unmap_peripheral(&bsc0);

    // Done!
```

```
}

// Exposes the physical address defined in the passed structure
using mmap on /dev/mem
int map_peripheral(struct bcm2835_peripheral *p)
{
   // Open /dev/mem
   if ((p->mem_fd = open("/dev/mem", O_RDWR|O_SYNC) ) < 0) {
      printf("Failed to open /dev/mem, try checking
permissions.\n");
      return -1;
   }

   p->map = mmap(
      NULL,
      BLOCK_SIZE,
      PROT_READ|PROT_WRITE,
      MAP_SHARED,
      p->mem_fd,  // File descriptor to physical memory virtual
file '/dev/mem'
      p->addr_p      // Address in physical map that we want this
memory block to expose
   );

   if (p->map == MAP_FAILED) {
        perror("mmap");
        return -1;
   }

   p->addr = (volatile unsigned int *)p->map;

   return 0;
}

void unmap_peripheral(struct bcm2835_peripheral *p) {

    munmap(p->map, BLOCK_SIZE);
    close(p->mem_fd);
}

void dump_bsc_status() {

    unsigned int s = BSC0_S;

    printf("BSC0_S: ERR=%d  RXF=%d  TXE=%d  RXD=%d  TXD=%d
RXR=%d  TXW=%d  DONE=%d  TA=%d\n",
        (s & BSC_S_ERR) != 0,
        (s & BSC_S_RXF) != 0,
        (s & BSC_S_TXE) != 0,
        (s & BSC_S_RXD) != 0,
        (s & BSC_S_TXD) != 0,
        (s & BSC_S_RXR) != 0,
        (s & BSC_S_TXW) != 0,
        (s & BSC_S_DONE) != 0,
        (s & BSC_S_TA) != 0 );
}
```

The code is all in one file for clarity, and can be compiled and run with: -

```
root@pi:~# gcc -o pcf8563_i2c_rtc pcf8563_i2c_rtc.c
root@pi:~# ./pcf8563_i2c_rtc
19:12:32 14/05/12 (UTC on PCF8563)
```

Parameters '-w' and '-s' are the same as those in hwclock – e.g. write hardware clock from system time, and set system time from hardware clock.

References

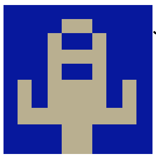
- PCF8563 Datasheet – RTC chip's datasheet.
- BCM2835 Datasheet – abbreviated datasheet for Broadcom SoC.
- Chris Boot's forked kernel – Debian image for the updated kernel.
- mmap() – Linux mmap function man page.
- Using the I2C Bus – a good overview of I2C wire communications.
- Olimex MOD-RTC – a more complete PCF8563 circuit with schematics.
- i2c-bcm2708 diffs – changes relating to I2C support in Chris Boot's kernel.

18 thoughts on "Raspberry Pi PCF8563 Real Time Clock (RTC)"

June 26, 2012 at 1:13 pm

Alexs

Fantastic
Thanks for contributing!!
Would it be possible to add gpio access routines and put the code in a shared library?
Unfortunately my understanding of C is very limited. I am working on a class for High School students and want to use free pascal / Lazarus to do some Hardware experiments and general programming.
Thanks!



★Kevin Sangeelee

June 26, 2012 at 3:27 pm

I'm about to write up the results of driving a relay from the GPIO peripheral, so there will be more C code available shortly (specifically for GPIO), however the code I've written is intended to show what's happening at a hardware (register) level, so wrapping it up in a library would maybe defeat that purpose. You might consider using the Shell Script example in http://elinux.org/RPi_Low-level_peripherals. This technique could be used to create a library in Pascal using nothing but file I/O.

Pingback: Raspberry Pi Relay using GPIO | SusaNET

Reiner Geiger

June 28, 2012 at 11:38 am

Using your BLOG as a start, I found everything working well with several RTC chips!

I will use the OLIMEX MOD-RTC (pcf8563 + 3V Lithium Battery) because it's easy to get in Germany and also has an affordable price (€5.95+Sipping).

All you have to do for a RTC backed Systime, is to add the following script files as root:

/etc/init.d/i2c-hwclock AND /sbin/i2c-hwclock
(WARNING !!!update-rc.d actions necessary!!!)

in directory /etc/init.d/

```
#!/bin/sh

### BEGIN INIT INFO
# Provides: i2c-hwclock
# Required-Start:
# Required-Stop: umountroot
# Should-Stop:
# X-Start-Before: checkroot
# Default-Start: S
# Default-Stop: 0 1 6
# Short-Description: Restore / save the current clock
# Description:
### END INIT INFO

set -e
case "${1:-}" in
stop|reload|restart|force-reload)
echo "Stopping i2c hwclock: saving system time."
i2c-hwclock save ;;

start)
echo "Starting i2c hwclock: loading system time."
i2c-hwclock load ;;

*)
echo "Usage: ${0:-}
{start|stop|status|restart|reload|force-reload}" >&2
exit 1
;;
esac
```

in directory /sbin

```
#!/bin/sh
#
# Trivial script to load/save current contents of the
kernel clock
# from an i2c RTC.
#
# This Version is made for Raspberry PI fake-hwclock
replacement
#
# Tested on: Debian Wheezy with kernel 3.2.21-
rp1+_5_armel.deb (2012-06-23)
# This kernel includes support for i2c and spi!
# --> http://www.bootc.net
#
# Using NTP is still recommended on these machines to get
to real time sync
# once more of the system is up and running.
#
# Copyright 2012 Reiner Geiger
```

```
#
# License: GPLv2, see COPYING

set -e

FILE=/dev/rtc0
COMMAND=$1

case $COMMAND in
save)
if [ -e $FILE ] ; then
hwclock -w
echo "Current system time: $(date -u '+%Y-%m-%d %H:%M')
written to RTC"
else
echo "No RTC device $FILE found"
fi
;;
load)
if [ -e $FILE ] ; then
hwclock -s
echo "Current system time: $(date -u '+%Y-%m-%d %H:%M')
written to RTC"
else
modprobe i2c-dev
modprobe rtc_pcf8563
command -- echo pcf8563 0x51 > /sys/class/i2c-adapter
/i2c-0/new_device
if [ -e $FILE ] ; then
hwclock -s
echo "Current system time: $(date -u '+%Y-%m-%d %H:%M')
written to RTC"
else
echo "No RTC device $FILE found"
fi
fi
;;
*)
echo $0: Unknown command $COMMAND
exit 1
;;
esac
```

Do not forget use to make the files owner executable:

Use

update-rc.d fake-hwclock remove
update-rc.d i2c-hwclock defaults
to
activate the new init.d configuration.

Use at your own risk

June 28, 2012 at 7:30 pm

Nicely done, with proper dependency based init scripts. Thanks for sharing your work.

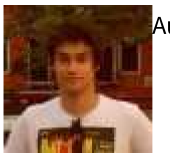
★ Kevin Sangeelee

July 6, 2012 at 11:36 pm

struct tm {} uses 0-11 for months and the RTC uses 1-12, and I failed to notice when writing the code – fixed now.

★ Kevin Sangeelee

Pingback: Rasperry Pi - The 25$ Computer - Seite 5



August 29, 2012 at 3:37 pm

Thanks for this! I had an old PCF8563 lying around and plugged it. After copy pasting & compiling the code I had it running in under 20 seconds. Very happy to have a working example of i2c with the pi. Thanks!

Pieter-Jan

August 29, 2012 at 4:44 pm

Good to hear – the I2C bus is wonderfully simple.

★ Kevin Sangeelee

October 1, 2012 at 12:27 am

anon

With this setup, is the RTC IC powered by the battery/capacitor the whole time, or just when the Raspberry Pi is off? It seems sensible to power the RTC chip from the GPIO header most of the time and only use the battery as backup; some rtc ics (eg. the DS1307) appear to have built in support for this mode of operation, but AFAICT the PCF8563 doesn't. Is this correct?

October 1, 2012 at 12:30 pm

★ Kevin Sangeelee

The capacitor is always charging when power is on, which what we want, and the diode in series with 3V3 prevents it from discharging to the Raspberry Pi when power is off. For a battery, you'd need to have two diodes, one in series with 3V3 (3V3 -> diode -> Vin) and one in series with the battery (+ve -> diode -> Vin).

Modelling this with 3V3 and a 3V cell, the cell only drains around 1uA when the chip is active and drawing, say, 200uA from 3V3. When the chip is in low-power mode, the drain on the cell will be as per the datasheet (e.g. 500nA).

I don't know anything about the DS1307, but I've not seen anything in the PCF8563 data-sheet to suggest there's an on-chip way to isolate two independent supplies.

October 18, 2012 at 4:53 pm

guzik

Why I must do this:
echo pcf8563 0×51 > /sys/class/i2c-adapter/i2c-0/new_device
after each restart? Is it possible to do this automatically? But better than in startup script.

October 18, 2012 at 5:16 pm

★ Kevin Sangeelee

I can't answer either question with any authority. I think that this can be done automatically in the udev/modprobe configuration, if you really don't want to use rc.local or a modified startup script. I guess the reason it needs to be done at all is so that the RTC driver (rtc-pcf8563.ko) knows what bus the chip is connected to. Perhaps someone else could correct me or elaborate.

Pingback: Moje Raspberry Pi już wie która godzina « guzik

October 22, 2012 at 4:19 am

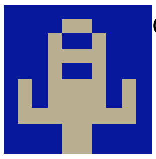
Dalmir Silva

Hello friend,
First of all I want to say: Great post!

This post gave to me the clarifications that I need to write a simple Wire library, with the same methods of the Arduino Wire library.

I made available in my github: https://github.com/dalmirdasilva/Wire

So, thank you!


October 22, 2012 at 1:12 pm

★ Kevin Sangeelee

Thanks for sharing this. Note that the V2 boards have changed pins 3 & 5 from BSC0 to BSC1, so the peripheral address will be different for these and future boards. I'm working on a PIC based RTC that will emulate a PCF8563 for compatibility, part of the code changes will be to incorporate support for V2 boards (when I get one), so I'll try to copy you in on updates.

Please don't forget GPL headers in all source files (though I often forget myself!).

October 22, 2012 at 6:05 pm

Dalmir Silva

Cool,

Yes, I always forget the GPL header!

I need to make the library more robust implementing buffer checking and board version compatibility checks.

Thank you in advance.

BTW: What PIC uC are you using? I have some projects using PIC18f4550 to drive GLCD and another things, but I don't like the SDCC compiler.

October 22, 2012 at 7:20 pm

★Kevin Sangeelee

I've only just discovered SDCC, but haven't actually tried it yet – I'd like a system that enables me to compile code on an RPi and then to program the PIC in-situ.

Up until now, I've mostly used HiTech C – the free version is good enough and, as long as you don't make too much use of the built in library functions (printf and scanf in particular), then the code is quite compact. I haven't warmed to the Eclipse based IDEs for PIC stuff (though I use Eclipse almost exclusively for Java), so I'm still on MPLAB V8.x.