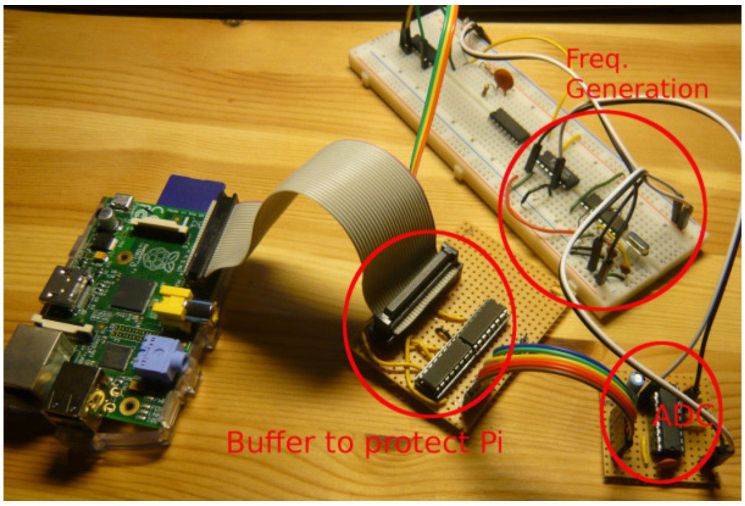
Raspberry Pi as an Oscilloscope @ 10 MSPS

— 20 Comments



How to make a 10MSPS scope from your Pi

Sooner or later when playing with electronics there comes a point where one needs to look at signals. For very slow signals a good way to analyze these is to use the sound card and a soft scope like xoscope [1]. But as soon as signals are a bit faster the sound card is far too slow. An intermediate step I took was the use of an Arduino and run it first with the internal ADC (Analog Digital Converter) 1 MSPS, and later with an external ADC with which I could reach around 5MSPS.

search here .
Go

Recent Posts

- Playing with the Cyclone V SoC system – DE0-Nano-SoC Kit/Atlas-So(
- Frequency generator based on the LTC6946
- Variable Gain Amplifier
- Using a PIC MCU to read out the MAX6675 temperature sensor (LCD,USB)
- PICkit2 programming with MPlabX – PIC16F690 UART ADC

Recent

This was still too slow for my purpose and I searched for a cheep way of getting even faster. What I had at home was a Raspberry Pi, which provides 17 GPIO (General Purpose Input Output) pins which can be used to interface the ADC and hopefully achieve a faster readout compared to the Arduino.The Raspberry Pi is a more or less general purpose computer running a Linux operation system which is definitely not real time!These are not the best initial conditions for such a project, since when reading out an external ADC one needs to make sure that the time between each sample point is the same. On a none real time operation system this is not guaranteed. There are other processes using the CPU and interrupts, especially from the GPU, that are making a real time read out impossible.
One very annoying interrupt, the adjustment of the refresh rate of RAM every 500ms, can be disabled by:

```
sudo sed -i '$s/$/\ndisable_pvt=1/'
/boot/config.txt
```

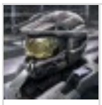
This is in general a good idea when working with fast GPIO operations.
After a lot of tests in user space and a lot of reading about interrupts and process control in Linux I decided to test something completely new, something which promised real time operations on sub second level, running without interrupts, direct access to hardware and register manipulation.

A Linux kernel module

Writing a Linux kernel module provides the possibility to do low level hardware operations. Which is exactly what we want to do. We need to run with the highest possible priority, disable the interrupts and read out the GPIO register as fast as possible.

In order to get started the easiest is to set up a cross compiler on a Linux machine and do the code writing and compilation on a faster computer [2]. Or you can do it directly on the Raspberry Pi by using the setup script in this article.
When the kernel compiles and is installed one can start

Comments



crmann on Playing with the Cyclone V SoC…

Archives

- August 2016
- June 2016
- February 2016
- January 2016
- December 2015
- October 2015
- September 2015
- July 2015
- June 2015
- May 2015
- April 2015
- March 2015
- December 2014
- October 2014
- September 2014
- August 2014
- June 2014
- April 2014
- February 2014
- January 2014

Categories

with the development of the kernel module. A useful documentation of Linux kernel development can be found in [3].
If we want to read and write registers on the Raspberry Pi we need to know their address.The documentation BCM2835-ARM-Peripherals [4] of the periphery of the Raspberry Pi is a good starting point.
When writing a kernel module we start with some basic include and function definitions:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/time.h>
#include <linux/io.h>
#include <linux/vmalloc.h>

int init_module(void);
void cleanup_module(void);
static int device_open(struct inode *
    struct file *);
static int device_release(struct ino
    struct file *);
static ssize_t device_read(struct fi
    char *, size_t, loff_t *);
static ssize_t device_write(struct f
    const char *, size_t, loff_t *);

#define SUCCESS 0
#define DEVICE_NAME "chardev"// Dev n
#define BUF_LEN 80//Max length of dev
```

The next step is to specify the hardware address of the periphery registers and specify some macros to do a more simple register manipulation. Here the starting address of the periphery address space is defined, some macros to set the bits in the periphery registers in the correct way. Some more details can be found in [4] and [5].

```
//Things for the GPIO Port

#define BCM2708_PERI_BASE       0x20
#define GPIO_BASE               (BCM
#define BLOCK_SIZE      (4*1024)

#define INP_GPIO(g)    *(gpio.addr +
#define OUT_GPIO(g)    *(gpio.addr +
//alternative function
#define SET_GPIO_ALT(g,a) *(gpio.add

#define GPIO_SET  *(gpio.addr + 7)  ,
#define GPIO_CLR  *(gpio.addr + 10) ,
```

- Uncategorized

Meta

- Register
- Log in
- Entries RSS
- Comments RSS
- WordPress.com

```

#define GPIO_READ(g)  *(gpio.addr + 1

//GPIO Clock
#define CLOCK_BASE               (BCM2
#define GZ_CLK_BUSY (1 << 7)
```

Further we need to connect our external ADC in some way to the GPIO. So we need to decide on which GPIO pins we want to connect. Since I'm using an 6-bit ADC for this example, I need 6 GPIO pins to connect my ADC. So the pins are defined for ADC 1 and optional for ADC 2.

```
//How many samples to capture
#define SAMPLE_SIZE     10000

//Define GPIO Pins

//ADC 1
#define BIT0_PIN 7
#define BIT1_PIN 8
#define BIT2_PIN 9
#define BIT3_PIN 10
#define BIT4_PIN 11
#define BIT5_PIN 25
//ADC 2
#define BIT0_PIN2 17
#define BIT1_PIN2 18
#define BIT2_PIN2 22
#define BIT3_PIN2 23
#define BIT4_PIN2 24
#define BIT5_PIN2 27
```

Then one needs some more function and variable definitions for the kernel module:

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


static int map_peripheral(struct bcm2
static void unmap_peripheral(struct b
static void readScope();//Read a samp


static int Major;       /* Major numb
static int Device_Open = 0; /* Is dev
static char msg[BUF_LEN];
static char *msg_Ptr;


static uint32_t *ScopeBuffer_Ptr;
static unsigned char *buf_p;


static struct file_operations fops =
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};
```

We need to assign the addresses of GPIO and the clock to a variable that we can find the hardware. A data structure is defined to hold our values we read out from the ADC, as well as the time from start of the readout to the end of the readout. This time is needed in order to calculate the time step between each sample. Additional two pointers are defined for later operations.

```
static struct bcm2835_peripheral myc

static struct bcm2835_peripheral gpi


static struct DataStruct{
    uint32_t Buffer[SAMPLE_SIZE];
    uint32_t time;
};

struct DataStruct dataStruct;

static unsigned char *ScopeBufferSta
static unsigned char *ScopeBufferSto
```

Since we want to manipulate hardware registers we need to map the hardware registers into memory. This can be done by two functions, one for the mapping and one for the unmapping.

```
static int map_peripheral(struct bcm2
{
    p->addr=(uint32_t *)ioremap(GPIO_
    return 0;
}

static void unmap_peripheral(struct b
    iounmap(p->addr);//unmap the addre
}
```

At this point we have defined most of the things needed in our kernel module. It’s time to start with the implementation of some code which is actually doing some work.
The readScope() function is probably the most important part of this work, since here the real readout is done. But

it is easy to understand. The first thing is to disable all interrupts. This is important since we want to run in real time and do not want to get interrupted. But it is very important that we do not do this too long… since basically everything freezes during this time. No network connection and file write operations are happening. In our case we are only taking 10k samples so not too much time. Before the sample taking we take a time stamp. Then we read out 10k times the GPIO register and save it in our data structure. The GPIO register is a 32bit value so it is made out of 32 ‘1’s and ‘0’s each defining if the GPIO port is high (3.3V) or low (GND). After the read out we take another time stamp and turn on all interrupts again. The two time stamps we took are important since we can calculate how long it took to read in the 10k samples. The time difference divided by 10k gives us the time between each sample point. In case the sample frequency is too high and should be reduced one can add some delay and waste some time during each readout step. Here the aim is to achieve the maximal performance.

```
static void readScope(){

    int counter=0;
    int Fail=0;

    //disable IRQ
    local_irq_disable();
    local_fiq_disable();

    struct timespec ts_start,ts_stop
    //Start time
    getnstimeofday(&ts_start);

    //take samples
    while(counter<SAMPLE_SIZE){
    dataStruct.Buffer[counter++]= *(
    }

    //Stop time
    getnstimeofday(&ts_stop);

    //enable IRQ
    local_fiq_enable();
    local_irq_enable();

    //save the time difference ns re
    dataStruct.time=
        timespec_to_ns(&ts_stop)-tim

    buf_p=&dataStruct;//cound maybe

    ScopeBufferStart=&dataStruct;

    ScopeBufferStop=
        ScopeBufferStart+sizeof(stru
}
```

In order to make a kernel module work the module needs some special entry functions. One of these functions is the init_module(void) which is called when the kernel module is loaded. Here the function to map the periphery is called, the GPIO pins are defined as inputs and a device file in /dev/ is created for communication with the kernel module. Additionally a 10 MHz clock signal on the GPIO Pin 4 is defined. This clock signal is needed in order to feed the ADC with an input clock. A 500 MHz signal from a PLL is used and the clock divider is set to divide by 50, which gives the required 10 MHz signal. More details on this clock can found in chapter 6.3 General Purpose GPIO Clocks in [4].

```
int init_module(void)
{
    Major = register_chrdev(0, DEVICI

    if (Major < 0) {
        printk(KERN_ALERT "Reg. char
        return Major;
    }

    printk(KERN_INFO "Major number %(
    printk(KERN_INFO "created a dev
    printk(KERN_INFO "'mknod /dev/%s
        DEVICE_NAME,Major);

    //Map GPIO

    if(map_peripheral(&gpio) == -1)
    {
        printk(KERN_ALERT,"Failed to
        return -1;
    }

    //Define Scope pins
    // Define as  Input
    INP_GPIO(BIT0_PIN);
    INP_GPIO(BIT1_PIN);
    INP_GPIO(BIT2_PIN);
    INP_GPIO(BIT3_PIN);
    INP_GPIO(BIT4_PIN);
    INP_GPIO(BIT5_PIN);

    INP_GPIO(BIT0_PIN2);
    INP_GPIO(BIT1_PIN2);
    INP_GPIO(BIT2_PIN2);
    INP_GPIO(BIT3_PIN2);
    INP_GPIO(BIT4_PIN2);
    INP_GPIO(BIT5_PIN2);

    //Set a clock signal on Pin 4
    struct bcm2835_peripheral *p=&my
    p->addr=(uint32_t *)ioremap(CLOC

    INP_GPIO(4);
    SET_GPIO_ALT(4,0);

    int speed_id = 6; //1 for 19Mhz 
    *(myclock.addr+28)=0x5A000000 | 

    //Wait untill clock is no longer
    while (*(myclock.addr+28) & GZ_C
    //Set divider //divide by 50
    *(myclock.addr+29)= 0x5A000000 |
    *(myclock.addr+28)=0x5A000010 | 

    return SUCCESS;
}
```

A clean up function is needed to clean up when the kernel module is unloaded.

```
void cleanup_module(void)
{
    unregister_chrdev(Major, DEVICE_N
    unmap_peripheral(&gpio);
    unmap_peripheral(&myclock);
}
```

Furthermore a function is needed which is called when the device file belonging to our kernel module is opened. When this happens the measurement is done by calling the readScope() function and saved in memory.

```
static int device_open(struct inode 
    struct file *file)
{
    static int counter = 0;

    if (Device_Open)
        return -EBUSY;

    Device_Open++;
    msg_Ptr = msg;

    readScope();//Read n Samples int

    try_module_get(THIS_MODULE);

    return SUCCESS;
}
```

Also a function which is called when closing the devise file is needed.

```
static int device_release(struct inode
    struct file *file)
{
    Device_Open--; /* We're now ready
    module_put(THIS_MODULE);
    return 0;
}
```

When the device is open we can read from it which calls the function device_read() in kernel space. This returns the measurement we made when we opened the device. Here one could also add a call of the function readScope() in order to do a permanent readout. As the code is right now one needs to open the device file for each new measurement, read from it and close it. But we leave it like this for the sake of simplicity.

```
static ssize_t device_read(struct fi
               char *buffer,
               size_t length,
               loff_t * offset)
{
    // Number of bytes actually writ
    int bytes_read = 0;

    if (*msg_Ptr == 0)
        return 0;

    //Check that we do not overfill t
    while (length && buf_p<ScopeBuffe

        if(0!=put_user(*(buf_p++), bu
            printk(KERN_INFO "Probler
        length--;
        bytes_read++;
    }
    return bytes_read;
}
```

The last step to make our kernel module complete is to define a function that is called when we want to write into the device file. But this functions does nothing except for writing an error message, since we do not want write support.

```
static ssize_t
device_write(struct file *filp, const
    size_t len, loff_t * off)
{
    printk(KERN_ALERT "This operation
    return -EINVAL;
}
```

Now all code is ready to compile the kernel module. This can be done by a “Makefile” containing:

```
obj-m += Scope-drv.o

all:
make -C /lib/modules/$(shell uname
-r)/build \
M=$(PWD) modules

clean:
make -C /lib/modules/$(shell uname
-r)/build M=$(PWD) clean
```

and in case one does cross compiling an addition “Makefile.crosscompile” which contains:

```
KERNEL_TREE=Set_your_path_here/linux-
rpi-x.x.x"
CCPREFIX=Set_your_path_here/arm-bcm2708-
linux-gnueabi-

obj-m += Scope-drv.o

all:
make -C ${KERNEL_TREE} ARCH=arm
CROSS_COMPILE=${CCPREFIX} M=$(PWD) modules
```

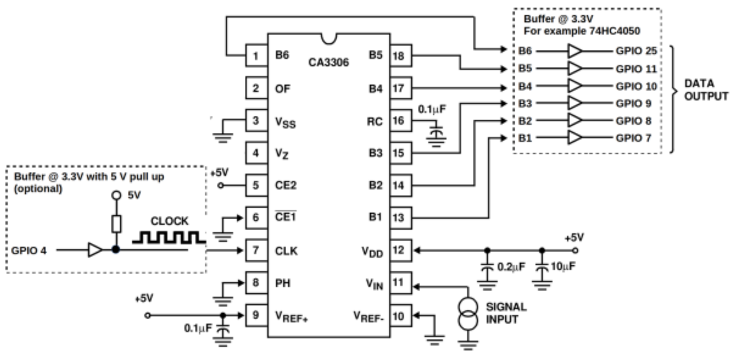
Now one can compile by a simple “make” or “make -f Makefile.crosscompile” the kernel module. Of course after having set up the kernel sources correctly [2].
With the command:

```
sudo insmod ./Scope-drv.ko
sudo mknod /dev/chardev c 248 0
```

The kernel module is loaded and the device file is assigned.

Connecting an ADC to the Raspberry Pi

Now we are able to read out an ADC connected to the GPIO of the Raspberry Pi, but of course we need to connect one first. This will be the next step of this tutorial. I'm using a CA3306 ADC from Intersil. This is a 6-bit 15 MSPS ADC with a parallel read out. I'm using this particular chip since it's very cheep and fast. But any other chip should also work even though of course one needs to check with the data sheet how to connect it. 6-bit means that we have 64 levels between ground level and reference voltage. Which is not much but enough for simple tests and our purpose. The ADC is operation at 5V which means we cannot connect it directly to the Raspberry Pi! We need a level converter in between. The simplest way to do level conversion is to use a buffer like the CMOS 74HC4050 buffer. Also for the clock signal which comes from GPIO 4 it is a good idea to put a buffer in between to protect the Raspberry Pi. Since the buffers should run at 3.3V it can be necessary to place a pull up resistor to 5V behind the clock buffer. Since 3.3V is not yet logic high level at 5V this will pull up the voltage level a bit to make the ADC trigger. In my case it was not needed since it worked without the pull up resistor.



How to do data acquisition?

Now we have everything in place to get started to look at signals. We have the ADC connected and we have a kernel module that can interface this piece of hardware. Now we need a program that communicates with the device file. I will present a simple program that does the readout and conversion of the bits and print out the result on the screen as a table. The result can further be

processed with for example a plot program like gnuplot [6].
First we start again with defining some variables and structures which are needed for further processing. Attention should be paid to the "DataPointsRPi" so that it has the same value as the "SAMPE_SIZE" in the kernel module.

```
#include <iostream>
#include <cmath>
#include <fstream>
#include <bitset>

typedef unsigned short    uint16_t;
typedef unsigned int      uint32_t;

const int DataPointsRPi=10000;
//This is the structre we get from tl
struct DataStructRPi{
    uint32_t Buffer[DataPointsRPi];
    uint32_t time;
};
```

After this we start building the main() function, we basically open the device file "/dev/chardev" and read from it. The data we get is binary and we put it into the data structure, from where we easily can access it.

```
int main(){

    //Read the RPi
    struct DataStructRPi dataStruct;
    unsigned char *ScopeBufferStart;
    unsigned char *ScopeBufferStop;
    unsigned char *buf_p;

    buf_p=(unsigned char*)&dataStruc
    ScopeBufferStart=(unsigned char*
    ScopeBufferStop=ScopeBufferStart-
        sizeof(struct DataStructRPi)

    std::string line;
    std::ifstream myfile ("/dev/charc
    if (myfile.is_open())
    {
      while ( std::getline (myfile,l
      {
        for(int i=0;i<line.size();i+
          if(buf_p>ScopeBufferStop)
            std::cerr<<"buf_p out of
          *(buf_p)=line[i];
          buf_p++;
        }
      }
      myfile.close();
    }
    else std::cerr << "Unable to oper
```

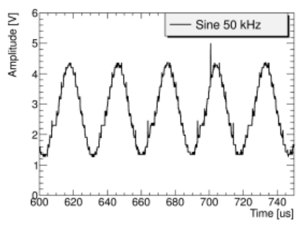
The next step is to calculate the time step between each sample point. Remembering that the duration of ADC readout was saved in the kernel module, we can now calculate the time step by dividing this time by 10k. Since GPIO bits are not laying beside each other they need some sorting. This can be done by performing bitwise shift operations. We remember that the GPIO register is 32 bits long we need to extract the bits for ADC1 and in case we have connected a second ADC also for ADC2. After this sorting we end up with a value for ADC1 and ADC2 which we print together with the time.

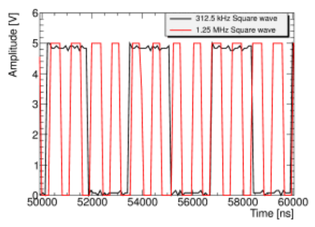
```
    //convert structure we get from R
    //time step in ns
    double time=dataStruct.time/(doul

    for(int i=0;i<DataPointsRPi;i++)·

        int valueADC1=0;//ADC 1
        //move the bits to the right
        int tmp = dataStruct.Buffer[:
        valueADC1=tmp>>(7);
        tmp = dataStruct.Buffer[i] &
        valueADC1+=(tmp>>(20));


        int valueADC2=0;//ADC2
        tmp = dataStruct.Buffer[i] &
        valueADC2=tmp>>17;
        tmp=dataStruct.Buffer[i] & ((
        valueADC2+=(tmp>>20);
        tmp=dataStruct.Buffer[i] & ((
        valueADC2+=(tmp>>22);

        std::cout<<i*time<<"\t"<<valu
            <<"\t"<<valueADC2*(5./63
    }
    return 0;
}//end main
```

The voltage is converted by multiplying with 5V/63 and we get a table of the form:

```
time [ns] Value ADC1 [V] Value ADC2 [V]
```

The data can now be analysed with for example gnuplot by plotting it.





How the data is analysed at the end is of course up to the individual user and the described code for data analysis should be understood as a starting point. I myself have written a server application which reads out the device file on the Raspberry Pi every few milliseconds and sends the data via Ethernet to a computer with a screen. On the computer I run a Qt application which is doing some raising edge triggering in order to get a stable signal, perform FFT and much more one requires from a real scope. If not willing to write that much code a good starting point could be to modify the xoscope [1], there a lot of things are already available. In the documentation they write that there is already a parallel port support so it should not be too complicated to use instead of parallel port the designed GPIO interface.

Get ready for kernel module compilation with the Raspberry Pi

```
#!/bin/bash

FV=`zgrep "* firmware as of" /usr/sha
| head -1 | awk '{ print $5 }'`

mkdir -p k_tmp/linux

wget https://raw.github.com/raspberry
wget https://raw.github.com/raspberry
-O k_tmp/Module.symvers

HASH=`cat k_tmp/git_hash`

wget -c https://github.com/raspberryp

cd k_tmp
tar -xzf linux.tar.gz

KV=`uname -r`

sudo mv raspberrypi-linux* /usr/src/
sudo ln -s /usr/src/linux-source-$KV

sudo cp Module.symvers /usr/src/linu

sudo zcat /proc/config.gz &gt; /usr/

cd /usr/src/linux-source-$KV/
sudo make oldconfig
sudo make prepare
sudo make scripts
```

[1] http://xoscope.sourceforge.net/
[2] http://elinux.org/RPi_Kernel_Compilation
[3] http://www.tldp.org/LDP/lkmpg/2.6/html/lkmpg.html
[4] http://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf
[5] http://www.pieter-jan.com/node/15
[6] http://www.gnuplot.info/

The described code can be found on GitHub
https://github.com/digibird1/RPiScope

It can be checked out with:

```
git clone https://github.com/digibird1/RPiScope
```

Share this:

Twitter Facebook 6 Google

Like

Be the first to like this.

20 comments on “Raspberry Pi as an Oscilloscope @ 10 MSPS”

someone says:

November 24, 2014 at 08:49

the set clock signal on pin 4 should be OUT_GPIO(4) !
Am I Right ?

Reply

digibird1 says:

December 4, 2014 at 17:57

Yes I believe you are right… thanks for pointing this out.

Reply

Malik says:

March 17, 2015 at 10:05

No, it's wrong, do SET_GPIO_ALT(4,0) set the pin 4 automatically in out mode because the alternate function 0 of the pin 4 is GPCLK0 , and you need to do INP_GPIO(g) before OUT_GPIO(g) or SET_GPIO_ALT(g).

Faraz says:

April 26, 2015 at 06:46

wooow, great job. Salute.
I'm going to do something like this, and it really helped me. but i need some modifications, to get long on-going buffer. and this solution without needs for another processor and memory look fantastic simple and low cost.
but It seems that is not possible, to have all sample buffered (no loss), and also give time to the OS for its jobs, since we turn interrupts off during logging, and when we turn interrupts on for OS, than we loose

some samples. do you think there may be any work around?

Reply

digibird1 says:
April 26, 2015 at 17:37

Thanks 🙂
Yes it is fully true that when having the interrupts on you will loose samples, the worst is actually that it is becoming unpredictable how much time is between the sampled points.

I actually think there is no way of having this speed and have at the same time the operation system running with interrupts turned on or continues buffering.

If you can lower the speed I believe it becomes possible, but it is on the cost of speed.

This issue is one of the reasons why I have moved to a FPGA design.

I think if you are not fixed to the Raspberry Pi you could use a processor which has a DMA controller and connect the ADC to its bus. But I have not looked into this solution.
Maybe the Beagle Bone can do this or one of the Arduinos. The Arduino Due had a DMA controller if I remember correctly, but will not run as easy the Linux as the RPi does. Maybe the Arduino tre.

Reply

kingspp says:
May 11, 2015 at 14:50

Hello digibird1,

Myself, Electronics and communication student. Great work, regarding RPiScope. I have completed RPiScope using MCP3208 with SPI Interface. I am able to get 5ksps at maximum. Later I built using CA3306 and followed your steps , but I am unable to achieve the results.

Model : Raspberry Pi B+
OS : Raspbian

cat /proc/version: Linux version 3.12.35+ (dc4@dc4-XPS13-9333) (gcc version 4.8.3 20140303 (prerelease) (crosstool-NG linaro-1.13.1+bzr2650 – Linaro GCC 2014.03)) #730 PREEMPT Fri Dec 19 18:31:24 GMT 2014

A/D IC : CA3306CE
Buffer: TC4050BP (5V –> 3.3V)
Function Generator: Standard Generator

Steps:
1. Cloned the git
2. Changed permission for GetReadyKernelCompile.sh and executed it
3. Moved to KernelMod folder and executed make command.
4. sudo insmod ./Scope-drv.ko
5. sudo mknod /dev/chardev c 248 0
6. Moved to ReadScope folder
7. g++ main.cpp main
8. ./main

I always get some constant noise value, if I plot it using GNU Plot.

Can you please guide me?

Reply

digibird1 says:
May 17, 2015 at 16:05

Hi kingspp,

thank you!
I have never tried with the RPi2. Ans also not with the latest distribution.

I have some thoughts suggestions how to track the problem.

1) The RPi 2 has a different pin GPIO compared to the RPi 1, could there be some pin differences?
2) Can you read out the ADC without the Kernel module? Just in user space? maybe the gpio library can help you to do the test or just look at some example how to read out the GPIO register. You can put a constant voltage on the ADC and check if you can read it from the ADC. If all this works you can look at the kernel module.

3) Try with an external clock to trigger the ADC, I had during development some issues with this.

4) Were there changes in the linux kernel? The RPi 2 has a different arm processor compared to the RPi1.

Reply

JWernette says:
April 6, 2016 at 17:14

Was this question ever resolved? I have built this project using the same steps, using the Raspberry Pi B. However I get noise in the beginning of the read and at the end with zero reading in between. I have tried troubleshooting the code by directly running a square wave to the GPIO pins but it still yields the same results. I read the GPIO pins during this process and they are triggering as they should. Any suggestions on where I should go from here?

JWernette says:
April 6, 2016 at 17:19

I have built this project using the same steps that Kingspp stated above, using the Raspberry Pi B. However I get noise in the beginning of the read and at the end with zero reading in between. I have tried troubleshooting the code by directly running a square wave to the GPIO pins but it still yields the same results. I read the GPIO pins during this process and they are triggering as they should. Any suggestions on where I should go from here?

Issam says:
May 17, 2015 at 15:40

Great Work !
i'm wondering about somthing ….
" Since we want to manipulate hardware registers we need to map the hardware registers into memory. This can be done by two functions, one for the mapping and one for the unmapping "

can i ask u why u do unmapping after mapping the register ?

Reply

digibird1 says:
May 17, 2015 at 16:12

Hi Issam,

thanks for your comment. I'm not 100% sure what your mean.

I map in init_module()
and unmap in the void cleanup_module(void)
function.

One for loading the module and one for unloading the module.

Reply

kemelloago says:
June 24, 2015 at 21:16

Hi! Awesome work! Congrats. Could you tell me which OS are you using… (I suppose raspbian) and which kernel?

Thanks,

Tiago

Reply

digibird1 says:
June 27, 2015 at 18:08

Hi Tiago,

thanks!

Yes I use the raspbian Linux version, I was doing this with kernel linux-rpi-3.11.y

Cheers

Reply

Joaquin says:
July 31, 2015 at 22:47

Hi, I´m not able to buy CA3306. If I use another ADC, does it have to be of 6 bits, and 15MSPS too?

Thanks

Reply

digibird1 says:
July 31, 2015 at 23:35

You should use an ADC which has a parallel readout, the more bits and speed you get the more expensive the ADC gets. It should be no problem to use for example 8 bits or more, as long as you have enough GPIO pins to read out. An option could for example be the ADC AD9057 which is 8 bits and is available in a 40 MSPS version. See one of my other projects were I have used this ADC. But you have to to modify the code and read out the two more additional bits. Keep in mind the ADC analog input voltage and the digital ouput voltage to not destroy the ADC or the GPIO of the RPi!

Reply

MakCuber says:
November 25, 2015 at 08:03

would this also work with the 80MSPS version? do you need a FPGA between the RPi and the ADC or can you just swap out the ADC you used in this project for any ADC and just modify the code you used?

digibird1 says:
November 25, 2015 at 19:45

@ MakCuber, Don’t know the specs of the 80 MSPS version but if you can run it at lower speed you probably can use it. The RPi can max read around 10 MSPS on the GPIO, this will not be changed by a faster ADC. Of cause you can plug some additional logic in between capture buffer and read out. But then you could also use a different interface like USB or so. Connecting a FTDI USB2.0 chip to an ADC should give you around 60 MSPS. If you use an FPGA see my other project where I use an FPGA, but it is a kind of overkill if you don't want to add DSP or so. A CPLD is cheaper compared to a FPGA. But as soon as one involves additional hardware I would integrate a buffer and move away from running without interrupts. The idea of the project was as little extra hardware as possible and achieving the highest speed.

darkvoice says:
November 19, 2015 at 22:20

Great project!

Do you think by this it is also possible to capture data form an old skool cpu to GPIO? Adress busses also. Just snooping in on x26fe and recording the 01001101 data written.

And what about using the cpu's E and Q to trigger (irq like) data writing to an address captured bij GPIO?

Thanks

Reply

digibird1 says:
November 25, 2015 at 19:48

You mean using the RPi as a logic analyses? This works and there is a project with GUI and all out there if I remember correctly. Search for Panalyzer. But I haven't tried.

Reply

CapTech outcomes – the Murgen project says:
January 13, 2016 at 22:20

[…] Raspberry Pi as an Oscilloscope @ 10 MSPS […]

Reply

Leave a Reply

Enter your comment here...