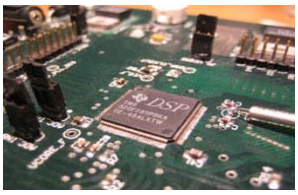
University of Waterloo

THE MARCONI PROJECT

www.sgsolutions.ca/marconi



Transparent Wireless Modem
Detailed Design

June 3rd, 2005

Department of Electrical and Computer Engineering
University of Waterloo
200 University Ave. W., Waterloo, ON

Aaron Cheung
Hardware Lead
00141344

Stefan Janhunen
Algorithms Lead
00057388

Vincent G. Liu
Marketing Lead
20069140

Shu Wu
Software Lead
20069140

Table of Contents

1 Duties and Responsibilities

The responsibilities for designing and implementing the different parts of the project are divided as shown in Table 1. It is important to note that the team discusses all design objectives together and the responsibilities are not exclusive. In addition, Table 1 does not give credit for all the smaller contributions that each team member has made.

Table 1 Responsibilities of Implementation

	Stefam Janhunen	Shu Wu	Vincent Liu	Aaron Cheung
PCB Layout	x			x
Radio Characterization	x			
Hardware Drivers	x			
Debug Interface		x		
Modem Architecture		x		
Viterbi Block		x		
Packeting Block		x	x	
Modulator and Demodulator Block	x			

2 Hardware Design

In addition to the four sub blocks that fall under the hardware section of the radio modem, an implicit sub block – the power supply – must be included. Hence, in total, there are five sub blocks within the hardware section, namely the power supply, serial interface, Digital Signal Processor (DSP), audio codec, and the audio interface. This section will examine the critical design decisions made in each of these sub blocks, as well as presenting some of the board layout considerations. The final board layout and the circuit schematic diagrams can be viewed in APPENDIX A - Hardware Design Schematics. A high-level abstraction of the hardware architecture is shown in Figure 1.

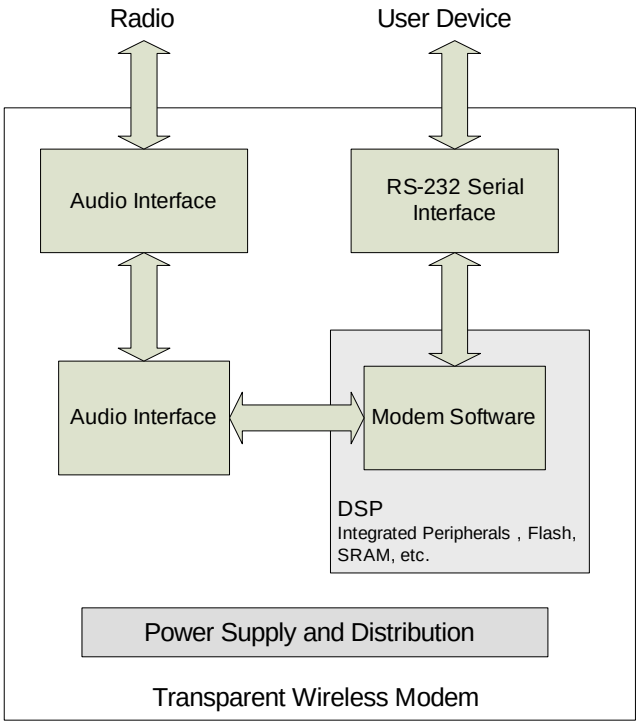


Figure 1 High-Level Hardware Abstraction

2.1 Power Supply Circuit Design

At the time of design, the top priority of the power supply circuit is reliability. Since the radio modem is not to be designed for mobile applications, it was decided to sacrifice board space, part count, and power efficiency for reliability and redundancy. The general approach taken to design the power supply circuit was to first decide on the components to be used on the radio modem, then based on each component's voltage and current requirements, the proper voltage supply components were chosen with ample redundancy.

After all components were selected, it was decided that one 1.8V power supply and two 3.3V power supply ICs were required. Since each of the 3.3V power supply is capable of supplying 1.5A of current, it was predicted that only 3.3V power supply will be

sufficient. However, since there are digital and analog components in the complete design, it is better to separate the digital components' power supply from the analog components' power supply in order to prevent any switching noise from the digital to the analog components. Hence it was decided to use one 3.3V power supply IC for the analog components only, and one 3.3V supplying the digital components, along with the 1.8V IC for the DSP's input/output power supply.

There are a few special design considerations paid on the power supply for the DSP, namely the power up sequencing and reset signal control. Two voltages are used in the DSP – 3.3V and 1.8V – for general purpose signalling and input/output signalling, respectively. At power up, the DSP requires the 3.3V power supply to be established first before the 3.3V supply takes effect. As a result, a special IC was required to ensure that there is a delay for the 1.8V power supply after the 3.3V power supply is detected. Aside from power up sequencing, the reset signal for the DSP and the audio codec need to be properly controlled, such that the proper power up sequence will be enforced after a power failure. In order to implement the above functions, a special IC was selected in conjunction with the 1.8V power supply IC to provide the power up sequencing function, as well as the reset signal control for the DSP and the audio codec.

2.2 Serial Interface Design

From the block verification document, it was decided that the hardware should have two serial ports, so as to facilitate software debugging during the software development phase. In order to allow two serial ports to function simultaneously, a dual input/output serial driver with RS-232 protocol was selected.

2.3 Interconnections with the Digital Signal Processor (DSP)

Almost all connections made with the DSP was based on the manufacturer's specifications. The only major design decision made regarding the DSP, was that an array of LEDs connected to the DSP's general purpose input/outs (GPIO) pins will be very useful in debugging any software issues. In total, the DSP provides 32 GPIO pins. For debugging reasons, it would be excessive if all 32 bits GPIO pins were used for driving LEDs. Since one ASCII character can be represented by one byte, it was decided that it will be sufficient to user eight out of the all 32 GPIO pins for debugging purposes. These GPIO pins, however, cannot supply high current. In order to drive eight LEDs, it is then necessary to use an eight bit driver to supply the necessary current for the LEDs.

2.4 Audio Codec and Audio Interface Design

Similar to the case for the DSP, most interconnections made for the audio codec was based on the specifications given by the manufacturer. The only relevant design decision was to use external opamps in the audio interface rather than the internal opamps provided by the audio codec. The main reason for using external op amps in a later stage rather than using the internal opamps in the audio codec is because the output signal from the audio codec is in differential mode, whereas the input to the transmitting radio module needs to be a single ended sinusoid. As such, it is necessary to transform

differential signals to/from single ended signals separate from the audio codec. Coincidentally, differential-sinusoidal signal transformations can be implemented alongside with signal amplification. As such, the opamps were used in such a way that they can achieve both operations at the same time.

A notable design decision made on the audio interface was that each input and output pair was duplicated exactly, such that there are two input and two output connections available. The motivation was to minimize reconfiguration time during testing. Since each input and output opamp circuit has a potentiometer for adjusting the gain to and from the radio modules, if only one input-output pair was present, then repetitive adjustments will need to be made each time a testing configuration is changed. For instance, if at one setting a group member needs to connect to a computer to generate or record audio signals for analysis, then the opamp gain settings will need to be changed when he/she needs to test the radio modem on actual radios. By having two separate input-output opamp circuits, a developer may switch between a specific opamp gain setting quickly and easily.

2.5 Design Considerations for the Board Layout

After the circuit was designed and the schematic diagram was entered, the actual circuit board needs to be laid out manually to ensure optimum operations. Three specific designs are worth mentioning here.

First, an attempt should be made to separate the digital components from the analog components, so as to minimize digital noise to the analog side and vice versa. When digital signal switches, the voltage or current changes very rapidly. If analog components are placed near the digital components, the digital components are capable of inducing a high frequency component noise to the analog signal. Similarly, the analog signals may also introduce some signal component at a specific frequency to the digital signals. In order to minimize this effect, the digital components should be placed together and far away from the analog components. Additionally, analog signal traces should not be placed in parallel with digital traces, so as to avoid crosstalk and signal coupling.

Second, a common and large ground fill should be used in between traces to minimize crosstalk and signal coupling. In addition to the separation of digital and analog devices, a common ground in between traces should minimize any crosstalk that may exist between traces, ensuring the best signal integrity on the design.

Third, decoupling capacitors should be placed close to the device or signal in concern. Decoupling capacitors should be used throughout the board to ensure that the voltage supply will be well regulated during digital switching of a signal. In order for decoupling capacitors to be functional, they should be placed near to the signal or device in question.

If the printed circuit board is designed with these guidelines, it is positive that the circuit board should have good signal integrity and minimum noise level.

3 DSP Software Architecture

3.1 Overview

The DSP software project is clearly divided into two parts, the lower-level firmware and the modem software and an optional debug component. The firmware is responsible for performing I/O operations and controlling peripherals. The debug interface produces a command-line interface which allows the developer to control the modem through the serial port. A high-level overview of the architecture is shown in Figure 2.

The modem software is platform independent; it can be executed on the PC, on the current hardware platform or another platform in the future.

The ability to run the modem on the PC is important because it allows the modem to be debugged in a familiar PC environment. Because there is minimal need to debug the modem in hardware, there is no need to purchase JTAG tools, which are relatively expensive.

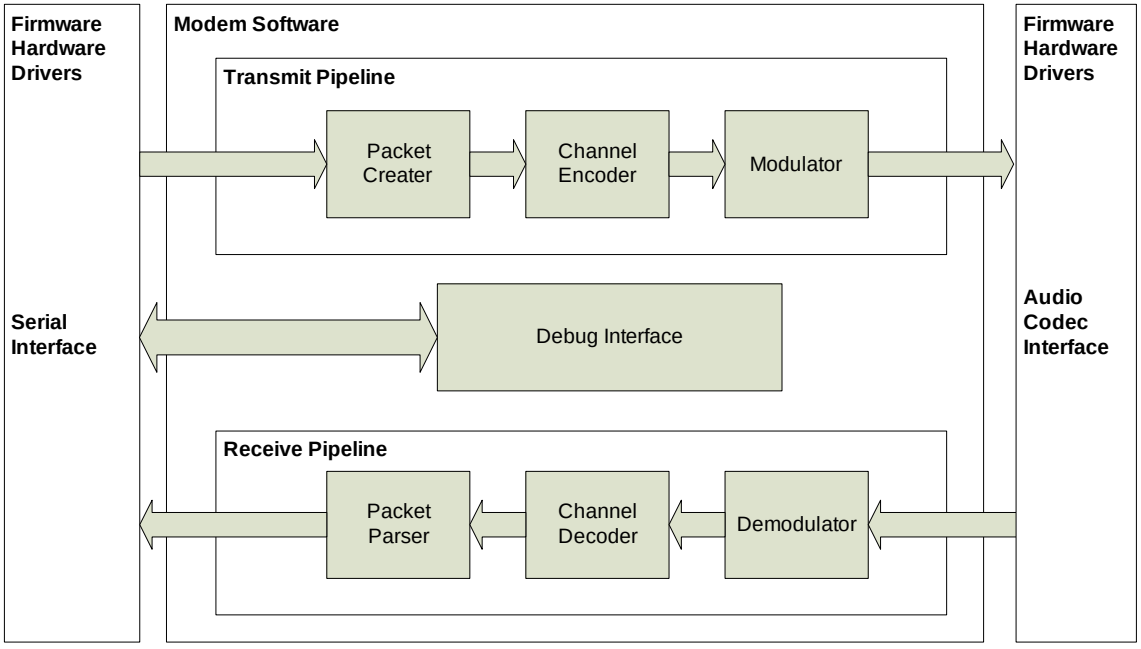


Figure 2 Software Architecture Overview

To implement the firmware and modem, the C language was chosen. The advantage of C is its wide acceptance and ease of learning. In comparison to higher level languages such as C++ and Java, C gives us more ability to optimize memory usage and execution speed.

3.1.1 Software Methodology

The modem software is divided into many pieces through an explicit object framework specially designed for this project. This object framework allows the development of each modem block independently while structuring each block in a way that they can fit together perfectly.

Another advantage of the object-oriented design is the ability to instantiate multiple modems in simulation. This makes it possible to perform simulations of point-to-point modem connections in the PC environment.

The top-level objects in the software are pipelines, which control the flow of data from one modem block to the next and arbitrate any conflicts between them. Each modem block is wrapped in an object called a controller, which introduces a standardized I/O procedure and initialization sequence. One configuration of this object design is shown in Figure 2.

3.2 Hardware Interface Drivers

In order to make development of the DSP firmware simple and efficient, a well-defined interface between the modem firmware and the drivers for the individual DSP hardware blocks must be defined. While numerous on-chip peripherals are present, only a few are actually required for this design. Drivers must be written for the UART (Universal Asynchronous Receiver Transmitter) serial ports used for RS-232, the synchronous serial port used to interface to the audio codec, and the GPIO (General Purpose Input Output) ports used to drive external hardware such as LEDs. Additionally, drivers are required for the hardware timers as well as to configure the DSP for proper operation after reset. Clear definition of this interface is important so that team members can simultaneously work on both sides of this dividing line without problems. Table 2 details the functional interface defined.

Table 2 Hardware interface functions organized by peripheral

Peripheral	*Interface Functions*
System Initialization	`HwInit(…)`

Asynchronous Serial (UART)	`HwUartOpen(…)`
	`HwUartClose(…)`
	`HwUartRd(…)`
	`HwUartRdAvail(…)`
	`HwUartWr(…)`
	`HwUartWrAvail(…)`

Audio Codec	`HwCodecOpen(…)`
	`HwCodecClose(…)`
	`HwCodecSetRxPGAGain(…)`
	`HwCodecSetTxPGAGain(…)`
	`HwCodecSetRxChannel(…)`
	`HwCodecSetLoopback(…)`
	`HwCodecRdAvail(…)`
	`HwCodecRd(…)`
	`HwCodecWrAvail(…)`
	`HwCodecWr(…)`

GPIO	`HwGpioSet(…)`
	`HwGpioClear(…)`
	`HwGpioSetOutputEnable(…)`
	`HwGpioClearOutputEnable(…)`
	`HwGpioRead(…)`

Hardware Timer	`HwTimerConfigure(…)`
	`HwTimerEnable(…)`
	`HwTimerDisable(…)`
	`HwTimerGetCurrentTicks(…)`
	`HwTimerGetTicksPerMS(…)`

Rather than expose the details of interrupts and hardware buffering, a clean and consistent interface is maintained across both the UART and the audio codec interface. Read and write functions are available that provide a non-blocking polling interface to

the transmit and receive FIFOs of each port. The drivers perform all interrupt servicing complete transparently. This allows the main application code to reside in an infinite loop while periodically polling the ports for data, processing and sending data as necessary.

Hardware initialization upon reset is accomplished with a single function call. Complete UART functionality is encapsulated in a file-like interface with open, close, read, and write functions. The audio codec is also represented with a file-based interface. Additional functions allow control of the programmable gain amplifiers and other special features. The GPIO pins are supported through set, clear, and read operations. Finally, the hardware timer can be configured, enabled, disabled, and polled for the current clock tick count.

While this overview presents the functional aspects of the hardware driver interface, please refer to APPENDIX B – Hardware Interface Header File for the complete specification of this interface in the form of a C header file.

3.3 Debug Interface.

There are two serial ports available on our hardware platform, and one is dedicated to the debug interface.

The debug command parser, shown in Figure 3 provides a dos-like command line interface to access modem features and access all objects in the software framework. The command structure used by the parser is created using a structured array and several command structures can be layered on top of each other.

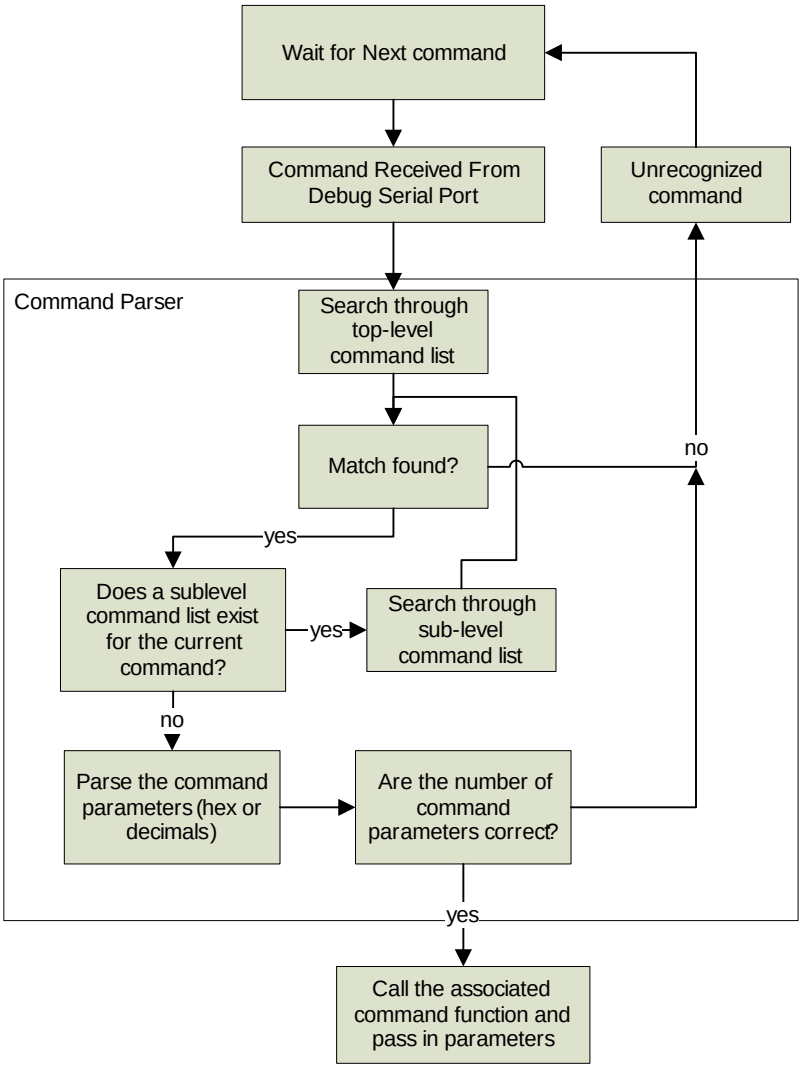


Figure 3 Command Parser Block Diagram

After a command match is found, multiple parameters in the form of hex or decimals can be parsed and passed into the function associated with that command.

3.4 Pipeline Framework

As shown in Figure 4, all blocks are wrapped within a controller object. The execution of the block and the flow of data from one block to the next are arbitrated by the pipeline object.

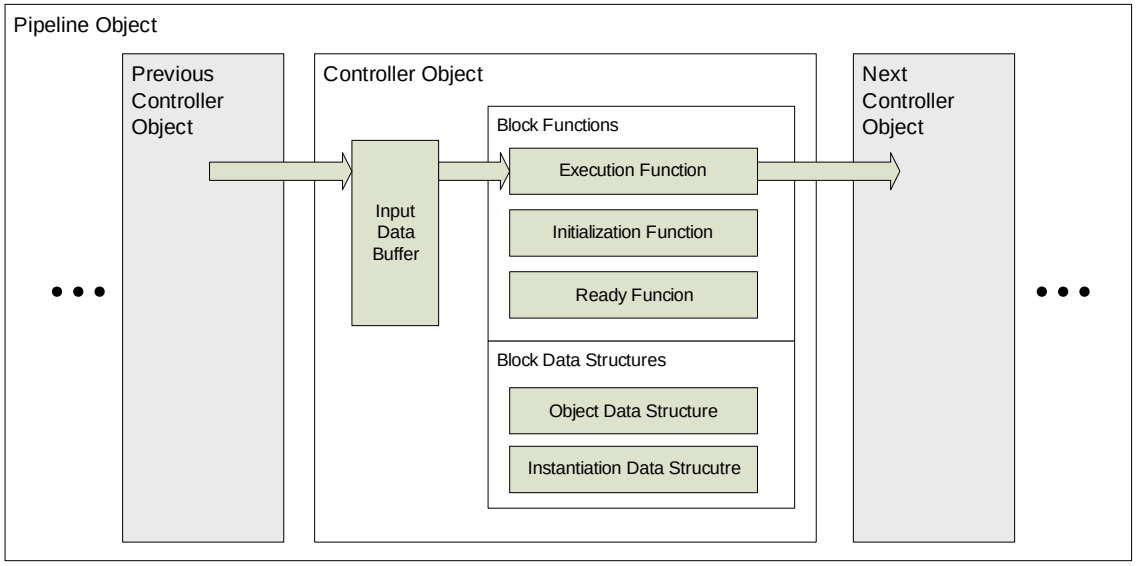


Figure 4 Pipeline and Controller Objects

A block is able to interact with the pipeline via the ready function, which tells the pipeline whether it is ready to be executed. Because there is a input buffer for each block, the blocks do not need to be executed in a linear sequence (ex. block1, block2, block3). They can execute whenever there is data available in the input buffer and when the next block's input buffer is not full.

The composition and order of the controller objects within the pipeline object can be changed dynamically. The data flow will be arbitrated automatically by the pipeline object.

3.5 Software Modem Blocks

The software modem blocks outlined below are the components of the receive and transmit pipelines in the modem. They are wrapped within a controller object that allows data flow from one block to the next to be arbitrated by the pipeline object.

3.5.1 Soft-Decision Viterbi Convolutional Codec Block Design

The Convolutional codec is a forward error correction technique which is widely used in wireless digital communication systems where low SNR makes it highly susceptible to bit errors.

The performance of a convolutional encoder and decoder, which known parameters, when subjected to Additive White Gaussian Noise is well defined and documented. These parameters are the output rate, code length, polynomial coefficients, and length of history bits kept inside the trellis. The codec is also soft-decision, which means that uncertainty levels of received binary signals are passed to the decoder. These parameters have to be set based on tests with real signals and adjusted for the processing power of the DSP used.

The Convolutional decoder is implemented using a Viterbi algorithm, which has a fixed runtime and relatively good performance. It is important to note that the decoding algorithm only has an affect on the processing requirements and does not error-correction performance when identical parameters are chosen.

At this stage of modem development, only the PC simulation environment is available, and so a simulated AWGN channel was created to compare the performance of the designed Convolutional codec as compared to references.

The codec parameters are configurable at run-time, which allows for easy testing. The results are shown below. The BER (Bit Error Rate) vs. SNR graph is normalized for different data rates (which some books usually omit).

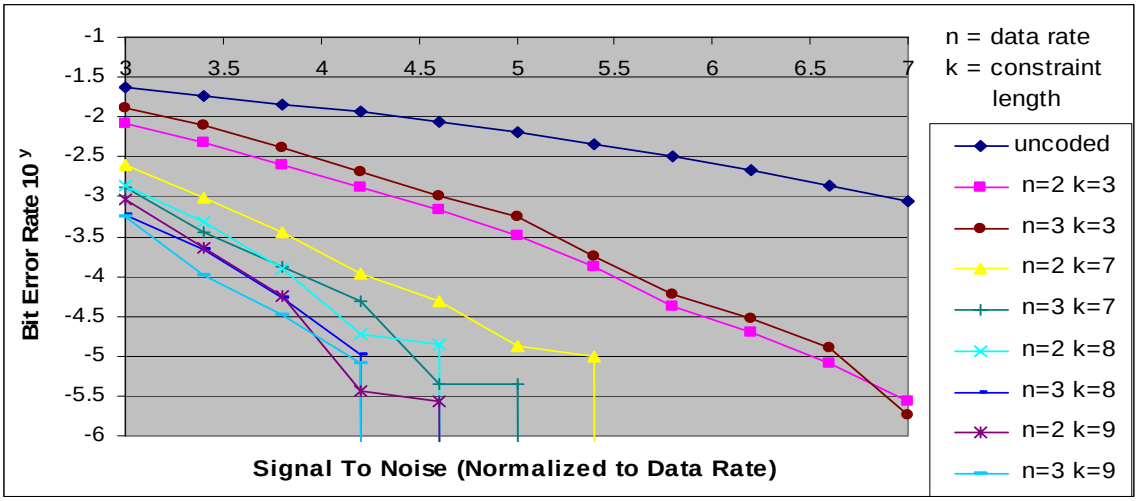


Figure 5 BER vs. SNR for Soft-Decision Viterbi Convolutional Codec

The n and k parameters, in the chart above, represent the output data rate and constraint length parameters respectively. As the convolutional data rate is increased, there is very little effect on the BER performance. This is because the chart is normalized with respect to higher data rates, which offsets any error-correction benefits.

However, the BER performance increases significantly as the constraint length is increased. Larger constraints cost nothing in terms of bandwidth, but are very expensive in terms of processing power. In fact, there is an inverse exponential relationship has constraint lengths are increased, as shown in the graph below.

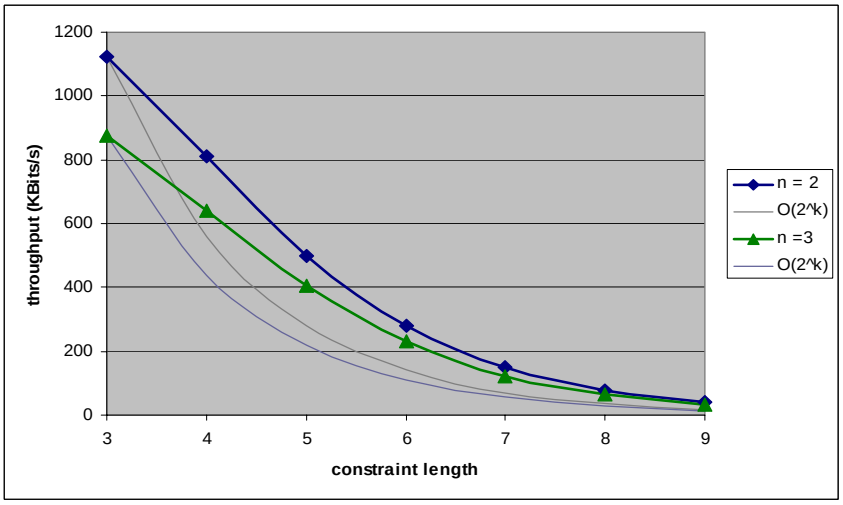


Figure 6 Codec performance vs. Constraint Length

In conclusion, the ability of the convolutional codec to improve Bit Error Rates is well proven and confirmed with simulation. The next step in the design phase is to optimize

the codec parameters for the intended application and to push constraint length as high as high as the processing resources allow.

3.5.2 Packeting Block Design

The purpose of this block is to establish transmission synchronization and data flow control by adding header information to blocks of data. The main motivation of this block is to improve the reliability of the modem data transmission (process?). Without this process (step?), there are no guarantees that all the packets sent to the receiver will be received. Incomplete packet transfer is a problem for our application, since every packet is crucial in our transmission session. Adding this packet encapsulation block will improve data error rate and accommodate interruptions during transmission by establishing "hand shaking" of the sender and the receiver.

The three major functions of this block are: to achieve transmission synchronization, to create packets on the sending end, and to parse packet information on the receiving end.

On the sending end, the block will encapsulate blocks of signal data and encapsulate them with meaningful information. Figure 1 displays the contents of the packet header.

16	4	4	4	4	8	variable
sync bits	SEQ. #	ACK. #	Header length	code bits	check sum	DATA

Figure 7 Contents and size of a typical packet

The functions of each part of the content are listed below.

Sync bits are used to allow the system to match a unique sequence created by the computer in order to notify the receiver the beginning of meaningful information. Then, the receiver will continue to parse information received until it finds the synch bits. If the received information is exhausted before the synch bits can be matched, the receiver will then initiate a resend.

Source port and destination port numbers: They are used to give the system the capability to carry out several different transmissions simultaneously. Each unique transmission will have a different source and destination number in order to differentiate amongst each other on the receiving end.

SEQ number and ACK number: Sequence number and acknowledge number is used to implement "hand shaking" between the sender and the receiver. Each octet of data will be given a unique sequence number. The receiver will send an acknowledgement number of (from?) the received sequence number plus another one to the sender (as notification upon receiving the block of data successfully?) in order to notify the sender that it has successfully received the block of data. When the sender receives this acknowledgment,

it will then send the next packet. This cycle will happen continuously until the end of the transmission. This is also called “handshaking”.

Header length: The header length field contains information about the length of the header.

Code bits: This field contains the information of what type of packet this is. (Ie, SEQ, ACK, etc.)

Check sum: A mechanism used to ensure that there are no errors in the bits received. If the check sum and the sum of all the bits received are not equal, it is clear that there are errors in the received data.

The following paragraph will entail the packet encapsulation block’s performance considerations.

Since the modem will not be used to stream high bandwidth, we, as a result, have decided to set 100 bps as our minimum for our primary design objective. This goal was set based on the assumption that the modem’s geographical operation areas will have low noise. We have also picked a simpler transmission scheme for our primary design objectives by sending and receiving one packet at a time. We have not yet decided on the exact size of our data blocks being encapsulated in a packet, because we still need to determine unknowns, such as actual noise of the transmission media and the time it takes for a sender to switch to a receiver.
All we know so far is that there is a large physical/mechanical delay caused by the switch pushed to change the radio’s(?) from sender to receiver. We have to assume that there will be low error occurrence in order to achieve our desired transmission speed. Another assumption is that since our range is short, signals sent will arrive at the receiving end instantaneously.

This specific design of the packet encapsulation block is used based on research of current industry standard. This type of encapsulation scheme is proven to be effective in other modems. Our group has also decided to use a simple and achievable scheme to ensure a successful and working product for our primary design objectives.

The sender and receiver will each have a packet encapsulator and a decapsulator (parser). The encapsulator will created the appropriate header information and added to the packet and the decapsulator will take the header information and parse through it for meaningful information.

Please consider the following graphical overview of the encapsulator.
Once the data is encapsulated and sent to the receiver, the decapsulator will parse through the information and make appropriate decisions based on the information received. A state-machine mechanism is used in the decapsulator because the block will go into different states depending on the received data.

3.5.3 Modulator and Demodulator Block Design

Previously, in the Block Verification stage of this project, two possible modulation schemes were proposed: 4-FSK (4-level Frequency Shift Keying) and 4-PSK (4-level Phase Shift Keying). A decision was ultimately made in favor of 4-FSK due to the simplicity of robust implementation. Since the design is non-coherent, it is not necessary to track phase and frequency variations in the received carrier(s). This simplifies the receiver design significantly but reduces error-rate performance when operating in very low SNR conditions.

To design the appropriate signal set to be used for 4-FSK modulation, it is necessary to consider once again the available bandwidth of the voice channel provided by the radios, as illustrated in Figure 8. As was discussed in the Block Verification, the outer null-to-outer null bandwidth of M-ary FSK can be calculated as:

$$BW = (M+1)(W/2) = (M+1)f_b$$

where M is the number of tones used, W is the width of the main lobe of the power spectrum for each tone, and f_b is the baud rate in symbols per second. According to project specifications, a net data rate of 100 bits per second is required. Since a ½ rate convolutional encoder will be used to encode the data, this necessitates a gross bit rate of 200 bits per second. 4-FSK uses four distinct symbols (signals) for transmission, corresponding to four distinct tones, allowing two bits of information to be encoded per symbol. Thus, the required baud rate is f_b = 100 symbols per second.

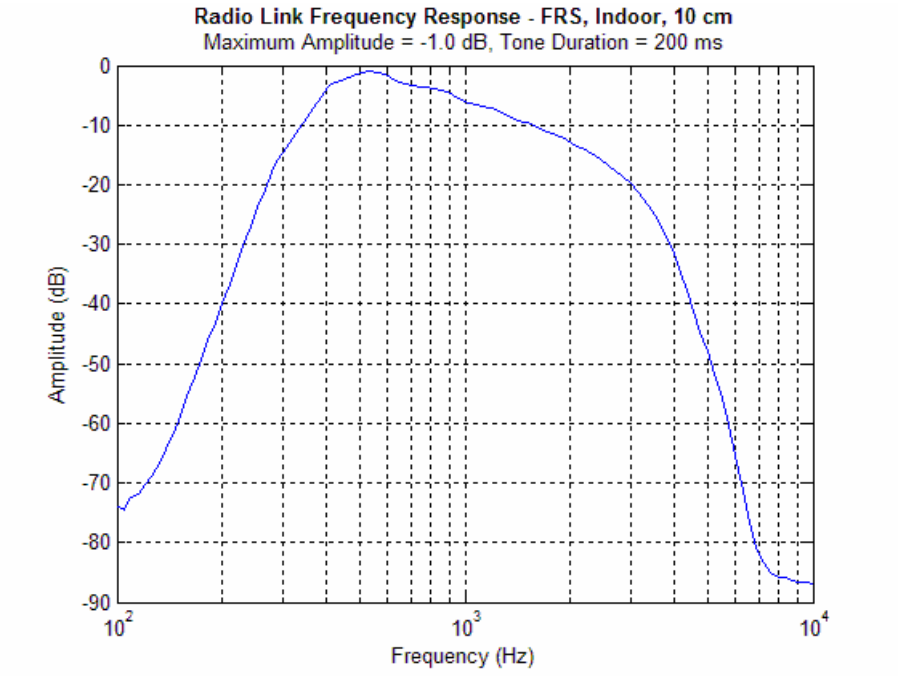


Figure 8 – Frequency response of radio voice channel

Using this, we can calculate the required bandwidth and design the required tone set:

$$BW = (M+1) f_b = (4+1)(100) = 500 \text{ Hz}$$

To maintain orthogonally among the tones, it is necessary that a given tone be located in the spectral nulls of all other tones in the set. Referring to Figure 9, it is evident that the minimum distance between tones for which this is possible is equal to f_b, the baud rate. This ensures that for any two tones $s_i(t)$ and $s_j(t)$ the condition

$$\int_{-\infty}^{\infty} s_i(t) s_j(t) dt = 0 \text{ for any } i \neq j$$

is satisfied. Essentially, this relationship states that the correlation between any two tones in the tone set is zero.

A tone set satisfying these criteria can now be chosen:

$$\begin{aligned} s_1(t) &= \cos(2\pi 600t) \quad 0 \le t < T_b \\ s_2(t) &= \cos(2\pi 700t) \quad 0 \le t < T_b \\ s_3(t) &= \cos(2\pi 800t) \quad 0 \le t < T_b \\ s_4(t) &= \cos(2\pi 900t) \quad 0 \le t < T_b \end{aligned}$$

These signals correspond to tones located at 600, 700, 800, and 900 Hz, respectively. This satisfies the spacing requirement and fits well within the bandwidth defined in Figure 8.

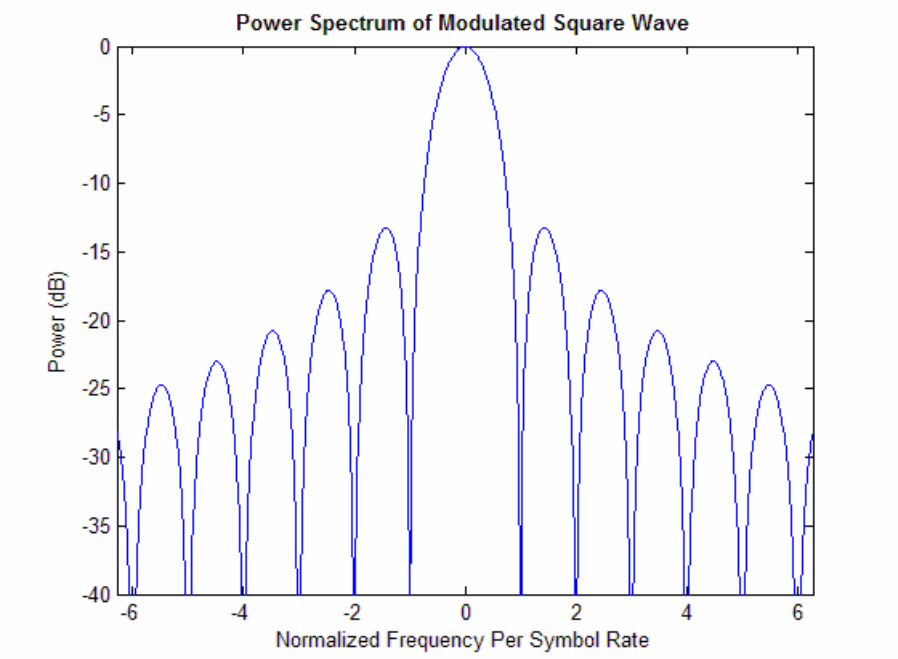


Figure 9 – Power spectrum of modulated square wave

The structure of the modulator and demodulator can now be discussed. Figure 10 illustrates the design of the 4-FSK modulator. Incoming bits are received from the output of the convolutional encoder in pairs. These bit pairs are passed through a 2-to-4 decoder to produce a one-hot output corresponding to the symbol for transmission. The output of the decoder changes at a rate equal to the symbol or baud rate (f_b). These decoder outputs must be up-sampled to the sampling rate of the audio interface (f_s) before further processing can take place. The output of each up-sampling block is a train of rectangular pulses, which have the power spectrum illustrated in Figure 9. To suppress the excess side lobe bandwidth, the rectangular pulses can be filtered with a pulse shaping filter $h_s(t)$. These filtered baseband pulses are then modulated up to one of the four tone frequencies by multiplication by the appropriate tone frequency. The four modulator outputs are summed to produce an orthogonal 4-FSK output signal.

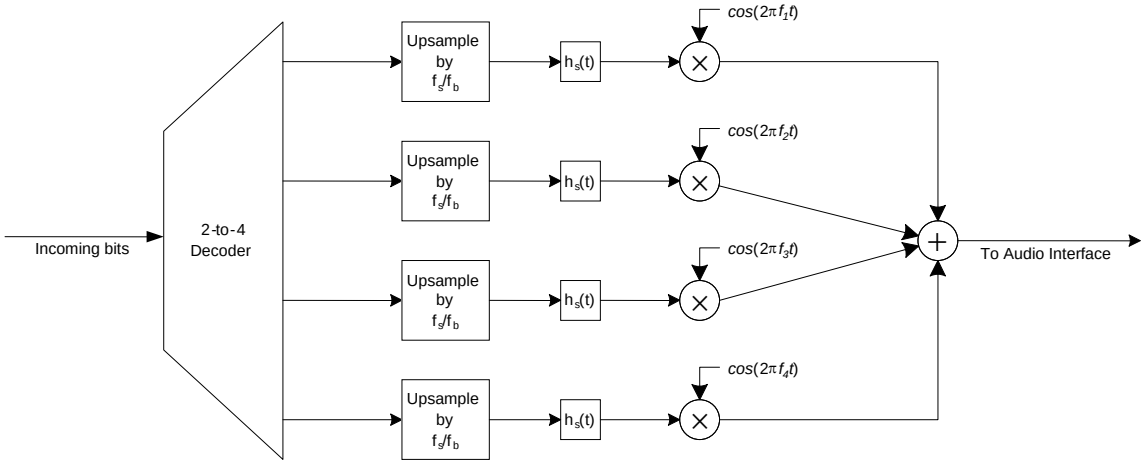


Figure 10 – Structure of 4-FSK modulator

In this form, the modulator structure is relatively easy to implement on a DSP. Most of the operations map nicely into MAC (Multiply-and-Accumulate) operations. Furthermore, reference waveforms for each tone can be stored in look-up tables for efficiency rather than being computed in real-time.

The demodulator structure is illustrated in Figure 11. Incoming samples from the audio interface are multiplied by four complex sinusoids, corresponding to the four tones. The result of this is a demodulated baseband pulse and a signal component at twice the tone frequency, which is removed with the filter $h_f(t)$. Using complex sinusoids for demodulation has the advantage of producing complex-baseband representation of the received pulses, which can be viewed as a convenient way of separating magnitude and phase information in the received signal. Since the demodulator is non-coherent, only magnitude information is used to make the symbol detection decision. Thus, after filtering the received pulse with a (matched) pulse filter $h_s(t)$ the magnitude information of the signal is extracted by taking the magnitude of the complex signal and discarding the phase information.

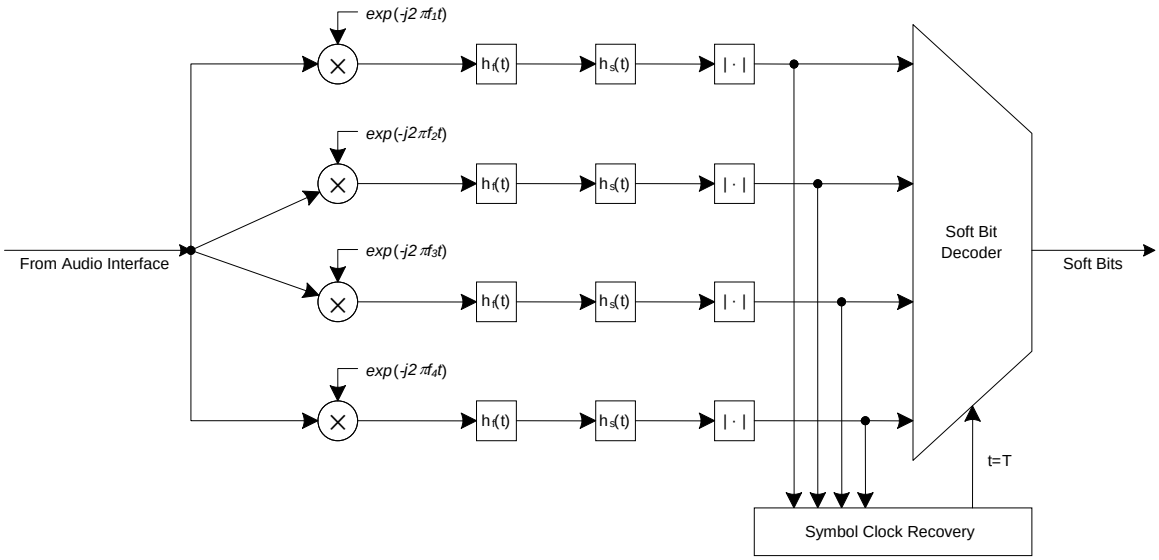


Figure 11 – Structure of 4-FSK demodulator

The magnitude output of the each matched filter detector is then fed into the soft-bit decoder. When triggered at the appropriate sampling instant, as determined by symbol clock recovery, this decoder simply chooses the tone with the greatest energy and produces the two output bits associated with the chosen symbol. Each of the output bits is weighted by the relative certainty of the decision as measured by the relative difference between the four matched filter outputs. These soft bits are then used by the Viterbi decoder to produce more accurate error metrics during decoding.

Several methods may be used to recover the symbol clock timing. Timing recovery is necessary to ensure that decoding decisions are made at the optimal point. As an example, Figure 12 illustrates a received and filtered pulse. The optimal sampling instant is at the peak of the pulse. At this point, received energy is maximized and interference from other tones is minimized, assuming the filters have been designed for minimal inter-symbol-interference. To discover this optimal sampling instant, a maximum-seeking search will be used to find the peak of the pulse shape. While this technique is computationally intensive since every possible offset across the symbol must be tested in a linear manner, it offers very fast synchronization acquisition, even on the initial symbol of a transmission.

As in the case of the modulator, most operations performed within the demodulator map nicely into MAC operations. Complex signals are represented as two separate signals, corresponding to in-phase and quadrature components, for all processing. Once again, all reference tones in both in-phase and quadrature form are stored in look-up tables for computational efficiency.

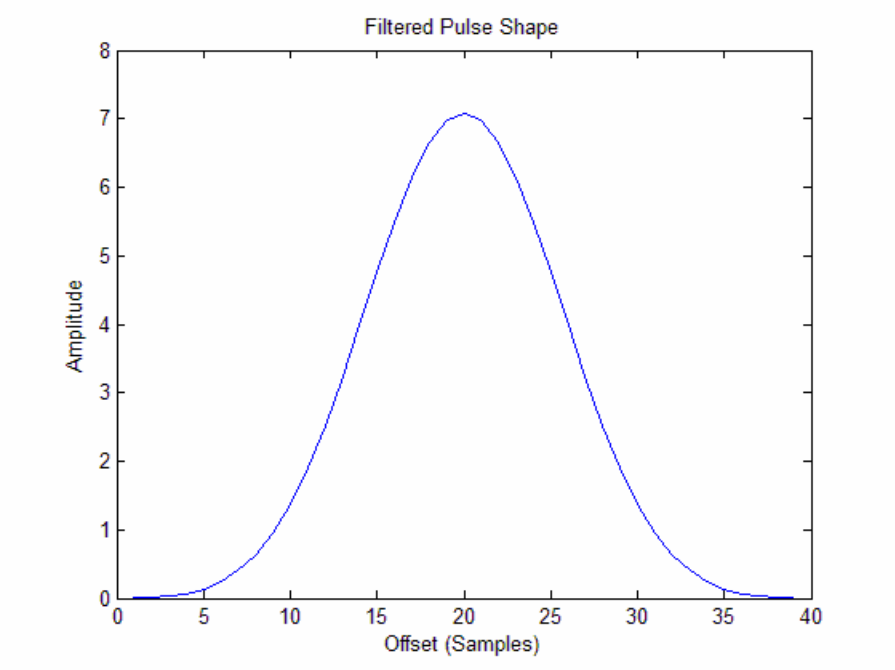
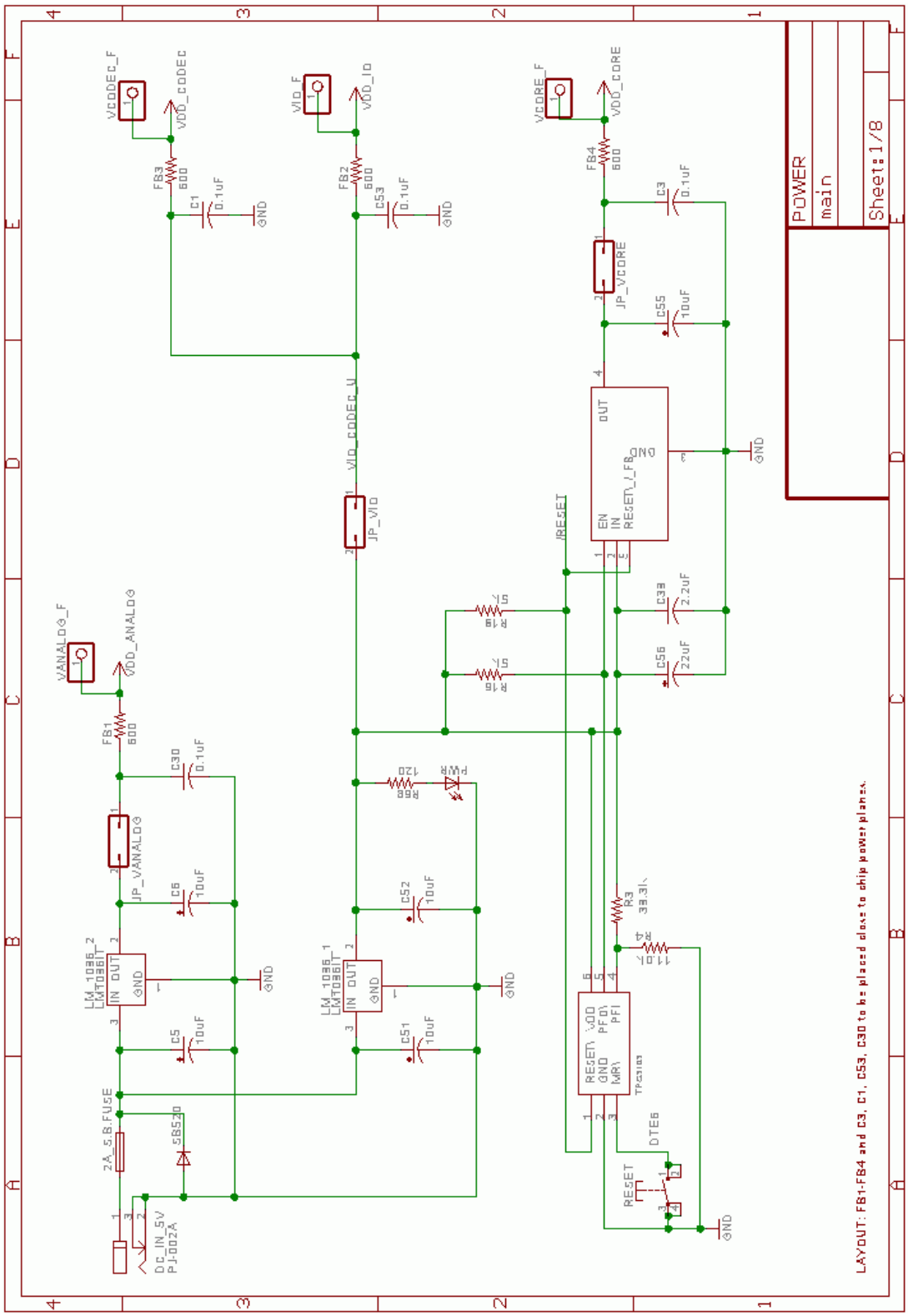
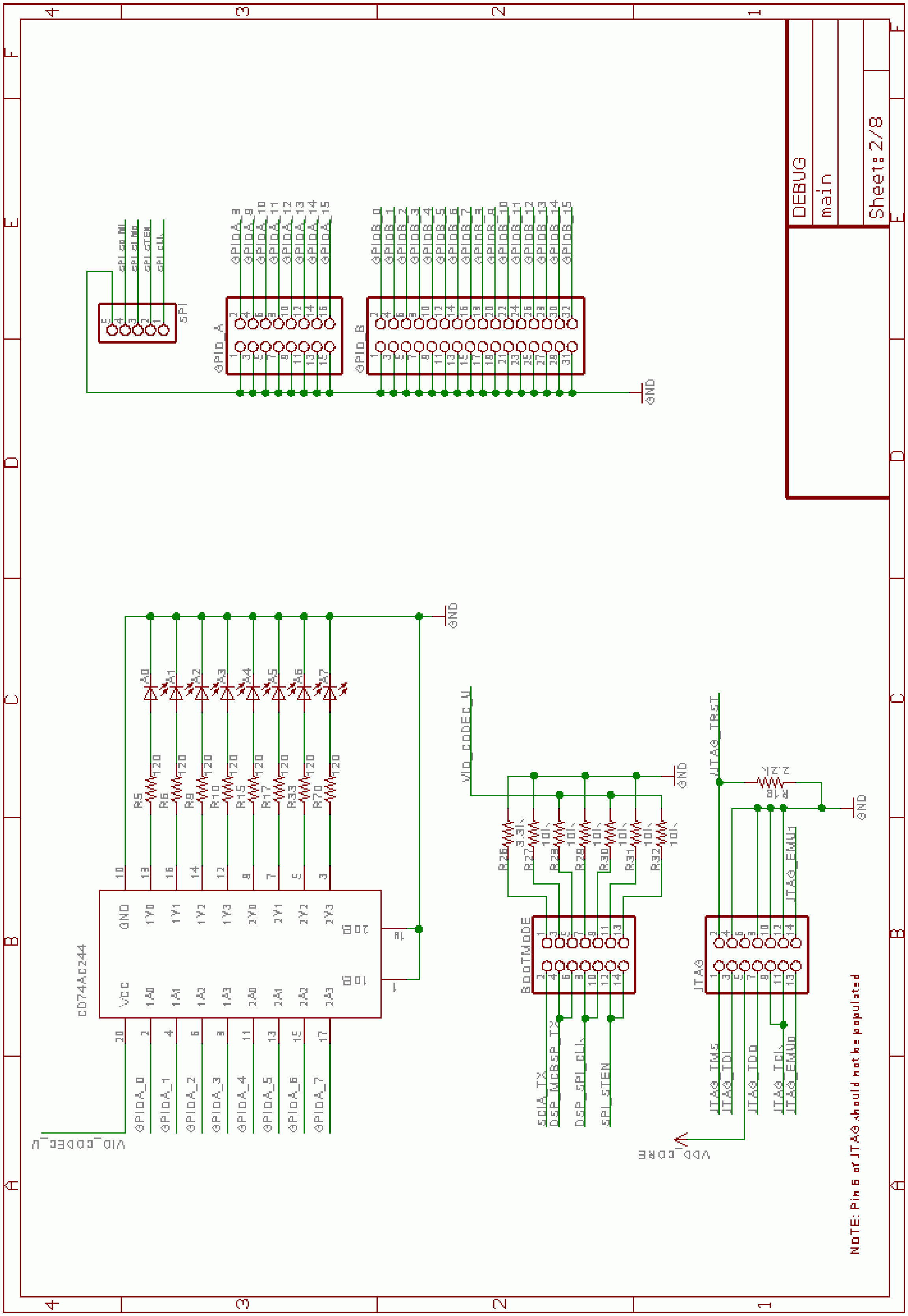
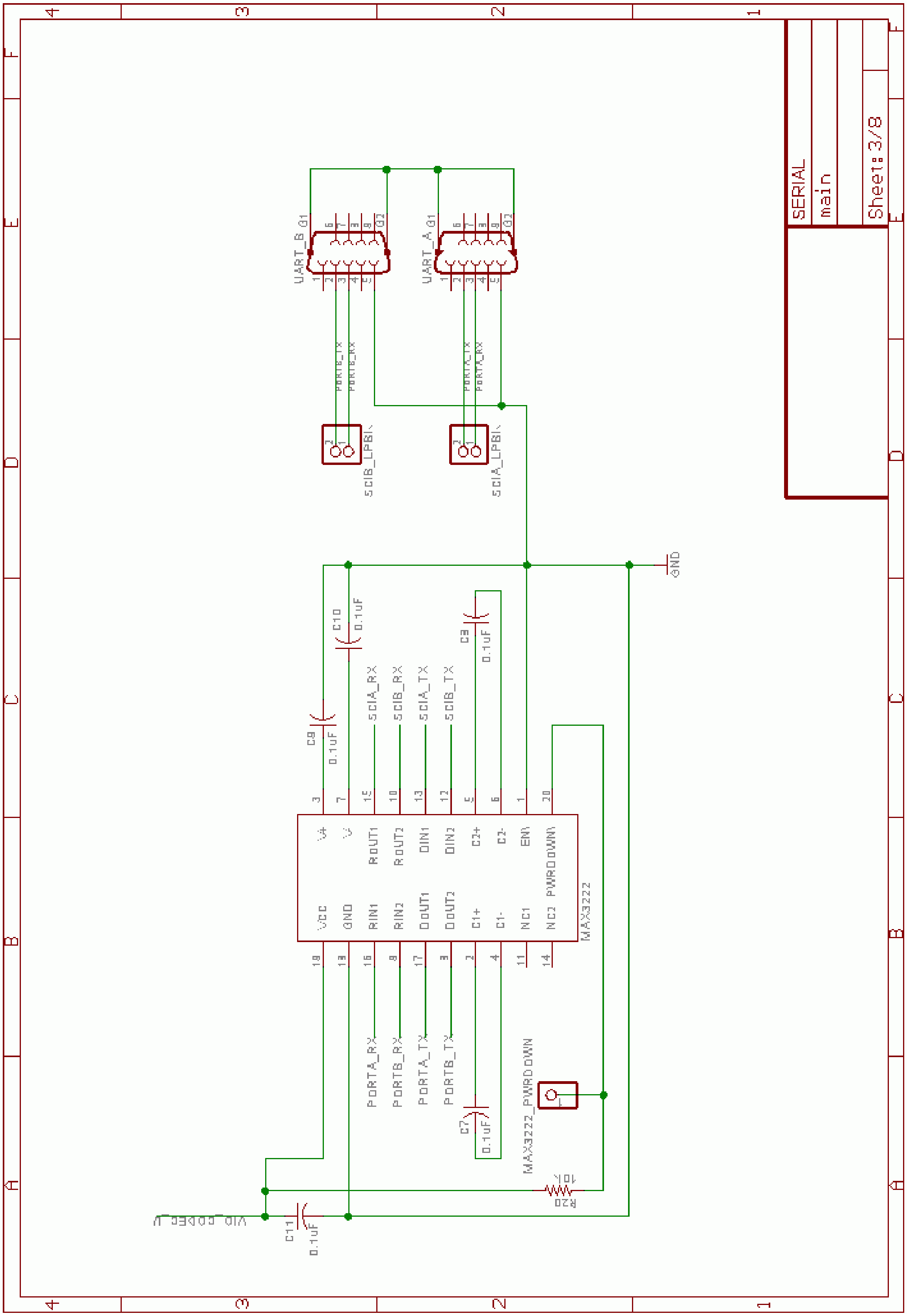


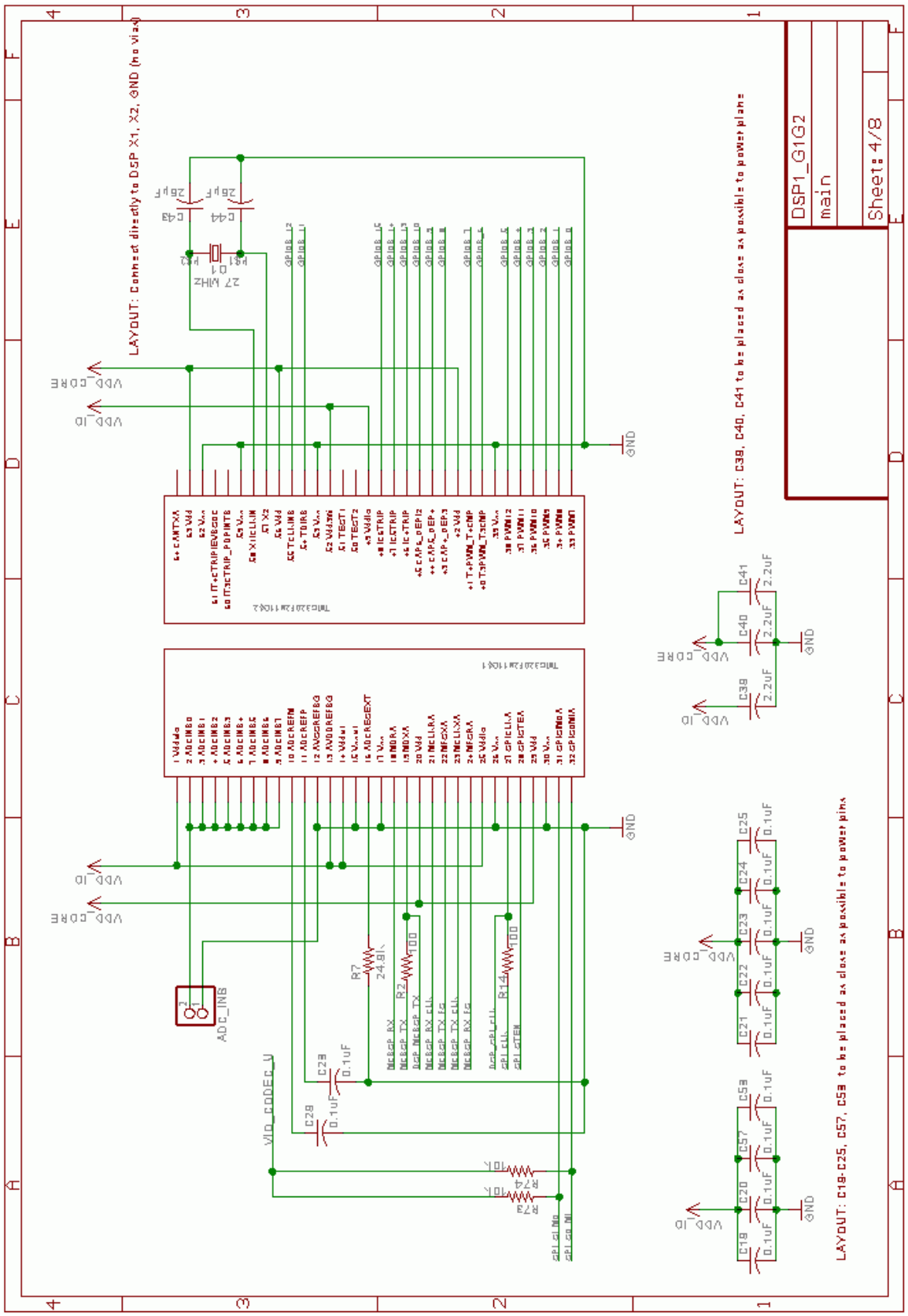
Figure 12 – Sample pulse shape after demodulation and matched filtering

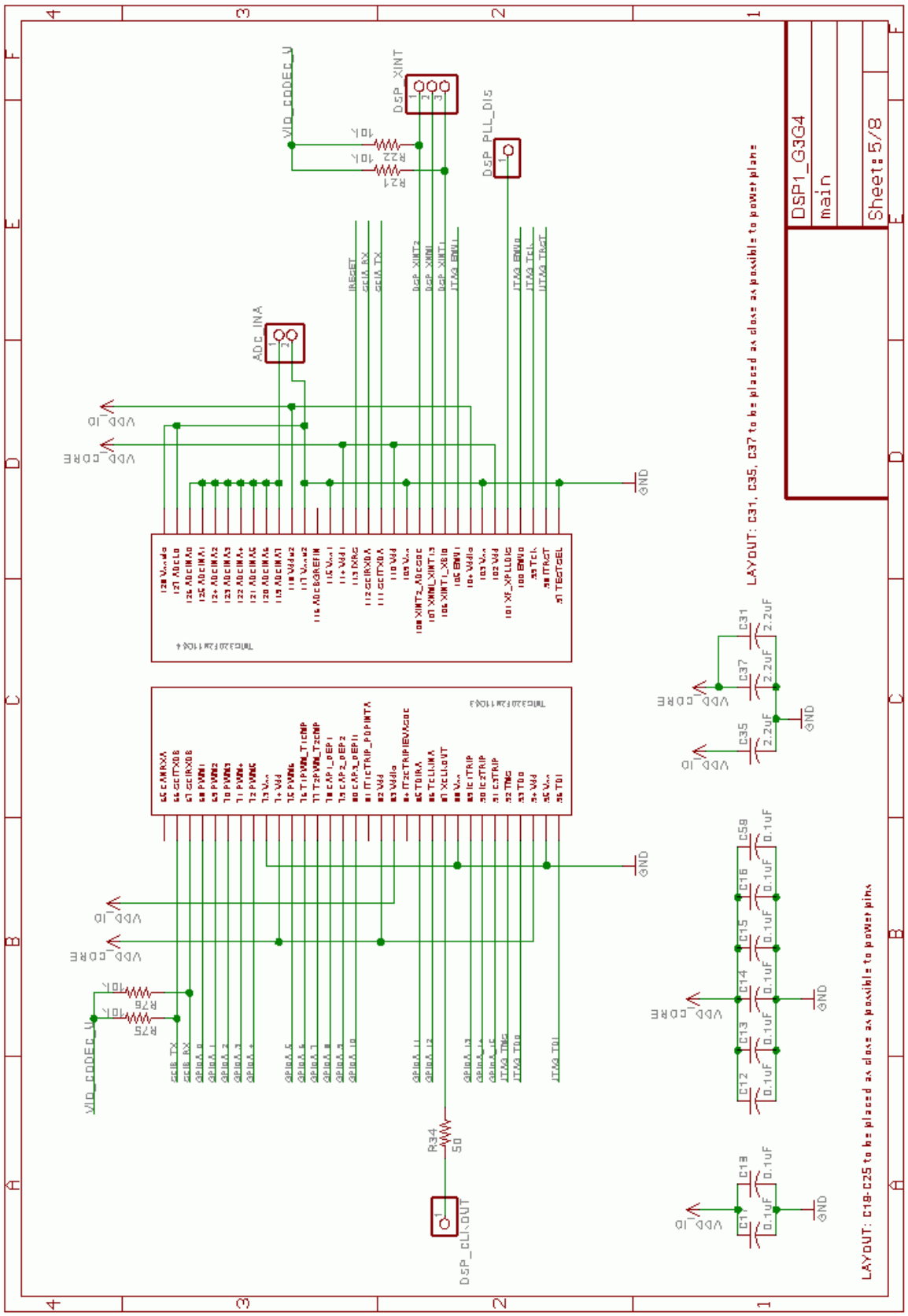
APPENDIX A - Hardware Design Schematics

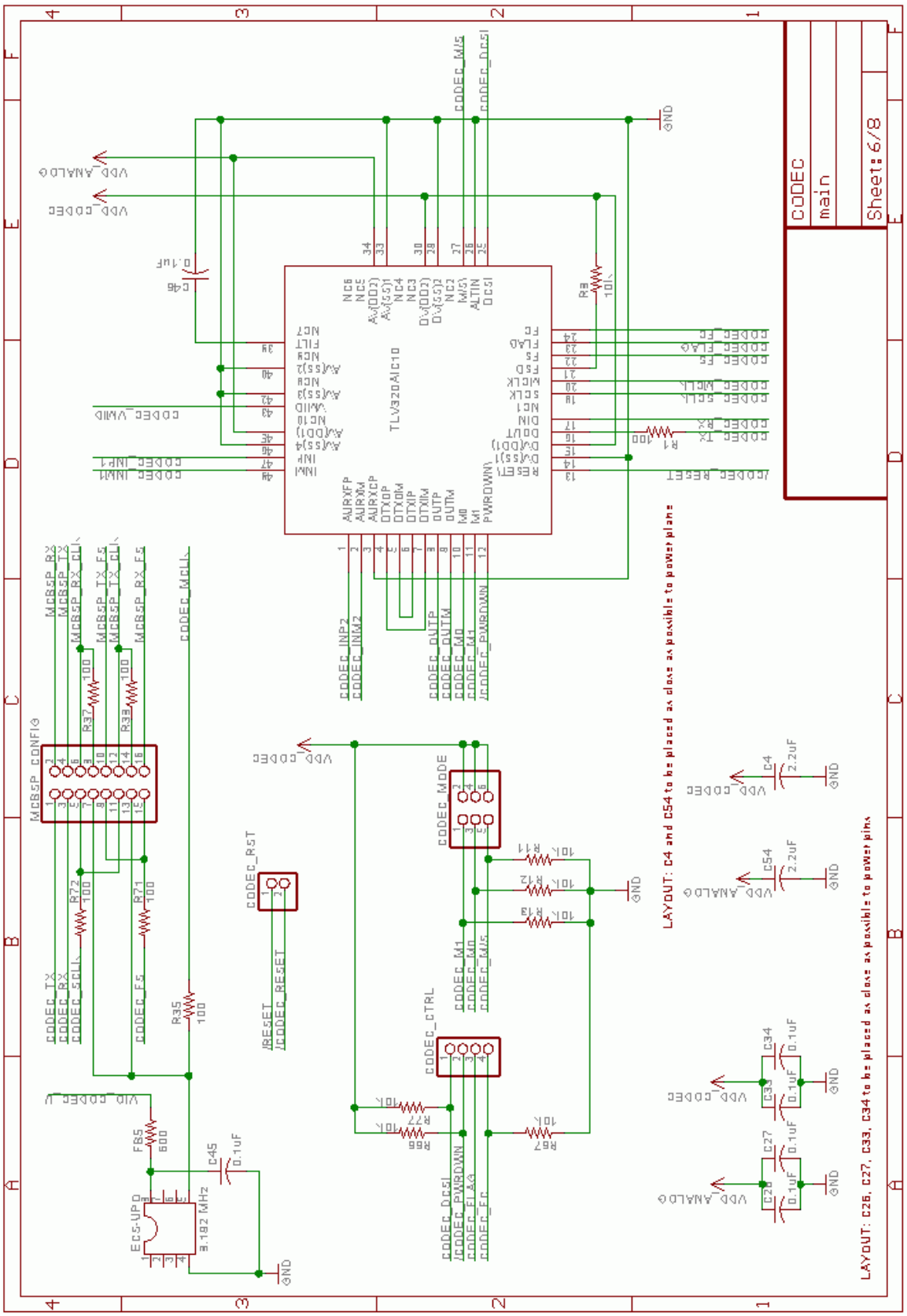


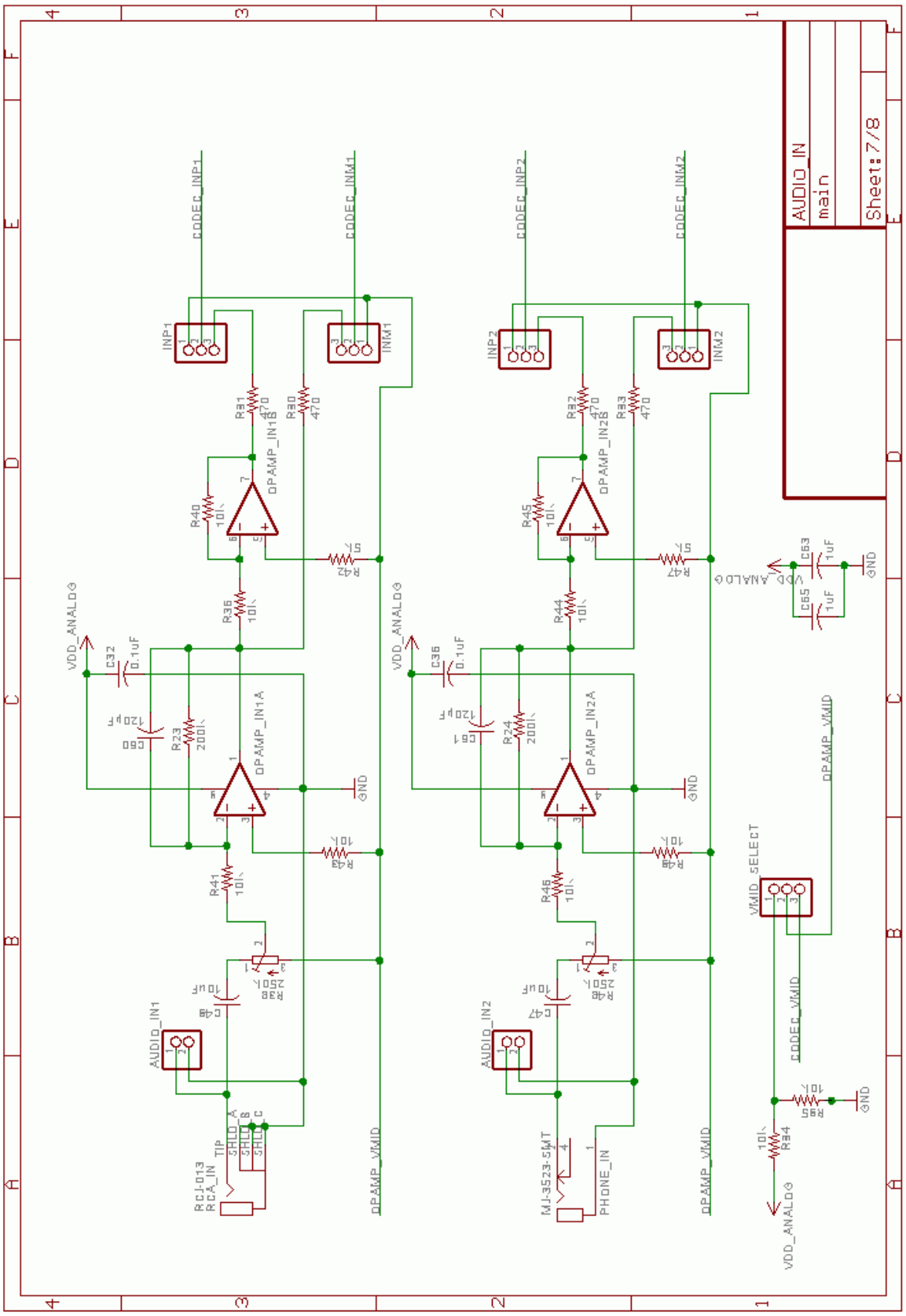

DEBUG
main
Sheet: 2/8
SPI
GPIO_A
GPIO_B
GND
CD74AC244
VID_CODEC_U
BOOTMODE
JTAG
VDD_CORE
NOTE: Pin 6 of JTAG should not be populated

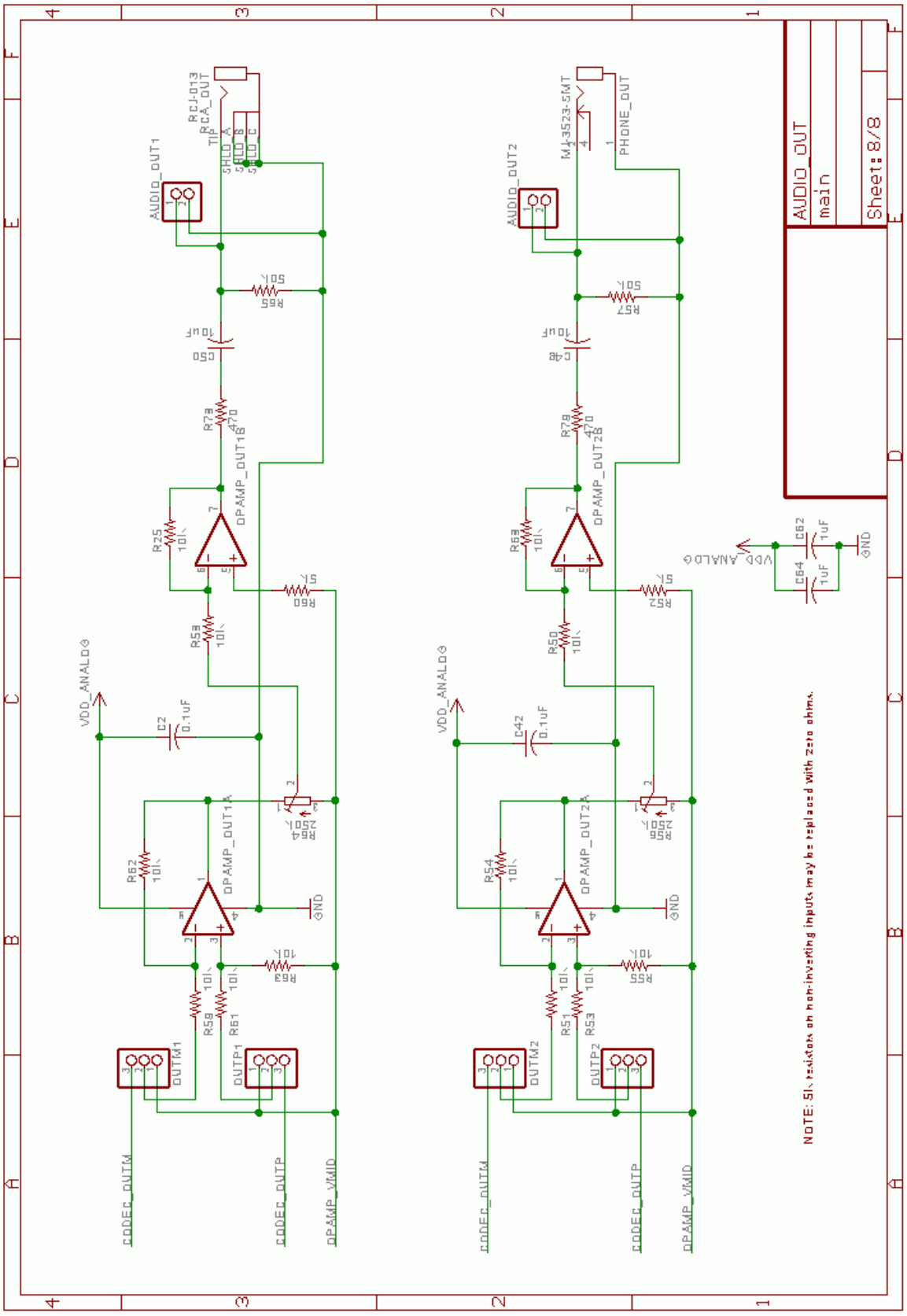

SERIAL
main
Sheet: 3/8
UART_B
UART_A
SCIB_LPBK
SCIA_LPBK
PORTB_TX
PORTB_RX
PORTA_TX
PORTA_RX
GND
C10
0.1uF
C8
0.1uF
C9
0.1uF
C7
0.1uF
C11
0.1uF
SCIA_RX
SCIB_RX
SCIA_TX
SCIB_TX
MAX3222
MAX3222_PWRDOWN
R20
10K


DSP1_G1G2
main
Sheet: 4/8
LAYOUT: Connect directly to DSP X1, X2, GND (no via)
LAYOUT: C38, C40, C41 to be placed as close as possible to power plane
LAYOUT: C18-C25, C57, C58 to be placed as close as possible to power plane
27 MHZ
C43
C44
28pF
VDD_CORE
VDD_IO
GND
ADC_INB
VID_CODEC_V
C41
C40
C38
2.2uF
C25
C24
C23
C22
C21
C58
C57
C20
C19
0.1uF
R7
24.9k
R2
100
R14
100
R73
10k
R74
10k


DSP1_G3G4
main
Sheet: 5/8
LAYOUT: C31, C35, C37 to be placed as close as possible to power pins
LAYOUT: C19-C25 to be placed as close as possible to power pins
DSP_XINT
DSP_PLL_DIS
ADC_INA
DSP_CLKOUT
VID_CODEC_V
VDD_IO
VDD_CORE
GND
R34
50
R75
R76
10K
C31
C35
C37
2.2uF
C59
C16
C15
C14
C13
C12
C19
C17
0.1uF


CODEC
main
Sheet: 6/8
TLV320AIC10
LAYOUT: C4 and C54 to be placed as close as possible to power pins
LAYOUT: C26, C27, C33, C34 to be placed as close as possible to power pins


AUDIO_IN
main
Sheet:7/8
CODEC_INP1
CODEC_INM1
CODEC_INP2
CODEC_INM2
INP1
INM1
INP2
INM2
OPAMP_IN1B
OPAMP_IN2B
OPAMP_IN1A
OPAMP_IN2A
VDD_ANALOG
GND
OPAMP_VMID
CODEC_VMID
VMID_SELECT
AUDIO_IN1
AUDIO_IN2
PHONE_IN
RCJ-013
RCA_IN
MJ-3523-5MT


AUDIO_OUT
main
Sheet: 8/8
RCJ-013
RCA_OUT
TIP
SHLD_A
SHLD_B
SHLD_C
AUDIO_OUT1
AUDIO_OUT2
MJ-3523-SMT
PHONE_OUT
R65
R57
C50
C48
10uF
R73
R79
470
OPAMP_OUT1B
OPAMP_OUT2B
R25
R63
10K
R60
R52
R58
R50
VDD_ANALOG
C2
C42
0.1uF
C62
C64
1uF
GND
OPAMP_OUT1A
OPAMP_OUT2A
R64
R56
250K
R62
R54
R59
R61
R51
R53
R55
OUTM1
OUTP1
OUTM2
OUTP2
CODEC_OUTM
CODEC_OUTP
OPAMP_VMID
NOTE: Si resistors on non-inverting inputs may be replaced with zero ohms.

APPENDIX B – Hardware Interface Header File

```
//*****************************************************************************
//
//              Copyright (C) 2005.  The Marconi Project
//      All rights reserved.  No part of this program may be reproduced.
//
//     http://www.sgsolutions.ca/marconi
//
//=============================================================================
//
// Interface to hardware drivers.
//
//*****************************************************************************

#ifndef __HWINTERFACE_H__
       #define __HWINTERFACE_H__

#include "GlobalDefs.h"

//*****************************************************************************
//  D E F I N E S
//*****************************************************************************


//*****************************************************************************
//  E N U M S
//*****************************************************************************


//*****************************************************************************
//  G L O B A L    D E F I N I T I O N S
//*****************************************************************************


typedef enum
{
       HW_UART_ID_A,
       HW_UART_ID_B,
       NUM_HW_UART_ID
} HW_UART_ID_TYPE;


typedef enum
{
    HW_UART_PARITY_NONE,
    HW_UART_PARITY_EVEN,
    HW_UART_PARITY_ODD,
    NUM_HW_UART_PARITY
} HW_UART_PARITY_TYPE;


typedef enum
{
       HW_GPIO_ID_A,
       HW_GPIO_ID_B,
       NUM_HW_GPIO_ID
} HW_GPIO_ID_TYPE;


typedef enum {
    HW_CODEC_GAIN_0_DB,
    HW_CODEC_GAIN_MINUS_36_DB,
    HW_CODEC_GAIN_MINUS_30_DB,
    HW_CODEC_GAIN_MINUS_24_DB,
    HW_CODEC_GAIN_MINUS_18_DB,
    HW_CODEC_GAIN_MINUS_12_DB,
    HW_CODEC_GAIN_MINUS_9_DB,
    HW_CODEC_GAIN_MINUS_6_DB,
    HW_CODEC_GAIN_MINUS_3_DB,
    HW_CODEC_GAIN_PLUS_3_DB,
```

```
    HW_CODEC_GAIN_PLUS_6_DB,
    HW_CODEC_GAIN_PLUS_9_DB,
    HW_CODEC_GAIN_PLUS_12_DB,
    HW_CODEC_GAIN_PLUS_18_DB,
    HW_CODEC_GAIN_PLUS_24_DB,
    HW_CODEC_GAIN_MUTE,
    NUM_HW_CODEC_GAIN
} HW_CODEC_GAIN_TYPE;


typedef enum {
    HW_CODEC_SAMPLE_RATE_4_KHZ,
    HW_CODEC_SAMPLE_RATE_8_KHZ,
    HW_CODEC_SAMPLE_RATE_16_KHZ,
    NUM_HW_CODEC_SAMPLE_RATE
} HW_CODEC_SAMPLE_RATE_TYPE;


typedef enum {
    HW_CODEC_RX_CHANNEL_1,
    HW_CODEC_RX_CHANNEL_2,
    NUM_HW_CODEC_RX_CHANNEL
} HW_CODEC_RX_CHANNEL_TYPE;


//*****************************************************************************
//  E X T E R N A L    V A R I A B L E S
//*****************************************************************************


//*****************************************************************************
//  F U N C T I O N    P R O T O T Y P E S
//*****************************************************************************

//*****************************************************************************
//
// FUNCTION:    HwInit - Initializes DSP hardware (PLL, flash wait states,
//                  clock enables, etc.)
//
// USAGE:       Must be called at startup before any hardware peripherals
//              are accessed.
//
//
// INPUT:       None
//
// OUTPUT:      None
//
//*****************************************************************************
void HwInit(void);


//*****************************************************************************
//
// FUNCTION:    HwUartOpen - opens the specified hardware UART with a
//              specific baud rate and parity setting.
//
// USAGE:       Must be called before any other calls are made for the
//              specified UART. After closing with HwUartClose(...), a
//              UART may be reopened with different settings.
//
// INPUT:       uartId - one of HW_UART_ID_A or HW_UART_ID_B
//              rate - the baud rate in bits / second
//              parity - one of HW_UART_PARITY_NONE, HW_UART_PARITY_EVEN,
//                  or HW_UART_PARITY_ODD
//
// OUTPUT:      none
//
//*****************************************************************************
void HwUartOpen(HW_UART_ID_TYPE uartId, DWORD baud, HW_UART_PARITY_TYPE parity);
```

```
//*****************************************************************************
//
// FUNCTION:    HwUartClose - closes the specified hardware UART.
//
// INPUT:       uartId - one of HW_UART_ID_A or HW_UART_ID_B
//
// USAGE:       Call when a UART is no longer in use or before
//              reconfiguring.
//
// OUTPUT:      none
//
//*****************************************************************************
void HwUartClose(HW_UART_ID_TYPE uartId);


//*****************************************************************************
//
// FUNCTION:    HwUartRd - reads one or more byes from the specified UART.
//
// INPUT:       uartId - one of HW_UART_ID_A or HW_UART_ID_B
//              buffer - pointer to buffer in which to store received data
//              length - number of bytes to read
//
// USAGE:       This function is non-blocking and returns immediately even
//              if an insufficient number of bytes are available for reading.
//
// OUTPUT:      return value - the actual number of bytes stored in buffer
//
//*****************************************************************************
WORD HwUartRd(HW_UART_ID_TYPE uartID, BYTE *buffer, WORD length);


//*****************************************************************************
//
// FUNCTION:    HwUartRdAvail - returns the number of received bytes
//              available for reading.
//
// INPUT:       uartId - one of HW_UART_ID_A or HW_UART_ID_B
//
// OUTPUT:      return value - number of received bytes available
//
//*****************************************************************************
WORD HwUartRdAvail(HW_UART_ID_TYPE uartId);


//*****************************************************************************
//
// FUNCTION:    HwUartWr - writes one or more bytes to the specified UART.
//
// INPUT:       uartId - one of HW_UART_ID_A or HW_UART_ID_B
//              buffer - pointer to buffer from which to write data
//              length - number of bytes to write
//
// USAGE:       This function is non-blocking and returns immediately even
//              if insufficient space exists for writing data.
//
// OUTPUT:      return value - number of bytes written from buffer
//
//*****************************************************************************
WORD HwUartWr(HW_UART_ID_TYPE uartID, const char *buffer, WORD length);
```

```
//*****************************************************************************
//
// FUNCTION:    HwUartWrAvail - returns number of bytes that can be written.
//
// INPUT:       uartId - one of HW_UART_ID_A or HW_UART_ID_B
//
// OUTPUT:      return value - number of bytes that may be written
//
//*****************************************************************************
WORD HwUartWrAvail(HW_UART_ID_TYPE uartId);


//*****************************************************************************
//
// FUNCTION:    Sets bits on the specified GPIO pin set.
//
// INPUT:       gpioId - one of HW_GPIO_ID_TYPE indicating the port to use
//              pattern - a 16-bit pattern corresponding to the 16 bits of the
//                  given GPIO port. If bit n is set to 1, GPIO pin n will be
//                  set. If bit n is set to 0, no change is made to pin state.
//
// OUTPUT:      none
//
//*****************************************************************************
void HwGpioSet(HW_GPIO_ID_TYPE gpioId, WORD pattern);


//*****************************************************************************
//
// FUNCTION:    Clears bits on the specified GPIO pin set.
//
// INPUT:       gpioId - one of HW_GPIO_ID_TYPE indicating the port to use
//              pattern - a 16-bit pattern corresponding to the 16 bits of the
//                  given GPIO port. If bit n is set to 1, GPIO pin n will be
//                  cleared. If bit n is set to 0, no change is made to pin
//                  state.
//
// OUTPUT:      none
//
//*****************************************************************************
void HwGpioClear(HW_GPIO_ID_TYPE gpioId, WORD pattern);


//*****************************************************************************
//
// FUNCTION:    HwGpioSetOutputEnable - sets the specified GPIO pins for output
//              mode (as opposed to input mode).
//
// INPUT:       gpioId - one of HW_GPIO_ID_TYPE indicating the port to use
//              pattern - a 16-bit pattern corresponding to the 16 bits of the
//                  given GPIO port. If bit n is set to 1, GPIO pin n will be
//                  enabled for output. If bit n is set to 0, no change is
//                  made.
//
// OUTPUT:      none
//
//*****************************************************************************
void HwGpioSetOutputEnable(HW_GPIO_ID_TYPE gpioId, WORD pattern);
```

```
//*****************************************************************************
//
// FUNCTION:    HwGpioClearOutputEnable - sets the specified GPIO pins for
//              input mode (as opposed to output mode).
//
// INPUT:       gpioId - one of HW_GPIO_ID_TYPE indicating the port to use
//              pattern - a 16-bit pattern corresponding to the 16 bits of the
//                  given GPIO port. If bit n is set to 1, GPIO pin n will be
//                  enabled for input. If bit n is set to 0, no change is
//                  made.
//
// OUTPUT:      none
//
//*****************************************************************************
void HwGpioClearOutputEnable(HW_GPIO_ID_TYPE gpioId, WORD pattern);


//*****************************************************************************
//
// FUNCTION:    HwGpioRead - returns the current GPIO input logic state.
//
// INPUT:       gpioId - one of HW_GPIO_ID_TYPE indicating the port to use
//
// OUTPUT:      return value - the input state of each pin of the specified
//              GPIO port, where bit n corresponds to pin n
//
//*****************************************************************************
WORD HwGpioRead(HW_GPIO_ID_TYPE gpioId);


//
// Codec Interface
//


void HwCodecOpen(HW_CODEC_SAMPLE_RATE_TYPE sampleRate);
void HwCodecClose(void);
BOOL HwCodecSetRxPGAGain(HW_CODEC_GAIN_TYPE gain);
BOOL HwCodecSetTxPGAGain(HW_CODEC_GAIN_TYPE gain);
BOOL HwCodecSetRxChannel(HW_CODEC_RX_CHANNEL_TYPE channel);
BOOL HwCodecSetLoopback(BOOL enable);
WORD HwCodecRdAvail(void);
WORD HwCodecRd(WORD *buffer, WORD count);
WORD HwCodecWrAvail(void);
WORD HwCodecWr(WORD *buffer, WORD count);


//
// Timer Interface
//


void HwTimerConfigure(WORD delayInMS, void (*funcPtr)(void));
void HwTimerEnable(void);
void HwTimerDisable(void);
DWORD HwTimerGetCurrentTicks(void);
DWORD HwTimerGetTicksPerMS(void);

#endif
```