Lab Project

Colin Boblet
Section: Wednesday

December 24, 2020

1 Introduction

This project uses a PSoC 5 and Raspberry Pi (RPi) to create a rudimentary oscilloscope. The PSoC 5 is used to convert analog signals to digital and send that data to the RPi. The RPi processes the data and prints the waveforms to a screen. The oscilloscope can be configured with various settings: trigger detection, trigger channel, trigger level, trigger slope, xscale, yscale, and sampling rate. The PSoC and RPi communicate using both I2C and USB; I2C is used for commands and potentiometer measurements, and USB is used to transfer data from both oscilloscope channels.

2 Design

The design is broken into several parts: makefile, command line processing, graphics, I2C, data collection, and data processing. The data collection is done on the PSoC. The command line processing, graphics, and data processing are done on the RPi. I2C is shared between the RPi and PSoC. Inspiration is taken from the CSE121 Canvas Files for the graphics, I2C, and USB. The high level operation of the RPi code can be seen in Figure 3.

2.1 Makefile

A makefile is used to compile and link all the necessary files for this project on the RPi. The files are main.c, i2c.c, i2c.h, draw.c, and draw.h. The i2c files contain the functions for I2C communication with the PSoC, and the draw files contain the functions necessary to display the oscilloscope on a monitor. The make file produces a single executable ”myscope” that needs to be run as a superuser.

2.2 Command Line Processing

When the program is executed on the RPi it can be started with several optional command line inputs. The inputs for this program control the oscilloscope settings: trigger detection, trigger channel, trigger level, trigger slope, xscale, yscale, and sampling rate. In the program variables for each setting are declared and initialized to their default values. In the command line each setting is prefaced by a unique string to tell the program which setting is being modified. For example to set the sampling rate the user needs to input ”-r” followed by the desired sampling rate in ksamples per second. If an invalid value is entered after one of the setting specifiers, the program will print an error message and exit. The command line inputs can be entered in any order.

An example command line input to set the sampling rate to 10 ksamples per second with no trigger is:

```
sudo ./myscope -r 10 -m free
```

An excerpt from the lab manual in Section 6 describes the full range of inputs and their defaults. A default sampling rate is not mentioned, so it is set to 10 ksamples per second.

2.3 Graphics

The graphics part of the project is implemented on the RPi in several functions in draw.h and draw.c. Graphics are implemented in 3 functions. One function sets the background color to black and creates 11 vertical lines and 6 horizontal lines for the x and y divisions. The second function prints the data waveforms. The amount of data points required is calculated from the sampling rate and xscale values. The xscale value gives the amount of time displayed on the screen while the sampling rate gives the amount of data points per second. The third function prints the oscilloscope settings to the top left corner of the screen.

2.4 I2C

The I2C implementation is shared between the RPi and PSoC. The RPi is set as the master, and the PSoC is set as the slave. There are three communications needed between the RPi and PSoC. The first communication is the start command. The start command defined as 0xff and will set the PSoC to start sending data over USB. A sampling rate command sets the sampling rate and is defined as a byte with the most significant bit as 1 but not 0xff. Both the start command and sampling rate are sent to the PSoC before the RPi reads from the USB buffer. The third message is the PSoC sending potentiometer values to the RPi and, as it is the only message sent to the RPi, requires no encoding. The PSoC sends the potentiometer values every time it loads data into the USB buffer. Data is collected from the potentiometers using a Sigma Delta ADC and an analog mux. The the mux is used to toggle the Sigma Delta ADC input between the 2 potentiometers.

2.5 Data Collection

Data is collected on the PSoC using 2 SAR ADCs and 2 DMA channels. The DMAs are driven with a variable frequency clock. The clock's frequency is set with a clock divider determined by the sample rate. Both sets of ADCs and DMAs operate identically except that they sample different channels. Each DMA channel has 2 TDs that each transfer 64 bytes to a ping pong buffer and trigger an interrupt. In the interrupt flags are set to tell the PSoC which buffer is ready to be transferred to the RPi. In the main loop, the PSoC transfers data with USB 64 byte bulk transfers when the flag is set.

2.6 Data Processing

On the RPi, once the command line inputs have been processed and I2C commands have been sent, a forever loop reads the USB buffers and potentiometer values. If mode is set to free, the data from the USB buffers are automatically stored until the correct data count has been reached. The data count is calculated using the xscale and sampling rate values to determine how many data points need to be displayed on the screen. When the mode is set to trigger, data is discarded until the trigger condition is met. Once the trigger condition is met data is stored until the data count has been reached. The first time through this loop or after data has been drawn to the screen the USB buffer is cleared to discard any old data. Each channel has its data counted separately, and they both need to reach the data count before any data can be printed. This can be seen as a block diagram in Figure 4.

3 Diagrams

This section contains all the diagrams for this Lab Project. The wiring diagram is shown in Figure 1 and the TopDesign is shown in Figure 2. The block diagram shown in Figure 3 is the high level block diagram for the RPi's main. The block diagram shown in Figure 4 is the operation of the RPi's forever loop and is the expanded version of Figure 3's Trigger Detection process.

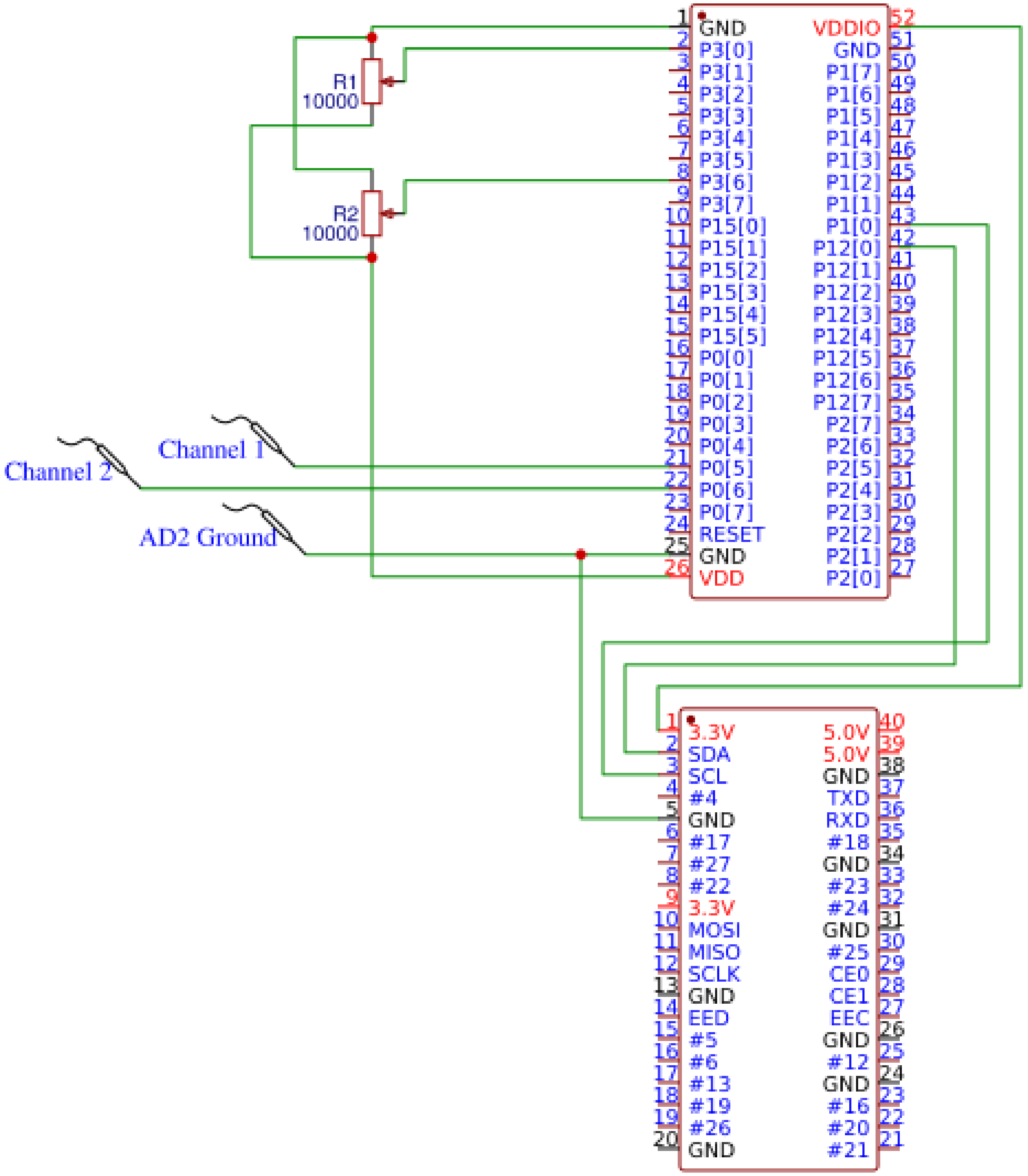

Figure 1: This is the wiring diagram for the Lab Project. The PSoC is shown above and the RPi is below. The PSoC and RPi are also connected with a USB cable.

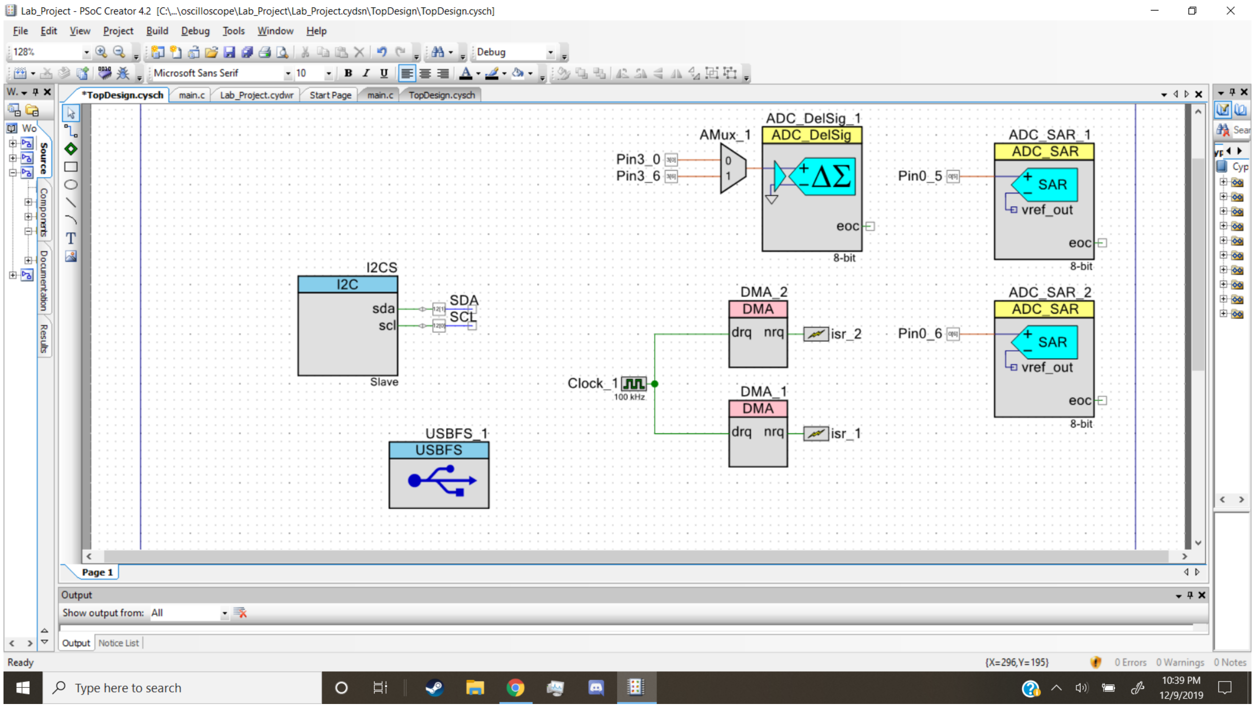

Figure 2: This is the TopDesign on the PSoC

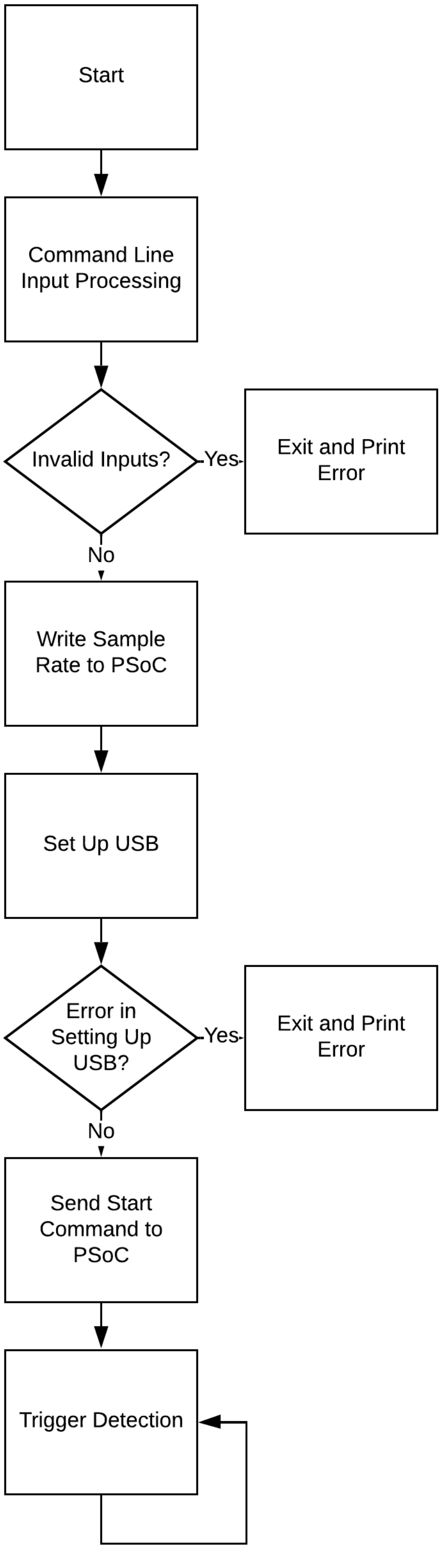

Figure 3: This is the top level block diagram for the RPi main. The Trigger Detection block is expanded in Figure 4.

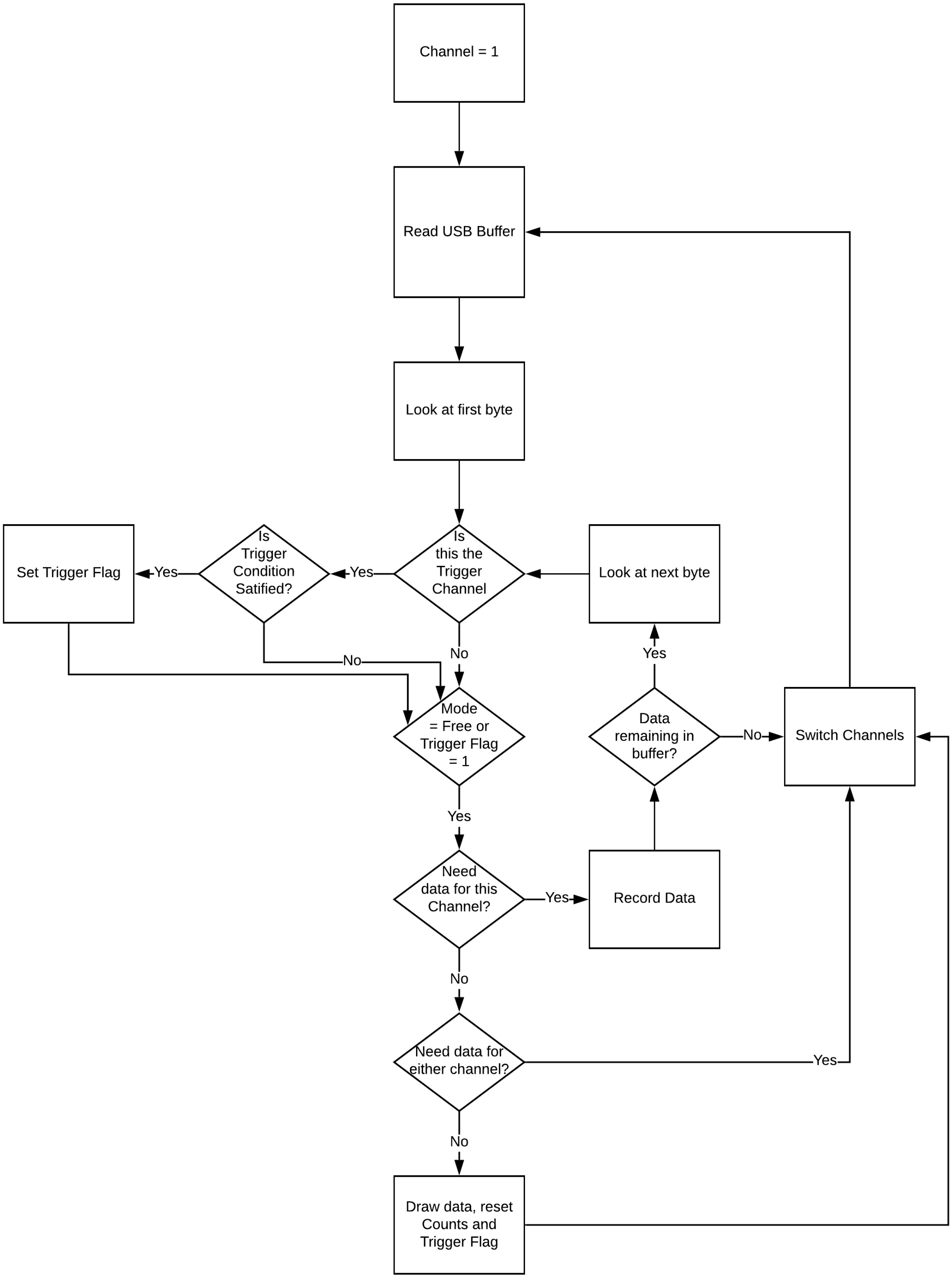
Channel = 1
Read USB Buffer
Look at first byte
Is this the Trigger Channel
Yes
Is Trigger Condition Satified?
Yes
Set Trigger Flag
Look at next byte
No
No
Mode = Free or Trigger Flag = 1
Yes
Need data for this Channel?
Yes
Record Data
Data remaining in buffer?
Yes
No
Switch Channels
No
Need data for either channel?
Yes
No
Draw data, reset Counts and Trigger Flag

Figure 4: This is the block diagram for the Trigger Detection block in Figure 3.

4 Testing

The system was created and tested in several parts: graphics, I2C, command line inputs, USB transfer, and trigger detection.

The graphics were created first and were based off examples in the CSE121 Canvas Files. The background was created first as it only required using given functions and drawing 17 predetermined lines. The data drawing was created next and tested with predefined data. Finally, printing the settings to the screen required only a few print functions and some if statements.

I2C was created next and is again based off examples in the CSE121 Canvas Files. Desoldering R15 was the hardest part of implementing I2C. After R15 was desoldered I2C only required a few lines of code to add the necessary functions to the PSoC and RPi, as I have worked with I2C and RPi's before.

Implementing command line inputs was simple as well. The solution I decided on was a long chain of if statements to cover all possible inputs. The amount of function inputs required to make this a function in a separate file made me decide to keep all the code in main. The only error I encountered in my testing was I forgot to check if a variable was a null pointer before I used it in a function.

USB transfer was almost a simple copy and paste from Lab 5. However, the clock driving the DMA needed to be variable and I used Sigma Delta ADCs for Lab 5.

Data processing and trigger detection caused the most errors. I initially had a separate loop for when the RPi was ready to collect data. The purpose of this loop was to eliminate extraneous if statements that could cause delay and data loss. However, the logic required made debugging more difficult and getting trigger detection working was a priority before data loss could be considered. Therefore, trigger detection and data collection was implemented in a single loop. This immediately proved effective, but the delay from drawing data caused data from the PSoC to become stale by the time the RPi returned from drawing. Therefore, the first data collected after drawing is discarded.

5 Conclusion

The oscilloscope accomplished all of the required functionality except for data loss at 100 ksamples per second. This project has obvious practical applications for being used as an oscilloscope. It also provides valuable experience in working with I2C and graphics.

However, there are some improvements that could be made. First trigger detection is not implemented across bulk transfers. The comparison variable is reset after each 64 bytes are processed. This is to prevent data loss from corrupting trigger detection. Practically this should not affect performance noticeably, but could result in several missed trigger events. The command line input processing could also be improved in 3 ways. Currently, when the user inputs an invalid value the program prints a single error message and exits. The error message should also tell the user what the acceptable inputs are and all error messages should be printed not just the first error found. The error checking only checks values following specified strings (ex. "-m"). If the user enters a random string before any of the specified strings an error should be raised. Additionally moving the waves up and down is not practically useful. The offset value should be printed to the screen. The offset also does not move the wave the same amount of pixels for each yscale setting. At higher yscales, the wave moves less, and at lower yscale values the wave moves much more. This would be an relatively easy change as it would only require adding in the offset after the data has been scaled. Lastly, the x and y divisions should have additional sub divisions and labels to facilitate reading an accurate voltage or frequency.

6 Command Line Inputs

-m <mode>

<mode> can be either free or trigger. In the free-run mode, the scope displays a free-running signal that may not be stable. The trigger mode aligns the sampled signal using a trigger level and trigger slope, so that repeating waveforms will appear stable on the screen. By default, the mode is set to trigger.

-t <trigger-level >

<trigger-level> specifies the trigger level in millivolts and can range from 0 to 5000 in steps of 100. By default, the trigger level is set to 2500 (that is, 2.5V).

-s <trigger-slope>

<trigger-slope> can be either pos or neg. Positive slope (pos) indicates that the time sweep of the waveform should start on a rising edge when the signal crosses the trigger level. Negative slope (neg) indicates that the time sweep of the waveform should start on a falling edge when the signal crosses the trigger level.By default, the trigger-level is set to pos.

-r <sample-rate>

This parameter defines the sampling rate for the ADCs of the two channels, in kilosamples per second. The sampling rate can be 1, 10, 20, 50 and 100 ksamples per second.

-c <trigger-channel>

<trigger-channel> is the channel (1 or 2) that is the source of the trigger. By default, channel 1 is selected as the trigger channel.

-x <xscale>

<xscale> defines the horizontal scale of the waveform display in microseconds. Its value can be 100, 500, 1000, 2000, 5000, 10000, 50000 or 100000. Its default value is 1000 (1 millisecond per division).

-y <yscale>

<yscale> defines the vertical scale of the waveform display in millivolts per division. Its value can be 100, 500, 1000, 2000 or 2500. Its default value is 1000 (1V per division).