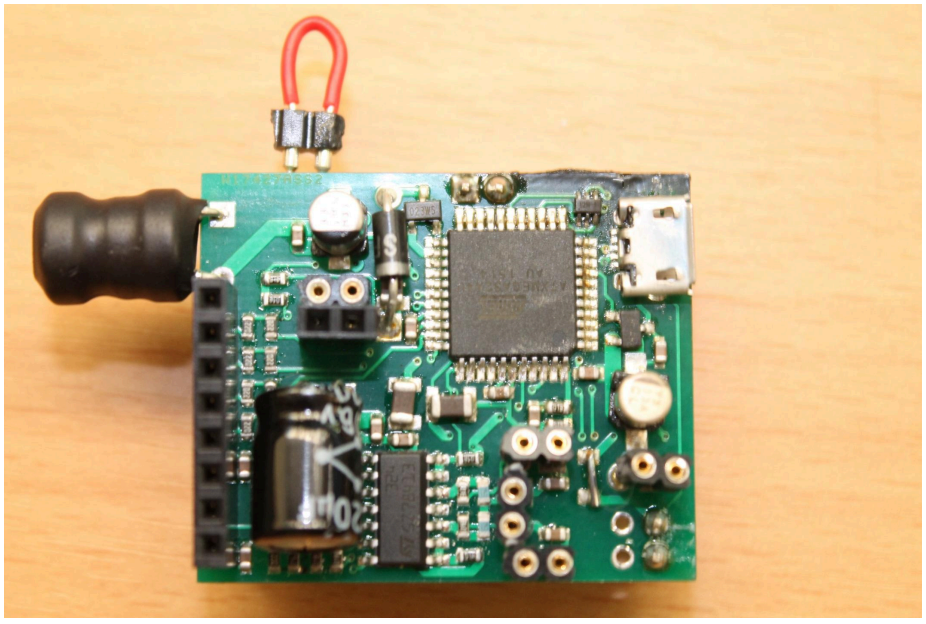
Labrador Development Documentation

Chris Esposito <admin@espotek.com>

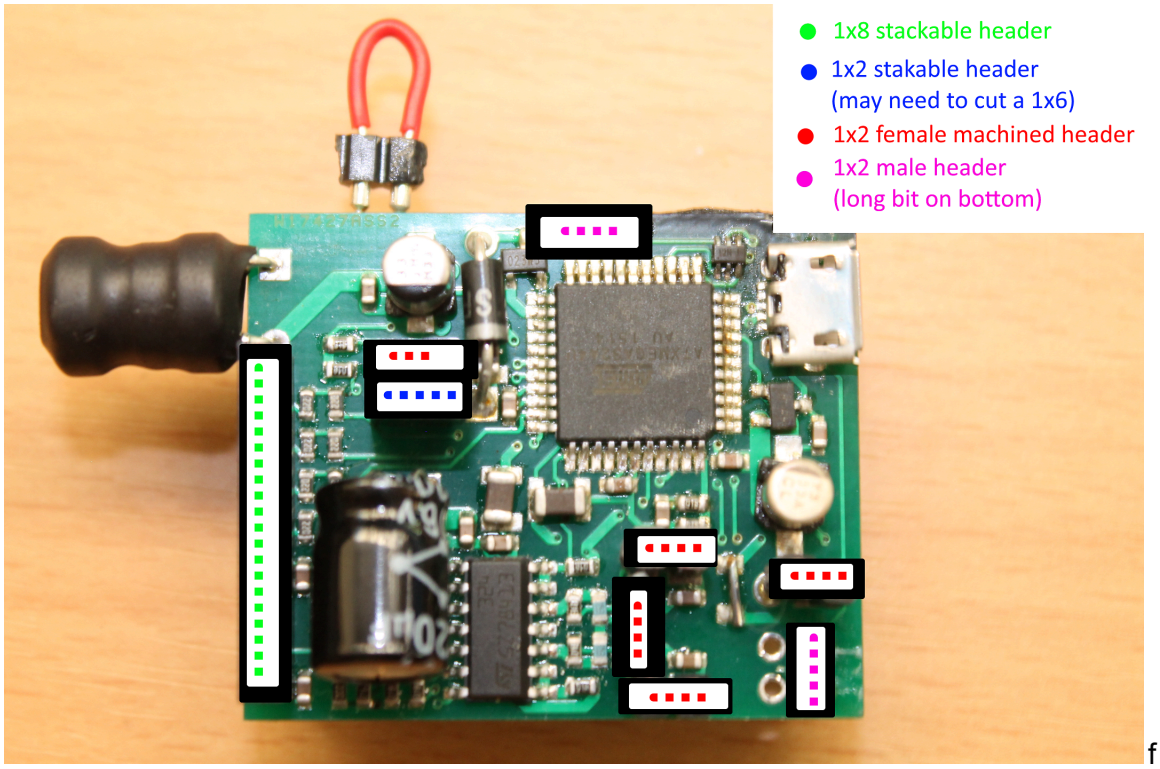
Assembling

Below are two high resolution shots that should help with the assembly!
Make sure to check both diagrams before soldering.

Unlabelled board:



Labelled diagram:



f

Building the Software

The sources are available at github.com/espotek/labrador.

The software should be built using Qt 5.7. The project file (labrador.pro) will automatically link the correct libraries and drivers for the different OSes - as well as compile slightly different versions to deal with platform differences.
The following instructions relate to building the software interface in the Qt Creator IDE. Note that both the Qt library and Qt Creator are free.
Firstly, ensure:

- Shadow Build is disabled (in the Projects menu on the left side, looks like a spanner)
- Compiler used is msvcc2013 x64 (Windows), gcc x64 (Linux), Clang x64 (Mac). Note that this means installing Visual Studio 2013 (Windows) or XCode 7.2.1 (Mac).
- If on Linux, ensure that it runs in terminal. It currently requires root access to access USB.
- If on Mac, XCode needs to be mounted when you compile. The installer for Qt Creator will tell you to use XCode 5. Do not use XCode 5.

Then follow the magic sequence in the build menu:

- Clean
- Run QMake
- Build

Hit the run button (green play button in the bottom left corner) and watch the software spring to life!
If it isn't, try a different USB port. One that isn't shared with an interrupt/iso device (keyboard, mouse etc.)

USB Interface

Control Endpoint (0x00)

The control endpoint handles both standard USB requests (e.g. requests to handle enumeration), as well as Labrador-specific (vendor) requests.

Board-side, both types are handled in ASF/common/services/usb/udc/udc.c, in usb_reqstd() and udc_reqvend(), respectively.
Desktop-side, the standard requests are handled by the driver and the vendor requests are handled by usbSendControl(), in winusbdriver.cpp.

This documentation covers only the vendor requests, since no standard requests are sent after initialisation. If you're interested in learning about the standard USB calls, I recommend USB Complete by Jan Axelson. USBPcap is also helpful for understanding this.

Note that wValue, wIndex and wLength are all 16 bits long and unsigned. Where their maximum value is not specified, assume it's 65535.

Below are all of the commands:

Command 0xa0 - Debug

This request tells the board to break when in debug mode by causing it to hit a breakpoint.
The setup packet sent looks like below:

bmRequestType	bRequest	wValue	wIndex	wLength
0x40	0xa0	0	0	0

No data packet is sent following the setup stage.

In winusbdriver.cpp, avrDebug(), causes this call to be sent.

Command 0xa1 - Signal Gen CH1

This request sends the waveform data to signal gen CH1, as well as sets up the timing.
The setup packet sent looks like below:

bmRequestType	bRequest	wValue	wIndex	wLength
0x40	0xa1	PER	CLKDIV	LEN

A data transaction of length LEN bytes is sent following the setup stage, containing the raw sample values as unsigned chars. LEN should not be higher than 512.

The timing of Labrador's signal gen is controlled by an internal timer. When the timer overflows, the next waveform sample is sent to the DAC. CLKDIV and PER determine the period of this timer.

CLKDIV sets the timer's clock prescaler according to the following table:

CLKDIV	0	1	2	3	4	5	6
Prescaler	1	2	4	8	64	256	1024

PER sets its period.

Example:
PER = 4000, CLKDIV = 3, LEN = 128.

The Labrador's base clock is 24MHz.
CLKDIV is 3, giving the timer a prescaler value of 8. Therefore, the timer's input clock runs at 3MHz.
PER is 4000, meaning samples are updated at a rate of 750Hz.
The whole waveform is 128 samples long, giving an overall frequency of 5.86Hz.

In winusbdriver.cpp, setFunctionGen() controls this call (among others) and performs the maths necessary to back-calculate CLKDIV, PER and LEN.

Command 0xa2 - Signal Gen CH2

This command works exactly the same way as 0xa1, but controls CH2 instead.
While

bmRequestType	**bRequest**	**wValue**	**wIndex**	**wLength**
0x40	0xa2	PER	CLKDIV	LEN

Command 0xa3 - PSU Voltage Control

This request sends the PSU output level to the device.
The setup packet sent looks like below:

bmRequestType	**bRequest**	**wValue**	**wIndex**	**wLength**
0x40	0xa3	VOUT	0	0

Note that VOUT should not be above 106 or below 21.

VOUT is not the voltage level itself, but is calculated in setPsu() according to the formula:
VOUT = <<Voltage Level>>/18.15 * 128.

Example: Set the power supply to 10V.
*To set the power supply to 10V, you'd send a packet with VOUT = 10/18.15*128 = 70.52 ~= 71.*

This command is sent periodically (if necessary) by psuTick().

Command 0xa4 - Signal Gen Triple

This request sends enables or disables op-amps that can boost the DAC output above 3.2V.
The setup packet sent looks like below:

bmRequestType	**bRequest**	**wValue**	**wIndex**	**wLength**
0x40	0xa4	TRIP	0	0

The signal generator is driven by an on-chip DAC that is limited to the range of 0.15V - 3.2V. To extend the range beyond this level, external amplifiers are used. 0xa4 sets these amplifiers to be either unity gain or 3x gain.

Bit 0 of TRIP controls the amplifier on the output of signal gen CH1. A 1 sets the gain to 3x, and a 0 sets it to 1x (unity gain buffer).
Bit 1 of TRIP controls the amplifier on the output of signal gen CH2 in the same way.

setFunctionGen() controls this call, enabling it when the user requests a voltage above 3.2V.

Command 0xa5 - Mode and Gain

This request changes the on-board mode and ADC gain variables.
The setup packet sent looks like below:

bmRequestType	**bRequest**	**wValue**	**wIndex**	**wLength**
0x40	0xa5	MODE	GAIN	0

MODE turns devices on and off in order to maximise bandwidth and share a single RAM buffer, according to the following table:

MODE	0	1	2	3	4	5	6	7
Oscilloscope CH1 (375ksps)	**X**	**X**	**X**					
Oscilloscope CH2 (375ksps)			**X**					
Oscilloscope CH1 (750ksps)							**X**	

Logic Analyzer CH1 (375ksps)		**X**		**X**	**X**			
Logic Analyzer CH2 (375ksps)					**X**			
Multimeter (375ksps, 12-bit)								**X**

An **X** in the box means that the device is enabled when MODE is equal to the value in the top row. For example, if MODE = 2, the oscilloscope CH1 and CH2 will be enabled at 375ksps, and all other devices in the table will be turned off. Devices not in the table, such as signal gen and power supply, are unaffected by MODE and always available.

GAIN controls a programmable gain amplifier (PGA) on the input to the oscilloscope ADC.
The lower byte of GAIN controls the gain on CH1's PGA, and the upper byte controls CH2's, according to the following table:

GAIN	0x1C	0x00	0x04	0x08	0x0c	0x10	0x14	0x18
PGA gain	1/2	1	2	4	8	16	32	64

When in mode 7, the multimeter uses CH1's PGA.

Note that both lower and upper byte should always be set to the same value. Separate gain on CH1 and CH2 is a deprecated feature and may have errors.
Also, perhaps foolishly, this gain value is stored and used later to convert raw samples to voltage levels, when reading the oscilloscope data in.

This command is sent by both setGain() and setDeviceMode().

Command 0xa6 - PSU Voltage Control

This turns digital outputs on and off.
The setup packet sent looks like below:

bmRequestType	**bRequest**	**wValue**	**wIndex**	**wLength**
0x40	0xa6	MASK	0	0

The lower 4 bits of MASK control the digital outputs. A 1 in bit *n* will turn output *n* on (3.3V), a 0 turns it off.

This is controlled by newDig().

Command 0xa7 - Reset

This command resets the device. Simple enough. :)
The setup packet sent looks like below:

bmRequestType	bRequest	wValue	wIndex	wLength
0x40	0xa7	0	0	0

Isochronous IN Endpoint (0x83)

This is the endpoint that handles all streaming data: the oscilloscope, logic analyzer and multimeter.
It reserves 1023 bytes per frame of bandwidth (100% of USB FS!!), but each packet sent is only 750 bytes long.

MODE	0	1	2	3	4	5	6	7
Oscilloscope CH1 (375ksps)	**X**	**X**	**X**					
Oscilloscope CH2 (375ksps)			**X**					
Oscilloscope CH1 (750ksps)							**X**	
Logic Analyzer CH1 (375ksps)		**X**		**X**	**X**			
Logic Analyzer CH2 (375ksps)					**X**			
Multimeter (375ksps, 12-bit)								**X**

In modes 1 - 4, the first 375 bytes of each packet contains the data streamed from the device listed closest to the top of the page; the second 375 bytes contain the data from the other device.

For example, in mode 1, the first 375 bytes contain Oscilloscope CH1's data, and the second 375 bytes contain Logic Analyzer CH1's.

In modes 6 and 7, the whole packet is dedicated to the Oscilloscope/Multimeter's data.

Oscilloscope Data

The oscilloscope data consists of a stream of sequential samples. Each sample is a signed 8-bit int that needs to be decoded PC-side to discover which voltage it represents (see isoDriver::analogConvert(); this function essentially reverses the effects of the frontend and PGA digitally). The samples are streamed at a constant data rate - either 375ksps or 750ksps depending on the mode.

Logic Analyzer Data

The logic analyzer data is similar to the osciloscope's, but they are streamed at 3Msps and each sample is a single bit long.

Multimeter Data

The multimeter data is similar to the oscilloscope's, but each sample is 12 bits long. Each of these 12-bit samples are stored in a 16-bit int (possibly with four leading zeroes?). The top four bits from each sample should be discarded and ignored.