21-SEP-2016; Roderick Young

Learnings About the Hantek 6022BE USB Oscilloscope

- The scope samples analog data at various sample rates.
- The samples returned are always 8 bits. They are unsigned values, with 0 corresponding to the most negative voltage, and 255 corresponding to the most positive. If these voltages are exceeded, the sample clamps at 0 or 255 as appropriate.
- There are two channels of data. It is not possible to read only one channel.
- The scope sends small buffers of data (probably about 1k per channel) to the driver. The driver assembles these buffers into longer buffers.
- At the highest sampling rate of 48 MSps, the scope and driver together cannot keep up for more than one small buffer, making the data size 1016 bytes only. At slower sampling rates, the buffers can be longer.
- Although there are 8 input voltage settings discussed in the SDK documentation, there is no variable gain input amplifier. Actually, there are just two real voltage settings, with a 10:1 attenuator internal to the scope that can be switched in. So the real native voltage settings are just ±5.0 V full scale (1V/div if there are 10 divisions) and ±0.5 V full scale (100 mV/div), corresponding to samples of 0 to 255. All other settings are derived by mathematically scaling one of these two. For example, to display 500 mV/div (±2.5V full scale), data is simply multiplied by 2, giving a full-scale precision of only 128 levels instead of 256.
- With the timebases set at 2 µS and faster, the driver takes significantly longer to return a sample buffer. It may be a half-second to a second of delay - definitely noticeable. Possibly, the driver is doing some sort of sample interpolation at these speeds.

Learnings About the Hantek SDK

Hantek6022BEX861.sys
Hantek6022BEX862.sys
These two driver files must be installed for the interface to the scope to work. An easy way to put these in the right place is to simply install the stock Hantek software. For 64-bit architectures, there are corresponding driver files with slightly different names.

HTMarch.dll
HTMarch.lib
This library interfaces with the scope, making settings, and retrieving data buffers. In theory, you could call functions from HTMarch.dll directly, but it's easier to statically link your program to HTMarch.lib, then place HTMarch.dll in the same directory as your executable program.

HTDisplayDll.dll
HTDisplayDll.lib
This library handles the drawing functions. You do not absolutely need this library if you are

doing your own drawing functions, but the libary is very convenient if you happen to want to put up an oscilloscope style display. Again, it is easiest to link to HTDisplayDll.lib, and then place HTDisplayDll.dll in the same directory as your program. RichardK on the web reverse engineered the drawing functions, and put out a source file, which you could use to avoid HTDisplayDll entirely.

Specific Instructions for Visual C++ 2010:

1. Create a new project. It can be a win32 Windows App, or Console App if no graphics are needed.

2. Put the files HTMarch.h, HTDisplayDll.h, HTMarch.lib and HTDisplayDll.lib files into the same directory as the source .cpp and .h files. This puts the header files in a place that will be visible to the compiler, and the library files in a place that will be visible to the linker. The .lib files will be statically linked to your program, and functions can be called in the regular way. These libraries handle the interface to the dynamically linked libraries (.dll files), automatically.

3. On the solution explorer pane, right-click on the project name, and select Properties. Select Linker -> Input -> Additional Dependencies, and type in the file names HTMarch.lib and HTDisplayDll.lib (one per line). Check the box that says to inherit default libraries. In my BasicScope program, I use Microsoft common controls, so need to add a line with comctl32.lib also.

4. Put the files HTMarch.dll and HTDisplayDll.dll in the same directory as the output (executable) file. If you are doing a Debug compile, the usual place is the Debug directory within the Project directory.

5. Here is some sample code for a console app:

```
// Example of calling Hantek library
//
#include "stdafx.h"  // this is from visual c++ otherwise <stdio.h>

// windows.h contains some definitions required by Hantek .h files.
// a windows app automatically includes windows.h in the standard header,
// but a console app needs an explicit #include
#include <windows.h>

// These are the Hantek header files, either from the SDK, or the enhanced
// ones from RichardK.
#include "HTMarch.h"
#include "HTDisplayDll.h"
```

```
int _tmain(int argc, _TCHAR* argv[])
{
	DeviceIndex = 0;
	if (dsoOpenDevice(DeviceIndex))
		printf("Opened Hantek Device\n");
	else
		printf("Failed to open Hantek Device\n");
	return 0;
}
```

6. The general sequence for taking a trace and displaying it is:
   a. Use dsoOpenDevice() to make sure the scope is there. If the device is there, it might not be a 6022BE, so it's worth checking further. A 6022BL can be supported with very little effort just by selecting the scope instead of the logic analyzer with dsoChooseDevice().
   b. Use dsoCalibrate() at the time division and voltage for which the measurement is intended. My experience is that changing the voltage and sampling frequency makes no difference to the calibration, but never know. Both scope probes must be grounded for this operation. Use dsoSetCalData() to store the calibration data back into the driver. Calibration is optional. It is possible to simply retrieve previously stored values by calling dsoGetCalData().
   c. Use dsoSetVoltDIV() twice, once to set the voltage range for each channel. One might think that this is unnecessary, as the Voltages are again specified in dsoReadHardData() below, but in fact, the values supplied to dsoReadHardData() appear to be ignored. Actually, necessary but not sufficient - low volt ranges auto amplification
   d. Use dsoSetTimeDIV() to set the sampling rate. It is imperative to use this call to set the sampling rate, as the value passed to dsoReadHardData() is ignored. Or is it?
   e. Use dsoReadHardData(). The calibration data is what was previously obtained from dsoCalibrate(). As far as I can see, nCH1VoltDIV, nCH2VoltDIV, nTimeDIV, and nHTrigPos are ignored by the function.
   f. Use HTDrawGrid() to draw a grid into a window.
   g. Use HTDrawWaveInYT() to plot the data into a window.