Interfacing I2C Devices To Your System

Inter-Integrated Circuit (I²C or more often written as I2C) is a multimaster serial single-ended computer bus invented by the Philips semiconductor division (see the wikipedia article for more information on I2C) and commonly used in many modern electronic devices including PC.

I²C uses only two bidirectional open-drain lines, Serial Data Line (SDA) and Serial Clock Line (SCL), pulled to logic level 1 by pullup resistors. Typical voltages used are +5 V or +3.3 V although systems with other voltages are permitted and are often encountered. This has an interesting implication: logic level 1 is achieved by doing nothing whilst logic level 0 needs to be pulled down to ground. Although there are bidirectional I2C voltage level shifters (like the PCA9306) it may be feasible to experiment simpler solutions if on the I2C bus you're not going to have many devices connected. See more on this in the “Voltage Level Shifting” chapter.

Preface

Most modern PC have several internal components that communicate vital information, like internal temperatures of critical components, over I2C bus. This type of stuff is factory wired into your PC and is normally dealt with lm_senosrs … what we want to do here is use an I2C bus on your computer to connect some external I2C sensor like an accelerometer. I'm tagging this in the ARM hardware section because I think that, excluding the lm-sensors stuff, most people will be doing this sort of thing on embedded ARM systems … but the concepts are applicable to any linux capable system with an I2C bus.

Should you want to hack one of the many I2C busses on your PC the easiest one to access is the one in the DIMM modules. Modern DIMM modules have an I2C eprom in them that the bios reads to find out the characteristics of the DIMM module. Since the DIMM module can be removed from PC and hacked separately without risking to damage permanently your PC it is probably your safest option. See chapter “Hacking I2C in PC DIMM modules” for more details.

Preparing Your Host System

Before you start you might want to make sure that the OS has all that's required to manage the I2C bus you will be using. The first thing to do is make sure you have the correct kernel driver for whatever physical layer implementation is on your system. You will have to research on the datasheets of your system's hardware … my Pi has bcm2708 so in my case it was a matter of loading the bcm2708_i2c module. It might also be necessary to load i2c-dev module too depending on your setup.

Once you have the drivers right you might like to have a user-space tool to help you detect buses, present devices and communicate with the I2C devices each bus. I use i2ctools for this but it's not packaged amongst the Slackware packages and I was unable to find ant third party providing an ARM Slackware package so I downloaded the sources and compiled it for myself. Sources can be gotten from here: I2CTools.

Connecting A Device on the I2C Bus

Provided you've sorted out the voltage level issues (see the “Voltage Level Shifting” chapter) adding a new device in the bus is really simple. The bus is multimaster meaning that you can have many devices (upto 101) on the same bus so all you have to do is make 4 connections: Power, Ground, SDA and SCL. If it's the first device you're connecting on the bus it may be necessary to install the pullups between Power-SDA and Power-SCL. It's as simple as that and if the system is ready with the appropriate drivers and user-land utilities you are ready to access the newly connected device.

Detecting Connected Devices

There are probably many ways to determine what's connected to an I2C bus, I chose to use stuff out of the i2ctools project. I was unable to find a Slackware ARM package for i2ctools so I compiled and installed it on my system. On some distribution the package may be called t2c-tools.

First thing you want to know is what I2C busses are present on your system as there may be more then one and looking in the wrong bus may be frustrating:

```
root@pi:~# i2cdetect  -l
i2c-0   i2c             bcm2708_i2c.0                           I2C adapter
i2c-1   i2c             bcm2708_i2c.1                           I2C adapter
root@pi:~#
```

If you're not sure which bus you connected your stuff on you might want to do this on on all the busses:

```
root@pi:~# i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- 1e --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: 40 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- 69 -- -- -- -- -- --
70: -- -- -- -- -- -- -- 77
root@pi:~#
```

Communicating With An I2C Device

Communication with I2C devices id done by reading and writing to it's registers. Each device has it's own register list and is something you need to look for in the device's datasheet. Some devices even need some preliminary calibration to be done before you can read any sensible data out of them so before you start using i2cget to read out some registers you should at least have an idea of what

registers you are interested in and fave figured out if your device needs calibration prior to reading any sensible data. Registers can be set by using i2cset but do this only if you have read the datasheet.

Even once you know if your device needs calibration and the registers involved, the content of the registers may not be in a convenient format for ready use. There are generally various scripts in perl or python that address calibrating and managing data from specific I2C devices so that human readable information can be produced.

At this point it's impossible to show any further info about I1C communication without going into some detail about a specific device so I'll pick one ot the devices on my IMU pcb. While detecting previously on my PI I found 4 devices with the flowing hex addresses: 1e,40,69 and 77. Generally each I2C device has a range of addresses than can set. We shall concentrate on the device with 0x69 address. By doing some cross reference search on the IMU pcb datasheet and on the single datasheets of each device present on my IMU it's revealed that 0x69 should be the address of the ITG3200 (gyro + temp sensor) and indeed the ITG3200 datasheet asserts that it can have either 0x68 or 0x69 address selectable by logic level on pin 9. No what we need is the ITG3200 register chart:

```
Addr  Addr     Register Name R/W
Hex  Decimal
  0     0      WHO_AM_I      R/W
 15    21     SMPLRT_DIV     R/W
 16    22       DLPF_FS      R/W
 17    23       INT_CFG      R/W
 1A    26     INT_STATUS     R
 1B    27     TEMP_OUT_H     R                          TEMP_OUT_H
 1C    28     TEMP_OUT_L     R                           TEMP_OUT_L
 1D    29    GYRO_XOUT_H     R                         GYRO_XOUT_H
 1E    30    GYRO_XOUT_L     R                         GYRO_XOUT_L
 1F    31    GYRO_YOUT_H     R                         GYRO_YOUT_H
 20    32    GYRO_YOUT_L     R                         GYRO_YOUT_L
 21    33    GYRO_ZOUT_H     R                         GYRO_ZOUT_H
 22    34    GYRO_ZOUT_L     R                          GYRO_ZOUT_L
 3E    62      PWR_MGM       R/W
```

I chose the ITG3200 because it has a temperature sensor inside and I'm hoping I can read that without having to do any calibration, just for the sake of keeping the example as simple as possible. According to the chart the temperature register addresses are 1b and 1c so let's go and try and get some data out of there:

```
root@pi:~# i2cdump  -y -r 0x1b-0x1c 1 0x69 b
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
10:                               c0 90                         ??
root@pi:~#
```

The above example dumps registers 1b and 1c from the ITG3200 the same result can be achieved with i2cget:

```
root@pi:~# i2cget -y  1 0x69  0x1b b
0xc0
root@pi:~# i2cget -y  1 0x69  0x1c b
```

```
0x90
root@pi:~#
```

So we got c090 as our temperature reading. According to the datasheet this value is a 2's compliment of the temperature. So let's try and figure out what that would be: c090 written in binary is 1100000010010000, the most significant bit is 1 so the result should be

```
16528 - 32768 = -16240
```

I was unable to find in the datasheet what units this reading is in but they did mention that there was an average offset of 13200. I did a little google search and found this formula:

```
35 + ((raw value + 13200) / 280))
35 + ((13200 - 16240)/280) = 24.14
```

Considering that my current ambient temperature is about 20 Celcius I guess that for uncalibrated data that's OK.

If you want a script that does the maths for you and just reads out the ITG3200 sensor data in a human readable format here's an example:

```
#!/bin/bash
BUS=1
ID=0x69
ATH=0x1b
ATL=0x1c
ARXH=0x1d
ARXL=0x1e
ARYH=0x1f
ARYL=0x20
ARZH=0x21
ARZL=0x22

#need upper case hex stripped of prefix "0x" or bc will not like the input
for VAR in TH TL RXH RXL RYH RYL RZH RZL
do
  CMD="$VAR=\$(i2cget -y $BUS $ID \$A$VAR  b |sed -e "s/^0x//" |tr "a-z" "A-
Z")"
  eval $CMD
  eval "echo $VAR = \$$VAR"
done

echo "Temp register: $(echo "ibase=16; $TH$TL" | bc -l) 0x$TH$TL ($(echo
"ibase=16; obase=2; $TH$TL" | bc -l))"

echo -n "Temp in Celcius: "
#this takes hex input and evaluates the followin formula in hexadecimal
#temp= 35 + ((raw + 13200) / 280))"
#where raw is the input reading un 2's compliment
#(to uncompliment the input I take away 0x10000 if input is larger then
```

```
0x8000)
echo "ibase=16; input=$TH$TL; if ( input >= 8000 ) { raw=input - 10000;}
else { raw=input;}; 23 + ((raw + 3390)/118);" |bc -l

echo "X Axis Rotation Register: $(echo "ibase=16; $RXH$RXL" | bc -l)
0x$RXH$RXL ($(echo "ibase=16; obase=2; $RXH$RXL" | bc -l))"
echo -n "X Axis Angula velocity degree/sec: "
echo "ibase=16; input=$RXH$RXL; if ( input >= 8000 ) { raw= input - 10000;}
else { raw=input;}; raw / E.177" |bc -l

echo "Y Axis Rotation Register: $(echo "ibase=16; $RYH$RYL" | bc -l)
0x$RYH$RYL ($(echo "ibase=16; obase=2; $RYH$RYL" | bc -l))"
echo -n "Y Axis Angula velocity degree/sec: "
echo "ibase=16; input=$RYH$RYL; if ( input >= 8000 ) { raw= input - 10000;}
else { raw=input;}; raw / E.177" |bc -l

echo "Z Axis Rotation Register: $(echo "ibase=16; $RZH$RZL" | bc -l)
0x$RZH$RZL ($(echo "ibase=16; obase=2; $RZH$RZL" | bc -l))"
echo -n "Z Axis Angula velocity degree/sec: "
echo "ibase=16; input=$RZH$RZL; if ( input >= 8000 ) { raw= input - 10000;}
else { raw=input;}; raw / E.177" |bc -l
```

The above script just does one simple dump of the register data and converts the values into human readable format, it does nothing with regards to calibration and averaging out vibrations. More consistent gyroscopic readings would be achieved if an average over 10 consecutive data samples was made thus averaging out most of the ambient vibrations.

A bash script is really not the most suitable way to read data from I2C devices, a faster means of managing the data from the devices is really mandatory in order to do calibration, data averaging and what more to make the information consistent and useful for further calculations. I found the Linux kernel i2c documentation a usefull reference (<kernel source tree>/Documentation/i2c/dev-interface); it's not the only way that data can be read but it's a good starting point.

I hate showing my poor C programming capabilities but here's some code that uses i2c-dev to read stuff from the ITG3200 and takes an average over 10 readings:

```
#include <sys/ioctl.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <linux/i2c-dev.h>
#include <fcntl.h>
#include <errno.h>

#define I2C_DEVICE "/dev/i2c-1"

/*ITG3200*/
#define ITG3200_ADDR 0x69
#define ITG3200_SELF 0x0
```

```
#define ITG3200_INT 0x1a
#define ITG3200_TH 0x1b /*2 bytes Hight byte and Low byte*/
#define ITG3200_TL 0x1c
#define ITG3200_XRH 0x1d /*2 byte Hight byte and Low byte*/
#define ITG3200_XRL 0x1e
#define ITG3200_YRH 0x1f /*2 byte Hight byte and Low byte*/
#define ITG3200_YRL 0x20
#define ITG3200_ZRH 0x21 /*2 byte Hight byte and Low byte*/
#define ITG3200_ZRL 0x22 /*2 byte Hight byte and Low byte*/
#define ITG3200_TEMP_RAW_OFFSET 13200
#define ITG3200_TEMP_RAW_SENSITIVITY 280
#define ITG3200_TEMP_OFFSET 35
#define ITG3200_ROT_RAW_SENSITIVITY 14.375

int twosc2int(int twoscomplimentdata)
{ int retval;
  if( twoscomplimentdata > 32768 ) retval = twoscomplimentdata - 65536;
  else retval = twoscomplimentdata;
  return retval;
}

float ITG3200_rot_conv(int rawdata)
{ float retval;
  int raw;

  raw=twosc2int(rawdata);
  retval = (float)raw / (float)ITG3200_ROT_RAW_SENSITIVITY;
  return retval;
}

float ITG3200_temp_conv(int rawdata)
{ float retval;
  int raw;

  raw=twosc2int(rawdata);
  retval = (float)ITG3200_TEMP_OFFSET + (((float)raw +
ITG3200_TEMP_RAW_OFFSET) / ITG3200_TEMP_RAW_SENSITIVITY);
  return retval;
}

void ITG3200_read (int file, int *raw, int *reg_array,int size)
{ __s32 res;
  int i,j,k;

  for(i=0;i<size;i++)
  { k=0;
    for (j=0;j<2;j++)
    {
      if( (res = i2c_smbus_read_byte_data(file,*(reg_array + i + j)) )< 0 )
      { printf("Failed to read from the i2c bus.\n");
```

```
        exit(1);
      }
      if (j == 0) k=(int)res << 8;
      else
      { k += (int)res;
        *(raw + (i/2))=k;
      }
    }
    i++;
  }
}

main ()
{ int file;
  int i,j,k;
  float data[4]={0};

  int ITG3200_REGS[8]={ITG3200_TH,ITG3200_TL,ITG3200_XRH,ITG3200_XRL,
    ITG3200_YRH, ITG3200_YRL,ITG3200_ZRH,ITG3200_ZRL};
  int ITG3200_RAW_DATA[4];
  float ITG3200_DATA[4];

  if ((file = open(I2C_DEVICE, O_RDWR)) < 0)
  { perror("Failed to open the i2c bus");
    exit(1);
  }

  if (ioctl(file, I2C_SLAVE, ITG3200_ADDR) < 0)
  { printf("Failed to acquire bus access and/or talk to slave.\n");
    exit(1);
  }

/*Take an avarage over 10 consecuitve readings on the ITG3200*/
  for (i=0;i<10;i++)
  {
ITG3200_read(file,&ITG3200_RAW_DATA[0],&ITG3200_REGS[0],sizeof(ITG3200_REGS)
/sizeof(ITG3200_REGS[0]));

    data[0] += ITG3200_temp_conv(ITG3200_RAW_DATA[0]);
    data[1] += ITG3200_rot_conv(ITG3200_RAW_DATA[1]);
    data[2] += ITG3200_rot_conv(ITG3200_RAW_DATA[2]);
    data[3] += ITG3200_rot_conv(ITG3200_RAW_DATA[3]);
  }
  for(i=0;i<4;i++) data[i] /= 10;

  printf("Temp. : %2.2f \n",data[0]);
  printf("Rot. X : %2.2f \n",data[1]);
  printf("Rot. Y : %2.2f \n",data[2]);
  printf("Rot. Z : %2.2f \n",data[3]);

  close(file);
```

```
}
```

Voltage Level Shifting

You may end up with heterogeneous voltage level devices and if you have many devices the correct way to work around this problem is by using bidirectional I2C voltage level shifters like the PCA9306, but if you only have a few devices all grouped up in a neat PCB like the 10DOF IMU unit you might want to give a simpler system a try. This is how I connected my 5v IMU pcb to a 3.3v I2C bus on my RaspberryPI: I took as educated guess that 4.4v (5v with a 4148 diode in series) would still be a tolerable power supply voltage for the whole IMU pcb, this would most likely allow all the I2C devices on the IMU pcb to recognize a minimum of 3.08v (4.4 * 0.7) as the lowest reliable logic level 1 tension allowing it to inter-operate with the PI's 3.3v logic levels. I was not able to find if the PI has internal pullups on the I2C bus or if they have to be externally placed so in doubt I put in 10k pullups between the 4.4v power line and the 2 data lines. I was the able to correctly detect the sensors on the IMU pcb.

Hacking I2C in PC DIMM modules

As mentioned in the Preface it is possible to hack, in fact, any one of the I2C busses on your PC it's just that the one in DIMM modules it the easiest one. You will need to identify these 4 connections on your DIMM module's eprom that respectively go to these pins on the DIMM module itself:

200 pin SO-DIMM 1/2:

Pin	Function
197	Vcc (3v)
193	SDA
195	SCL
185	GND

204 pin SO-DIMM 3:

Pin	Function
199	Vcc (3v)
200	SDA
202	SCL
203/204	GND

260 pin SO-DIMM 4:

Pin	Function
255	Vcc (2.5v)
254	SDA
253	SCL
251/252	GND

Once you have identified them you can stack up another I2C device on the bus provided it will not conflict with the addresses in use on this bus.

I have an old laptop with a 4Gb SO-DIMM 3 that is perfect for experimenting. The DIMM has onboard a ST M34: a I2C bus Serial EEPROM, SPD for DRAM. I downloaded the datasheet and found that on the eprom itself

Pin	Function
8	Vcc (3v)
5	SDA
6	SCL
4	GND

and I double-checked that these pins are actually connected to the respective pins on the SO-DIMM module. I was particularly lucky and this module has unused solder pads for a second eprom unit, making it super easy to solder some wires on the unused pats to hack another I2C device into the bus.

Next you will need to identify which bus is the one reading the eprom on the DIMM modules, there is another tool that is part of the i2ctool that comes in handy for this: decode-dimms. At the beginning it will sit out a line like this

Decoding EEPROM: /sys/bus/i2c/drivers/ee1004/0-0050

The last part 0-0050 identifies bus 0 address 50.

Looking at what devices are present on bus 0

```
# i2cdetect -y 0
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:                         08 -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: 30 -- -- -- -- 35 UU UU -- -- -- -- -- -- -- --
40: -- -- -- -- 44 -- -- -- -- -- -- -- -- -- -- --
50: UU -- 52 -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

So as long as whatever device I hack into this bus does not use address 8,30,35,36,37,44,50 or 52 it should work fine, provided the voltages are compatible.

Sources

- Originally written by louigi600

howtos, hardware, arm, author louigi600