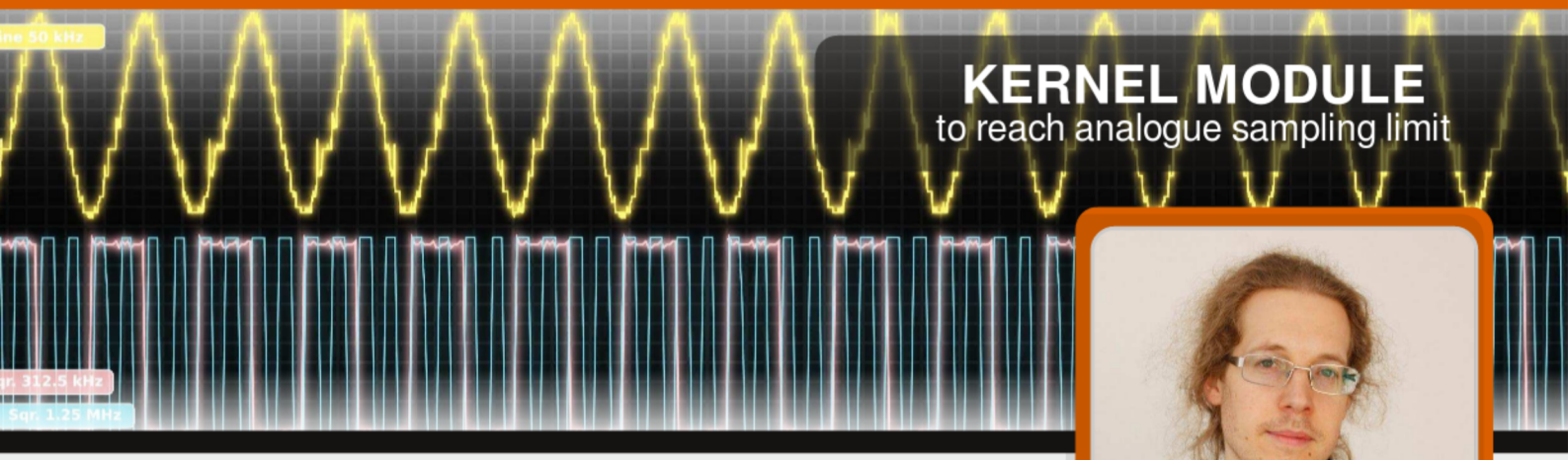


Building an oscilloscope with a Raspberry Pi

Daniel Pelikan

Guest Writer

SKILL LEVEL : ADVANCED

An oscilloscope can be very useful to analyse a circuit or experiment. Slow signals can be sampled with a sound card and xoscope http://xoscope.sourceforge.net/ However, sound cards cannot run at sampling frequencies above 100kHz.

To achieve higher sampling rates, I tried using an Arduino. With the Arduino internal Analog Digital Converter (ADC), I was able to reach 1,000,000 samples every second (1MSPS). Next, I tried using the Arduino with an external ADC, which reached around 5MSPS. However, this was still too slow for my application and I searched for a cheap way to reach a higher sampling speed.

I decided to try the Raspberry Pi, which provides 17 General Purpose Input Output (GPIO) pins that can be used to interface with ADC chips. Although we could use an ADC that connects via SPI or I^2C to the Raspberry Pi, these protocols result in sampling rates that are as slow as a sound card readout or slower. One needs to use an ADC that has a parallel data output, which can be connected to a Raspberry Pi.

Parallel ADC and realtime readout

A parallel ADC can be used to take a sample on the rising edge of a clock signal and output the sample on the data pins on the falling edge. The aim is to clock the ADC at our required sample rate and read all of the data pins between each sample.

The Raspberry Pi is a general purpose computer that can run a Linux operation system. However, Linux operating systems do not normally run processes in realtime. This is because the operating system listens for inputs from other devices, rather than just processing one command at a time. When reading an external ADC, one needs to make sure that the time between each sample point is the same. Without a realtime operating system, this is not guaranteed.

After a lot of tests and a lot of reading about interrupts and process control in Linux, I decided to write a Linux kernel module to try to solve the realtime readout problem.

A Linux kernel module

Writing a Linux kernel module provides the possibility to perform low level hardware operations. We need to run with the highest possible priority, reading the GPIO register with the system interrupts disabled for as short a time as possible.

Compiling a kernel module

The text that follows assumes that a Raspberry Pi is being used to compile the Linux kernel. However, another Linux PC can be used to perform cross-compilation to speed up the process: http://elinux.org/RPi_Kernel_Compilation

To set up the build environment correctly, copy the Bash script from the top of the next page into a file, make it executable and then run it.

Before proceeding, it may be useful to read over some Linux kernel development documentation:
http://www.tldp.org/LDP/lkmpg/2.6/html/lkmpg.html

To read and write registers on the Raspberry Pi, we need to know their addresses in memory. This information can be found in the BCM2835-ARM-Peripherals documentation:
http://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf

```
#!/bin/bash
# A script to setup the Raspberry Pi for a kernel build
FV=`zgrep "* firmware as of" /usr/share/doc/raspberrypi-bootloader/changelog.Debian.gz | head -1 | awk '{ print $5 }'`
mkdir -p k_tmp/linux
wget https://raw.github.com/raspberrypi/firmware/$FV/extra/git_hash -O k_tmp/git_hash
wget https://raw.github.com/raspberrypi/firmware/$FV/extra/Module.symvers -O k_tmp/Module.symvers
HASH=`cat k_tmp/git_hash`
wget -c https://github.com/raspberrypi/linux/tarball/$HASH -O k_tmp/linux.tar.gz
cd k_tmp
tar -xzf linux.tar.gz
KV=`uname -r`

sudo mv raspberrypi-linux* /usr/src/linux-source-$KV
sudo ln -s /usr/src/linux-source-$KV /lib/modules/$KV/build
sudo cp Module.symvers /usr/src/linux-source-$KV/
sudo zcat /proc/config.gz > /usr/src/linux-source-$KV/.config
cd /usr/src/linux-source-$KV/
sudo make oldconfig
sudo make prepare
sudo make scripts
```

Bash script

Writing the kernel module

Create a C file called `Scope-drv.c` that contains:

Scope-drv.c (1/10)

```
#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 inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t,
  loff_t *);
static ssize_t device_write(struct file *, const char *,
  size_t, loff_t *);

#define SUCCESS 0
#define DEVICE_NAME "chardev" /* Device name */
#define BUF_LEN 80 /* Maximum length of device message */
```

To set address values and allow simpler register manipulation in the C code, append the preprocessor macros at the bottom of this page to the C source file. More information on these macros can be found at: http://www.pieter-jan.com/node/15

The next piece of code that should be added to the C source file defines the GPIO connections that are used to connect to the ADC. For this article, a six bit ADC was used. Therefore, six GPIO connections are needed per ADC:

Scope-drv.c (3/10)

```
/* Number of 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
```

The numbering scheme used follows the BCM numbering scheme given at: http://elinux.org/RPi_Low-level_peripherals

Scope-drv.c (2/10)

```
/* Settings and macros for the GPIO connections */
#define BCM2708_PERI_BASE 0x20000000
#define GPIO_BASE (BCM2708_PERI_BASE + 0x200000) /* GPIO controller */

#define INP_GPIO(g)  *(gpio.addr + ((g)/10)) &= ~(7<<(((g)%10)*3))
#define SET_GPIO_ALT(g,a) *(gpio.addr + (((g)/10))) |= (((a)<=3?(a) + 4:(a)==4?3:2)<<(((g)%10)*3))


/* GPIO clock */
#define CLOCK_BASE              (BCM2708_PERI_BASE + 0x00101000)
#define GZ_CLK_BUSY (1 << 7)
```

Now add the remaining function and variable definitions given below to the C file.

Scope-drv.c (4/10)

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

static int map_peripheral(struct bcm2835_peripheral *p);
static void unmap_peripheral(struct bcm2835_peripheral *p);
static void readScope(void); /* Read a sample */

static int Major; /* Major number set for device driver */
static int Device_Open = 0; /* Store device status */
static char msg[BUF_LEN];
static char *msg_Ptr;

static unsigned char *buf_p;

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

static struct bcm2835_peripheral myclock = {CLOCK_BASE};
static struct bcm2835_peripheral gpio = {GPIO_BASE};
struct DataStruct{
  uint32_t Buffer[SAMPLE_SIZE];
  uint32_t time;
};

struct DataStruct dataStruct;

static unsigned char *ScopeBufferStart;
static unsigned char *ScopeBufferStop;
```

The first part of this code defines a struct to hold the address information. A pointer of this struct type is passed to the `map_peripheral()` and `unmap_peripheral()` functions, which are used to map the hardware registers into memory and release the mapping.

The `myclock` and `gpio` structs are assigned the register addresses of the GPIO and clock pins, such that they might be used later. The `DataStruct` is defined to hold the time and voltage data read from the ADC. The time information is needed in order to calculate the time between each sample. In addition two pointers `ScopeBufferStart` and `ScopeBufferStop` are defined for later use.

Now that all of the function declarations have been made, the implementation of each function must be added to complete the kernel module.

Memory mapping functions

```
static int map_peripheral(struct bcm2835_peripheral *p){
  p->addr=(uint32_t *)ioremap(GPIO_BASE, 41*4);
  return 0;
}
```

```
static void unmap_peripheral(struct bcm2835_peripheral *p){
  iounmap(p->addr);//unmap the address
}
```

Scope-drv.c (5/10)

Add these two functions to the end of the C source file.

The readScope function

The `readScope()` function is responsible for the ADC readout. Add the code below to the end of the C file.

```
static void readScope(){
  int counter=0;
  struct timespec ts_start,ts_stop;

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

  getnstimeofday(&ts_start); /* Get start time in ns */

  /* take samples */
  while(counter<SAMPLE_SIZE){
    dataStruct.Buffer[counter++]= *(gpio.addr + 13);
  }

  getnstimeofday(&ts_stop); /* Get stop time in ns */

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

  /* Store the time difference in ns */
  dataStruct.time =
    timespec_to_ns(&ts_stop) - timespec_to_ns(&ts_start);

  buf_p = (unsigned char*)&dataStruct;
  ScopeBufferStart = (unsigned char*)&dataStruct;

  ScopeBufferStop = ScopeBufferStart+
    sizeof(struct DataStruct);
}
```

Scope-drv.c (6/10)

The first action in this function is to disable all interrupts to provide realtime readout. It is very important that the time while the interrupts are disabled is minimised, since the interrupts are needed for many other Linux processes such as the network connections and other file operations. Reading 10,000 samples takes approximately 1ms, which is small enough not to cause interrupt related problems.

Before reading out the ADC, the current time in nano seconds is stored. Then the full GPIO register is read out 10,000 times and the data are saved in `dataStruct`. After the readout, the current time is requested again and the interrupts are enabled again. The time between each sample point is calculated from the time difference divided by 10,000.

The init_module function

In order to make a kernel module work, the module needs some special entry functions. One of these functions is the `init_module()`, which is called when the kernel module is loaded. Add the C code below to the end of the C source file.

Scope-drv.c (7/10)

```
int init_module(void){
  struct bcm2835_peripheral *p=&myclock;
  int speed_id = 6; /* 1 for 19MHz or 6 for 500 MHz */

  Major = register_chrdev(0, DEVICE_NAME, &fops);
  if(Major < 0){
    printk(KERN_ALERT "Reg. char dev fail %d\n",Major);
    return Major;
  }
  printk(KERN_INFO "Major number %d.\n", Major);
  printk(KERN_INFO "created a dev file with\n");
  printk(KERN_INFO "'mknod /dev/%s c %d 0'.\n",
    DEVICE_NAME,Major);

  /* Map GPIO */
  if(map_peripheral(&gpio) == -1){
    printk(KERN_ALERT "Failed to map the GPIO\n");
    return -1;
  }

  /* Define input ADC connections */
  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 */
  p->addr=(uint32_t *)ioremap(CLOCK_BASE, 41*4);

  INP_GPIO(4);
  SET_GPIO_ALT(4,0);
  *(myclock.addr+28)=0x5A000000 | speed_id;

  /* Wait until clock is no longer busy (BUSY flag) */
  while(*(myclock.addr+28) & GZ_CLK_BUSY) {};

  /* Set divider to divide by 50, to reach 10MHz. */
  *(myclock.addr+29)= 0x5A000000 | (0x32 << 12) | 0;

  /* Turn the clock on */
  *(myclock.addr+28)=0x5A000010 | speed_id;

  return SUCCESS;
}
```

This function registers the new device (`/dev/chardev`), maps the GPIO address and configures the input connections. It then sets the clock to run at 500MHz and uses a divider to provide a 10MHz clock signal on GPIO pin 4.

The device file `/dev/chardev` provides a mechanism for an external program to communicate with the kernel module. The 10MHz clock signal on GPIO Pin 4 is used to drive the ADC clock for sampling. More details on setting the clock can be found in chapter 6.3 General Purpose GPIO Clocks in http://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf

The GPIO bit samples may not be synchronised with the clock. This can cause bits to be read from the same sample twice or be missed. This can be improved by setting the clock as close as possible to the frequency of the GPIO readout.

Clean up and device functions

Add the C code below to the end of the `Scope-drv.c` file.

Scope-drv.c (8/10)

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

This function is called when the kernel module is unloaded. It removes the device file and unmaps the GPIO and clock.

The implementation of four more functions need to be added to the C file, to handle connections to the device file associated with the kernel module:

```
static int device_open(struct inode *inode,
  struct file *file){
  static int counter = 0;
  if(Device_Open) return -EBUSY;
  Device_Open++;
  sprintf(msg,"Called device_open %d times\n",counter++);
  msg_Ptr = msg;
  readScope(); /* Read ADC samples into memory. */
  try_module_get(THIS_MODULE);
  return SUCCESS;
}

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

static ssize_t device_read(struct file *filp,char *buffer,
  size_t length,loff_t * offset){
  int bytes_read = 0; /* To count bytes read. */
  if(*msg_Ptr == 0) return 0;

  /* Protect against going outside the buffer. */
  while(length && buf_p<ScopeBufferStop){
    if(0!=put_user(*(buf_p++), buffer++))
      printk(KERN_INFO "Problem with copy\n");
    length--;
    bytes_read++;
  }
  return bytes_read;
}
```

Scope-drv.c (9/10)

```
static ssize_t device_write(struct file *filp,
  const char *buff, size_t len, loff_t * off) {
  printk(KERN_ALERT "This operation isn't supported.\n");
  return -EINVAL;
}
```
Scope-drv.c (10/10)

The `device_open()` function is called when the device file associated with the kernel module is opened. Opening the device file causes the ADC to be read out 10,000 times, where the results are saved in memory. The `device_release()` function is called when the device file is closed. The `device_read()` function is called when a process reads from the device file. This function returns the measurements that were made when the device file was opened. The last function `device_write()` is needed to handle the case when a process tries to write to the device file.

Building and loading the module

Create a Makefile in the same directory as the `Scope-drv.c` file. Then add

```
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
```

where the indents should be a single tab. (More information on Makefiles is given in Issue 7 of The MagPi.)

The kernel module can now be compiled on a Raspberry Pi by typing

```
make
```

Once the module has been successfully compiled, load the module by typing:

```
sudo insmod ./Scope-drv.ko
```

Then assign the device file, by typing:

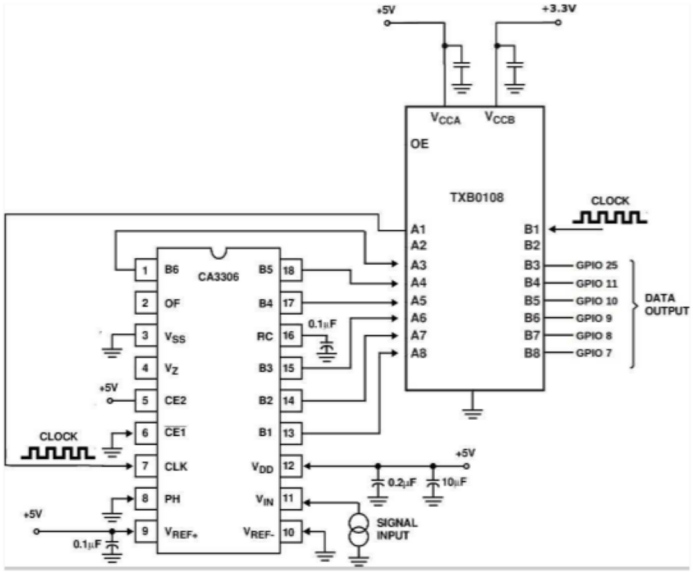
```
sudo mknod /dev/chardev c 248 0
```

Connecting the ADC

Now that the kernel module has been described, an ADC is needed to provide the input data. For this article, a CA3306 ADC from Intersil was used. This is a 6-bit 15 MSPS ADC with a parallel read out. This ADC is very cheap and fast. Many other ADC chips with parallel readout could be used, although it is necessary to check the datasheet for connection details and clock speed settings, etc..

For the selected ADC, 6-bit implies that between the ground level (0V) and the reference voltage (5V) there are 64 divisions to represent the signal. This is quite course, but is enough for simple applications.

The selected ADC operates with 5V logic, but the Raspberry Pi uses 3V3 logic. Therefore, a level converter is needed to protect the Raspberry Pi from being damaged. The simplest way to achieve this is to use a dedicated level converter, such as the TXB0108 from Texas Instruments. To ensure that stable readings are obtained from the ADC, it is recommended that a separate 5V supply is used as your VREF+ and VDD supply. This prevents voltage drops that can occur if the power supply is shared with the Raspberry Pi. However, a common ground (GND) connection should be used for the external



supply, ADC and Raspberry Pi.

Data acquisition

Once the ADC has been connected and the kernel module has been loaded, data can be read from the ADC by connecting to the device file associated with the kernel module. To connect to the kernel module, another program is needed. This program could be written in several different programming languages. For this article, C++ was chosen. Create a new file called `readout.cpp` and add the C++ given below and on the next page.

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

typedef unsigned int uint32_t;

/* To match kernel module data structure */
const int DataPointsRPi=10000;
struct DataStructRPi{
  uint32_t Buffer[DataPointsRPi];
  uint32_t time;
};
```

```
int main(){
  //Read the RPi
  struct DataStructRPi dataStruct;
  unsigned char *ScopeBufferStart;
  unsigned char *ScopeBufferStop;
  unsigned char *buf_p;

  buf_p=(unsigned char*)&dataStruct;
  ScopeBufferStart=(unsigned char*)&dataStruct;
  ScopeBufferStop=ScopeBufferStart+
    sizeof(struct DataStructRPi);

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

  // Now convert data for text output

  // Time in nano seconds
  double time=dataStruct.time/(double)DataPointsRPi;

  for(int i=0;i<DataPointsRPi;i++){
    int valueADC1=0;//ADC 1
    // Move the bits to the right position
    int tmp = dataStruct.Buffer[i] & (0b11111<<(7));
    valueADC1=tmp>>(7);
    tmp = dataStruct.Buffer[i] & (0b1<<(25));
    valueADC1+=(tmp>>(20));

    int valueADC2=0;//ADC2
    tmp = dataStruct.Buffer[i] & (0b11<<(17));
    valueADC2=tmp>>17;
    tmp=dataStruct.Buffer[i] & (0b111<<(22));
    valueADC2+=(tmp>>20);
    tmp=dataStruct.Buffer[i] & (0b1<<27);
    valueADC2+=(tmp>>22);

    // Print the values of the time and both ADCs
    std::cout<<i*time<<"\t"<<valueADC1*(5./63.)
             <<"\t"<<valueADC2*(5./63.)<<std::endl;
  }
  return 0;
}
```

This program includes the definition of the data struct that matches the version in the kernel module.

The `main()` function connects to the `/dev/chardev` device, which causes the kernel module to readout the ADC and store the values. Then the data are read from the memory buffer and copied into the local buffer within the `main()` function. Finally, the data are converted into a time in nano seconds and voltage values. The time and two voltage values are then printed in columns.

The voltage values read by the ADCs are encoded as six bits. The bits are decoded using bit shift operations and bitwise and operations.

To compile the data acquisition program, type:

```
g++ -o readout readout.cpp
```

Then run the program by typing:

```
./readout > data.txt
```

The data file can be displayed using gnuplot. Install gnuplot by typing:

```
sudo apt-get install -y gnuplot-x11
```

Then type gnuplot and enter the macro given below:

```
set key inside right top
set title "ADC readout"
set xlabel "Time   [ns]"
set ylabel "Voltage   [V]"
plot "data.txt" using 1:2 title 'ADC1' with lines, \
     "data.txt" using 1:3 title 'ADC2' with lines
```

More information on gnuplot can be found at:
http://www.gnuplot.info/

gnuplot could also be run directly from the readout program as discussed in The C Cave article in Issue 6 of The MagPi. Alternatively, gnuplot can be used within a Bash script as described in the Bash Gaffer Tape article in Issue 12 of The MagPi.