PCI Based Digitizer & Threaded Programming (Lab 8)

Introduction

Digitizer is a device which converts the level of an analog signal into an integer number which is closest to the real value of the signal in terms of ratio. There is a limit for the number of choices of this integer which is determined by the number of bits used for the digitization.

Example: We have a voltage signal whose range is [0, 1] volt, and we have a digitizer which has just 2 bits. We can have 4 different integer numbers with 2 bits. Each of these numbers will correspond to a range of voltage level, say, according to the following table:

Voltage	***Digitizer Bits***	***Integer***
0.00 - 0.25	00	0
0.25 - 0.50	01	1
0.50 - 0.75	10	2
0.75 - 1.00	11	3

This kind of conversion is performed at regular intervals. Sometimes, we call this “sampling”, and the frequency of the sampling is called “sampling frequency”. Thus, digitization is not only performed in the signal domain, but also in the time domain.

Also note that, to improve the precision of the measurement:

1) We want to increase the number of bits, so each corresponding range is small enough.
2) We want to sample at high frequency, and even more importantly, we want precise sampling clock, such that the deviation between the two sampling time is small.

In this laboratory exercise, we will build a sampling ADC using a slow ADC. To do this, we will:

- Review a minimalist (*kernel space*) PCI device driver written for this exercise,
- Study the structure of a *users space* Read Out Controller (ROC) for the board,
- Study the VMOD 12E16 ADC card manufactured by Janz,
- Study a root based analysis program which implements ring-buffer and threads.

The ADC card we will be using has 12 bit voltage resolution and a typical conversion time of 15 μ sec. This specific ADC card is usually used for slow control automation which requires a few conversion per second.

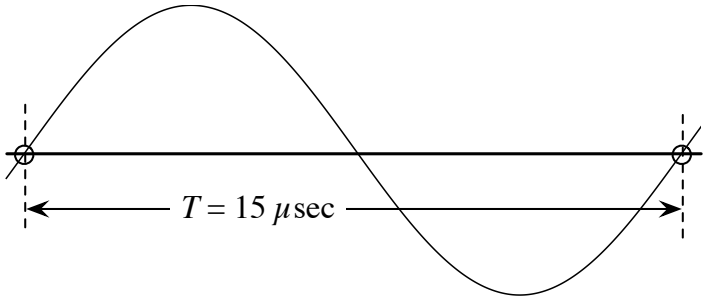
However, in this exercise, we will force its limits and make a 12-bit sampling ADC out of it, by continuously starting a conversion with a clock in sync with the PCI clock of the PC we are using.

What we will be measuring is simple: The period of a scintillator pulse simulated by an arbitrary function generator.

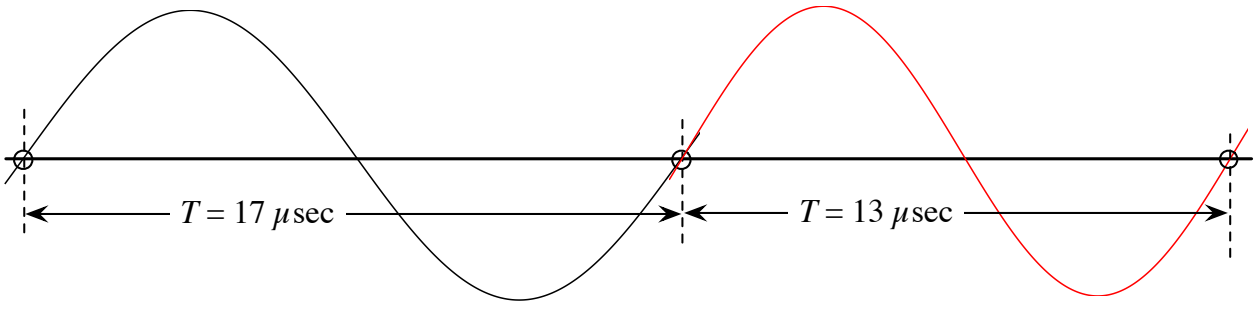
The Measurement

We are using a digitizer board from janz: VMOD 12E16 ADC shown in the figure below.

This has MODULbus form factor. It can be fitted into a PCI carrier card to be used in a regular PC. Its "vertical" resolution is 12 bit, and we know that the lower limit for the conversion time is 15 μsec. Let us assume that we sample every time a clock signal make a transition from negative to positive value as shown with "o" in the following figure.

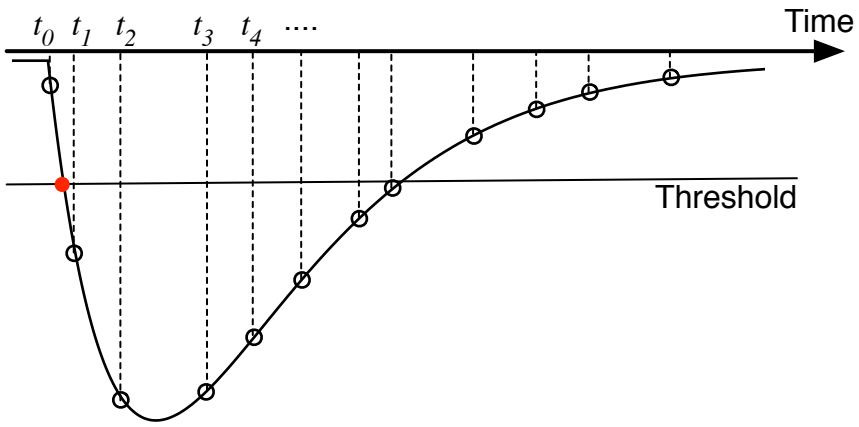


The question is how accurate is this timing? If this is accurate enough, our time resolution can be much better than the sampling period, once we interpolate our signal to find a specific time. Thus, our time resolution depends directly on the stability and reproducibility of this "clock". The sketch below is an example of changes in this "clock" signal from one period to another:



This may happen, and it is called “jitter”. Also note that, this does not mean that “sampling frequency” is changing, because by “sampling frequency” we usually mean “the average sampling frequency.”

Now consider the following signal, which are sampled by a clock with “bad” jitter:



If we want to determine the time when the level of the signal is equal to a threshold value (from high to low), first, we have to find t_0 and t_1, the time of two samplings in which that kind of transition happens, then, we interpolate the time of interest (red dot). At, this point the accuracy of sampling times become important.

Accuracy of such a time resolution depends on many other factors, such as the noise level in the analog part of the board, the quality of the cables being used, and most importantly, the quality of the sampling clock used with the digitizer.

Overall, this is called the **intrinsic time resolution** of the system. The aim of this exercise is to measure this quantity. This will put a lower limit on timing resolution of any other measurement which is using this instrument, namely “a specification” of the system.

We will measure **intrinsic time resolution** of the system built in this exercise.

Building a Linux Device Driver for the PCI Card

First, we need to control the hardware. The lowest level software for this system resides in the kernel of the OS as a “device driver.” There are certain control/status registers on the ADC card. These registers can be accessed just like a regular memory access in a C program.

As an example, let us consider a 16 bit FIFO output pointed by “`fifo_out_reg`”. The following two-line C code will read a new 16-bit word from the card and print out the results:

```
uint16_t *fifo_out_reg = get_register_address_somehow();
for (i=0;i<100;i++) printf(“%d\n”, *fifo_out_reg);
```

Note that, although the pointer is not changing, it can print 100 different values read from the same location/ register. Similarly, we can configure the hardware by writing configuration settings into the registers:

```
uint16_t *configuration_reg = get_register_address_somehow2();
```

```
*configuration_reg = ARM_THE_SYSTEM | MAKE_THIS_AND_THAT;
```

In short, once we get the registers of the device and map it to the virtual memory of the system, we can control the hardware, and gather/send data from/to it.

As you may notices all the trick lies in the hypothetical "`get_register_address_somehow()`" function. The PCI device driver we will be writing will do the following (follow `vmod.c` file shown by your tutor.)

1) Probe the card. If it does not exist, quit immediately. This is done by calling

```
ad1500_pci_dev = pci_get_device(VMOD_VENID, VMOD_DEVID, NULL);
```

function, here the first argument is the vendor ID (the ID reserved for the manufacturer "Janz"), and second argument is device ID (the ID of the PCI device -- carrier card). The device's hardware manual must provide these. The following table is coppied from the manual of our PCI carrier:

Purpose	Value	Found in
Vendor ID	`0x10b5`	CFG space register 0
Device ID	`0x9030/0x9050`	CFG space register 0
Subsystem Vendor ID	`0x13C3`	CFG space register 0x2c
Subsystem ID	`0x02??`	CFG space register 0x2c

This function returns pci device handler or NULL if there is no such card found in the PCI bus.

2) Then, we enable the pci device:

```
pci_enable_device(ad1500_pci_dev);
```

This initializes the device.

3) Every PCI device has a PCI Configuration Header (Google "PCI Configuration Header" for details.) Within this header, what is most valuable for us is Base Address Registers (BAR) in addition to the Vendor/Device ID. These registers are used to determine and allocate the type, amount and location of PCI I/O and PCI memory space that the device can use. There may be six of them for the same device. The purpose of these must be provided by the hardware manual as well. Our manual says:

PCI base address register	Local address space	Description	Size
0	-	Local configuration registers (memory mapped)	128Bytes
1	-	Local configuration registers (I/O mapped)	128Bytes
2	0	MODULbus memory space, little endian access	4KBytes
3	1	MODULbus memory space, big endian access	4KBytes
4	2	On-Board registers	4KBytes
5	3	Unused	-

So, this table tells us that, the first two BAR (BAR0 and BAR1) is used for the same purpose with two different access method. It controls the carrier card itself. We will not need them, the default should work out fine. BAR2 and BAR3 are used for two MODULbus cards carried by our PCI carrier card. Again, the purposes are identical; the difference is the endianness of the access. Intel architecture uses little endian. So, BAR2 is what we are looking for. BAR4 is used to access the on-board registers which is disabling/ enabling the interrupt signals coming from the MODULbus cards. Thus, we need BAR4 as well. We call the following functions to read the relevant BAR configuration from the PCI configuration header:

```
bar=2; // and, then bar=4;
mem_start_phys = pci_resource_start(ad1500_pci_dev, bar);
mem_end_phys   = pci_resource_end(ad1500_pci_dev, bar);
mem_length     = pci_resource_len(ad1500_pci_dev, bar);
```

These give the exact location and and length of the physical address of the registers of the ADC card which can be read/written as regular memory element. It should also be noted that, some registers must be read/written with specific data-width. Our beloved manual says that the registers on the ADC card can be accesses only by 16-bit operations. So, we have to use 16-bit (`uint16_t*`) pointers for these. Anything else will result in I/O errors (your application will receive a SIGBUS signal.)

So, this is the outline of the hidden part behind the magical "`get_register_address_somehow()`" function. But, it is half of the story. Two problems: 1) We are still in the kernel space, 2) The address we got is the physical address, but all the memory read/write operations must be via virtual addressing (beyond the scope of this exercise). Thus, 1) we need to map the physical address to the virtual address, so we use them via regular memory operations, 2) we send the virtual address to the user-space (as a user-space virtual address), so our application can use it directly. Thus the following steps:

4) We map the physical address to the virtual address:

```
mem_virtual = ioremap_nocache(mem_start_phys, mem_length);
```

That was easy! This much is sufficient for most of the device drivers you are using for regular computer hardware such as serial port, network cards, which are somewhat smarter than the ADC card we are using. However, for each conversion we have to write into a proper control register. If we do this in the kernel space we may have to call about 40,000 systems calls per second which is the only way to communicate with the kernel. And, a system call is an expensive call. So, we decided to handle this in the user space. In general, the disadvantage of this is, if the user is not careful, it is possible to bring down the whole system. But, for this case, there is no such risk, and we can perform some polling for near-realtime processing, without mush hit to the system performance.

5) We send the register address to the user space via "mmap" system call. This is done through a device file (will appear as /dev/vmod), and mmap(2). The mmap kernel fop (file operation) has the following key function call:

```
remap_pfn_range(vma,
                vma->vm_start,
                mem_start_phys >> PAGE_SHIFT,
                vma->vm_end-vma->vm_start,
                vma->vm_page_prot);
```

And in the use space, we call:

```
fd = open("/dev/vmod", O_RDWR);
vmod = (uint16_t *)mmap(NULL, 0x80, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

From this point forward, "`vmod`" can be used for the array of registers dealing with the ADC card.

Read Out Controller (ROC)

Now, we have the register address in the user-space. And, the manual of the ADC card list the functionality of the each register as follows: (See the manual for detailed descriptions.)

Control Register															0x00 - wo
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
-unused-										Amplific.		Channel Ch #			

Data Register															0x00 – ro
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
-unused-				Converted Data											

Ready Register															0x04 - ro
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Rdy	-unused-														

Interrupt Disable Register															0x06 – r/w
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
IntDis	-unused-														

Our current goal is to initiate a conversion as regularly as possible, and to read the converted digital value when the data is ready in the register... 40,000 times per second. We don't want to use interrupts for this, because interrupts are too expensive to use for each conversion. Thus, we can either use "UALARM(3)" for period sleeps (this is possible, only when the kernel can allow to sleep micro-sec precision), or by polling. We try to make this as simple as possible, so polling will be used for this exercise.

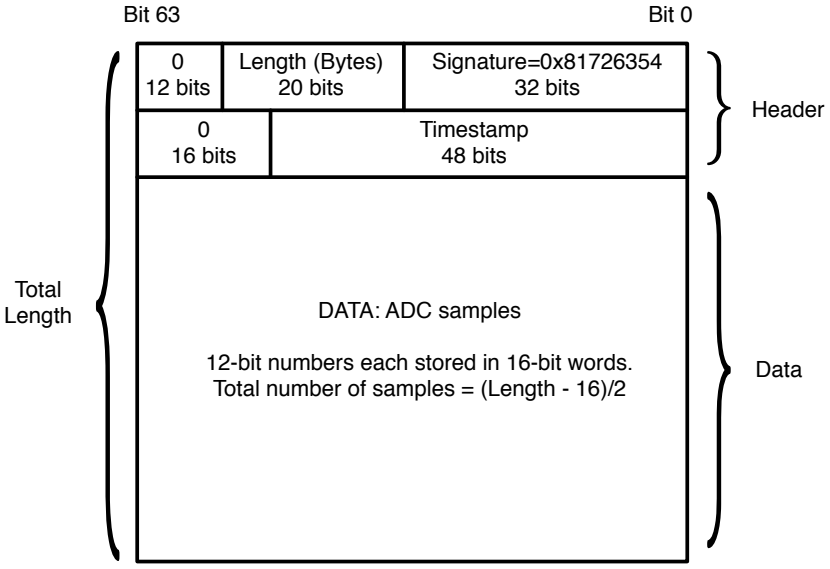
The manual says that a conversion starts whenever a write operation is performed on the Control Register (0x00) which can be accessed by vmod[0]. Bit-15 of the Ready Register (0x04) accessed by vmod[2], tells when the data is available. Then we read from the Data Register (0x00) accessed by vmod[0] again. So, the simplest ROC which will print out the converted data to the stdio will look like the following:

```
int fd;
uint16_t *vmod;
fd = open("/dev/vmod", O_RDWR);
vmod = (uint16_t *)mmap(NULL, 0x80, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
for (;;) {
   vmod[0] = 0;                // start conversion
   while (vmod[2]&0x8000) ;    // wait until the data becomes ready
   printf("%d\n", vmod[0]);    // read data and print it
}
munmap((void *)vmod, 0x80);
close(fd);
```

(Off course, you should take care of regular error handling.) Write this code immediately and test it with some signal in the 1st channel of the ADC. Then, study a little more advanced ROC which performs periodic conversions, and send data into a FIFO, to be read by the analyzer.

2nd Level Trigger and Analyzer

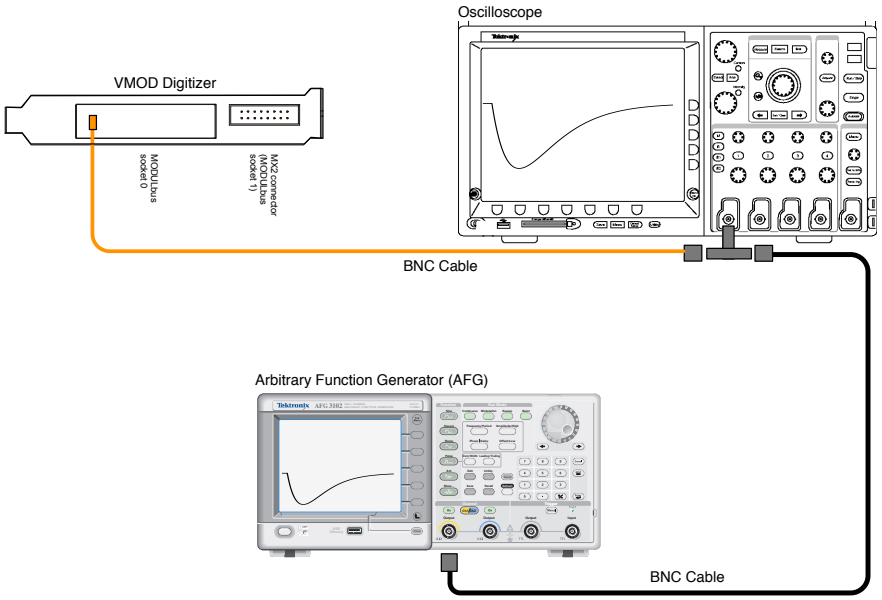
The 2nd level trigger (first level was trigger by the clock) reads samples from the FIFO, and looks for a trigger condition defined by a simple threshold value. Then it constructs an event packet and sends it to the analyzer. The event packet is described as follows:



The rest of the task is relatively simple; the analyzer interpolated the time of threshold crossing, and find the pulse time, and calculates the pulse-to-pulse time (pulse period), and creates a histogram of pulse period. If everything would be perfect, what we expect would be a dirac-delta function. Which is not the case.

Experiment

1) Construct/confirm the experimental setup according to the following sketch:



The signal generated by the AFG unit is supplied first to the oscilloscope, and passed from here to the Ch-0

input of the digitizer card. Make sure that the impedance of the oscilloscope is adjusted to be 50 Ω, and use a T-BNC to pass the signal over the oscilloscope.

2) Login to the PC and reset the lab8 directories, so all the work/changes done by the previous group are removed and a fresh copy of the files are installed. Do this by the following command:

```
tdaq@adc:~$ reset_lab8
```

"`tdaq@adc:~$`" is your prompt, "`reset_lab8`" is the command. This will create two directories in your home directory, called "vmod" and "afg." *vmod* directory contains the ROC, analyzer and the device driver. *afg* contains a utility to configure the *A*rbitrary *F*unction *G*enerator. In addition to this, you will see an editor window containing the relevant source codes for this exercise.

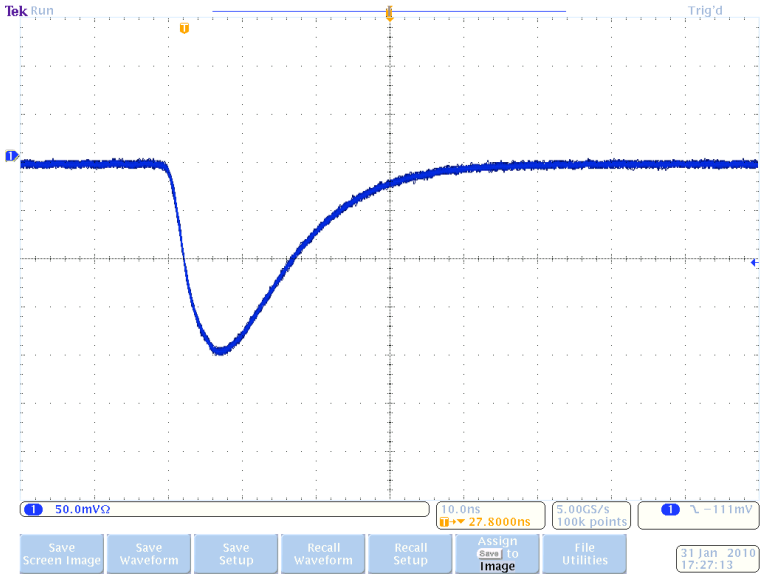
3) Change AFG function (if needed) such that you get the following function through the Ch1 output of the AFG:

$$\frac{At}{\tau} e^{-t/\tau+1}$$

where *A* = −4 volts, $\tau = 120\,\mu\text{sec}$. There is also a PULSE_FREQ definition in the function.c . The function that will be generated will be repeated with this frequency. Set PULSE_FREQ to be 400 Hz (2.5 msec period). Save function.c and give the following command:

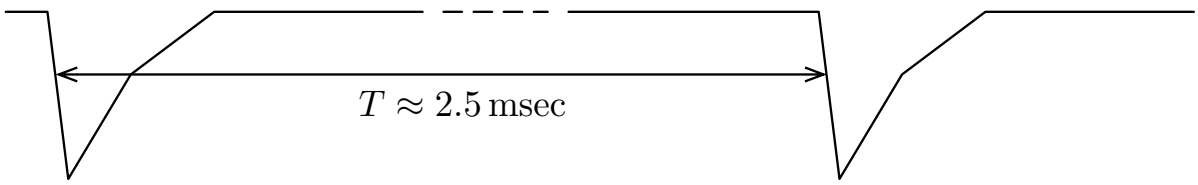
```
tdaq@adc:~$ ./setafg
```

This will setup the function generator. You can see a signal on the oscilloscope that will look like:



Arbitrary function generator is used to simulate a scintillator type pulse. It is extremely predictable and reproducible. We will use these pulses, and measure the time intervals from one pulse to the other.

4) The setup of the experiment is completed. Now, the aim is to measure the time intervals between these repetitive pulses as shown below:



We will measure these intervals tens of thousands times per second and create a histogram showing the distribution of T. The usage of the online analysis program is as follows:

```
Usages:
  ./analyze_gui [-h]                Help/usage
     [-n buffer_size]    Buffer size
     [-w width]          Pulse: width
     [-p pre_width]      Pulse: width before threshold]
     [-t threshold]      Pulse: trigger threshold
     [-b bins]           Histogram: Number of bins
     [-B bin_width]      Histogram: Bin width
     [-l bin_low]        Histogram: Lower limit
     [-r bin_high]       Histogram: Upper limit
     [-T period]         Histogram: Refresh rate
```

This program will do the following:

a) Will configuration parameters to the digitizer (threshold/width/pre_width), and ARM it to start the acquisition.

b) A thread will read the events from the PCI card and put into a local buffer.

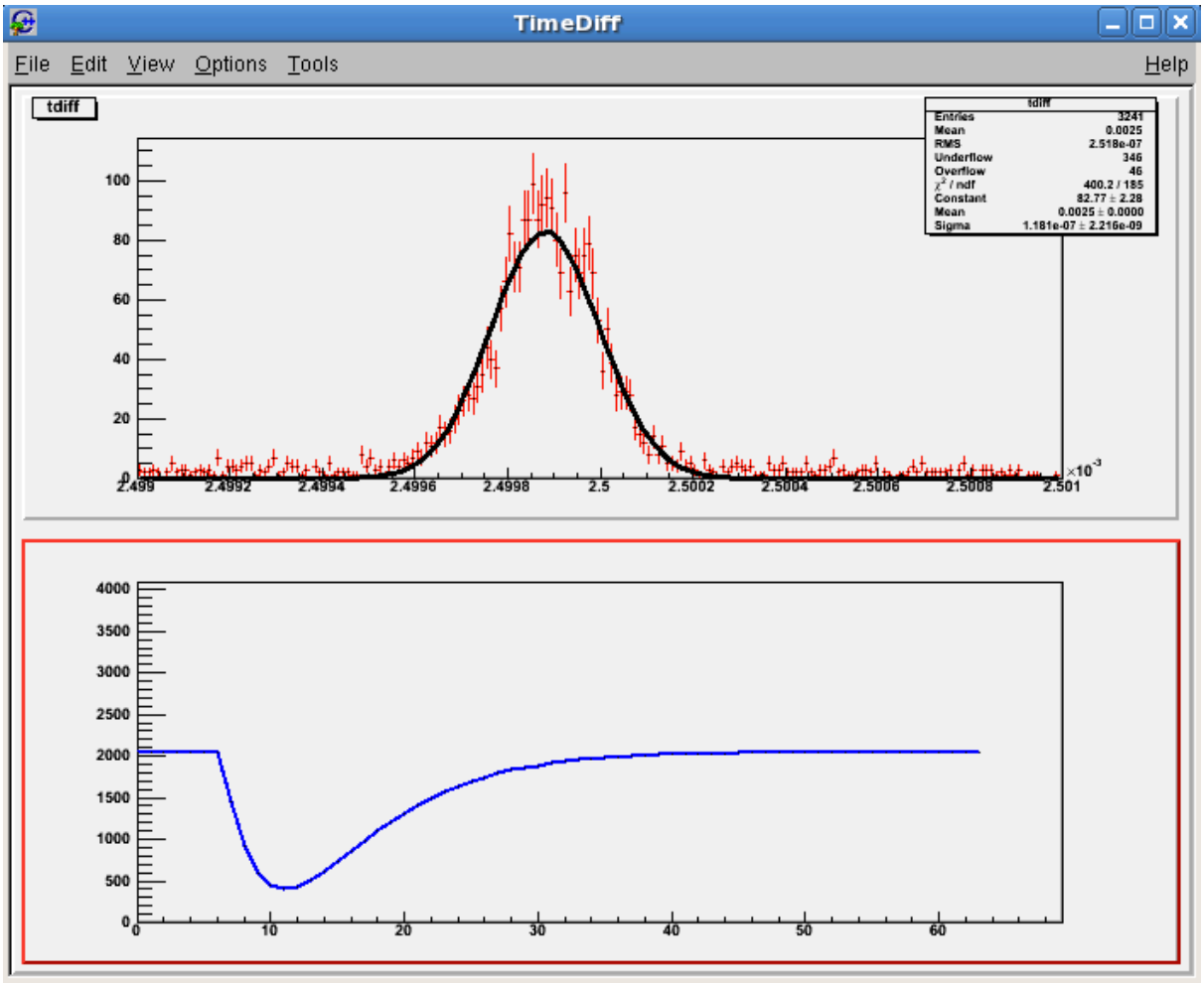
c) Another thread will read the events one by one, and it will find the crossing point of the threshold time by interpolating a given number of samples around the interested point.

d) After each time determination, its difference from the previous value is found and fed into a histogram you see on the screen.

e) It also shows the samples of the signal, just like an oscilloscope, on the screen every second. This is the raw data we get from the board. It is for diagnostic.

Start this program with: threshold=1250, bin_width=64, pre_width=8, bin_width=10e-9, low_limit=2.499e-3, high_limit=2.501e-3.

5) Turn off the AFG signal by pressing the “On” of Ch1 on the AFG, once you collect at least 10,000 events which is enough to determine the deviation of the distribution precisely. The histogram on the screen should look like:



Perform a gaussian fit on the tdiff histogram, and note the fit parameters below:

Mean-period = ...

Sigma-period = ...

We expect to get about 2.5 msec for the mean value, which what we programmed into the AFG. The question is how stable is this value? Sigma (the width of the distribution) is the quantity of interest. If it is stable, then we should always get the same value for the period. If there is a jitter/error in the period, the time of the sampling is represented by $t_i \pm \sigma_t$, where i is the sampling number. Note, however that $T_i = t_i - t_{i-1}$, thus its error is $\sigma_T = \sqrt{2}\sigma_t$. This is what we measured. Then, determine σ_t, and note this below:

Sigma-sampling-time = ...