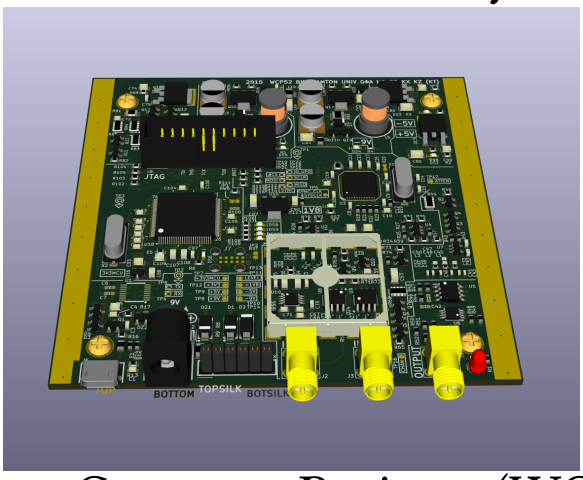
USB Gain/Phase Analyzer



Watson Capstone Projects (WCP52)

Christopher M. Pavlina, EE
Kenneth M. Zach, CoE
Harrison B. Owens, CoE

Faculty Advisor: Kyle J. Temkin

May 1, 2015
Revision: -

Submitted in partial fulfillment of the requirements of EECE 487 in the Spring Semester of 2015.

Thomas J. Watson School of Engineering and Applied Science
State University of New York at Binghamton

Executive Summary

In the current market, electronics lab equipment that can analyze a frequency response by gain and phase (a vector analyzer) is very expensive, often reaching into the tens of thousands of dollars. They are large, heavy instruments, and are not portable between home, class and a lab. A direct result of this high cost is that they are generally unobtainable for teachers to use in classrooms and labs to enhance their education techniques and for students to use as a hands-on tool for circuit analysis projects. Due to their lack of portability, those teachers who do have access to a vector analyzer can not use its vast functionality for demonstrations of a circuit's behavior in front of a classroom or in a lab with many students.

It is our goal to remedy this situation by creating a low cost, portable device that can provide an analysis of gain and phase in a graph.

This gain/phase analyzer can be produced and sold for approximately $100. The device is portable and can be carried between home, work, labs, classrooms, etc and is very easy to set up with a USB connection to a computer and a power outlet.

Additionally, our device is an open source project, with all hardware and software source available online. Thus, any capable students or engineers will have ability to learn how it works and modify the design any way they choose. This can be a great way for students to gain some hands-on experience in learning some advanced circuit design techniques and microcontroller programming. Additionally, one can apply onefis own creativity and ingenuity to improve the design of this project; making a better device for future teachers and students.

Table of Contents

1 Problem Definition **3**
- 1.1 Problem Scope . . . 3
- 1.2 Technical Review . . . 3
- 1.3 Design Requirements . . . 3
  - 1.3.1 Context-Level Constraints . . . 3
  - 1.3.2 System-Level Constraints . . . 4

2 Design Description **5**
- 2.1 Overview . . . 5
- 2.2 Detailed Description . . . 6
  - 2.2.1 Synthesizer . . . 6
  - 2.2.2 Output Signal Path . . . 7
  - 2.2.3 Input Signal Path . . . 7
  - 2.2.4 Phase Measurement . . . 8
  - 2.2.5 Microprocessor . . . 8
  - 2.2.6 Analysis . . . 8
- 2.3 Use . . . 8

3 Implementation **8**
- 3.1 Hardware Prototype PCBs . . . 9
- 3.2 Microprocessor Software Prototyping . . . 10
- 3.3 PC Software . . . 10

4 Evaluation **10**
- 4.1 Overview . . . 10
- 4.2 Testing and Results . . . 11
- 4.3 Assessment . . . 14

5 Schedule and Budget **14**

6 Future Plans **15**

7 References **16**

A Project Requirements **17**

B Test Procedures **20**

C Microprocessor Code **24**

D PC Code **43**

E CAD Drawings **52**

F PCB Layouts **66**
- F.1 Layer 1 . . . 67
- F.2 Layer 2 . . . 68
- F.3 Layer 3 . . . 69
- F.4 Layer 4 . . . 70

1 Problem Definition

Tools such as vector analyzers are heavy and expensive, costing tens of thousands of dollars. There is need of a tool for students and teachers to help teach electronics and similar courses. Currently, there are no feasibly affordable tools to show the frequency response and phase of a circuit, amplifier, or control loop, and the learning experience is hindered by this gap.

1.1 Problem Scope

This project will produce a portable frequency response analyzer. It will communicate data with a computer through USB in order to form a Bode plot of the frequency and phase response of the circuit.

1.2 Technical Review

Proper circuit analysis is a fundamental practice in the field of engineering, since it is necessary for every electronic device on the market. As a result, it is the goal of every engineering institution to give students a strong basis in circuit design and analysis.

Currently, this is done mainly through lecture and labs. A professor stands in front of a classroom and delivers a lecture based on PowerPoint slides that they have created. It is expected that, from this, students grasp the concepts necessary to make them proficient in working with electronics. Although this may be a seemingly efficient way to broadcast the fundamentals to many people, it must be augmented with practice.

Then, there is the lab section of the class. Students are asked to put together a circuit and then observe it using a measuring device. Unfortunately, few such devices are available for frequency-domain work.

It is the goal of this project to help remedy this situation. This USB Gain/Phase Analyzer is a cheap, portable tool. With this, a teacher would have the ability to bring circuits to class and actually demonstrate fundamentals to their students. This would inevitably enhance the effectiveness of their lectures because students would see how these circuits respond in application. The aim, here, is to help a teacher captivate his/her students.

Additionally, the analyzer is a tool that students can afford. In a lab, they could now be asked to design something, increasing the creative thinking of the prospective engineer. With this tool, they can now do more meaningful analysis and see their devices' behaviors on real world signals. Also, the device is portable enough that students could take it home and do lab work there, without needing to spend thousands of dollars on lab equipment.

1.3 Design Requirements

1.3.1 Context-Level Constraints

This project is producing one gain/phase analyzer system. As shown in Figure 1, the device connects to a PC for user control and viewing of data. It has one Drive output, with which it applies a stimulus to a Device Under Test (DUT), and two Sense inputs, with which it detects the amplitude and phase of signals before and after the DUT. The device will also have an Adapter port, with which it can connect to external adapters for measuring various types of DUTs. The gain/phase analyzer system, in addition to the hardware, comprises PC Software with which the end user may start analyses and view results.

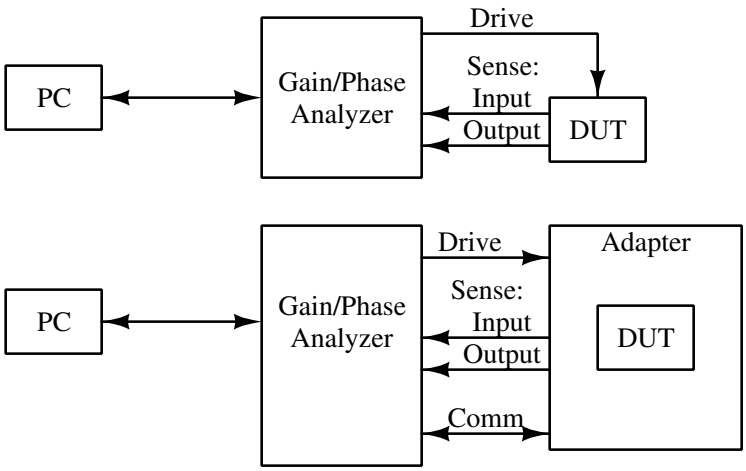


Figure 1: Gain/Phase Analyzer Context Diagram

1.3.2 System-Level Constraints

As shown in Figure 2, the analyzer uses a *Synthesizer* to generate the stimulus signal, and an *Output Amplifier* to provide the stimulus signal to the DUT, at up to 1.25 V RMS and up to 150 MHz. *Input Filters*, an *Input Switching Network* and the *Input Detector* provide a signal corresponding to the amplitude of the signals at the Sense ports. The Input Switching Network can also select a *Phase Reference* to be summed with the signals for phase measurement. These are digitized by an *Analog-Digital Converter* to be processed by the *Microprocessor*. The Microprocessor then interfaces with the *Software* via the *PC Interface*.

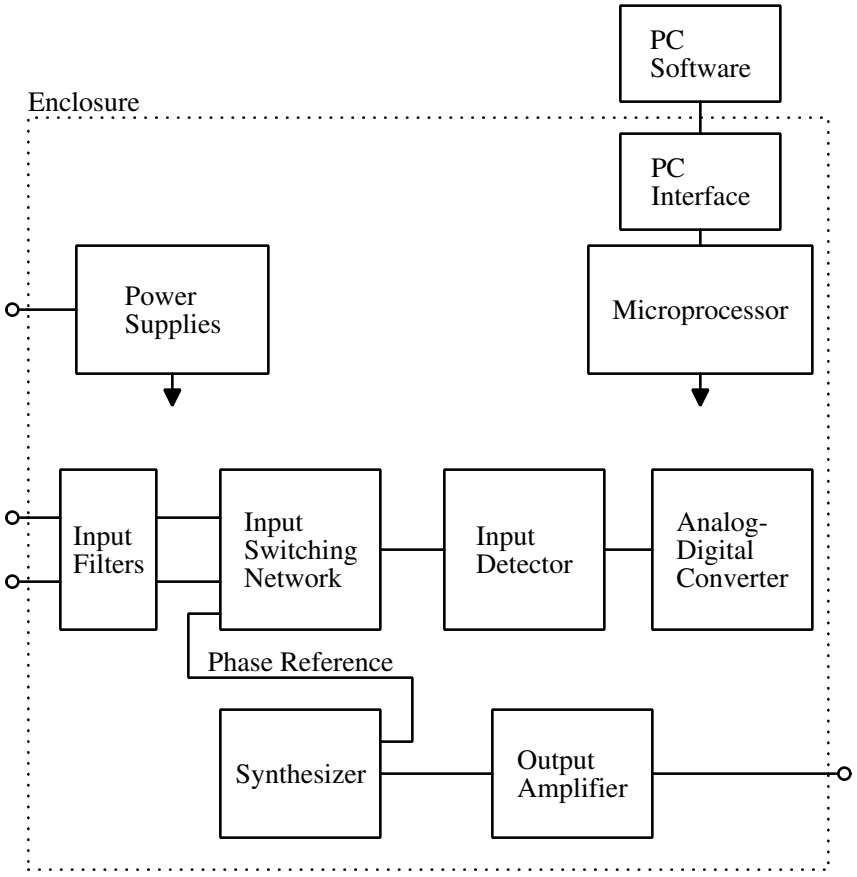


Figure 2: Gain/Phase Analyzer System Diagram

2 Design Description

2.1 Overview

A Gain/Phase Analyzer is an instrument used to plot the frequency response of a network or amplifier. This project is a small, portable gain/phase analyzer, which is controlled by a PC via USB and can perform analysis from 1 kHz to 150 MHz.

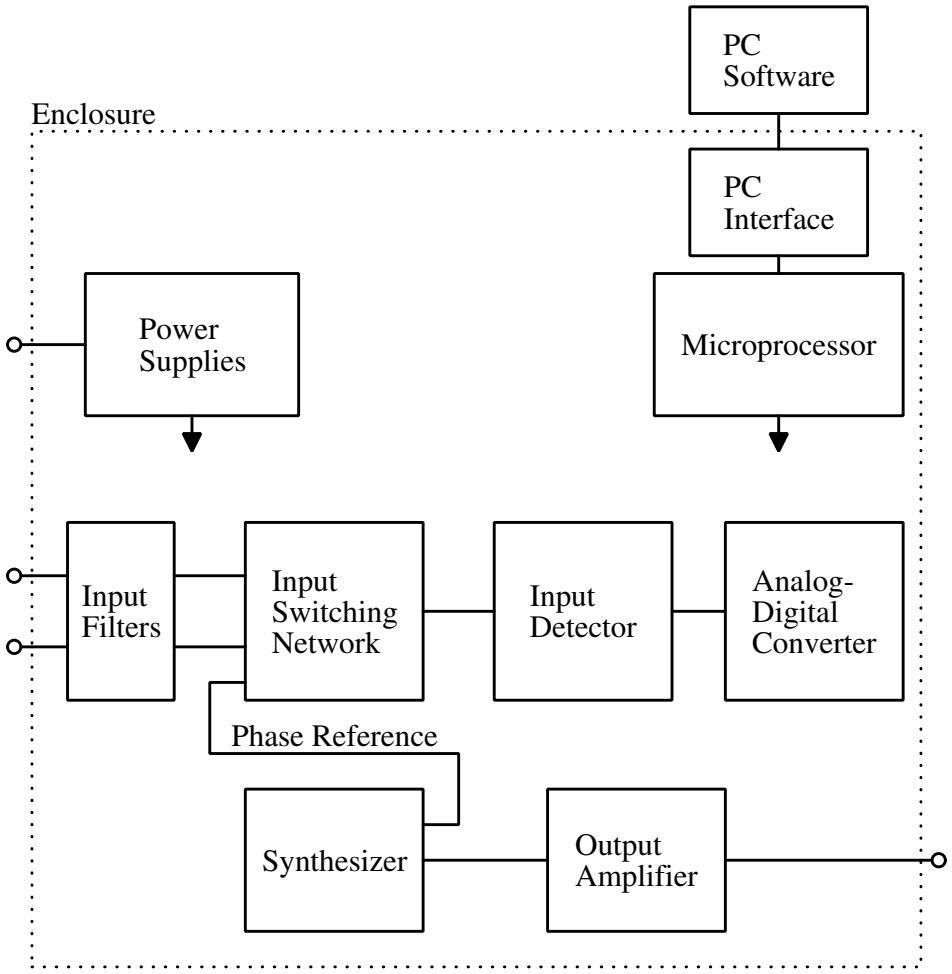


Figure 3: System diagram

Gain/phase analyzers work by generating a sinusoidal stimulus signal, which is applied to the device under test, and then measuring the signal that comes back out of the device. This signal is generated by the Synthesizer, and feeds into the Output Amplifier where it is boosted to an appropriate level for analysis. After it returns from the devuce under test, it passes through input filtering and protection, into a switch network. This allows one of two ports to be measured. It is summed with a second signal from the synthesizer, which is used for phase detection, and then passes to a logarithmic Detector which provides very sensitive detection of the signal power level. The output of this detector is measured by an Analog-Digital Converter that is under contrlo of a Microprocessor, and the data is returned to the PC Software via the PC Interface.

2.2 Detailed Description

2.2.1 Synthesizer

This instrument needs to generate signals over a very wide range, from 1 kHz to 150 MHz (a range of over five orders of magnitude). To achieve this, a monolithic, digital sinusoid synthesizer was used: the `AD9958` from Analog Devices. This is a dual DDS (Direct Digital Synthesizer) operating at a sample rate of 500 MHz, with a pair 10-bit digital-analog converters [3].

This device generates differential current-mode outputs, which need to be converted to a single-ended signal for transfer through the rest of the signal path. A pair of wideband video amplifiers was used as differential amplifiers

with 0 dB gain to convert the signal.

2.2.2 Output Signal Path

Sometimes, it is necessary to decrease the output amplitude significantly, when analyzing devices with high gain. The DDS features amplitude control, but this is performed digtally, meaning that the digital resolution relative to the full-scale signal amplitude decreases with the amplitude itself. This is acceptable only for small variations. To facilitate larger shifts in amplitude, an analog, switchable, 15 dB attenuator comes next in the signal path. The M/A-COM `MAADSS0008` is perfect for the application: a DC-coupled GaAs (gallium arsenide) switched 15 dB attenuator with low insertion loss well past our bandwidth [5]. A simple arrangement of discrete transistors level-shifts a control signal from the microprocessor to provide the necessary negative bias to switch this.

After the attenuator, the signal passes through an antialiasing lowpass filter. This is necessary because sampled systems inherently generate *aliases*, or copies of the main signal that appear above the *Nyquist rate*, which in this case is at 250 MHz. These are undesirable as they could stimulate responses in the device under test at frequencies we do not intend to measure. A seventh-order elliptical filter provides a very sharp cutoff at approximately 150 MHz to remove these from the signal.

The clean signal can now be amplified to drive the output. To achieve the specified 1.25 V RMS output level, 30 dB of gain is required, which is difficult because of the high bandwidth. This translates to a gain-bandwidth product of about 3.2 GHz and a maximum slew rate of 1110 V/µs. A chain of two very advanced, high speed operational amplifiers was used, at 15 dB each. The first is an Analog Devices `AD8000`, which offers 1.5 GHz bandwidth and up to 4100 V/µs slew rate [1]. This amplifier is not sufficient to be the *final* output amplifier, though, as it cannot operate at a high enough voltage and does not have sufficient slew rate under our operating conditions. The second amplifiers is a Texas Instruments `THS3001`, which provides a lower 420 MHz bandwidth but an extremely high 6500 V/µsslew rate and can operate on supplies up to +16 V and −16 V [8].

2.2.3 Input Signal Path

Two input connectors are provided, to allow relative measurements from the input to the output of a filter. The two input signals first pass through a small input protection network. This is not hugely significant, as anything that is would risk affecting the measurements. A pair of TVS diodes per input clamp off excessively large signals, with a small series resistor providing an impedance for them to clamp against. This provides enough power handling to quickly clamp off inputs from resonant devices under test, but the user will have to exercise care if testing powered devices like amplifiers.

After the protection networks, the signal enters a pair of switches. These are DC-coupled, GaAs switches with 2 GHz bandwidth, very similar to the attenuators used in the output path [6]. An identical level-shifting circuit generates the 0 V and −5 V bias voltages, and the two switches are connected with their outputs in parallel and bias inputs interchanged to form a double-throw configuration.

The selected signal is buffered to prevent further circuitry from affecting the device under test, which could interfere with measurements. A discrete buffer made from a very high bandwidth (9 GHz) bipolar transistor is used — the 'excessively' high bandwidth comes with an extremely low input capacitance, allowing the signal to be applied with a purely resistive termination and no reactive impedance matching.

A phase reference signal (see subsection 2.2.4) is summed with the signal coming out of the buffer, and the sum passes through a simple low-pass filter (fifth-order Butterworth). This provides two functions. First, it removes external high-frequency interference that may be picked up before the input connectors, and second, it removes Nyquist aliases from the phase reference signal.

After the filter, the signal enters a logarithmic detector. This device provides an output signal corresponding to the input power in decibels (nominally, 24 mV/dB [2]), which can be filtered down to a much lower frequency range and directly sampled by an analog-to-digital converter.

2.2.4 Phase Measurement

Phase detection is not as simple to implement as amplitude detection. There are many ways to do it, some of which are only realistically implemented for signals of known amplitude.

The method used by this device is a null-finding approach. A second test signal (called the "phase reference"), of identical frequency to the primary test signal and variable phase, is summed with the input signal. The software steps the phase offset of this signal over the range from 0° to 360°, and tests the amplitude at each point. When the input signal to the instrument and this phase reference are exactly 180° apart, they should interfere destructively, summing to zero. The software performs a null search, testing for the phase offset that produces this zero signal. To allow a high-precision search but minimize the number of points tested, a recursive search is used: the entire range from 0° to 360° is tested with a relatively small number of points, and then these points are used to select a narrower range to re-test. The algorithm stops when the search range is narrower than the desired measurement precision.

2.2.5 Microprocessor

The microprocessor in the instrument has a fairly simple job. It provides a bridge between the PC and the instrument, via USB, generates control signals to the various circuits in the instrument, and contains the analog-digital converter that samples the signal from the logarithmic detector. The Atmel SAM4S is a series of ARM Cortex-M4 processors with all the necessary peripherals, a generous 120 MHz clock speed, and full JTAG debugging support [7]; this was used to provide all of these functions, of which it is directly capable.

To provide backups in case this device proved troublesome, two alternatives were selected. The system is designed to allow the use of an alternate microcontroller, an Atmel ATMEGA32U4, and also can be used with an external USB interface controller, an FTDI FT230XS. Neither of these was necessary in the end.

A Micro-USB connector provides interface to the PC. Signals pass through a pair of termination resistors and a low-capacitance TVS diode array, and then directly into the microprocessor. Power passes through a ferrite choke and into an simple circuit that provides both inrush current limiting and power switchover, so that the microcontroller is powered when either USB or the external power pack is present.

2.2.6 Analysis

The function of the PC software is relatively simple. To analyze a device under test, the instrument is configured for logarithmically spaced test frequencies over the desired test range, and the reoprted amplitude is queried. Then, the null-finding algorithm (subsection 2.2.4) is applied to find the phase information.

If desired, this frequency response can then be saved and used as a template for normalization. In this case, the saved amplitude (in decibels) and phase are subtracted from a second analysis, to compensate for the attenuation and phase shift of the analyzer itself and any test setup.

2.3 Use

The purpose of this device is provide students with tangible data when analyzing circuits. As shown in Figure 1, the device is to be connected between a PC and a device under test; the user can initiate an analysis using the supplied software.

3 Implementation

In order to verify the concepts, a prototype was built very early before building the final system.

3.1 Hardware Prototype PCBs

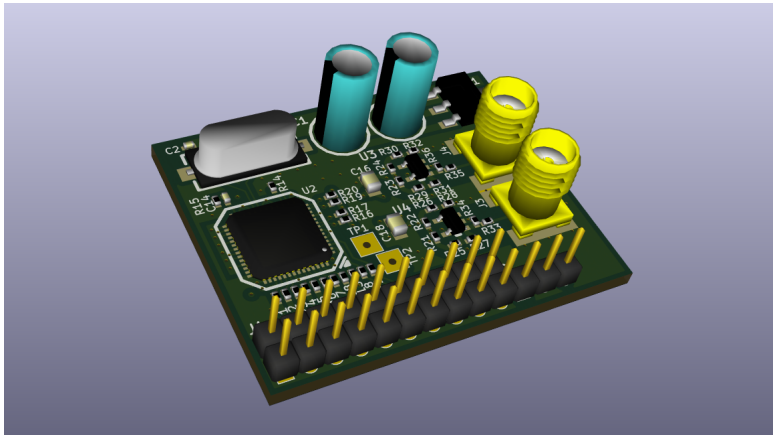


Figure 4: Synthesizer PCB, 3D render

Figure 4 shows the synthesizer prototype. This contains the AD9958 DDS and its associated support circuitry. A 10 ppm quartz crystal [9] was selected as the frequency response, and testing showed that the frequency stability of the AD9958's internal PLL (which was necessary to derive a 500 MHz clock from the 25 MHz crystal) was sufficient for the accuracy we required.

This board also contained the differential amplifiers, as the output circuit of the AD9958 is inconvenient to connect via external cables. Texas Instruments' inexpensive LMH6714 400 MHz [4] video amplifier was used.

This design worked exactly as expected, and the circuit on this board was carried through to the final build with no modification.

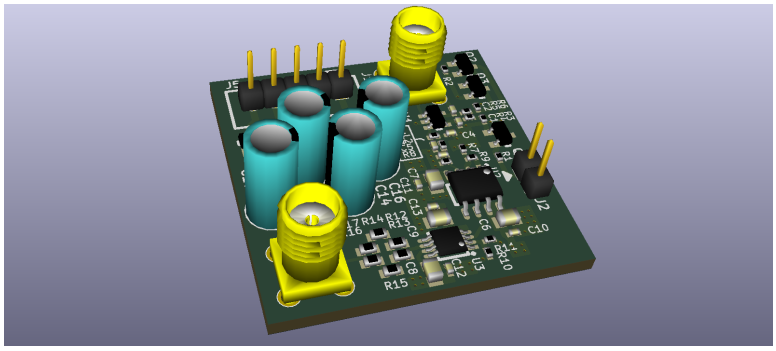


Figure 5: Output Amplifier PCB, 3D render

Figure 5 shows the output amplifier prototype. This prototype board accepts the signal directly from the synthesizer. The signal passes through a MAADSS0008 switched 15 dB attenuator, through a fifth-order Butterworth low-pass filter, and into the amplifiers. Testing revealed two problems with this design, both caused by the low-pass filter. The gradual roll-off of the fifth-order filter meant that the filter was both inadequate at removing all Nyquist aliases, and had too much loss at the high end of the desired passband.

For the final build, a precision, seventh-order elliptical filter was used. This filter design was prototyped and tested on its own, to verify that it would have the desired cutoff properties; it passed this test and so was used in the final design. No other changes were made here.

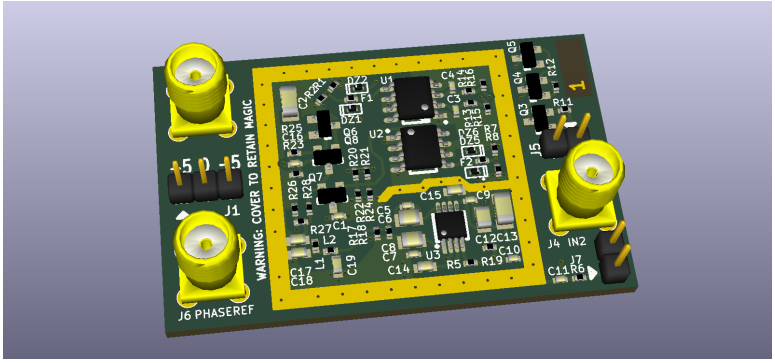


Figure 6: Input Front-end PCB, 3D render

?? shows the input frontend prototype. This contains the input protection network, the switches, the buffer, the filter, and the logarithmic detector. It has two coaxial inputs for the primary input signals, and one to be directly fed the phase reference signal from the synthesizer. This design also worked exactly as intended. The circuit was kept unchanged, though the layout was redone to fit the overall layout of the final board.

3.2 Microprocessor Software Prototyping

To prototype the microprocessor software, a development board was used rather than designing a custom PCB. An Atmel 'SAM4S Xplained' board, which holds a `SAM4S16C` (the largest member of the SAM4S series), an SRAM module, and an on-board programmer, was connected to the prototype boards. The final firmware was small, so the smaller but otherwise identical `SAM4S4C` was used in the final build.

3.3 PC Software

PC software was developed using Python. The TkInter graphical user interface toolkit was used to build the application, 'matplotlib' (a plotting library) was used to draw plots, and pySerial was used to provide communcations with the instrument.

4 Evaluation

4.1 Overview

Our system was designed through a combination of computer simulation, and prototype testing. After carefully designing each subsystem, per-subsystem PCBs were fabricated and interconnected to build a prototype, which was thoroughly tested before the final PCB was designed and fabricated.

As our project is a piece of electrical test equipment, the majority of the tests are electrical in nature. The tests for requirements 3.2.2 and 3.2.3 (output signal characteristics) require an oscilloscope with at least 300 MHz bandwidth. The remaining electrical tests require only a basic multimeter, and often use the instrument to verify itself. For example, 3.2.4 (sensitivity) is verified by measuring the reported amplitude from the output amplifier, and comparing that to the input noise floor. Requirement 3.2.6 (accuracy) is verified by examining the normalized sweep of a flat-response attenuator. Some requirements, for example 3.2.1 (type of plot), 3.3.1 – 3.3.3 (interface design), and 3.6.4 (direct control) are verified simply by observing how the instrument responds to PC control. Others, for example 3.3.4 (panel connectors), 3.6.1 (Operator's Manual), 3.6.2 (Protocol Guide), and 3.6.3 (surface-mount technology) are verified by observing the instrument and accompanying materials themselves.

The full project requirements can be found in Appendix A, and the full test procedures can be found in Appendix B.

4.2 Testing and Results

The full test procedures can be found in Appendix B.

Requirement WCP52.3.1.1 — Required mode: Idle

This requirement simply means that the device must have a mode in which it is not performing analysis. We measured the output signal in this state, and demonstrated that the "sample" annunciator was not lit.

Maximum output amplitude in "idle": 6.0 mV p-p, 750 µV RMS (mostly amplifier noise)

Requirement WCP52.3.1.2 — Required mode: Analysis

This requirement means that the device must have a non-idle mode in which it is performing its primary function. This does not require a separate demonstration; the fact that it performs correctly in the demonstration for WCP52.3.2.6 shows that it must be capable of performing analysis.

Requirement WCP52.3.2.1 — Bode plot

By providing sample data to the PC software and causing it to display a plot, we demonstrated that the software is capable of displaying a Bode plot.

Requirement WCP52.3.2.3 — Test signal frequency

We used an oscilloscope to measure the output of the instrument at nominal frequencies of 1 kHz, 10 MHz, 75 MHz, and 150 MHz. The maximum deviation from nominal was 0.13%, which is within the 2.5 % required by the test procedure.

Nominal	Measured	Error
1 kHz	1.000 kHz	0%
10 MHz	10.01 MHz	0.1%
75 MHz	74.95 MHz	0.06%
150 MHz	150.2 MHz	0.13%

Requirement WCP52.3.2.3 — Test signal amplitude

We used an oscilloscope to measure the amplitude of the instrument's output at nominal frequencies of 1 kHz, 20 MHz, and 100 MHz.

Test Frequency	Amplitude
100 MHz	1.267 V RMS
20 MHz	1.360 V RMS
1 kHz	1.516 V RMS

These are all above the required minimum of 1.25 V RMS.

Requirement WCP52.3.2.4 — Sensitivity

First, we measured the instrument's noise floor at 100 kHz, which is the absolute limit to its sensitivity. **The noise floor was -87.5 dB.** This ensures that any measured amplitude above this is the true amplitude, not the noise floor itself. Then, we measured the relative attenuation of a 40 dB nominal attenuator, verifying that the signal could be seen above the noise floor. This shows that the instrument is capable of viewing signals at least 40 dB less than its output amplitude.

Requirement WCP52.3.2.5 — Extended sensitivity

The fact that the noise floor above was also under a threshold of -65 dB, and that a -60 dB signal was visible above it, satisfies the requirement for extended sensitivity.

Requirement WCP52.3.2.6 — Accuracy — Amplitude

We measured an attenuator specified for **10.0 dB**, and the instrument claimed an attenuation between **9.1 dB and 10.4 dB**. This is within the required 3 dB accuracy limit.

Requirement WCP52.3.2.6 — Accuracy — Phase

We measured the phase shift due to propagation delay of a section of RG-316 coaxial cable. The nominal phase shifts are:

Frequency	Phase shift
10 kHz	0.02°
1 MHz	1.51°
10 MHz	15.1°
25 MHz	37.8°

We measured phase shifts of:

Frequency	Phase shift
10 kHz	1.76°
1 MHz	0.70°
10 MHz	19.0°
25 MHz	42.2°

These measurements are within 4.4°of nominal, which satisfies the 5°requirement.

Requirement WCP52.3.2.7 — Extended accuracy

The 4.4°error does *not* meet the optional requirement for extended accuracy, which requires a maximum error of 1°. However, the 0.9 dB amplitude error *does* meet the extended accuracy requirement for amplitude.

A possible source of phase error is that tests were performed without the shielding metal enclosure. Electromagnetic interference in the area around the analyzer tends to cause errors in the sensitive null-finding algorithm.

Requirement WCP52.3.2.8 — Interface safety

To test this, we issued the `LOWLEVEL:SET GPIO_LEVEL` command, and recieved back the message `Pin 'GPIO_LEVEL' is not an output!`. This verifies that the instrument will not set an output value on an input pin.

Requirement WCP52.3.3.1 — Interface

This requirement is satisfied by the fact that WCP52.3.2.1 through WCP52.3.2.8 were satisfied, which all required a PC interface.

Requirement WCP52.3.3.2 — Communications type

We connected a serial terminal to the device and issued the `*IDN?` command, which produced `WCP52, GPA, 1, 0`. These are both text commands, verifying that the instrument uses a text-based serial protocol.

Requirement WCP52.3.3.3 — Communications medium

The use of a USB-compatible serial terminal in verifying WCP52.3.3.2 also demonstrated WCP52.3.3.3.

Requirement WCP52.3.3.4 — Panel connectors

We demonstrated that the connectors on the front panel are SMA connectors.

Requirement WCP52.3.3.5 — Auxiliary connector

We measured the voltages on the power supply pins of the auxiliary connector; they were +8.76 V and −9.28 V, and could supply 93 mA and 190 mA respectively.

Requirement WCP52.3.6.1 — Operator's manual

We presented the PDF operator's manual, and showed the 'Theory of Operation' and 'Operational Instructions' sections in it. The latter section had test setups for characterizing both filters and control loops.

Requirement WCP52.3.6.2 — Protocol guide

We showed that there is a protocol guide inside the Operator's Manual, which lists the SCPI commands and their formats.

Requirement WCP52.3.6.3 — SMT

We showed that all parts on the PCB were surface-mount with the following exceptions:

DS1 — front panel LED — exemption: front panel

J1 — power jack — exemption: connector, front panel

J2 — input jack #1 — exemption: connector, front panel

J3 — input jack #2 — exemption: connector, front panel

J4 — output jack — exemption: connector, front panel

J5 — auxiliary jack — exemption: connector, front panel

J8 — JTAG port — exemption: connector

L6 — buck-boost inductor — exemption: inductor $>$ 5 mm diam

L7 — buck inductor — exemption: inductor $>$ 5 mm diam

Also, we showed that the only leadless parts were those that are directly responsible for the device's function:

U3 — DDS — directly satisfies WCP52.3.2.2

Requirement WCP52.3.6.4 — Direct control

We showed the following commands in the protocol guide: `LOWLEVEL:SET`, `LOWLEVEL:GET`, `LOWLEVEL:SPITX`, `LOWLEVEL:ADC`.

Requirement WCP52.3.6.5 — Electrical safety

We measured the voltages present on the pins of the auxiliary connector with a multimeter; they were, respectively, *3.03 V, 3.03 V, 8.76 V, −9.28 V, 3.03 V, 0.0 V*. We then measured the maximum amplitude of a signal from the output connector; the maximum peaks were 4.4 V and −4.4 V. These are all within the 15 V limit.

4.3 Assessment

In the end, our design met all of the mandatory requirements, and most of the optional ones. This is overall a useful design. It can measure filters over a wide range with reasonable accuracy. It is simple to use, robust against input overloads, output overloads, and incorrect power supply. The system is fully open-source, so users can modify and develop on it as they please, and students can learn from its operation.

The design has a few drawbacks, however. First, power consumption is high, as the power supply is inefficient. This was necessary within the development budget and time, as a more efficient power supply design would also produce more noise that could interfere with the measurements. Multiple prototypes and significant analysis and testing could have been required. Second, output phase noise is relatively high at high frequencies. This could cause measurement error particularly when measuring devices with nonlinearities or when attempting to measure very close to a strong resonance. However, this noise is inherent in a design with a fixed sample rate, and correcting this would require a significant increase in complexity of clock generation. Third, the output can have somewhat significant DC offset voltages (up to around 100mV), which is not a huge problem but could be surprising to some users. A coupling capacitor after the differential amplifier stage would fix this, but using the correct one would require significant testing and would risk introducing amplitude loss at either the high or the low end. Fourth, the frontend is not selective; it always measures the total input power over its full bandwidth. A frontend with a selective mixer to measure only signals at the same frequency as the output frequency would greatly increase the measurement noise floor and improve the behavior with nonlinear devices-under-test.

5 Schedule and Budget

The project budget is shown in Table 1.

Item	**Original Estimate ($)**	**Actual to Date ($)**	**Estimate to Completion ($)**	**Estimate at Completion ($)**
Synthesizer prototype and parts	$60	$60	$0	$60
Input prototype and parts	$70	$70	$0	$70
Output amplifier proto. and parts	$90	$91	$0	$91
Power supply proto. and parts	$0	$0	$0	$0
Final PCB and parts	$210	$210	$0	$210
Enclosure	$20	$21	$0	$21
Misc.	$50	$0	$0	$0
Total	$500	$452	$0	$452

Table 1: Project Budget

Table 2 shows the project schedule.

Description	Percent Complete	Date Completed
Requirements specification	100%	October 17, 2014
Development plan	100%	October 31, 2014
Synthesizer prototype built	100%	November 14, 2014
Input frontend prototype built	100%	December 1, 2014
Output amplifier prototype built	100%	December 5, 2014
Microcontroller experimentation	100%	December 16, 2014
Corrections made to prototypes	100%	February 26, 2015
Power supply design	100%	February 28, 2015
USB communications completed	100%	March 4, 2015
Final PCB layout	100%	March 14, 2015
Command parser completed	100%	March 19, 2015
Final PCB assembled	100%	April 8, 2015
Final testing	100%	April 30, 2015
PC software completed	100%	April 30, 2015
Owner's manual	100%	May 1, 2015

Table 2: Project Schedule

6 Future Plans

As it is, this is a solid design, and ready to be delivered. As the full board can be assembled almost entirely by automated pick-and-place, these can be delivered as a prepared package, or they could be delivered as kits with components and an assembly guide.

However, we recommend simultaneously beginning a second revision. The drawbacks mentioned in the Assessment should be corrected. Additionally, the BOM cost could be lowered in a few areas, such as replacing expensive wideband operational amplifiers with transistor amplifiers, or even possibly redesigning the synthesis subsystem to avoid using the purpose-specific DDS integrated circuit (it might be possible to use a relatively high speed FPGA, at a reduction of about half the cost).

Additionally, the PC software can be enhanced. More analysis functions can be written, and the software could be made to integrate with existing packages such as Octave.

7 References

[1] Analog Devices, Inc., "1.5 GHz Ultrahigh Speed Op Amp," AD8000 datasheet, March 2013 [Revision B]. http://www.analog.com/media/en/technical-documentation/data-sheets/AD8000.pdf

[2] Analog Devices, Inc., "Fast, Voltage-Out, DC to 440 MHz, 95 dB Logarithmic Amplifier," AD8310 datasheet, June 2010 [Revision F]. http://www.analog.com/media/en/technical-documentation/data-sheets/AD8310.pdf

[3] Analog Devices, Inc., "2-Channel, 500 MSPS DDS with 10-Bit DACs," AD9958 datasheet, April 2013 [Revision B]. http://www.analog.com/media/cn/technical-documentation/data-sheets/AD9958.pdf

[4] Texas Instruments, "Wideband Video Op Amp; Single, Single with Shutdown and Quad," LMH6714/LMH6720/LMH6722/LMH6722-Q1 datasheet, April 2013 [Revision G]. http://www.ti.com/lit/ds/symlink/lmh6714.pdf

[5] M/A-COM Technology, "Digital Attenuator, 1-Bit, 15 dB, DC-2.0 GHz," MAADSS0008 datasheet [Revision V1]. http://cdn.macom.com/DataSheets/MAADSS0008.pdf

[6] M/A-COM Technology, "GaAs SPST Switch," MASWSS0162 datasheet [Revision V3]. http://cdn.macom.com/DataSheets/MASWSS0162.pdf

[7] Atmel, "SAM4S Series SMART ARM-based Flash MCU," SAM4S datasheet, April 2015 [Revision 11100I]. http://www.atmel.com/Images/Atmel-11100-32-bit%20Cortex-M4-Microcontroller-SAM4S_Datasheet.pdf

[8] Texas Instruments, "420 MHz High-Speed Current-Feedback Amplifier," THS3001 datasheet, Sept. 2009 [Revision H]. http://www.ti.com.cn/cn/lit/ds/slos217h/slos217h.pdf

[9] TXC Corp., "9C Series Quartz Crystals," TXC-9C datasheet, http://www.txccrystal.com/images/pdf/9c-accuracy.pdf

A Project Requirements

3.1: Required States and Modes

This system requires the following modes:

3.1.1: Idle

The signal source and detection subsystems are inactive, and the system is waiting for commands.

3.1.2: Analysis

The system is performing a gain/phase analysis.

3.2: System Capability Requirements

3.2.1: Bode plot

The analyzer shall be able to display a plot on the operator's PC in the form of a Bode plot.

3.2.2: Test signal frequency

The analyzer shall be capable of sourcing test signals between 1 kHz and 150 MHz. (Not applicable: state WCP52-3.1.1 — idle)

3.2.3: Test signal amplitude

The analyzer shall be capable of output amplitudes up to 1.25 V RMS at frequencies up to 100 MHz. (Not applicable: state WCP52-3.1.1 — idle)

3.2.4: Sensitivity

The analyzer shall be able to detect signals down to at least 40 dB below the output amplitude. (Not applicable: state WCP52-3.1.1 — idle)

3.2.5: Extended sensitivity

The analyzer should be able to detect signals down to at least 60 dB below the output amplitude. (Not applicable: state WCP52-3.1.1 — idle)

3.2.6: Accuracy

Amplitude accuracy shall be within 3 dB, and phase accuracy within 5°.

3.2.7: Extended accuracy

Amplitude accuracy should be within 1 dB, and phase accuracy within 1°, for frequencies less than 20 MHz.

3.2.8: Interface safety

The hardware shall not be able to be damaged by its remote interface, unless an "unlock" command has been issued.

3.3: System External Interface Requirements

3.3.1: Interface

The analyzer shall interface with a PC.

3.3.2: Communications type

The analyzer should use a text-driven protocol.

3.3.3: Communications medium

The analyzer should use a common, standard communication protocol, for example, USB-CDC.

3.3.4: Panel connectors

The analyzer shall use either SMA or BNC connectors to interface to the device under test (DUT).

3.3.5: Auxiliary connector

The analyzer shall provide power via a front-panel connection for use with external DUT adapters. The voltage should be at least ± 7 V, and up to 40 mA should be available.

3.4: System Internal Interface Requirements

All internal interfaces are left to the system designers.

3.5: System Internal Data Requirements

All decisions about internal data are left to the system designers.

3.6: Other System Requirements

3.6.1: Operator's Manual

The system should include a simple operator's manual, which should include a brief Theory of Operation explaining its design, instructions for using each function, and example test setups for characterization of filters and control loops.

3.6.2: Protocol Guide

The system shall include a protocol guide, showing how to communicate with it.

3.6.3: SMT

The PCB shall be produced using surface-mount technology as much as is reasonable, without no-lead packages unless absolutely required.

3.6.4: Direct control

The interface should expose direct control of the hardware functions, allowing additional features to be implemented.

3.6.5: Electrical safety

The system shall have no voltages greater than 30 V peak-to-peak accessible externally.

3.7: Precedence and Criticality of Requirements

All requirements have equal weight.

B Test Procedures

3.1.1: Required mode: Idle

The idle state of the signal source is to be verified by viewing the signal from the "output" connector on an oscilloscope. Any signal present must be less than 5 V peak to peak.

The idle state of the detection subsystem is verified by viewing the status annunciators on the printed circuit board. The annunciator designated "sample" must not light.

3.1.2: Required mode: Analysis

This requirement is satisfied peripherally by the completion of requirement 3.2.6.

3.2.1: Bode plot

Cause the software to display any data, and verify that it is in the form of a Bode plot. This means that there should be two plots, both with a horizontal axis of 'frequency', one with a vertical axis of 'gain' in decibels, and one with a vertical axis of 'phase' in degrees.

This data can be generated by the instrument (satisfying this requirement peripherally by the completion of requirement 3.2.6), or by any other means.

3.2.2: Test signal frequency

To test this requirement, connect a patch cable between the "Output" connector and an oscilloscope with at least 150 MHz bandwidth. Either using the PC software or manual control via a serial terminal, command the instrument to generate signals with frequencies of 1 kHz, 10 MHz, 75 MHz, and 150 MHz. Using the oscilloscope's frequency display, verify that each signal's frequency is within 2.5 % of its nominal value.

3.2.3: Test signal amplitude

To test this requirement, connect a patch cable between the "Output" connector and an oscilloscope with at least 300 MHz bandwidth. Either using the PC software or manual control via a serial terminal, command the instrument to generate signals with frequencies of 1 kHz, 20 MHz, and 100 MHz, all with maximum amplitude. Using the oscilloscope's amplitude display, verify that all signals have an amplitude of at least 1.25 V RMS.

3.2.4: Sensitivity

1. Connect a patch cable between "Output" and "Input 1".
2. Command instrument to generate a signal of 100 kHz and maximum amplitude.
3. Query reported amplitude.
4. Remove patch cable, query reported amplitude. This noise floor must be at least 45 dB lower than the amplitude in step 3.
5. Reinstall patch cable with a 40 dB attenuator in series. The −40 dB signal must be visible.

3.2.5: Extended sensitivity

To test this optional requirement, perform the same test as in requirement 3.2.4, but verify that the final reported amplitude is at least 65 dB lower than the first reported amplitude.

3.2.6: Accuracy

Amplitude

Connect a pair of patch cables in series using a cable coupler, and connect this pair between the "Output" connector and the "Input 1" connector. Configure the PC software for a full sweep from 1 kHz to 150 MHz not including phase analysis. Perform normalization. Then, remove the cable coupler and insert a 50 Ω attenuator with specified attenuation between 5 dB and 15 dB and bandwidth of at least 200 MHz. Perform analysis. Verify that the reported gain is within 3 dB of the specified gain of the attenuator at all frequencies.

Phase

Connect a patch cable with length under 150 mm between "Output" and "Input 1". Configure for a sweep from 10 kHz to 25 MHz, normalize. Using an SMA coupler, add a patch cable made from 1 m of RG-316 coaxial cable in series. This cable is expected to exhibit the following phase shift due to propagation delay:

Frequency	Phase shift
10 kHz	0.02°
1 MHz	1.74°
10 MHz	17.4°
25 MHz	43.5°

Ensure that the propagation delay reported is within 5° of this at these frequencies.

3.2.7: Extended accuracy

To test this optional requirement, perform the test for requirement 3.2.6, with the following modifications: the upper sweep limit should be 20 MHz, reported gain must be within 1 dB of the specified attenuator gain, and phase shift must be within 1° of zero (above −1° and below +1°) at all frequencies.

3.2.8: Interface safety

To test this requirement, use the interface control commands to attempt to set a GPIO pin which serves as a signal input to be an output instead. The system must respond with an error.

3.3.1: Interface

This requirement is satisfied peripherally by the completion of requirements 3.2.1 through 3.2.8, which all required PC interfacing to complete.

3.3.2: Communications type

To test this requirement, connect the instrument to a PC. Connect a serial terminal application to it, and perform at least one command that produces a response. Demonstrate that the command and response comprise displayable text symbols.

3.3.3: Communications medium

This requirement is demonstrated peripherally as part of demonstrating requirement 3.3.2: the ability for a standard serial console to talk to the instrument demonstrates that the instrument was using a standard protocol.

3.3.4: Panel connectors

To test this requirement, observe the front panel of the device, and see that the RF connectors are either SMA or BNC.

3.3.5: Auxiliary connector

Using a multimeter, measure all pins on the "Auxiliary" connector, verifying that there are power supply pins with a voltage present of at least $+7$ V and -7 V. -7 V. Then, switch the multimeter to ammeter mode, and connect in series a selected resistor. Probe these pins again, and verify that at least 40 mA is supplied.

The resistor is to be selected such that it will draw at least 40 mA at the voltage that was measured on the connectors. This resistance may depend on the actual voltage that was present, which is allowed to be *above* the required ± 7 V.

3.6.1: Operator's Manual

Demonstrate the existence of a manual in digital format, containing at least the following sections: Theory of Operation, Operational Instructions. Demonstrate that there exist example test setups for characterization of filters and of control loops.

3.6.2: Protocol Guide

Demonstrate the existence of documentation explaining the protocol with which a PC communicates with the instrument. This documentation may be part of the Operator's Manual, or it may be separate.

3.6.3: SMT

Demonstrate that the parts on the PCB are surface-mount devices, with the following allowed exceptions:

- Connectors: through-hole connections have higher mechanical stability
- Inductors and capacitors larger than 5 mm in diameter: through-hole parts have higher mechanical stability

Demonstrate that the only leadless devices on the PCB are essential for its function. This may be a result of directly satisfying an above requirement. An anticipated example is device U3, an Analog Devices AD9958BCPZ digital frequency synthesizer, which directly satisfies requirement 3.2.2 (test signal frequency between 1 kHz and 150 MHz).

3.6.4: Direct control

Viewing the protocol guide, demonstrate the existence of:

- A command to set the output value of an arbitrary GPIO pin.
- A command to query the input value of an arbitrary GPIO pin.
- A command to transmit arbitrary data via the on-board SPI interface.
- A command to query the direct output value of the analog-to-digital converter.

3.6.5: Electrical safety

To test this requirement, connect the “common” input of a multimeter to the instrument ground, and verify that no signals on the “Auxiliary” connector are more positive than 15 V or more negative than −15 V. Then, connect a patch cable between the “Output” connector and an oscilloscope. Do not terminate the end. Command the instrument to generate a frequency of 100 kHz and maximum amplitude. Using the oscilloscope’s amplitude display, verify that the positive and negative peaks are no more positive than 15 V and no more negative than −15 V.

C Microprocessor Code

```c
/**
 * \file
 * Signal acquisition-related functions
 */

// ASF includes
#include <adc.h>
#include <pmc.h>
#include <sysclk.h>

#include <inttypes.h>
#include "conf_board.h"

#include "acquisition.h"

static volatile uint8_t RECEIVED = 0;
static volatile uint32_t ADCVAL = 0;

void ADC_Handler(void)
{
    if (adc_get_status(ADC) & ADC_ISR_DRDY) {
        uint32_t result = adc_get_latest_value(ADC);
        ADCVAL = result;
        RECEIVED = 1;
    }
}

void adc_setup(void)
{
    pmc_enable_periph_clk(ID_ADC);
    adc_init(ADC, sysclk_get_main_hz(), ADC_CLOCK, 8);
    adc_configure_timing(ADC, 0, ADC_SETTLING_TIME_3, 1);
    adc_set_resolution(ADC, ADC_MR_LOWRES_BITS_12);
    adc_enable_channel(ADC, ADC_CHANNEL_3); // TODO: wrong channel
    adc_enable_interrupt(ADC, ADC_IER_DRDY);
    adc_configure_trigger(ADC, ADC_TRIG_SW, 0);
    NVIC_EnableIRQ(ADC_IRQn);
    adc_start(ADC);
}


double acq_get_values (unsigned count)
{
    double total = 0.0;
    size_t n;
    for (n = 0; n < count; ++n) {
        RECEIVED = 0;
        adc_start(ADC);
        while (!RECEIVED);
        total += ADCVAL;
    }
    total /= count;
    return total;
}
```

```
/**
 * \file
 * Main loop and initializers.
 */

// libc includes
#include <assert.h>
#include <ctype.h>
#include <inttypes.h>
#include <stdio.h>
#include <string.h>

// Atmel ASF includes
#include <pio.h>
#include <spi.h>
#include <spi_master.h>
#include <stdio_usb.h>
#include <sysclk.h>
#include "conf_board.h"
#include "usb-functions.h"

// Software libraries
#include "scpi/scpi.h"
#include "scpi-def.h"
#include "util.h"

// Hardware support
#include "acquisition.h"
#include "synth.h"

static void pins_init(void);

/**
 * Initialize the board
 */
// Cannot declare static, as board.h declares this extern
void board_init(void)
{
    // Disable watchdog timer
    WDT->WDT_MR = WDT_MR_WDDIS;

        sysclk_enable_peripheral_clock(ID_PIOA);
        sysclk_enable_peripheral_clock(ID_PIOB);
        sysclk_enable_peripheral_clock(ID_PIOC);

    pins_init();
    pmc_enable_periph_clk(ID_PIOB);
}

/**
 * Initialize the SPI controller.
 */
static void spi_init(void)
{
    /* Configure an SPI peripheral. */
    spi_enable_clock(SPI_MASTER_BASE);
    spi_disable(SPI_MASTER_BASE);
    spi_reset(SPI_MASTER_BASE);
    spi_set_lastxfer(SPI_MASTER_BASE);
    spi_set_master_mode(SPI_MASTER_BASE);
    spi_disable_mode_fault_detect(SPI_MASTER_BASE);
    spi_set_peripheral_chip_select_value(SPI_MASTER_BASE, SPI_CHIP_PCS);
    spi_set_clock_polarity(SPI_MASTER_BASE, SPI_CHIP_SEL, SPI_CLK_POLARITY);
    spi_set_clock_phase(SPI_MASTER_BASE, SPI_CHIP_SEL, SPI_CLK_PHASE);
    spi_set_bits_per_transfer(SPI_MASTER_BASE, SPI_CHIP_SEL,
            SPI_CSR_BITS_8_BIT);
    spi_set_baudrate_div(SPI_MASTER_BASE, SPI_CHIP_SEL,
            (sysclk_get_cpu_hz() / 100000uL));
    spi_set_transfer_delay(SPI_MASTER_BASE, SPI_CHIP_SEL, SPI_DLYBS,
            SPI_DLYBCT);
```

```
    spi_enable(SPI_MASTER_BASE);
}

/**
 * Configure all device pins
 */
static void pins_init(void)
{
    size_t i;
    for (i = 0; PIN_TABLE[i].description; ++i) {
        if (!PIN_TABLE[i].index_str) continue; // Pin group, not pin
        pio_configure_pin(PIN_TABLE[i].index, PIN_TABLE[i].flags);
    }
}

/**
 * Main function.
 *
 * \return Unused (ANSI-C compatibility).
 */
int main(void)
{
    // Initialize all used peripherals.
    sysclk_init();
    irq_initialize_vectors();
    cpu_irq_enable();
    board_init();
    spi_init();
    adc_setup();
    stdio_usb_init();
    pio_set_pin_high(GPIO_LED1);

    SCPI_Init(&G_SCPI_CONTEXT);

    const size_t SMBUFFER_SIZE = 10;
    char smbuffer[10];
    for (;;) {
        // Get into smbuffer until either full, or a \r or \n
        size_t i;
        i = 0;
        while (i < SMBUFFER_SIZE - 1) {
            char ch = 0;
            if (G_CDC_ENABLED) {
                ch = udi_cdc_getc();
            } else {
                continue;
            }
            if (!ch) {
                continue;
            }
            if (ch == '\r' || ch == '\n') {
                smbuffer[i] = ch;
                ++i;
                break;
            } else {
                smbuffer[i] = ch;
                ++i;
            }
        }
        smbuffer[i] = 0; // Terminate!
        SCPI_Input(&G_SCPI_CONTEXT, smbuffer, strlen (smbuffer));
    }

    return 0;
}
```

```
// Atmel ASF includes
#include <pmc.h>

#include <stdio.h>
#include <inttypes.h>
#include "scpi/scpi.h"
#include "scpi-def.h"

/* These functions are required by the SCPI library to interact with the
 * console.
 */

/**
 * Write back to the SCPI console.
 * \param context   Active SCPI context
 * \param data      Data to write
 * \param len       Length of data
 * \return  Number of bytes written
 */
size_t SCPI_Write(scpi_t *context, const char * data, size_t len) {
    (void) context;
    return fwrite (data, 1, len, stdout);
}

/**
 * Flush the SCPI console. May be a no-op if not possible or sensible.
 * \param context   Active SCPI context
 * \return  Success or failure; returns success on no-op.
 */
scpi_result_t SCPI_Flush (scpi_t *context) {
    (void) context;
    return SCPI_RES_OK;
}

/**
 * Report a SCPI error. Not all SCPI interfaces report this; some require
 * polling. This may be a no-op in that case.
 * \param context   Active SCPI context
 * \param err       SCPI error number
 * \return  zero
 */
int SCPI_Error (scpi_t *context, int_fast16_t err) {
    (void) context;
    fprintf(stderr, "**ERROR: %" PRIdFAST16 ", \"%s\"\r\n", err, SCPI_ErrorTranslate(err)
);
    return 0;
}

/**
 * Handle the SCPI control commands.
 * \param context   Active SCPI context
 * \param ctrl      The control command given. See scpi_ctrl_name_t in
 *                  scpi/types.h for a list.
 * \param val       The value passed with the command.
 * \return  Success or failure
 */
scpi_result_t SCPI_Control(scpi_t * context, scpi_ctrl_name_t ctrl, scpi_reg_val_t val) {
    (void) context;
    if (SCPI_CTRL_SRQ == ctrl) {
        fprintf(stderr, "**SRQ: 0x%X (%d)\r\n", val, val);
    } else {
        fprintf(stderr, "**CTRL %02x: 0x%X (%d)\r\n", ctrl, val, val);
    }
    return SCPI_RES_OK;
}

/**
 * Handle the SCPI *TST? command.
 * \param context   Active SCPI context
 * \return  Success or failure
```

```
 */
scpi_result_t SCPI_Test(scpi_t * context) {
    (void) context;
    fprintf(stderr, "**Test\r\n");
    return SCPI_RES_OK;
}

/**
 * Handle the SCPI *RST command.
 * \param context   Active SCPI context
 * \return  Success or failure
 */
scpi_result_t SCPI_Reset(scpi_t * context) {
    (void) context;

    // Here, we are going to reset the entire chip. This requires writing to
    // the RSTC (reset controller) control registers.

    // Zero out the control register first, so we don't reset prematurely
    RSTC->RSTC_CR = 0;

    // Set the mode register
    RSTC->RSTC_MR = 0
        | RSTC_MR_KEY_PASSWD    // Chip won't reset without the password
        ;

    // Set the control register
    RSTC->RSTC_CR = 0
        | RSTC_CR_KEY_PASSWD    // Chip won't reset without the password
        | RSTC_CR_PROCRST       // Reset processor
        ;

    // This return won't happen, of course :D
    return SCPI_RES_OK;
}
```

```
/**
 * \file
 *
 * \brief SCPI command declarations
 */

/*-
 * Modified heavily in 2014 by wcp52.
 * Copyright (c) 2012-2013 Jan Breuer,
 *
 * All Rights Reserved
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 * 1. Redistributions of source code must retain the above copyright notice,
 *    this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHORS ''AS IS'' AND ANY EXPRESS OR
 * IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
 * OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN
 * IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */


#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "scpi/scpi.h"
#include "scpi-def.h"
#include "scpi-test.h"
#include "scpi-lowlevel.h"

static const scpi_command_t scpi_commands[] = {
    /* IEEE Mandated Commands (SCPI std V1999.0 4.1.1) */
    { .pattern = "*CLS", .callback = SCPI_CoreCls,},
    { .pattern = "*ESE", .callback = SCPI_CoreEse,},
    { .pattern = "*ESE?", .callback = SCPI_CoreEseQ,},
    { .pattern = "*ESR?", .callback = SCPI_CoreEsrQ,},
    { .pattern = "*IDN?", .callback = SCPI_CoreIdnQ,},
    { .pattern = "*OPC", .callback = SCPI_CoreOpc,},
    { .pattern = "*OPC?", .callback = SCPI_CoreOpcQ,},
    { .pattern = "*RST", .callback = SCPI_CoreRst,},
    { .pattern = "*SRE", .callback = SCPI_CoreSre,},
    { .pattern = "*SRE?", .callback = SCPI_CoreSreQ,},
    { .pattern = "*STB?", .callback = SCPI_CoreStbQ,},
    { .pattern = "*TST?", .callback = SCPI_CoreTstQ,},
    { .pattern = "*WAI", .callback = SCPI_CoreWai,},

    /* Required SCPI commands (SCPI std V1999.0 4.2.1) */
    {.pattern = "SYSTem:ERRor[:NEXT]?", .callback = SCPI_SystemErrorNextQ,},
    {.pattern = "SYSTem:ERRor:COUNt?", .callback = SCPI_SystemErrorCountQ,},
    {.pattern = "SYSTem:VERSion?", .callback = SCPI_SystemVersionQ,},

    //{.pattern = "STATus:OPERation?", .callback = scpi_stub_callback,},
    //{.pattern = "STATus:OPERation:EVENt?", .callback = scpi_stub_callback,},
    //{.pattern = "STATus:OPERation:CONDition?", .callback = scpi_stub_callback,},
    //{.pattern = "STATus:OPERation:ENABle", .callback = scpi_stub_callback,},
    //{.pattern = "STATus:OPERation:ENABle?", .callback = scpi_stub_callback,},
```

```
    {.pattern = "STATus:QUEStionable[:EVENt]?", .callback = SCPI_StatusQuestionableEventQ
,},
    //{.pattern = "STATus:QUEStionable:CONDition?", .callback = scpi_stub_callback,},
    {.pattern = "STATus:QUEStionable:ENABle", .callback = SCPI_StatusQuestionableEnable,}
,
    {.pattern = "STATus:QUEStionable:ENABle?", .callback = SCPI_StatusQuestionableEnableQ
,},

    {.pattern = "STATus:PRESet", .callback = SCPI_StatusPreset,},

    // Low-level
    {.pattern = "LOWlevel:SETpin", .callback = LOWLEVEL_PIN_ACTION,},
    {.pattern = "LOWlevel:CLRpin", .callback = LOWLEVEL_PIN_ACTION,},
    {.pattern = "LOWlevel:GETpin", .callback = LOWLEVEL_PIN_ACTION,},
    {.pattern = "LOWlevel:LISTpins", .callback = LOWLEVEL_LIST_PINS,},

    /* Test commands */
    {.pattern = "Test:SPI", .callback = TEST_SPI,},
    {.pattern = "Test:INIF", .callback = TEST_INIF,}, /* Init interface */
    {.pattern = "Test:INCK", .callback = TEST_INCK,}, /* Init clock */
    {.pattern = "Test:FREQ", .callback = TEST_FREQ,},
    {.pattern = "Test:PHASE", .callback = TEST_PHASE,},
    {.pattern = "Test:AMPlitude", .callback = TEST_AMPLITUDE,},
    {.pattern = "Test:SAMple", .callback = TEST_SAMPLE,},
    {.pattern = "Test:CHannel", .callback = TEST_CHANNEL,},
    SCPI_CMD_LIST_END
};

static scpi_interface_t scpi_interface = {
    .error = SCPI_Error,
    .write = SCPI_Write,
    .control = SCPI_Control,
    .flush = SCPI_Flush,
    .reset = SCPI_Reset,
    .test = SCPI_Test,
};

#define SCPI_INPUT_BUFFER_LENGTH 256
static char scpi_input_buffer[SCPI_INPUT_BUFFER_LENGTH];
static scpi_reg_val_t scpi_regs[SCPI_REG_COUNT];
scpi_t G_SCPI_CONTEXT = {
    .cmdlist = scpi_commands,
    .buffer = {
        .length = SCPI_INPUT_BUFFER_LENGTH,
        .data = scpi_input_buffer,
    },
    .interface = &scpi_interface,
    .registers = scpi_regs,
    .units = scpi_units_def,
    .special_numbers = scpi_special_numbers_def,
    .idn = {"WCP52", "GPA1", "1", "0"},
};
```

```c
/**
 * @file
 * SCPI LOWlevel:* commands
 */

// Atmel ASF includes
#include <pio.h>
#include <string.h>

#include "scpi/scpi.h"
#include "scpi-lowlevel.h"
#include "conf_board.h"
#include "util.h"

/**
 * SCPI pin action: set
 */
scpi_result_t pin_action_set (uint32_t pin, uint32_t flags, const char *name)
{
    if ((flags & PIO_TYPE_Msk) != PIO_OUTPUT_0 && (flags & PIO_TYPE_Msk) != PIO_OUTPUT_1)
 {
        printf ("Pin '%s' is not an output!\r\n", name);
        return SCPI_RES_ERR;
    }

    pio_set_pin_high(pin);
    return SCPI_RES_OK;
}

/**
 * SCPI pin action: clr
 */
scpi_result_t pin_action_clr (uint32_t pin, uint32_t flags, const char *name)
{
    if ((flags & PIO_TYPE_Msk) != PIO_OUTPUT_0 && (flags & PIO_TYPE_Msk) != PIO_OUTPUT_1)
 {
        printf ("Pin '%s' is not an output!\r\n", name);
        return SCPI_RES_ERR;
    }

    pio_set_pin_low(pin);
    return SCPI_RES_OK;
}

/**
 * SCPI pin action: get
 */
scpi_result_t pin_action_get (uint32_t pin, uint32_t flags, const char *name)
{
    (void) flags;
    (void) name;
    printf ("%d\r\n", pio_get_pin_value(pin) ? 1 : 0);
    return SCPI_RES_OK;
}

/**
 * SCPI: low-level pin actions
 * LOWlevel:SETpin NAME
 * LOWlevel:CLRpin NAME
 * LOWlevel:GETpin NAME
 *
 * @param context - active SCPI context
 * @return success or failure
 */
scpi_result_t LOWLEVEL_PIN_ACTION (scpi_t *context)
{
    char const *str;
    size_t len;
    scpi_result_t (*pin_action) (uint32_t, uint32_t, const char *);
```

```
    if (!SCPI_ParamString(context, &str, &len, true)) {
        return SCPI_RES_ERR;
    }

    if (SCPI_IsCmd(context, "LOWlevel:SETpin")) {
        pin_action = &pin_action_set;
    } else if (SCPI_IsCmd(context, "LOWLEVEL:CLRpin")) {
        pin_action = &pin_action_clr;
    } else {
        pin_action = &pin_action_get;
    }

#define buflen 32
    char buffer[buflen];
    if (len > buflen - 1) {
        puts ("String too long\r");
        return SCPI_RES_ERR;
    }

    memcpy(buffer, str, len);
    buffer[len] = 0;

    bool found_pin = false;
    size_t i;
    for (i = 0; PIN_TABLE[i].description; ++i) {
        if (!PIN_TABLE[i].index_str) continue; // Pin group, not pin
        if (!strcmp (PIN_TABLE[i].pin_name_str, buffer)) {
            pin_action(PIN_TABLE[i].index, PIN_TABLE[i].flags, buffer);
            found_pin = true;
            break;
        }
    }

    if (!found_pin) {
        printf ("Pin '%s' not found!\r\n", buffer);
        return SCPI_RES_ERR;
    } else {
        return SCPI_RES_OK;
    }
}

scpi_result_t LOWLEVEL_LIST_PINS (scpi_t *context)
{
    (void) context;

    size_t i;
    for (i = 0; PIN_TABLE[i].description; ++i) {
        if (PIN_TABLE[i].index_str) {
            // Pin
            printf ("%-20s %-30s %s\r\n",
                    PIN_TABLE[i].pin_name_str,
                    PIN_TABLE[i].flags_str,
                    PIN_TABLE[i].description);
        } else {
            // Pin group
            printf ("==== %s ====\r\n", PIN_TABLE[i].description);
        }
    }

    return SCPI_RES_OK;
}
```

```c
/**
 * \file
 * \brief SCPI Test:* commands
 */

// Atmel ASF includes
#include <pio.h>
#include <spi.h>

#include "scpi/scpi.h"
#include "scpi-test.h"
#include "conf_board.h"
#include "synth.h"
#include "acquisition.h"
#include "util.h"

/**
 * SCPI: Send arbitrary SPI data.
 * Test:SPI DATA
 *
 * \param context   Active SCPI context
 * \return  Success or failure
 */
scpi_result_t TEST_SPI(scpi_t *context)
{
    int32_t val;
    if (!SCPI_ParamInt(context, &val, true)) {
        return SCPI_RES_ERR;
    }
    uint8_t val_byte = (uint8_t) val;
    printf("Transmitting value %02x\r\n", val_byte);
    spi_write(SPI_MASTER_BASE, val_byte, 0, 0);
    return SCPI_RES_OK;
}

/**
 * SCPI: Initialize DDS interface
 * Test:INIF
 *
 * \param context   Active SCPI context
 * \return  Success or failure
 */
scpi_result_t TEST_INIF(scpi_t *context)
{
    (void) context;
    synth_initialize_interface();
    return SCPI_RES_OK;
}

/**
 * SCPI: Initialize DDS system clock
 * Test:INCK
 *
 * \param context   Active SCPI context
 * \return  Success or failure
 */
scpi_result_t TEST_INCK(scpi_t *context)
{
    (void) context;
    synth_initialize_clock();
    return SCPI_RES_OK;
}

/**
 * SCPI: Set DDS frequency
 * Test:FREQ channel, frequency
 *
 * \param context   Active SCPI context
 * \return  Success or failure
 */
```

```
scpi_result_t TEST_FREQ(scpi_t *context)
{
    int32_t ch;
    unsigned ch_uns;
    scpi_number_t freq;

    if (!SCPI_ParamInt(context, &ch, true)) {
        return SCPI_RES_ERR;
    }

    if (!SCPI_ParamNumber(context, &freq, true)) {
        return SCPI_RES_ERR;
    }

    ch_uns = ch;

    printf("Setting frequency %u to %f\r\n", ch_uns, freq.value);
    synth_set_frequency(ch_uns, freq.value);
    return SCPI_RES_OK;
}

/**
 * SCPI: Set DDS phase
 * Test:PHASE channel, phase
 *
 * \param context   Active SCPI context
 * \return  Success or failure
 */
scpi_result_t TEST_PHASE(scpi_t *context)
{
  int32_t ch;
  unsigned ch_uns;
  scpi_number_t phase;


    if (!SCPI_ParamInt(context, &ch, true)) {
        return SCPI_RES_ERR;
    }

    if (!SCPI_ParamNumber(context, &phase, true)) {
        return SCPI_RES_ERR;
    }

    ch_uns = ch;
    printf("Setting phase %u to %f\r\n", ch_uns, phase.value);
    synth_set_phase(ch_uns, phase.value);
    return SCPI_RES_OK;

}

/**
 * SCPI: Set DDS amplitude
 * Test:AMPlitude channel, amp
 *
 * \param context   Active SCPI context
 * \return  Success or failure
 */
scpi_result_t TEST_AMPLITUDE(scpi_t *context)
{
  int32_t ch;
  unsigned ch_uns;
  scpi_number_t amplitude;


    if (!SCPI_ParamInt(context, &ch, true)) {
        return SCPI_RES_ERR;
    }

    if (!SCPI_ParamNumber(context, &amplitude, true)) {
        return SCPI_RES_ERR;
```

```
    }

    ch_uns = ch;
    printf("Setting amplitude %u to %f\r\n", ch_uns, amplitude.value);
    synth_set_amplitude(ch_uns, amplitude.value);
    return SCPI_RES_OK;

}

/**
 * SCPI: Sample the input.
 * Test:SAMple
 *
 * \param context   Active SCPI context
 * \return  Success or failure
 */
scpi_result_t TEST_SAMPLE(scpi_t *context)
{
    int32_t num_samples;
    if (!SCPI_ParamInt(context, &num_samples, true)) {
        return SCPI_RES_ERR;
    }

    printf ("%f\r\n", acq_get_values (num_samples));
    return SCPI_RES_OK;
}

/**
 * SCPI: Set active input channel.
 * TEST:CHannel
 *
 * \param context   Active SCPI context
 * \return  Success or failure
 */
scpi_result_t TEST_CHANNEL(scpi_t *context)
{
  int32_t ch;


    if (!SCPI_ParamInt(context, &ch, true)) {
        return SCPI_RES_ERR;
    }


    util_set_pin (GPIO_CHANSEL, ch);

    return SCPI_RES_OK;
}
```

```
/**
 * \file
 * DDS synthesizer control functions
 */

// Atmel ASF includes
#include <delay.h>
#include <pio.h>
#include <spi.h>

#include "conf_board.h"
#include <string.h>

#include "synth.h"

// Synthesizer registers
uint8_t G_FR1[FR1_LEN];
uint8_t G_FR2[FR2_LEN];
uint8_t G_CFR[CFR_LEN];
uint8_t G_CFTW0[CFTW_LEN];
uint8_t G_CFTW1[CFTW_LEN];

uint8_t G_CPOW0[CPOW_LEN];
uint8_t G_CPOW1[CPOW_LEN];


uint8_t G_CACR0[CACR_LEN];
uint8_t G_CACR1[CACR_LEN];


/**
 * Reset the DDS IO system. This aborts a current IO cycle and prepares for
 * the next one.
 */
static void syncio(void)
{
    pio_set_pin_high(GPIO_DDS_SYNCIO);
    pio_set_pin_low(GPIO_DDS_SYNCIO);
}

/**
 * Commit loaded data into the DDS registers.
 */
static void io_update(void)
{
        pio_set_pin_high(GPIO_DDS_IOUPDATE);
        pio_set_pin_low(GPIO_DDS_IOUPDATE);
}

/**
 * Wait for a SPI transaction to finish.
 */
static void spi_wait(void)
{
        while (!spi_is_tx_empty(SPI_MASTER_BASE));
}

/**
 *  Send a control register to the DDS device.
 *  \param addr     Register address
 *  \param data     Register data array
 *  \param data_length  Length of data array
 */
static void send_control_register(
        uint8_t addr, const uint8_t *data, size_t data_length)
{
    pio_set_pin_low(GPIO_DDS_nCS);
    syncio();
    spi_write(SPI_MASTER_BASE, addr, 0, 0);
    for (size_t i = 0; i < data_length; ++i) {
```

```
        spi_write(SPI_MASTER_BASE, data[i], 0, 0);
    }
    spi_wait();
    pio_set_pin_high(GPIO_DDS_nCS);
    io_update();
}

/**
 * Send a channel register to the DDS device.
 * @param addr  Register address
 * @param data  Register data array
 * @param data_len      Length of data array
 * @param channel_num   Channel number (0 or 1)
 */
static void send_channel_register(
        uint8_t addr, const uint8_t *data,
        size_t data_length, unsigned channel_num)
{
    if (channel_num > 1) return;

    pio_set_pin_low(GPIO_DDS_nCS);
    syncio();

    // First, set the proper 'channel enable' bit
    spi_write(SPI_MASTER_BASE, 0x00, 0, 0);
    spi_write(SPI_MASTER_BASE, channel_num ? 0x42 : 0x82, 0, 0);

    // Now, send the data
    spi_write(SPI_MASTER_BASE, addr, 0, 0);
    for (size_t i = 0; i < data_length; ++i) {
        spi_write(SPI_MASTER_BASE, data[i], 0, 0);
    }

    spi_wait();
    pio_set_pin_high(GPIO_DDS_nCS);
    io_update();
}

/**
 * Initialize the DDS interface.
 */
void synth_initialize_interface(void)
{
    // Ensure sane pin defaults
        pio_set_pin_low(GPIO_DDS_PWRDN);
        pio_set_pin_low(GPIO_DDS_nCS);
        pio_set_pin_low(GPIO_DDS_IOUPDATE);

    // Make sure to delay after PWRDN goes low.
    // No idea how long! I can't find it in the datasheet...
    delay_ms(50);

    // Perform a master reset
        pio_set_pin_high(GPIO_DDS_MRST);
        pio_set_pin_low(GPIO_DDS_MRST);

    // Configure standard SPI mode
    syncio();
        spi_write(SPI_MASTER_BASE, 0x00, 0, 0);
        spi_write(SPI_MASTER_BASE, 0x02, 0, 0);

    spi_wait();
    pio_set_pin_high(GPIO_DDS_nCS);
    io_update();
}

/**
 * Initialize the DDS system clock.
 */
void synth_initialize_clock(void)
```

```
{
    memset(G_FR1, 0, FR1_LEN);
    REGSET(G_FR1, FR1_VCOGAIN, 1); /* VCO gain set for high frequency */
    REGSET(G_FR1, FR1_PLLRATIO, 20); /* PLL: 25 MHz crystal * 20 = 500 MHz */

    send_control_register(FR1_ADDR, G_FR1, FR1_LEN);

    memset(G_FR2, 0, FR2_LEN);
    REGSET(G_FR2, FR2_ALL_AUTOCLEAR_PHASE, 1); /* Autoclear phase accumulator on IOup*/
    send_control_register(FR2_ADDR, G_FR2, FR2_LEN);
}

/**
 * Set the DDS frequency on a channel.
 * \param channel   Channel number, either 0 or 1
 * \param freq      Frequency in Hz.
 */
void synth_set_frequency(unsigned channel, double freq)
{
    if (channel > 1) return;

    // First, convert 'freq' to a frequency tuning word.
    // From AD9958 datasheet, page 18: Fout = (FTW)(Fsys) / 2^32
    // So FTW = 2^32 * Fout / Fsys

    double ftw = (4294967296. * freq) / SYSCLK_FREQ;
    uint32_t ftw32 = (uint32_t) ftw;

    // Prepare the FTW as an array of four bytes
    uint8_t *cftw = channel ? G_CFTW1 : G_CFTW0;
    cftw[0] = (ftw32 & 0xff000000uL) >> 24;
    cftw[1] = (ftw32 & 0x00ff0000uL) >> 16;
    cftw[2] = (ftw32 & 0x0000ff00uL) >> 8;
    cftw[3] = (ftw32 & 0x000000ffuL);

    // Transmit the bytes
    send_channel_register(CFTW_ADDR, cftw, CFTW_LEN, channel);
}


/**
 * Set the DDS phase on a channel.
 * \param channel   Channel number, either 0 or 1
 * \param freq      Phase in degrees.
 */
void synth_set_phase(unsigned channel, double phase)
{

  if (channel > 1) return;
  // convert "phase" to phase offset word
  // equation according to datasheet page 18
  double pow =  (16384. * phase) / 360 ;
  uint16_t pow14 = (uint16_t)pow & 0x03FFF;

   //prepare pow as array of 2 bytes
    uint8_t *cpow = channel ? G_CPOW1 : G_CPOW0;
    cpow[0] = (pow14 & 0x0000ff00uL) >> 8;
    cpow[1] = (pow14 & 0x000000ffuL);

    send_channel_register(CPOW_ADDR, cpow, CPOW_LEN, channel);

}

/**
 * Set the DDS amplitude on a channel.
 * \param channel   Channel number, either 0 or 1
 * \param freq      Amplitude
 */
void synth_set_amplitude(unsigned channel, double amplitude)
{
```

```
     if (channel > 1) return;
     double acr =  1023 * amplitude ;
     //
     uint16_t acr16 = (uint16_t)acr & 0x03FF;
     //setting the amplitude enable bit high
     acr16 |= 0x1000;

    uint8_t *cacr = channel ? G_CACR1 : G_CACR0;
    cacr[0] = 0; // Ramp rate
    cacr[1] = (acr16 & 0x0000ff00uL) >> 8;
    cacr[2] = (acr16 & 0x000000ffuL);

    send_channel_register(CACR_ADDR, cacr, CACR_LEN, channel);
}
```

```c
// Atmel ASF includes
#include <pio.h>
#include <stdio_usb.h>

#include <inttypes.h>
#include "udi_cdc.h"
#include "usb_protocol_cdc.h"
#include "conf_board.h"

volatile bool G_CDC_ENABLED = false;

void main_sof_action(void) { }

void main_resume_action(void) { }

void main_suspend_action(void) { }

bool callback_cdc_enable(uint8_t port)
{
    (void) port;
    stdio_usb_enable();
    pio_set_pin_high(GPIO_LED2);
    G_CDC_ENABLED = true;
    return true;
}

void callback_cdc_disable(uint8_t port)
{
    (void) port;
    G_CDC_ENABLED = false;
    pio_set_pin_low(GPIO_LED2);
    stdio_usb_disable();
}

void callback_cdc_set_coding_ext(uint8_t port, usb_cdc_line_coding_t *cfg)
{
    (void) port;
    (void) cfg;
}

void callback_cdc_set_dtr(uint8_t port, bool enable)
{
    (void) port;
    (void) enable;
}

void callback_cdc_rx_notify(uint8_t port)
{
    (void) port;
}
```

```
/**
 * \file
 * Miscellaneous utilities
 */

#include "util.h"
#include "conf_board.h"

const struct pin_info PIN_TABLE[] = {

#define XPINGROUP(_grp) {.description = _grp},
#define XPIN(_pin, _idx, _flags, _descr) \
    {   .pin_name_str = #_pin, \
        .index_str = #_idx, \
        .flags_str = #_flags, \
        .description = _descr, \
        .index = PIO_ ## _idx ## _IDX, \
        .flags = (_flags) },
PIN_LIST
#undef XPINGROUP
#undef XPIN
    {NULL} // Sentinel

};
```

D PC Code

```
#!/usr/bin/env python

try:
    import tkinter as tk
except:
    import Tkinter as tk
import pygubu
import matplotlib
from tkinter import messagebox
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure
from bode import Bode

class GUI:
    def __init__(self, master):

        self.builder = builder = pygubu.Builder()
        builder.add_from_file("GUI.ui")
        master.title("USB Gain/Phase Analyzer")
        #master.rowconfigure(0, weight=1)
        #master.columnconfigure(0, weight=1)
        self.top = builder.get_object("top", master)

        #Set up entry boxes
        self.min_var = tk.DoubleVar()
        self.max_var = tk.DoubleVar()
        self.min_entry = tk.Entry(self.top, name="entry_min",textvariable=self.min_var, w
idth=10)
        self.min_entry.grid(column=1,row=1)
        self.max_entry = tk.Entry(self.top, name="entry_max", textvariable=self.max_var,w
idth=10)
        self.max_entry.grid(column=1,row=2)

        #set up canvas to hold plots
        self.freq_fig = Figure()
        self.phase_fig = Figure()
        self.freq_canvas = FigureCanvasTkAgg(self.freq_fig, master)
        self.phase_canvas = FigureCanvasTkAgg(self.phase_fig, master)

        #connect callbacks
        builder.connect_callbacks(self)
        self.calibrate = False

        self.bode = Bode()

    def on_plot(self):
        self.freq_fig.clear()
        self.phase_fig.clear()
        phase = self.builder.get_variable('do_phase').get()
        linear = self.builder.get_variable('do_linear').get()


        #If we have not calibrated, then we need to make sure that these
        #Bode attributes have been set
        if not self.calibrate:
            lower_bound = self.min_var.get()
            upper_bound = self.max_var.get()
            # need error here
            #while lower_bound != upper_bound:
            if lower_bound >= upper_bound:
                error = tk.messagebox.showerror("error","These values are not valid.\n Tr
y again min must be lower than max" )
            #        break
            self.bode.set_lower_bound(lower_bound)
            self.bode.set_upper_bound(upper_bound)
            self.bode.set_do_phase(phase)
            self.bode.generate_freqs()

        #Have bode generate data
        self.bode.run()
```

```
        #Get frequency response data from bode.
        freq_response = self.bode.get_freq_response_data()
        if self.calibrate:
            freq_plt = self.freq_fig.add_subplot(111, xlabel="Frequency", ylabel="Gain, c
alibrated")
            freq_calibrate_data = self.bode.get_freq_calibration_data()
            assert len(freq_calibrate_data) == len(freq_response), (len(freq_calibrate_da
ta), len(freq_response))
            for i in range(len(freq_response)):
                freq_response[i] = freq_response[i] - freq_calibrate_data[i]
        else:
            freq_plt = self.freq_fig.add_subplot(111, xlabel="Frequency", ylabel="Gain, u
ncalibrated")

        #Generate frequency response plot
        freqs_f = self.bode.get_freqs_f()
        if not linear:
            freq_plt.semilogx(freqs_f, freq_response)
        else:
            freq_plt.plot(freqs_f, freq_response)
        freq_plt.grid(True)
        self.freq_canvas.show()
        self.freq_canvas.get_tk_widget().place(relx=.26, rely=0.01,relheight=0.49,relwidt
h=0.7)

        #If we want phase, we go through the process again
        if phase:
            phase_response = self.bode.get_phase_response_data()
            if self.calibrate:
                phase_plt = self.phase_fig.add_subplot(111, xlabel="Frequency", ylabel="P
hase, calibrated")
                phase_calibrate_data = self.bode.get_phase_calibration_data()
                for i in range(len(phase_response)):
                    phase_response[i] = phase_response[i] - phase_calibrate_data[i]
            else:
                phase_plt = self.phase_fig.add_subplot(111, xlabel="Frequency", ylabel="P
hase, uncalibrated")

            freqs_p = self.bode.get_freqs_p()

            if not linear:
                phase_plt.semilogx(freqs_p, phase_response)
            else:
                phase_plt.plot(freqs_p, phase_response)

            phase_plt.grid(True)
            self.phase_canvas.show()
            self.phase_canvas.get_tk_widget().place(relx=.26,rely=0.505,relheight=0.49,re
lwidth=0.7)

    def on_calibrate(self):
        #set lower and upper bounds for bode
        lower_bound = self.min_var.get()
        upper_bound = self.max_var.get()
        self.bode.set_lower_bound(lower_bound)
        self.bode.set_upper_bound(upper_bound)
        #need error box here
        #while lower_bound != upper_bound:
        if lower_bound >= upper_bound:
            error = tk.messagebox.showerror("error","These values are not valid.\n Try ag
ain min must be lower than max")
        #        break
        #tell bode whether or not it will do a phase plot
        phase = self.builder.get_variable('do_phase').get()
        self.bode.set_do_phase(phase)

        #generate the frequencies to sample across
        self.bode.generate_freqs()
```

```
        #have bode perform calibration
        self.bode.calibrate()
        self.calibrate = True

if __name__ == "__main__":
    root = tk.Tk()
    gui = GUI(root)
    root.mainloop()
```

```
from serial_comm import *
from response import *
import numpy as np

class Bode:
    def __init__(self):
        self.s = connect_gpa()
        mc_init(self.s)
        synth_init(self.s)
        frontend_init(self.s)
        self.freq_calibration = []
        self.phase_calibration = []
        self.freq_response = []
        self.phase_response = []
        self.freqs_f = []
        self.freqs_p = []
        self.upper_bound = 0.0
        self.lower_bound = 0.0
        self.do_phase = False

    #set the upper bound of sampling frequencies
    def set_upper_bound(self, upper_bound):
        self.upper_bound = upper_bound

    #sets lower bound of sampling frequencies
    def set_lower_bound(self, lower_bound):
        self.lower_bound = lower_bound

    def set_do_phase(self, do_phase):
        self.do_phase = do_phase

    def get_freq_calibration_data(self):
        return self.freq_calibration

    def get_phase_calibration_data(self):
        return self.phase_calibration

    def get_freq_response_data(self):
        return self.freq_response

    def get_phase_response_data(self):
        return self.phase_response

    #generate lists of all sampling frequencies
    def generate_freqs(self):
        self.freqs_f = np.logspace(np.log10(self.lower_bound), np.log10(self.upper_bound)
, 60) # 1 kHz to 150 MHz
        if self.do_phase:
            self.freqs_p = np.logspace(np.log10(self.lower_bound), np.log10(self.upper_bo
und), 30) # 1 kHz to 150 MHz

    def get_freqs_f(self):
        return self.freqs_f

    def get_freqs_p(self):
        return self.freqs_p

    def calibrate(self):
        self.freq_calibration = get_freq_response(self.s, self.freqs_f)
        if self.do_phase:
            self.phase_calibration = get_phase_response(self.s, self.freqs_p)

    def run(self):
        self.freq_response = get_freq_response(self.s, self.freqs_f)
        if self.do_phase:
            self.phase_response = get_phase_response(self.s, self.freqs_p)
```

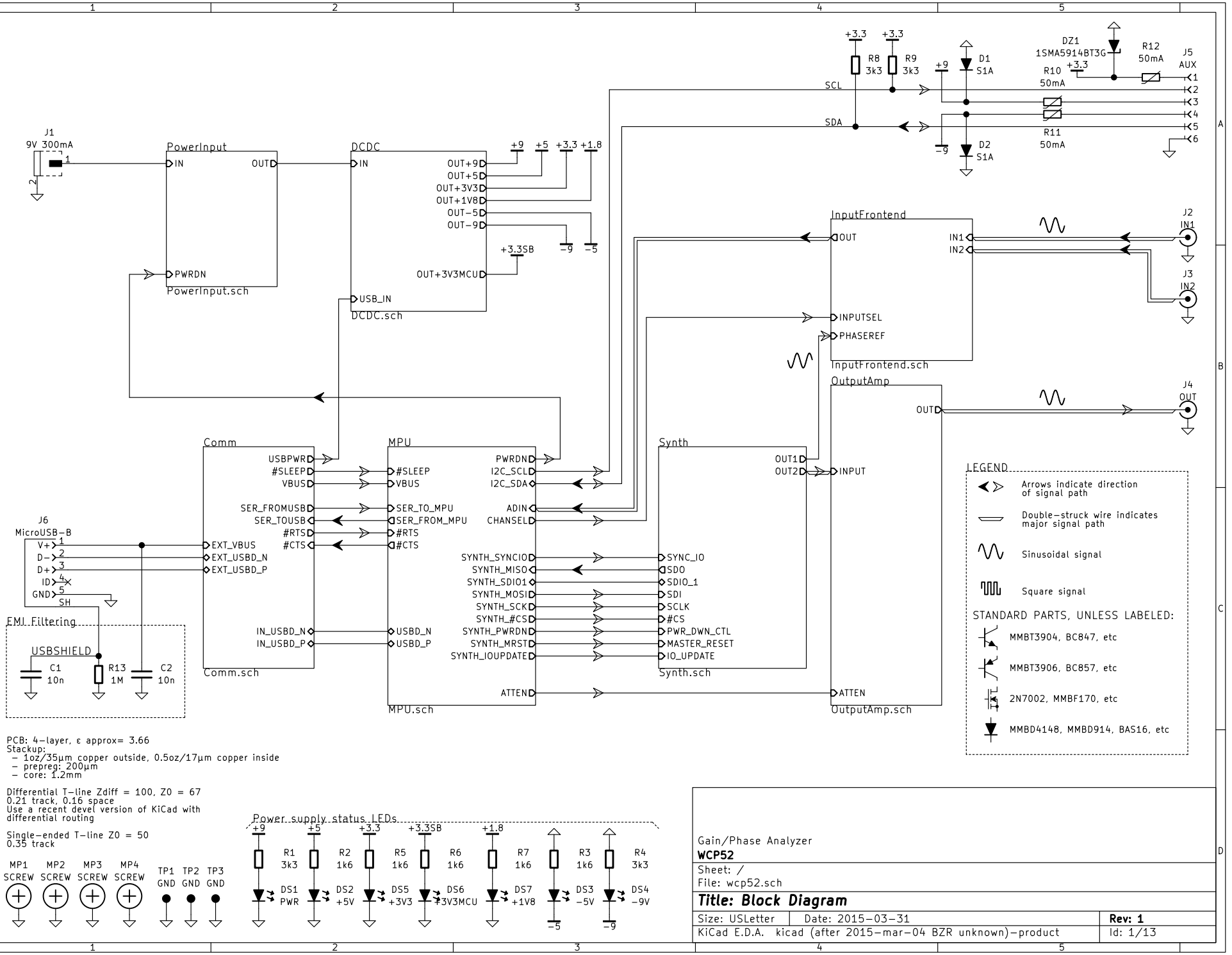
```
#!/usr/bin/python
import serial
import time
import sys
from serial_comm import getline
import serial_comm as defs

# The first few samples come out wrong. Repeat them
N_REPEAT = 6

def get_freq_response(s, freqs):
    print ("Collecting data... ")
    data = []

    freqs = list(freqs[:N_REPEAT]) + list(freqs)

    for i in freqs:
        nSamples = max(((1/i)*50)//1000000, 2048)
        s.write (("T:FREQ %d, %f\r\n" % (defs.CH_MAIN, i)).encode ('ascii'))
        getline (s)
        s.write(b"LOW:CLR GPIO_ATTEN\r\n")
        time.sleep(.005)
        s.write (("T:SAM %d\r\n" % nSamples).encode ('ascii'))
        level = float (getline (s))

        #if level >= 2900, attenuate the input signal
        if level >= 3500:
            print("this is running...")
            s.write(b"LOW:SET GPIO_ATTEN\r\n")
            time.sleep(.005)
            s.write (("T:SAM %d\r\n" % nSamples).encode ('ascii'))
            level = float (getline (s))
            db = level / (4095 * 24e-3 / 3.3) + 15
        else:
            db = level / (4095 * 24e-3 / 3.3)

        print ("%.2f Hz\t%.2f dB" % (i, db))
        data.append (db)

    # Remove repeated measurements
    data = data[N_REPEAT:]
    #data = [i-data[0] for i in data]
    return data


def get_phase_response(s, freqs):
    print ("Collecting phase data...")
    N_POINTS_PER_RANGE = 16
    PRECISION = 1.
    data = []
    freqs = list(freqs[:N_REPEAT]) + list(freqs)
    for i in freqs:
        # Set frequency. Then, search phases for a null
        s.write (("T:FREQ %d, %f\r\n" % (defs.CH_PHASE, i)).encode ('ascii'))
        getline (s)
        s.write (("T:FREQ %d, %f\r\n" % (defs.CH_MAIN, i)).encode ('ascii'))
        getline (s)
        s.write (("T:AMP %d, 1.0\r\n" % (defs.CH_PHASE)).encode ('ascii'))
        getline (s)
        s.write(b"LOW:SET GPIO_ATTEN\r\n")
        s.write (("T:AMP %d, 0.178\r\n" % (defs.CH_MAIN)).encode ('ascii'))
        getline (s)

        phase_bound_left = 0
        phase_bound_right = 360
        while phase_bound_right - phase_bound_left > PRECISION:
            width = phase_bound_right - phase_bound_left
            pitch = width / N_POINTS_PER_RANGE
            phases = [(i * pitch) + phase_bound_left for i in range (N_POINTS_PER_RANGE)]
            lowest_phase = None
```

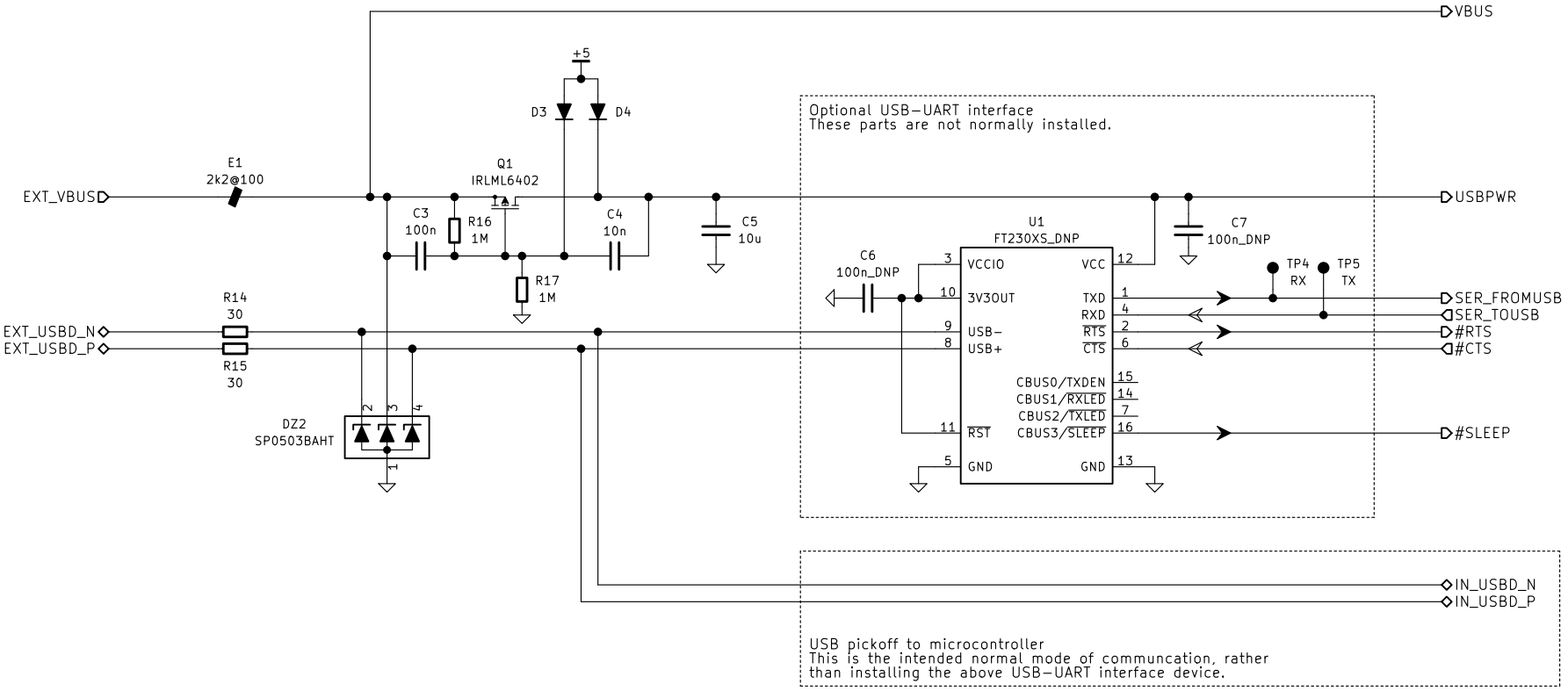
```
            lowest_amp = float("inf")
            lowest_i = None
            for phase_i, phase in enumerate(phases):
                nSamples = max(((1/i)*50)//1000000, 2048)
                phase = phase % 360.
                s.write (("T:PHASE %d, %f\r\n" % (defs.CH_PHASE, phase)).encode ('ascii')
)
                getline (s)
                s.write (("T:SAM %d\r\n" % nSamples).encode ('ascii'))
                level = float (getline (s))
                if level < lowest_amp:
                    lowest_amp = level
                    lowest_phase = phase
                    lowest_i = phase_i
                print (".", end='')
                sys.stdout.flush ()
            if lowest_i == 0:
                # Slide the range left
                width = phase_bound_right - phase_bound_left
                phase_bound_left -= width / 2
                phase_bound_right -= width / 2
            elif lowest_i == len(phases) - 1:
                # Slide the range right
                width = phase_bound_right - phase_bound_left
                phase_bound_left += width / 2
                phase_bound_right += width / 2
            else:
                # Narrow the range
                phase_bound_left = phases[lowest_i - 1]
                phase_bound_right = phases[lowest_i + 1]

        phases = list (range (0, 360, 40))
        lowest_phase -= 180.
        phase = lowest_phase

        # Smooth phase discontinuities
        if len(data):
            last_phase = data[-1]
            offset_phases = [phase + (360. * i) for i in range(-3,4)]
            phase_errors = [abs(i - last_phase) for i in offset_phases]

            best_phase, best_error = min (zip (offset_phases, phase_errors), key=lambda x
: x[1])
            phase = best_phase


        print ("%.2f Hz\t%f deg" % (i, phase))
        data.append (phase)
    data = data[N_REPEAT:]
    return data
```

```
#!/usr/bin/python
import serial
import time
import os

CH_MAIN = 0
CH_PHASE = 1
CH_IN_1 = 1
CH_IN_2 = 0

USB_ID = "1209:4757"


def getline (s):
    return s.readline ().decode ('ascii').strip ()

def printline (s, indent=2):
    """Print a line from serial, indented."""
    line = getline (s)
    print ((" " * indent) + line)
    return line

def read(fn):
    with open (fn) as f:
        return f.read().strip()

def usb_id(path):
    if not os.path.isfile (path + "/idProduct"):
        return None
    if not os.path.isfile (path + "/idVendor"):
        return None
    product = read(path + "/idProduct")
    vendor = read(path + "/idVendor")
    return vendor.lower() + ":" + product.lower()

def get_tty(path):
    for subdir in os.listdir(path):
        if not os.path.isdir(path + "/" + subdir):
            continue
        if os.path.isdir(path + "/" + subdir + "/tty"):
            this_subdir = subdir
            break
    return "/dev/" + os.listdir(os.path.join(path, subdir, "tty"))[0]


def find_device():
    DEVS = "/sys/bus/usb/devices"
    for dev_dir in os.listdir(DEVS):
        path = DEVS + "/" + dev_dir
        if usb_id(path) == USB_ID:
            return DEVS + "/" + dev_dir

def connect_gpa():
    devnode = find_device()
    if devnode is None:
        raise Exception("No GPA found!")
    tty = get_tty(devnode)
    print ("Connecting to GPA...")
    return serial.Serial (tty, 1, timeout=1)

def mc_init(s):
    print ("Initializing microcontroller...")
    s.write (b"*IDN?\r\n")
    idnstr = getline (s)
    print ("  identity string = " + idnstr)
    while not idnstr.startswith ("WCP52"):
        print ("  bad response, retrying")
        s.write (b"\r\n\r\n\r\n")
        time.sleep (0.5)
        s.flushOutput ()
```

```
        s.flushInput ()
        s.write (b"*IDN?\r\n")
        idnstr = getline (s)
        print ("  identity string = " + idnstr)
    print ("  OK")

def synth_init(s):
    print ("Initializing synthesizer...")
    s.write (b"T:INIF\r\n")
    s.write (b"*OPC?\r\n")
    getline (s)
    time.sleep (0.25)
    s.write (b"T:INCK\r\n")
    s.write (b"*OPC?\r\n")
    getline (s)
    time.sleep (0.25)
    print ("  OK")

def frontend_init(s):
    print ("Initializing frontend... ")
    s.write (("T:FREQ %d, 0\r\n" % (CH_PHASE)).encode('ascii'))
    printline (s)
    s.write (("T:AMP %d, 0\r\n" % (CH_PHASE)).encode('ascii'))
    printline (s)
    s.write (("T:CH %d\r\n" % (CH_IN_1)).encode('ascii'))
    s.write (b"*OPC?\r\n")
    getline (s)
    print ("  OK")
```

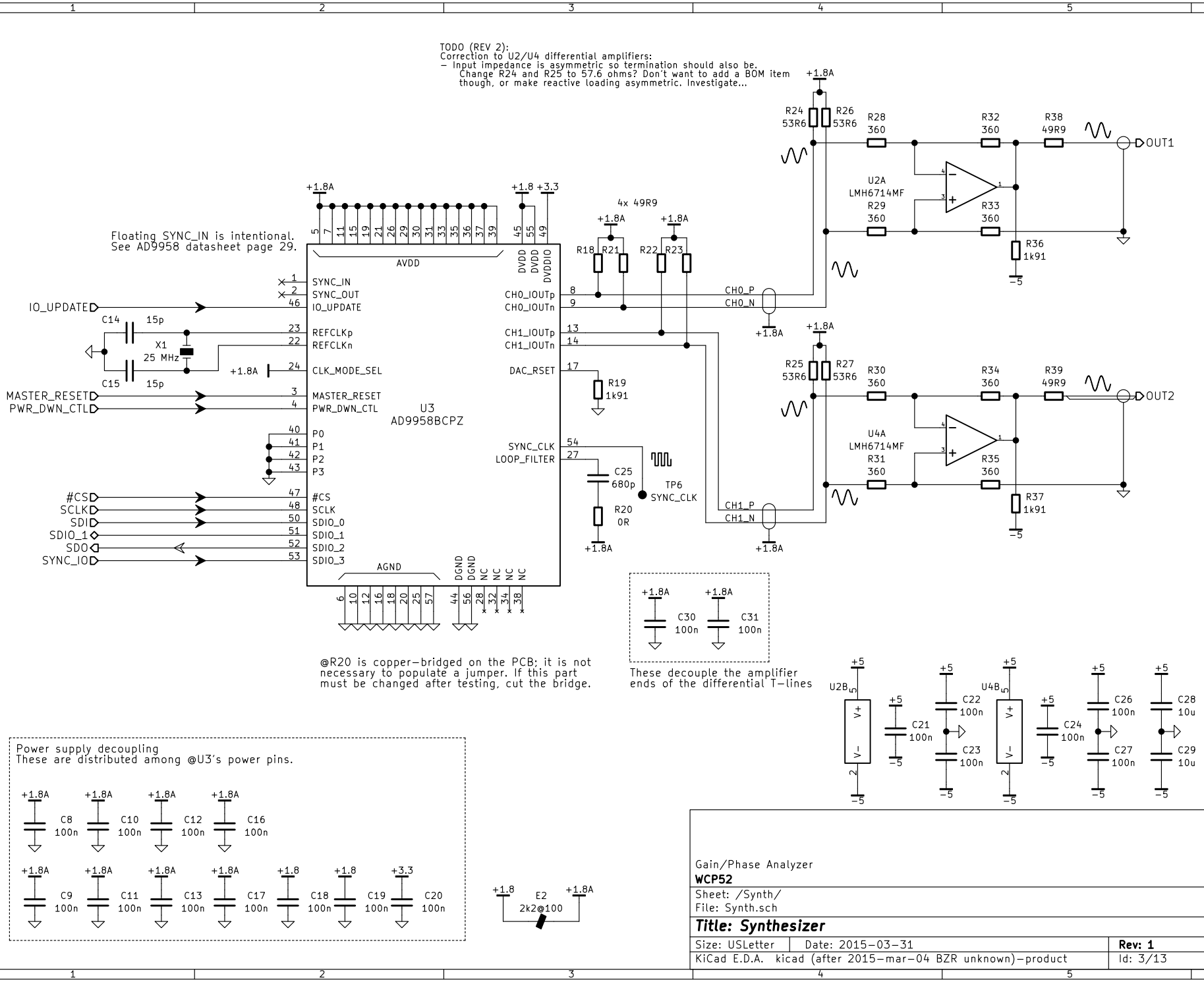
E CAD Drawings


J1
9V 300mA
PowerInput
IN
OUT
PWRDN
PowerInput.sch
DCDC
IN
OUT+9
OUT+5
OUT+3V3
OUT+1V8
OUT-5
OUT-9
OUT+3V3MCU
USB_IN
DCDC.sch
+3.3SB
+3.3
R8 3k3
R9 3k3
SCL
SDA
+9
D1 S1A
D2 S1A
DZ1 1SMA5914BT3G
R12 50mA
R10 50mA
R11 50mA
J5 AUX
InputFrontend
OUT
IN1
IN2
INPUTSEL
PHASEREF
InputFrontend.sch
J2 IN1
J3 IN2
OutputAmp
OUT
INPUT
ATTEN
OutputAmp.sch
J4 OUT
Comm
USBPWR
#SLEEP
VBUS
SER_FROMUSB
SER_TOUSB
#RTS
#CTS
EXT_VBUS
EXT_USBD_N
EXT_USBD_P
IN_USBD_N
IN_USBD_P
Comm.sch
MPU
PWRDN
I2C_SCL
I2C_SDA
ADIN
CHANSEL
SER_TO_MPU
SER_FROM_MPU
USBD_N
USBD_P
SYNTH_SYNCIO
SYNTH_MISO
SYNTH_SDIO1
SYNTH_MOSI
SYNTH_SCK
SYNTH_#CS
SYNTH_PWRDN
SYNTH_MRST
SYNTH_IOUPDATE
ATTEN
MPU.sch
Synth
OUT1
OUT2
INPUT
SYNC_IO
SDO
SDIO_1
SDI
SCLK
#CS
PWR_DWN_CTL
MASTER_RESET
IO_UPDATE
Synth.sch
J6
MicroUSB-B
V+
D-
D+
ID
GND
SH
EMI Filtering
USBSHIELD
C1 10n
R13 1M
C2 10n
LEGEND
Arrows indicate direction of signal path
Double-struck wire indicates major signal path
Sinusoidal signal
Square signal
STANDARD PARTS, UNLESS LABELED:
MMBT3904, BC847, etc
MMBT3906, BC857, etc
2N7002, MMBF170, etc
MMBD4148, MMBD914, BAS16, etc
PCB: 4-layer, ε approx= 3.66
Stackup:
- 1oz/35µm copper outside, 0.5oz/17µm copper inside
- prepreg: 200µm
- core: 1.2mm
Differential T-line Zdiff = 100, Z0 = 67
0.21 track, 0.16 space
Use a recent devel version of KiCad with differential routing
Single-ended T-line Z0 = 50
0.35 track
MP1 SCREW
MP2 SCREW
MP3 SCREW
MP4 SCREW
TP1 GND
TP2 GND
TP3 GND
Power supply status LEDs
R1 3k3
R2 1k6
R5 1k6
R6 1k6
R7 1k6
R3 1k6
R4 3k3
DS1 PWR
DS2 +5V
DS5 +3V3
DS6 +3V3MCU
DS7 +1V8
DS3 -5V
DS4 -9V
Gain/Phase Analyzer
WCP52
Sheet: /
File: wcp52.sch
Title: Block Diagram
Size: USLetter
Date: 2015-03-31
Rev: 1
KiCad E.D.A. kicad (after 2015-mar-04 BZR unknown)-product
Id: 1/13



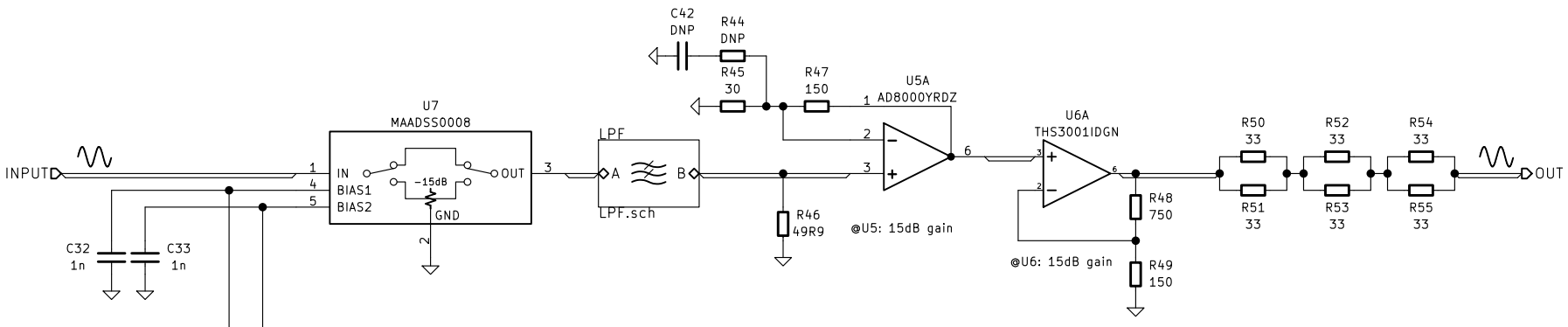
TODO (REV 2):
– Remove unnecessary U1, tidy routing

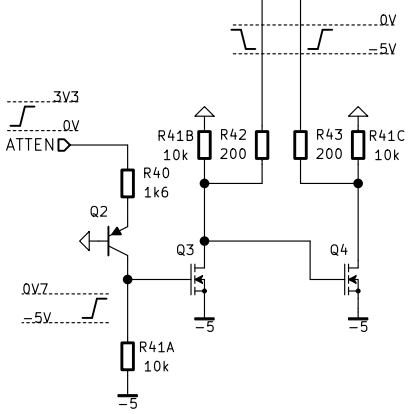
Gain/Phase Analyzer **WCP52**		
Sheet: /Comm/ File: Comm.sch		
Title: USB Communications		
Size: USLetter	Date: 2015–03–31	**Rev: 1**
KiCad E.D.A. kicad (after 2015–mar–04 BZR unknown)–product		Id: 2/13


TODO (REV 2):
Correction to U2/U4 differential amplifiers:
– Input impedance is asymmetric so termination should also be.
Change R24 and R25 to 57.6 ohms? Don't want to add a BOM item
though, or make reactive loading asymmetric. Investigate...
Floating SYNC_IN is intentional.
See AD9958 datasheet page 29.
U3
AD9958BCPZ
SYNC_IN
SYNC_OUT
IO_UPDATE
REFCLKp
REFCLKn
CLK_MODE_SEL
MASTER_RESET
PWR_DWN_CTL
P0
P1
P2
P3
#CS
SCLK
SDIO_0
SDIO_1
SDO
SDIO_2
SYNC_IO
SDIO_3
AVDD
DVDD
DVDD
DVDDIO
AGND
DGND
DGND
NC
CH0_IOUTp
CH0_IOUTn
CH1_IOUTp
CH1_IOUTn
DAC_RSET
SYNC_CLK
LOOP_FILTER
4x 49R9
R18
R21
R22
R23
R19
1k91
R20
0R
C25
680p
TP6
SYNC_CLK
C14
15p
C15
15p
X1
25 MHz
CH0_P
CH0_N
CH1_P
CH1_N
R24
53R6
R26
53R6
R28
360
R29
360
R32
360
R33
360
R36
1k91
R38
49R9
U2A
LMH6714MF
OUT1
R25
53R6
R27
53R6
R30
360
R31
360
R34
360
R35
360
R37
1k91
R39
49R9
U4A
LMH6714MF
OUT2
@R20 is copper–bridged on the PCB; it is not
necessary to populate a jumper. If this part
must be changed after testing, cut the bridge.
C30
100n
C31
100n
These decouple the amplifier
ends of the differential T–lines
U2B
U4B
C21
100n
C22
100n
C23
100n
C24
100n
C26
100n
C27
100n
C28
10u
C29
10u
Power supply decoupling
These are distributed among @U3's power pins.
C8
100n
C10
100n
C12
100n
C16
100n
C9
100n
C11
100n
C13
100n
C17
100n
C18
100n
C19
100n
C20
100n
E2
2k2@100
Gain/Phase Analyzer
WCP52
Sheet: /Synth/
File: Synth.sch
Title: Synthesizer
Size: USLetter
Date: 2015–03–31
Rev: 1
KiCad E.D.A. kicad (after 2015–mar–04 BZR unknown)–product
Id: 3/13

TODO (REV 2):
- Can we capacitively couple the signal into U5? The diff amp can produce significant offset. Cap-coupling 50R is impractical, but directly before U5 would allow removing the bias post-termination.
- Along those lines, can U5 become a transistor amplifier? BFR540?

@C42 and @R44 are optional compensation devices to boost gain slightly at high frequency, if post-fab testing shows that stray capacitance from layout results in a rolloff. In the first revision, do not populate them.

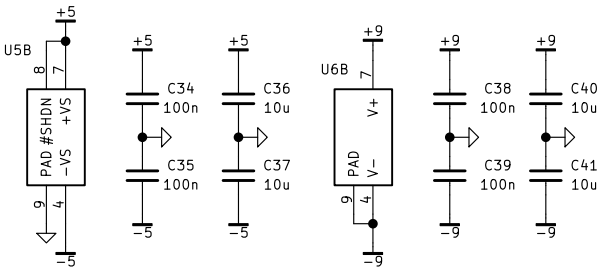




GaAs switch bias and truth table:
Low = -5V High = 0V

BIAS1	BIAS2	ATTENUATION
L	H	-15dB
H	L	0dB
L	L	Switches open
H	H	Both paths closed

CTRL	BIAS1	BIAS2	SELECTED
L	H	L	0dB
H	L	H	-15dB

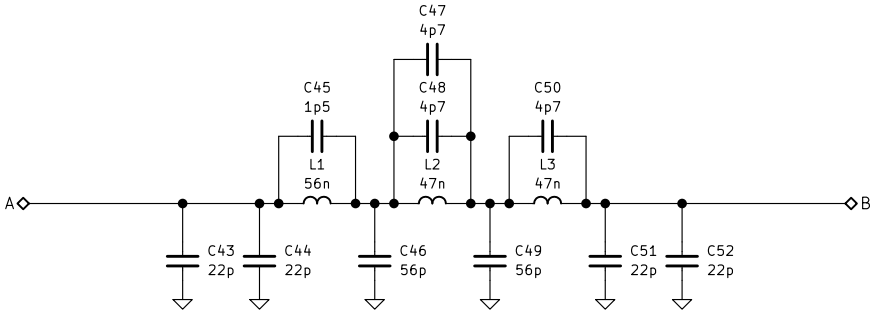


Gain/Phase Analyzer
WCP52

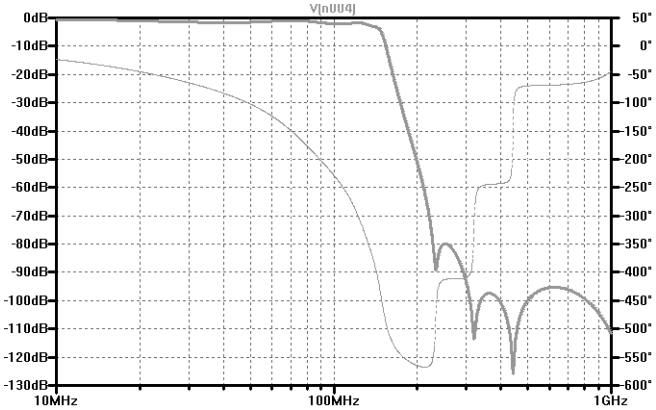
Sheet: /OutputAmp/
File: OutputAmp.sch

Title: Output Amplifier

Size: USLetter	Date: 2015-03-31	**Rev: 1**
KiCad E.D.A. kicad (after 2015-mar-04 BZR unknown)-product		Id: 4/13



Intended frequency response of filter:

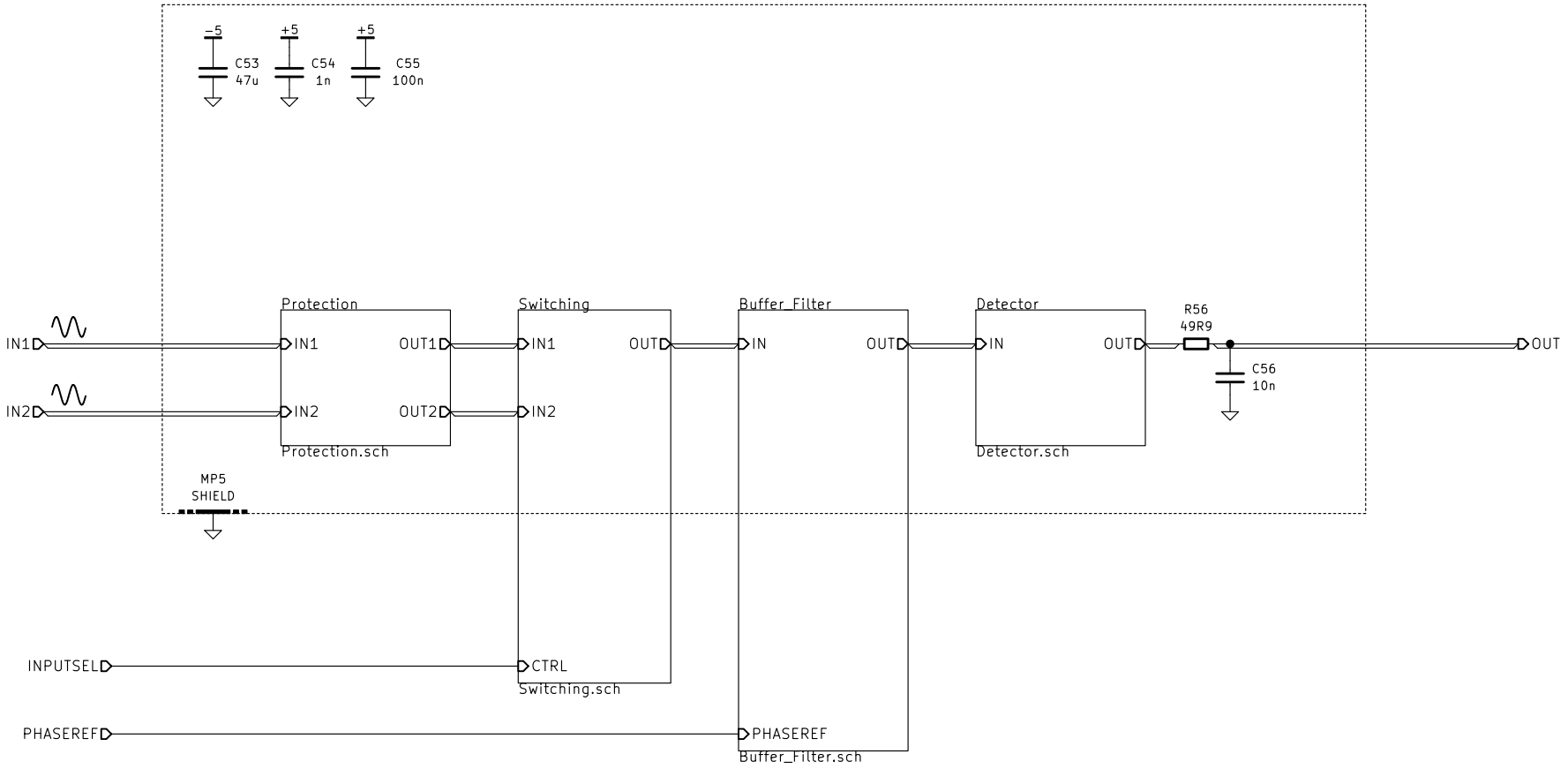


Gain/Phase Analyzer
WCP52

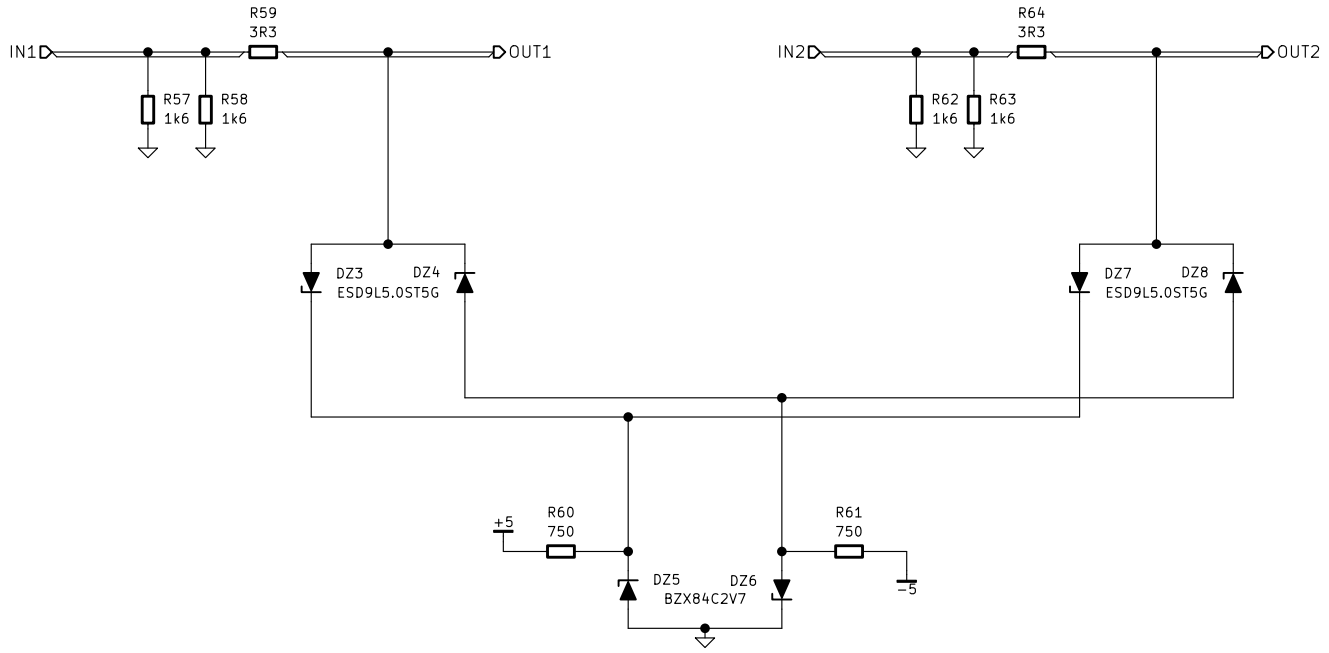
Sheet: /OutputAmp/LPF/
File: LPF.sch

Title: LPF for Output Amplifier

Size: USLetter	Date: 2015-03-31	**Rev: 1**
KiCad E.D.A. kicad (after 2015-mar-04 BZR unknown)-product		Id: 5/13



Gain/Phase Analyzer **WCP52**		
Sheet: /InputFrontend/ File: InputFrontend.sch		
Title: Input Frontend		
Size: USLetter	Date: 2015-03-31	**Rev: 1**
KiCad E.D.A. kicad (after 2015-mar-04 BZR unknown)-product		Id: 6/13

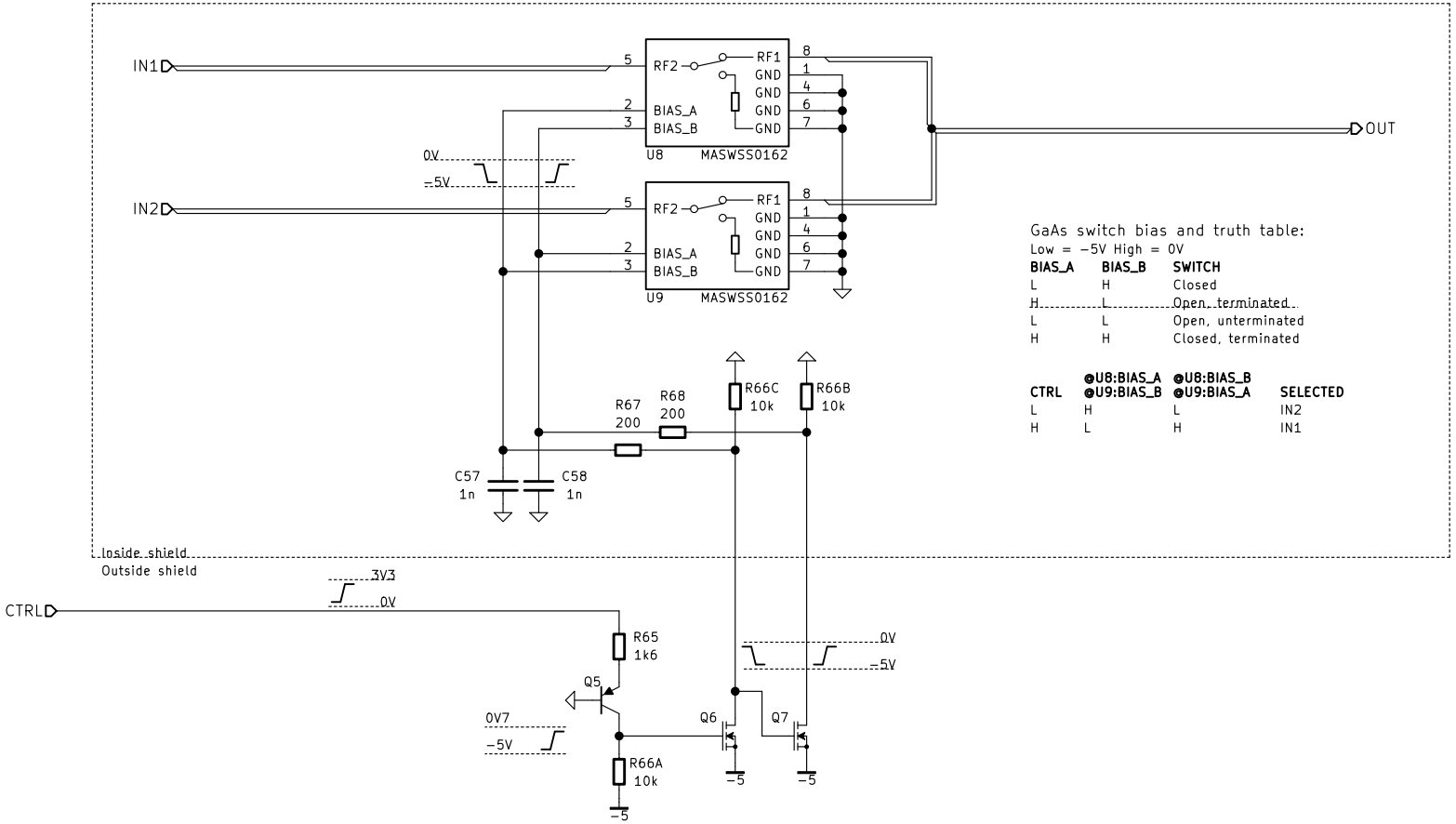


Gain/Phase Analyzer
WCP52

Sheet: /InputFrontend/Protection/
File: Protection.sch

Title: Input Protection

Size: USLetter	Date: 2015-03-31	**Rev: 1**
KiCad E.D.A. kicad (after 2015-mar-04 BZR unknown)-product		Id: 7/13



GaAs switch bias and truth table:
Low = −5V High = 0V

BIAS_A	BIAS_B	SWITCH
L	H	Closed
H	L	Open, terminated
L	L	Open, unterminated
H	H	Closed, terminated

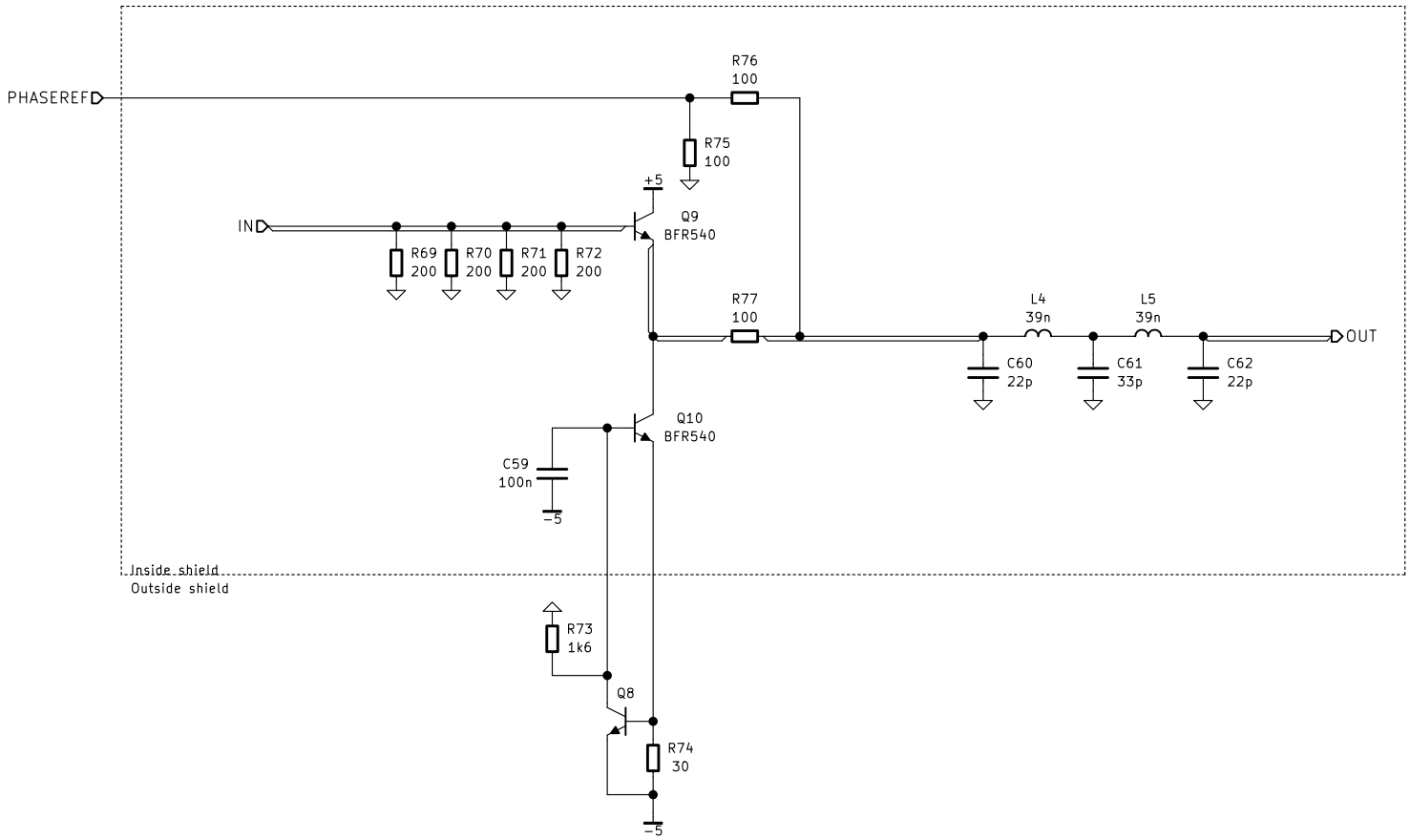
CTRL	@U8:BIAS_A @U9:BIAS_B	@U8:BIAS_B @U9:BIAS_A	SELECTED
L	H	L	IN2
H	L	H	IN1

Gain/Phase Analyzer
WCP52

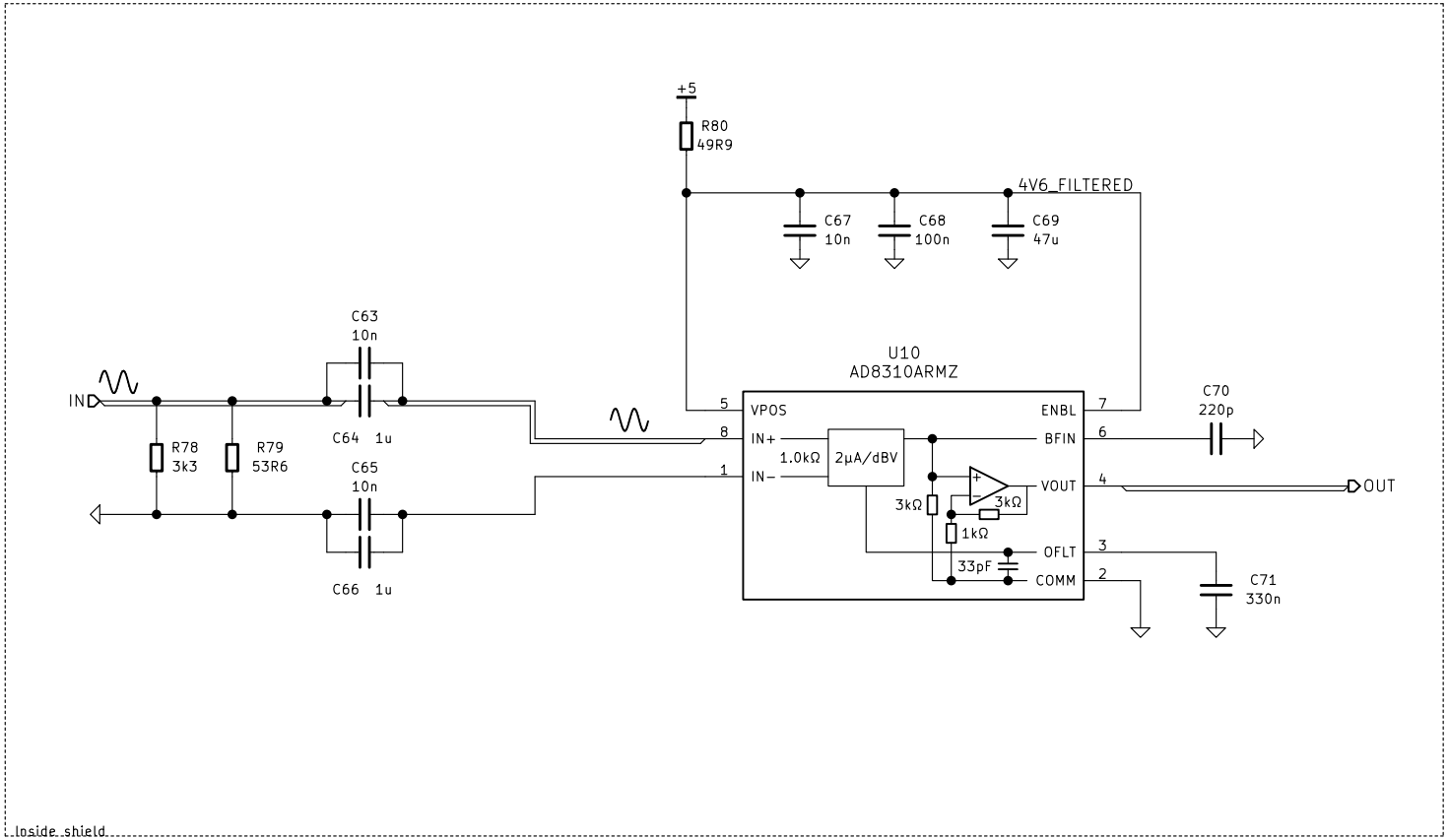
Sheet: /InputFrontend/Switching/
File: Switching.sch

Title: Input Switching

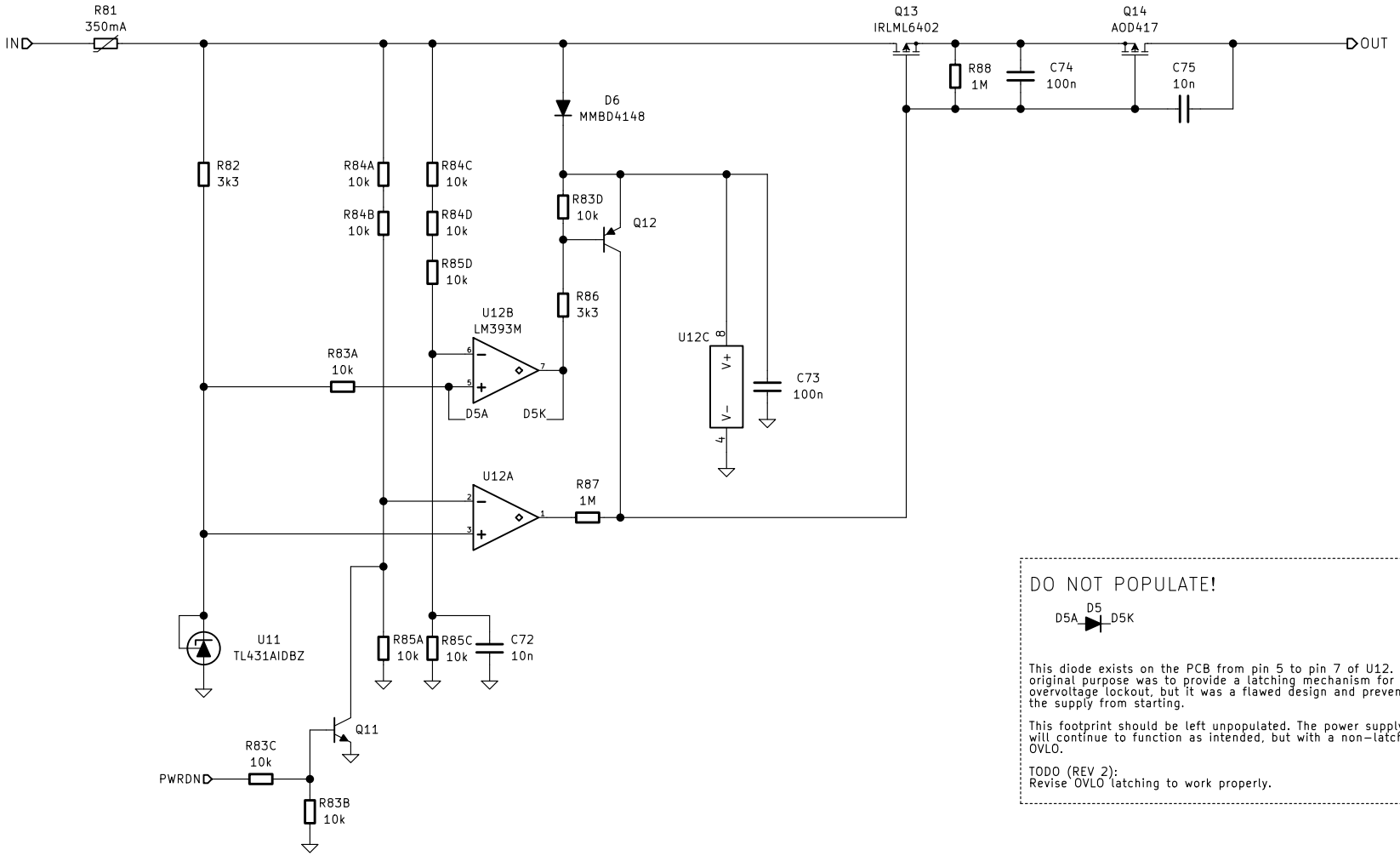
Size: USLetter	Date: 2015−03−31	**Rev: 1**
KiCad E.D.A. kicad (after 2015−mar−04 BZR unknown)−product		Id: 8/13



Gain/Phase Analyzer **WCP52**		
Sheet: /InputFrontend/Buffer_Filter/ File: Buffer_Filter.sch		
Title: Input Buffer and Filter		
Size: USLetter	Date: 2015-03-31	**Rev: 1**
KiCad E.D.A. kicad (after 2015-mar-04 BZR unknown)-product		Id: 9/13



Gain/Phase Analyzer **WCP52**		
Sheet: /InputFrontend/Detector/ File: Detector.sch		
Title: Logarithmic Detector		
Size: USLetter	Date: 2015-03-31	**Rev: 1**
KiCad E.D.A. kicad (after 2015-mar-04 BZR unknown)-product		Id: 10/13



DO NOT POPULATE!

This diode exists on the PCB from pin 5 to pin 7 of U12. The original purpose was to provide a latching mechanism for the overvoltage lockout, but it was a flawed design and prevented the supply from starting.

This footprint should be left unpopulated. The power supply will continue to function as intended, but with a non-latching OVLO.

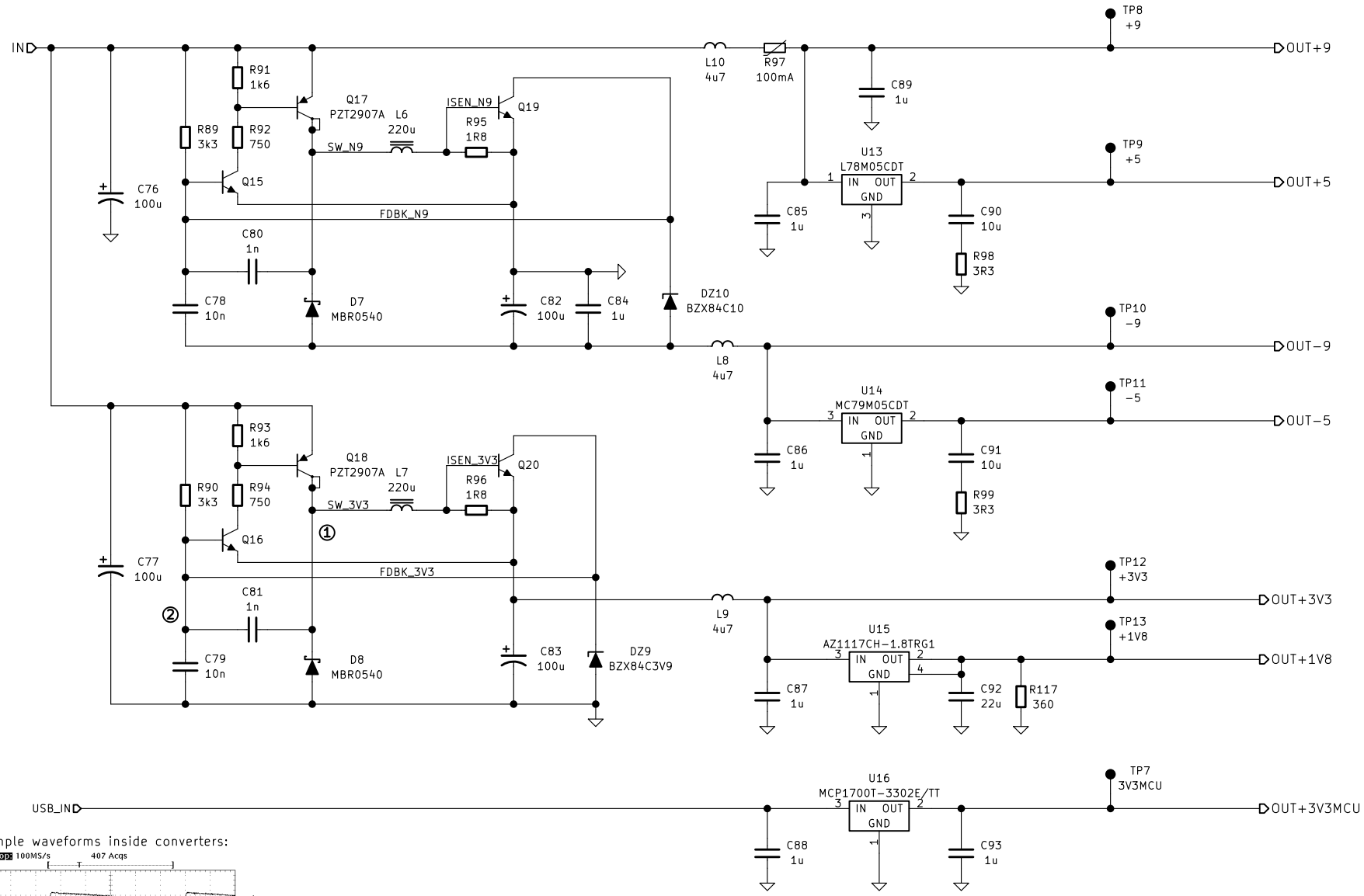
TODO (REV 2):
Revise OVLO latching to work properly.

Gain/Phase Analyzer
WCP52

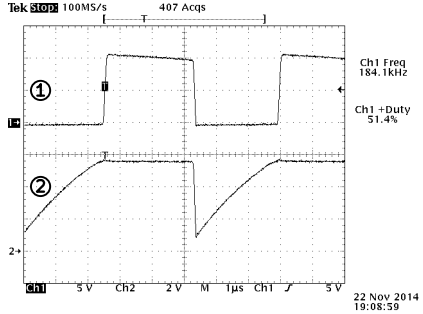
Sheet: /PowerInput/
File: PowerInput.sch

Title: Power Input Circuit

Size: USLetter	Date: 2015-03-31	**Rev: 1**
KiCad E.D.A. kicad (after 2015-mar-04 BZR unknown)-product		Id: 11/13



Example waveforms inside converters:



TODO (REV 2):
- C92 could be 2x 10uF, eliminating a BOM item.
- Can Q17 and Q18 be upgraded to PMOS? In simulation testing it was difficult to switch them off quickly and they lost significant power. However, the low current gain of the PZT2907A required changing the current−limiting resistors to 1R8 and sacrificing output current.

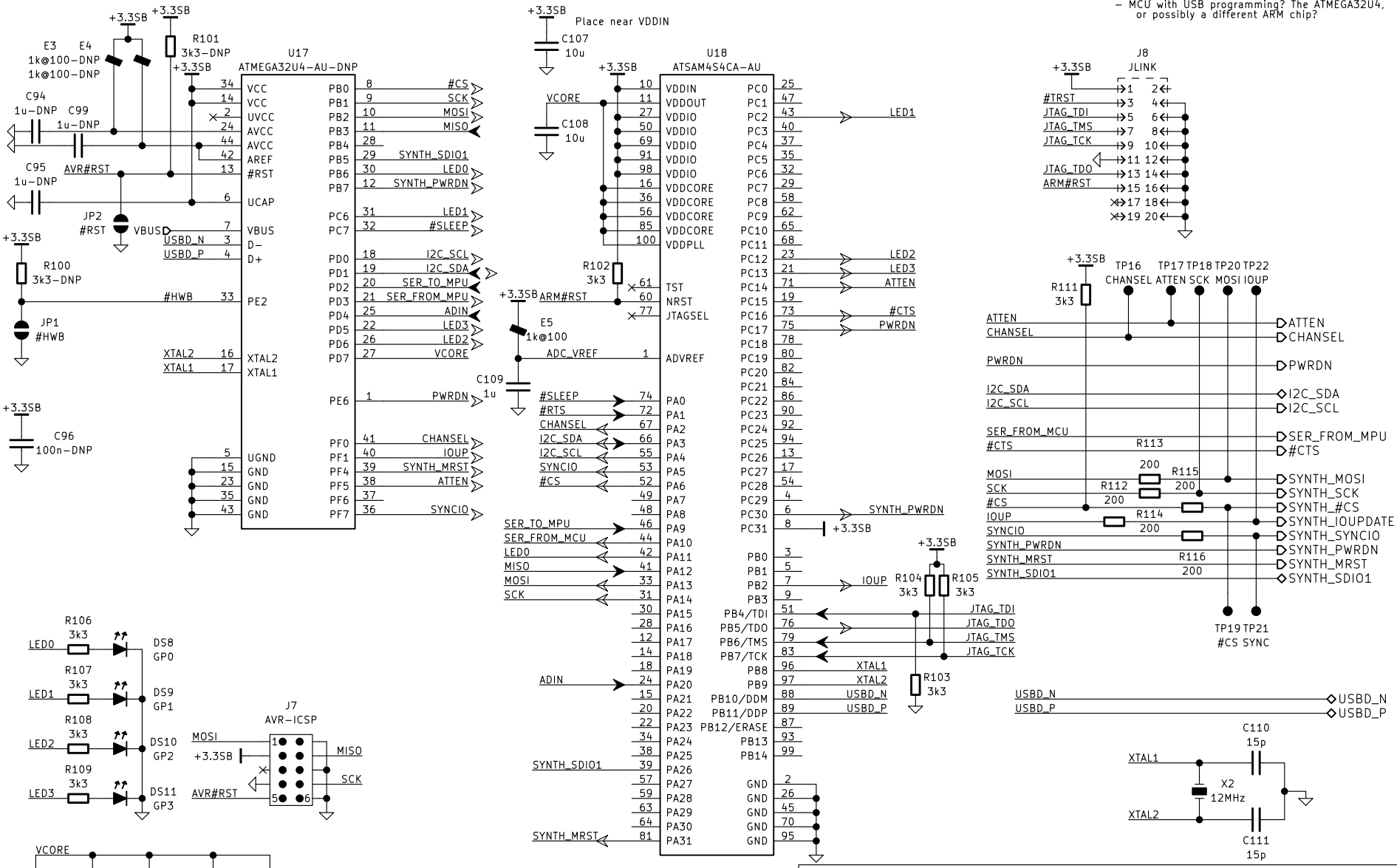
  Also possible: higher gain BJTs? Darlingtons/Sziklais? (probably too slow)

Gain/Phase Analyzer
WCP52

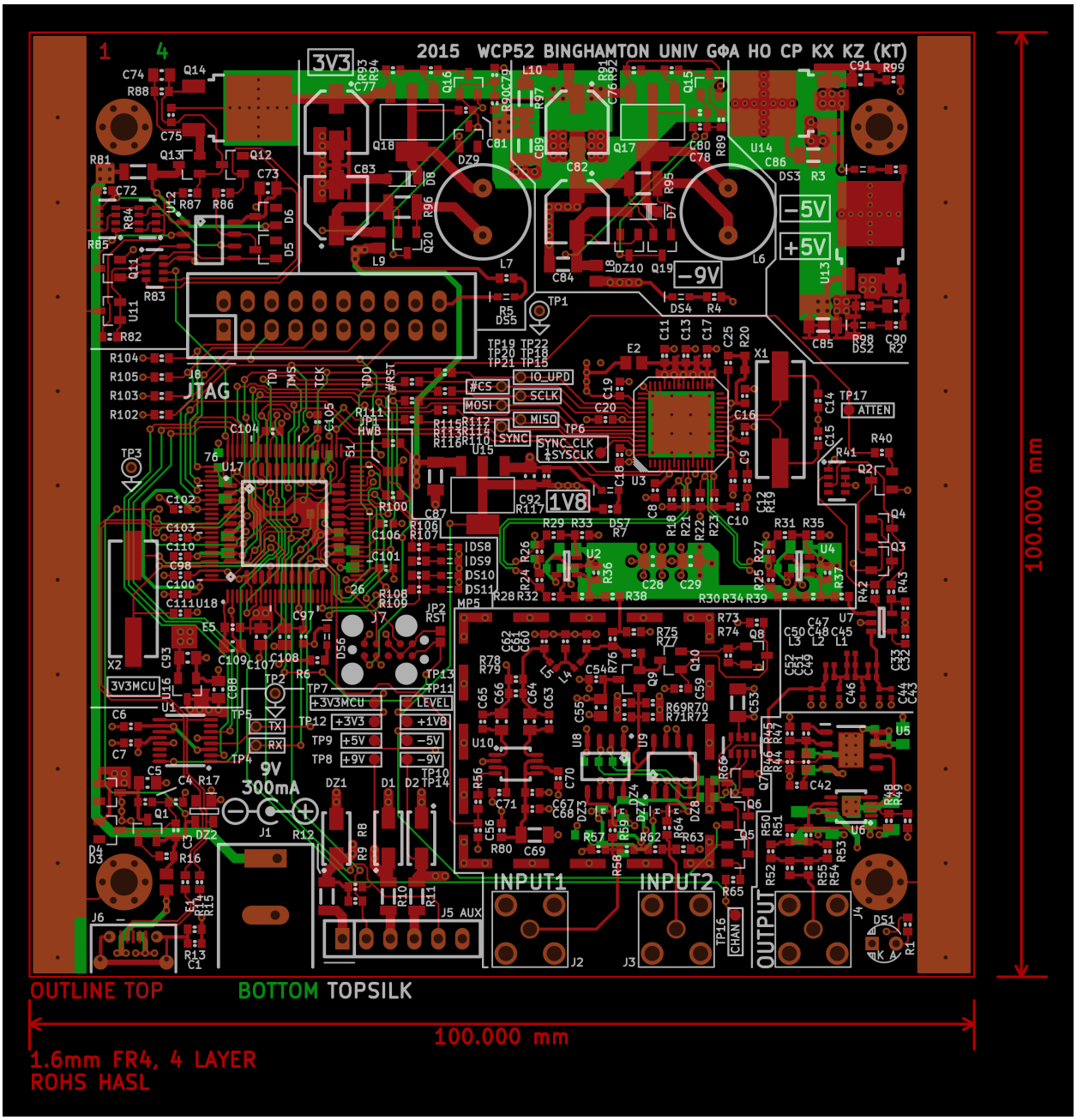
Sheet: /DCDC/
File: DCDC.sch

Title: DC−DC Converters

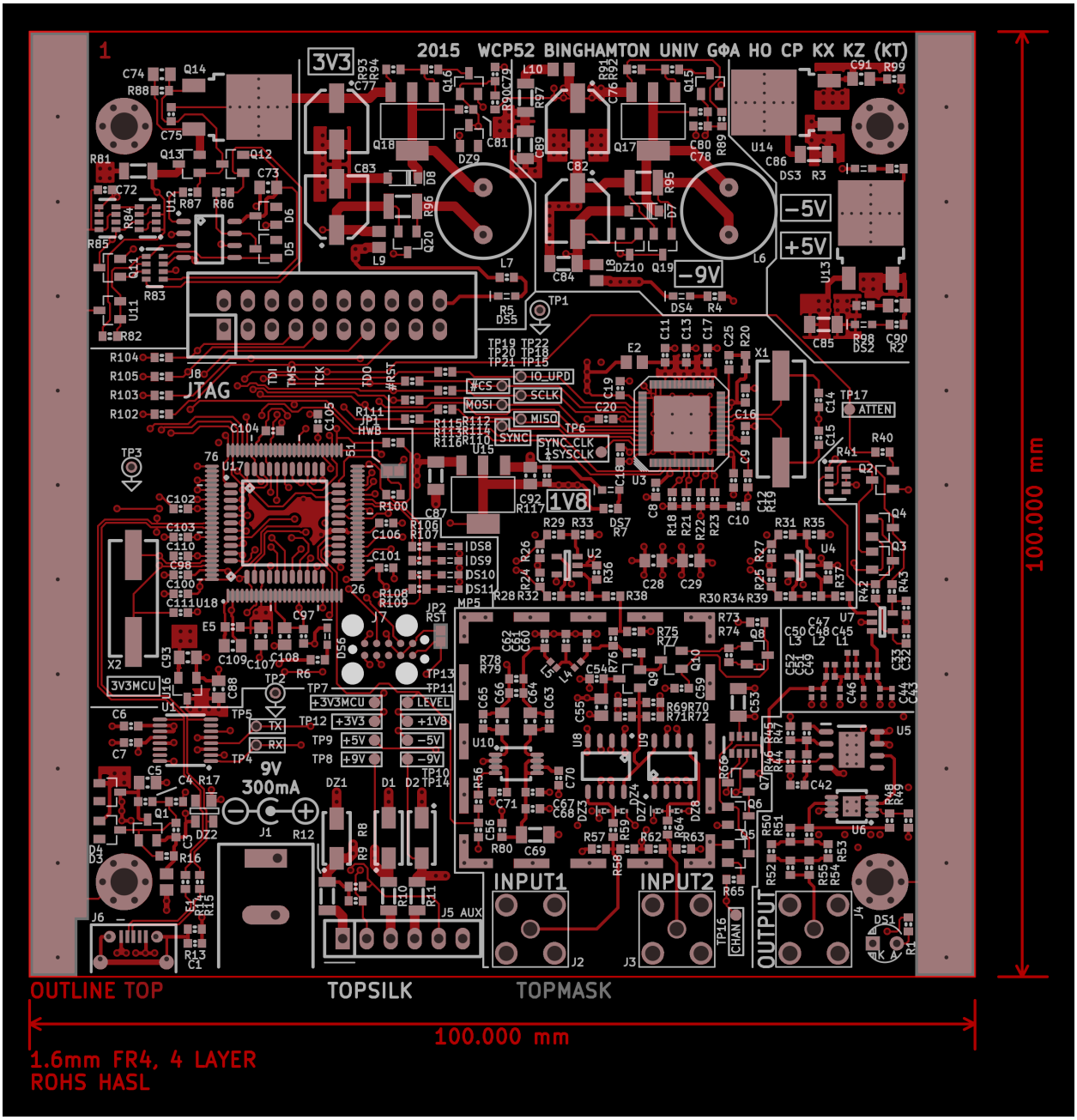
Size: USLetter	Date: 2015−03−31	**Rev: 1**
KiCad E.D.A. kicad (after 2015−mar−04 BZR unknown)−product		Id: 12/13


TODO (REV 2):
– MCU with USB programming? The ATMEGA32U4, or possibly a different ARM chip?
U17
ATMEGA32U4-AU-DNP
U18
ATSAM4S4CA-AU
J8
JLINK
J7
AVR-ICSP
Place near VDDIN
E3 E4
1k@100-DNP
1k@100-DNP
R101 3k3-DNP
C94 1u-DNP
C99 1u-DNP
C95 1u-DNP
C96 100n-DNP
JP2 #RST
JP1 #HWB
R100 3k3-DNP
C107 10u
C108 10u
C109 1u
R102 3k3
E5 1k@100
R103 3k3
R104 3k3
R105 3k3
R106 3k3
R107 3k3
R108 3k3
R109 3k3
R110 200
R111 3k3
R112 200
R113 200
R114 200
R115 200
R116 200
C110 15p
C111 15p
X2 12MHz
C97 100n
C98 100n
C100 100n
C101 100n
C102 100n
C103 100n
C104 100n
C105 100n
C106 100n
TP14 LEVEL
TP15 MISO
TP16 CHANSEL
TP17 ATTEN
TP18 SCK
TP20 MOSI
TP22 IOUP
TP19 #CS
TP21 SYNC
DS8 GP0
DS9 GP1
DS10 GP2
DS11 GP3
+3.3SB
VCORE
AVR#RST
#HWB
XTAL1
XTAL2
UCAP
VBUS
USBD_N
USBD_P
#CS
SCK
MOSI
MISO
SYNTH_SDIO1
LED0
SYNTH_PWRDN
SYNTH_MRST
ATTEN
SYNCIO
LED1
#SLEEP
I2C_SCL
I2C_SDA
SER_TO_MPU
SER_FROM_MPU
ADIN
LED3
LED2
VCORE
PWRDN
CHANSEL
IOUP
#RTS
ADC_VREF
ARM#RST
JTAGSEL
TST
NRST
LED1
LED2
LED3
ATTEN
#CTS
PWRDN
SYNTH_PWRDN
IOUP
JTAG_TDI
JTAG_TDO
JTAG_TMS
JTAG_TCK
XTAL1
XTAL2
USBD_N
USBD_P
#TRST
JTAG_TDI
JTAG_TMS
JTAG_TCK
JTAG_TDO
ARM#RST
SER_FROM_MCU
#CTS
MOSI
SCK
#CS
IOUP
SYNCIO
SYNTH_PWRDN
SYNTH_MRST
SYNTH_SDIO1
SYNTH_MOSI
SYNTH_SCK
SYNTH_#CS
SYNTH_IOUPDATE
SYNTH_SYNCIO
SYNTH_PWRDN
SYNTH_MRST
SYNTH_SDIO1
SYNTH_MISO
@U17 is an alternative microcontroller. Not used in first revision.
Note duplicate connections. These are used to facilitate routing. Make sure to set all duplicate connections as inputs.
Gain/Phase Analyzer
WCP52
Sheet: /MPU/
File: MPU.sch
Title: MPU
Size: USLetter
Date: 2015-03-31
Rev: 1
KiCad E.D.A. kicad (after 2015-mar-04 BZR unknown)-product
Id: 13/13

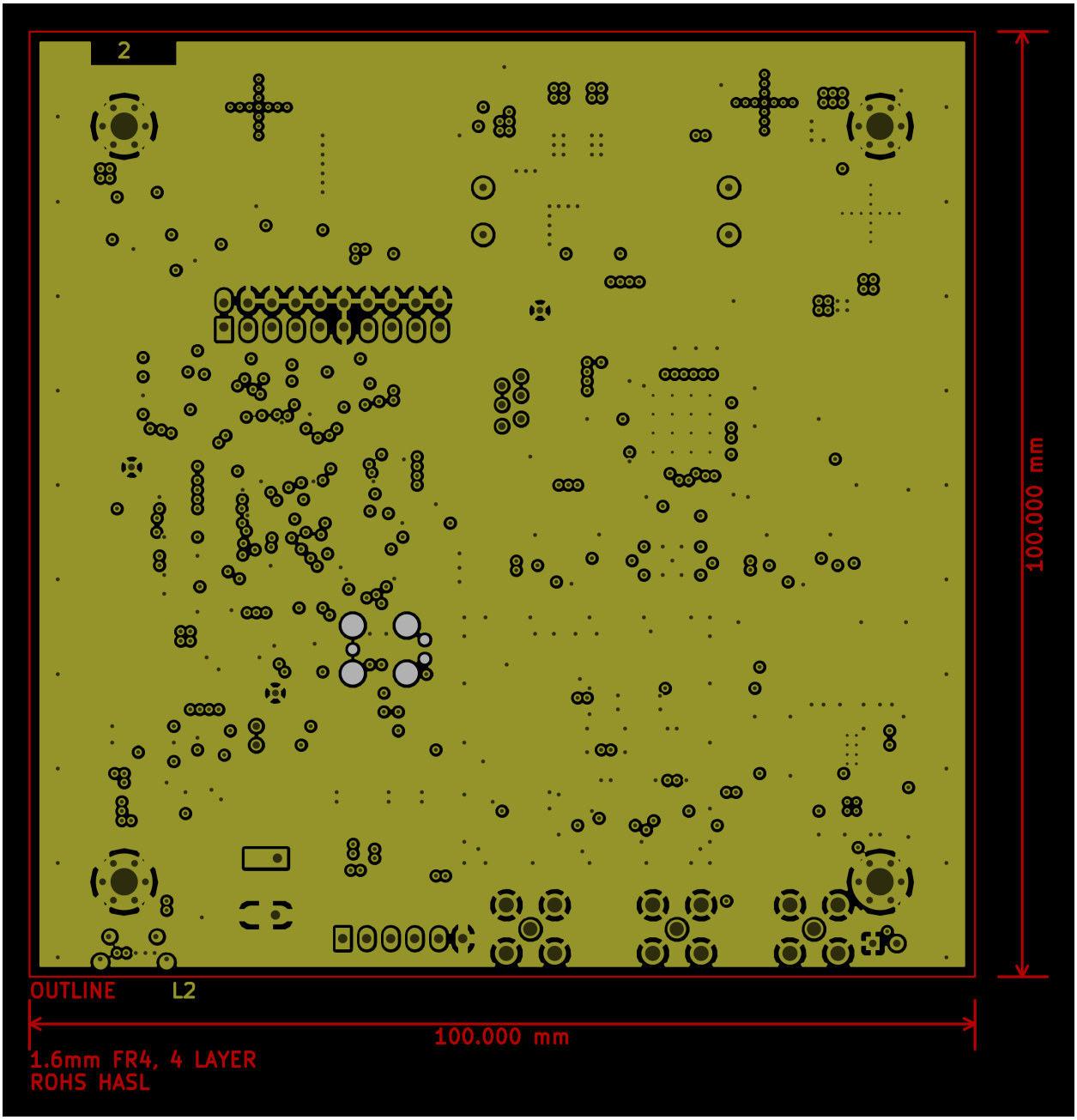
F PCB Layouts



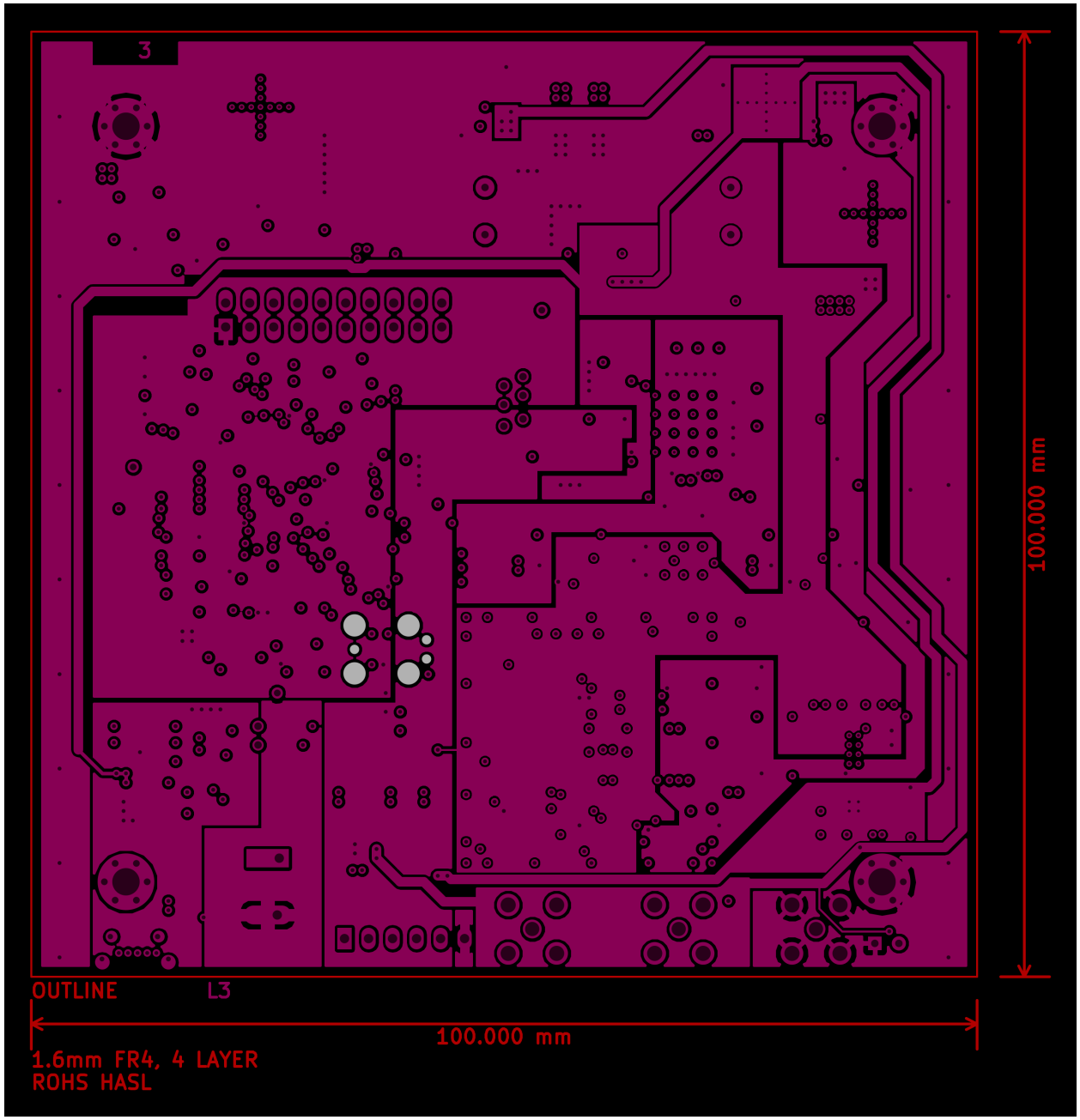
F.1 Layer 1


2015 WCP52 BINGHAMTON UNIV GΦA HO CP KX KZ (KT)
3V3
-5V
+5V
-9V
1V8
JTAG
INPUT1
INPUT2
OUTPUT
9V
300mA
100.000 mm
OUTLINE TOP
TOPSILK
TOPMASK
100.000 mm
1.6mm FR4, 4 LAYER
ROHS HASL

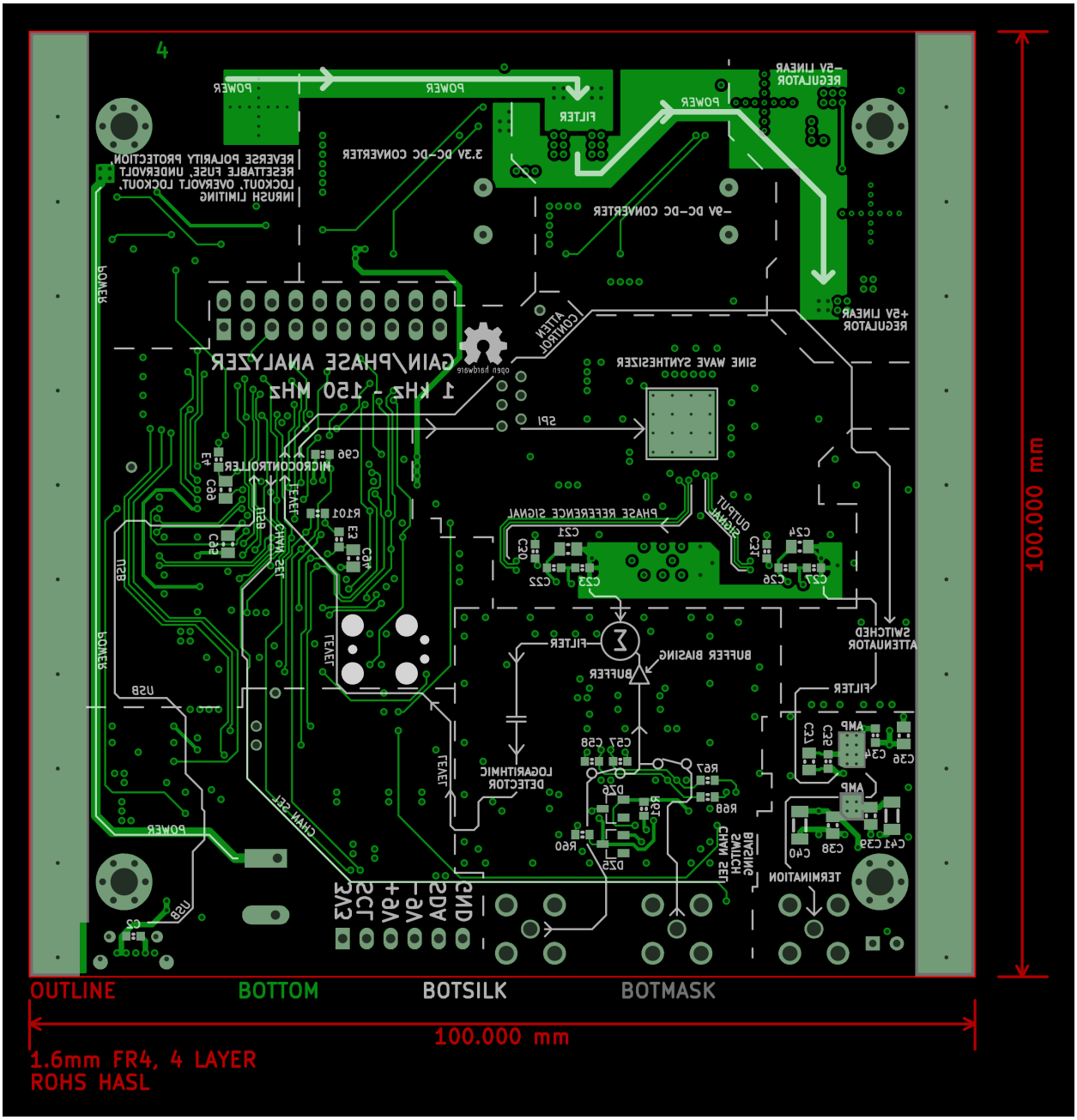
F.2 Layer 2



F.3 Layer 3



F.4 Layer 4


4
POWER
FILTER
3.3V DC-DC CONVERTER
-5V LINEAR REGULATOR
-9V DC-DC CONVERTER
+5V LINEAR REGULATOR
REVERSE POLARITY PROTECTION, RESETTABLE FUSE, UNDERVOLT LOCKOUT, OVERVOLT LOCKOUT, INRUSH LIMITING
GAIN/PHASE ANALYZER
1 kHz - 150 MHz
SINE WAVE SYNTHESIZER
SPI
MICROCONTROLLER
PHASE REFERENCE SIGNAL
OUTPUT SIGNAL
BUFFER BIASING
BUFFER
SWITCHED ATTENUATOR
AMP
LOGARITHMIC DETECTOR
TERMINATION
3V3
SCL
+5V
-5V
SDA
GND
100.000 mm
100.000 mm
OUTLINE
BOTTOM
BOTSILK
BOTMASK
1.6mm FR4, 4 LAYER
ROHS HASL