SILICON LABORATORIES

AN147

WIRELESS DIGITAL FULL-DUPLEX VOICE TRANSCEIVER

Relevant Devices

This application note applies to the following devices: C8051F330.

1. Introduction

Wireless voice applications require a microcontroller to quickly compress and decompress audio, efficiently control an RF transceiver, and accurately sample and output an audio stream. With its small, 4x4 mm size and high level of integration combining an on-chip ADC, DAC, and a 25 MIPS peak CPU, the C8051F330 is an ideal choice for wireless audio designs. This reference design demonstrates how an F330 can be utilized for such applications.

This document includes the following:

- A description of the system hardware
- PCB design notes
- A discussion about the software system
- Notes on system usage
- Example code to sample, compress, transmit, receive, decompress, and output a voice signal
- A schematic, bill of materials, and an example board layout showing the F330, RF transceiver, and support components

Key Points

- The F330's on-chip ADC, DAC, SPI, calibrated internal oscillator, and small 4x4 mm package allow designers to decrease PCB size and cost in wireless voice applications.
- The voice signal sampled by the ADC is compressed before transmission using the high-speed 25 MIPS peak CPU to overcome RF bandwidth limitations.
- Software synchronizes RX and TX mode switching between endpoints to maintain a stable RF link and to support full-duplex audio.
- Buffer overflows and underflows along the signal path can be prevented by dynamically adjusting ADC and DAC sampling rates.
- To minimize power consumption, the software system only transmits and receives data when the audio signals are at audible volume levels.
- Careful component placement during board layout is required to ensure optimal RF performance.

2. Overview

Figure 1 shows the voice signal path as it is filtered, sampled, compressed, transmitted, received, decompressed, and output by the microcontroller and the transceiver. Each stage of the process is discussed in the following sections.

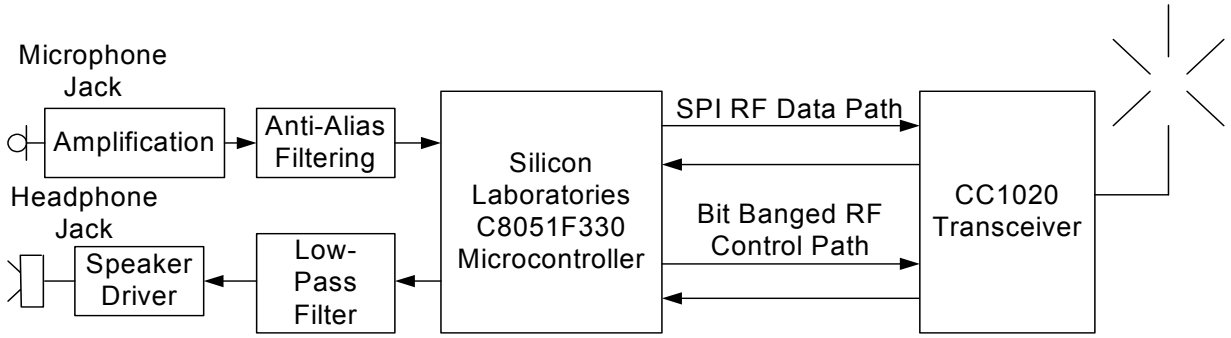


Figure 1. Signal Flow Diagram

2.1. Input Signal Filtering

The audio signal enters the board through a 3.5 mm microphone jack. Before the signal reaches the ADC, an op-amp amplifies the signal to a full-scale level of 0 to 2.43 V. The signal then passes through a low-pass anti-aliasing filter before reaching the microcontroller.

2.2. Signal Path in the Microcontroller

The 10-bit ADC samples at a rate of 8 kHz and pushes each sample onto an ADC buffer. Bytes are pulled from the buffer and compressed to a size of 4 bits using a DPCM algorithm discussed in "4.13.Audio Compression" on page 11. Compressed bytes are then pushed onto the SPI transmit buffer.

The RF transceiver sends and receives data wirelessly between endpoints. The data interface to the IF transceiver is implemented via software bit banging.

The receiving microcontroller stores the transmitted, compressed voice stream into a receive buffer. Compressed data is pulled from this buffer by a DPCM decompression routine that converts each compressed byte to two 10-bit full-scale DAC output values. Uncompressed samples are pushed onto a DAC output buffer and pulled from this buffer by the DAC at a rate of 8 kHz.

2.3. Signal Output

To help reduce noise, the DAC output is conditioned using a 5-pole Butterworth low-pass filter. After filtering, the signal is biased to a mid-rail value and boosted in power before it reaches the speaker driver for output.

3. Hardware Description

The Appendices contain a schematic and sample board layout of the design discussed below.

3.1. Microphone Input

The microphone audio signal is amplified in order to take advantage of the ADC's full-scale input voltage of 2.43 V. A gain of 10 provides adequate amplification while also avoiding audio clipping.

When sampling audio at a given frequency, signals above one-half the sampling frequency will appear in the sample as an aliased signal and can degrade audio performance. Placing a low-pass filter with a corner frequency at half the sampling frequency attenuates these aliased signals and improves audio quality. In this design, the input signal passes through a 3-pole filter with a corner frequency of 4 kHz.

3.2. Peripherals

This design utilizes three of the C8051F330's peripherals: the ADC, SPI, and DAC. Each peripheral's functionality is described in the following sections.

3.2.1. 10-Bit ADC

The 10-bit ADC is set to sample at 8 kHz using Timer 2 overflows as the start-of-conversion clock source. The ADC uses the on-chip voltage reference of 2.43 V. As is recommended by the C8051F330's data sheet, a 0.1 µF capacitor and a 4.7 µF capacitor are placed in parallel between the V_{REF} pin and ground for noise filtering.

3.2.2. SPI

The SPI peripheral in slave mode provides the data interface between the RF transceiver and the microcontroller. Controlled by an RF transceiver-sourced 153.6 kbps clock, the SPI peripheral shifts in and out compressed bytes of audio data. Data flow for the SPI depends on whether the transceiver has been configured to receive or transmit. The MISO and MOSI lines of the SPI are tied together and connected to the RF transceiver's bi-directional DIO data pin. Contention between the three pins is avoided by managing the output mode of MOSI and MISO using the port configuration registers.

Because the SPI's NSS pin is not controlled by the RF transceiver, the peripheral has no hardware-driven indication of byte boundaries in the incoming data stream, and so this data framing process must then be handled in software. The procedure whereby a "Sync Word" is used to byte-align the SPI is discussed later in "4.8.Receiving a Data Packet" on page 7. SPI's interrupt service routine contains the RX/TX State Machine that controls the RF link. For a detailed discussion on this state machine, refer to "4. Software Description".

3.2.3. 10-Bit DAC

The C8051F330's Current DAC outputs samples at a frequency of 8 kHz using Timer 3 interrupts. A 1 kΩ resistor and a 33,000 pF capacitor are connected in parallel between the IDAC output pin and ground to convert the output current into a voltage and to provide some filtering.

3.3. PCB Design Considerations

Following some simple guidelines in board layout will ensure optimal audio and RF performance. When placing components, power supply decoupling capacitors should be located as close to the power and ground pins as possible. To prevent digital noise from coupling into nearby voice and RF analog traces, digital and analog traces should not be routed next to each other. When routing traces that carry high frequency digital data signals, avoid the use of vias in order to minimize the amount of radiated noise.

3.4. RF Transceiver Placement

Special care must be taken when placing the RF transceiver and its associated components. For optimal performance, follow the recommendations and guidelines given by the RF transceiver manufacturer.

4. Software Description

The software system facilitates a full-duplex voice signal stream by creating a bi-directional signal flow path across a half-duplex RF link. The software system takes advantage of both continuously executing foreground routines and peripheral-controlled interrupts to guarantee defined data flow rates. The RF link is maintained while audible audio signal is present on either endpoint's audio input.

The sections below discuss each part of the software system, including idle channel power conservation methods, active channel detection, the RX/TX synchronization method, and an example compression algorithm. All code described in these sections can be found in "Appendix D—Firmware Listing" on page 18.

4.1. RF Transceiver Initialization and Calibration

Before the microcontrollers can communicate across the RF link, the RF transceiver must be initialized and calibrated to transmit and receive. At powerup, the RF transceiver's registers controlling all aspects of its functionality, including bit rates, filter bandwidth, and PLL-related settings are set to optimal values for transmission at 153.6 kbps. Settings for each of the RF transceiver's registers can be computed by running Chipcon's SmartRF Studio configuration wizard, which can be found at www.chipcon.com.

After the registers have been correctly configured, the RF transceiver must be calibrated successfully. The CC1020 data sheet contains a detailed description of the calibration procedure

4.2. Disabling the Watchdog Timer

During microcontroller reset, the Watchdog Timer is automatically enabled by hardware. Approximately 250 ms after being enabled, the Watchdog Timer will overflow and issue a microcontroller reset. In most software systems, the Watchdog Timer is typically disabled at the beginning of the main() function.

In this design, many variables are initialized in XDATA space before main() is executed, and the Watchdog Timer must be disabled before these variables are initialized. To disable the Watchdog Timer, the file STARTUP.A51 must be modified. See the code in "Appendix D—Firmware Listing" on page 18 for an example of a modified version of STARTUP.A51 that disables the watchdog timer immediately after device reset.

4.3. Idle Channel System State

When neither endpoint's audio endpoint is of great enough amplitude to warrant transmission, the Communication Channel can be considered Idle and the RF link need not be maintained.

The RF link will not be established until one of two events occurs: either the local audio signal assumes sufficient amplitude to warrant transmission or a remote audio signal is detected during a search on the RF link's transmission frequency. When one of these two events occurs, the software configures the system to operate as a Master or a Slave on the Communication Channel, respectively.

4.4. Establishing the Communication Channel

When the Communication Channel becomes Active, one endpoint is designated as Master and the other as Slave. The Master on the channel is responsible for maintaining timing and RX/TX switching synchronization across the link.

4.5. Master Mode Endpoint

The endpoint that first captures an audible signal is designated as the Master endpoint. This endpoint configures the RX/TX state machine to transmit the first data packet of the Communication Session and controls packet transmission timing, switching, and the termination of the Communication Channel. Figure 2 shows how a data packet is formatted. Data packet structure is discussed in greater detail below.

4.6. Slave Mode Endpoint

When one endpoint initiates the creation of a Communication Channel by transmitting, the other endpoint must detect this transmission and respond in order for the Communication Channel to be maintained. Detection is accomplished by configuring the RF transceiver to receive mode and then examining the data stream for a valid data packet. Once a packet has been detected, the RX/TX state machine is configured to operate in Slave mode. This endpoint periodically synchronizes its RX/TX State Machine clock to the Master's clock.

Figure 2. Data Packet Format

4.7. Transmitting A Data Packet

The Transmit State Machine is shown in Figure 3. The transmission of a data packet begins with the configuration of the RF transceiver to transmit mode. This process is allowed a set amount of time to complete defined by the constant *SPI_CalibrationWaitTime*. To compensate for the possibility that the RF transceiver's PLL may not be locked at CalibrationWaitTime, the state machine then delays progress for a length of time defined by *SPI_SlopTimeOut*. The length of time the transceiver takes to calibrate can change depending on the configuration settings of the RF transceiver.

Next, the system transmits a number of bytes of the character '0x55'. These bytes, known as Preamble, appear to a receiver to be a stream of alternating 1s and 0s. Preamble allows the receiving transceiver to calibrate its data slicer, which is the component that distinguishes between 0s and 1s in the data stream. The constant *TX_NumBytesPreamble* defines the number of bytes transmitted. After all bytes of Preamble have been sent, a Sync Word is transmitted. In this system, the Sync Word is defined as "0xFFFFEE". This 24-bit word allows the receiver to align the SPI along byte boundaries. This process is discussed in greater detail in "4.8.Receiving a Data Packet" on page 7.

The local Audio State variable is transmitted next. This byte indicates to the receiver whether or not the transmitter's audio stream is "quiet" or "loud", meaning that the audio amplitude is less than or greater than a defined audio threshold. The Audio State is discussed in "4.12.Communication Channel Shutdown Process" on page 11. After the Audio State variable has been sent, compressed audio bytes are transmitted. These bytes are pulled from the TransmitFIFO buffer.

When a number of data bytes defined by *RXTX_BytesOfData* has been transmitted, the RX/TX state machine tweaks the ADC sampling rate through a process discussed in "4.11.Buffer Management" on page 10. Once the *SPI_Timer* reaches its terminal value, the state machine enters the *RXTX_TimeOut* state. This state is discussed in "4.9.End-of-Packet Decision Process" on page 10.

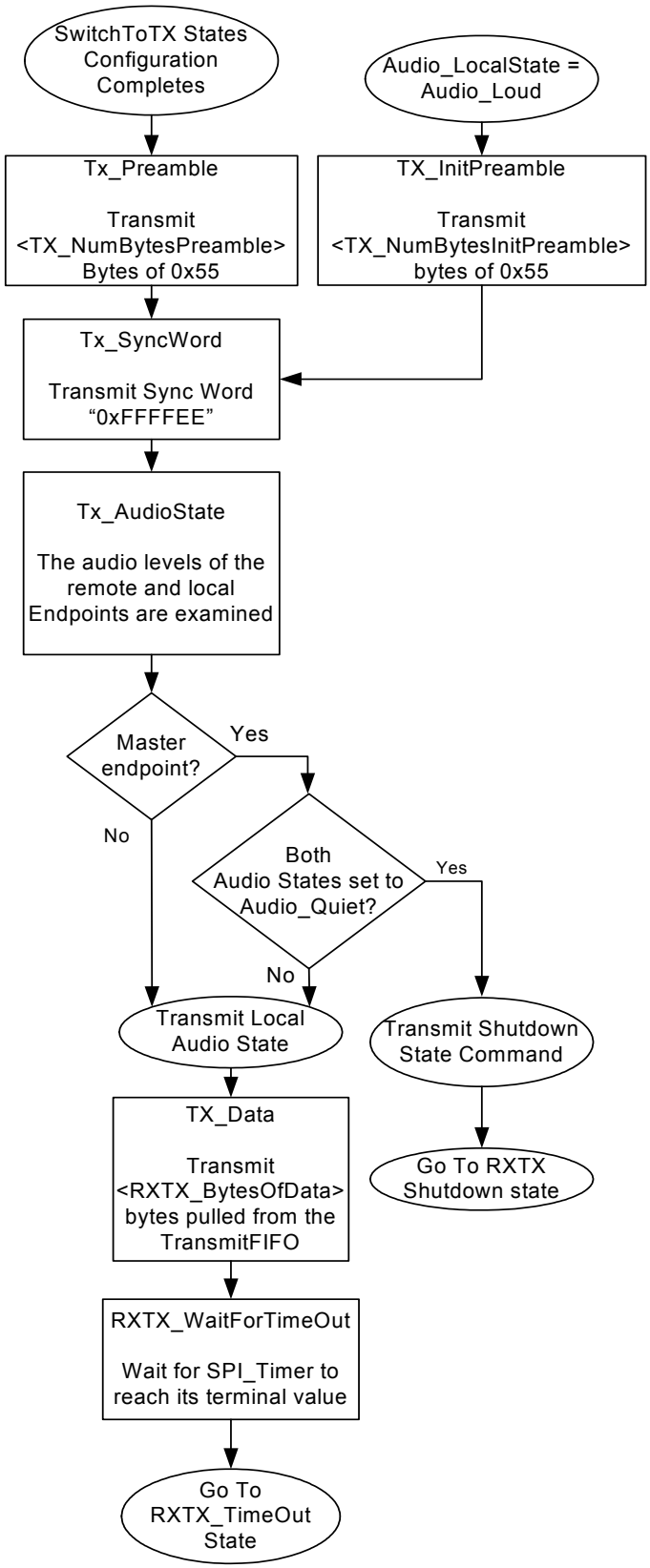


Figure 3. Transmit State Machine

4.8. Receiving a Data Packet

Figure 4 shows the Receive State Machine. After the transceiver has been configured to receive, the state machine searches for a stream of Preamble. Once the state machine has detected a number of bytes of Preamble defined by *RX_MinBytesPreamble*, it searches for a Sync Word, which is examined to determine where byte boundaries fall within the data stream. The receiver looks for the byte “0xFF”, which it must receive at some point during the reception of the stream containing the Sync Word. When the state machine finds 0xFF, it can be sure that Preamble has been completely received, and the bits currently being shifted into the microcontroller must be part of the Sync Word.

Once “0xFF” is received, the state machine disables SPI and enables a PCA module that has been configured to edge-triggered capture mode and routed through the crossbar to use the DIO line as its input. The third transmitted byte of the Sync Word is “0xEE”, which in binary is received as “11101110”. The PCA module is set to interrupt upon detecting a falling edge on an input pin, and upon reception of the first “0” in the word, the PCA interrupt flag will be set.

Once inside the PCA interrupt service routine (ISR), the system watches DIO to detect the second “0”, which is the last bit of the Sync Word. Whenever this “0” is found, the system then waits for a rising edge at the DCLK pin. Examination of DCLK is necessary because data is clocked out of the RF transceiver on the falling edge of DCLK, and should be read during the following DCLK rising edge. Once the next rising edge is detected, the PCA is disabled and SPI is re-enabled and routed back to the port pins using the crossbar. The re-enabled SPI is now aligned so that the first bit clocked in is the MSB of the first data byte of the packet.

The transmitting endpoint’s Audio State is received next, and transmission decisions at the receiving endpoint can be made based on this newly acquired information. Next, a number of data bytes defined by *RXTX_BytesOfData* are received. After the last byte of data has been received, the receiving endpoint performs the sample rate synchronization task discussed in "4.11.Buffer Management" on page 10. Also, if the receiving endpoint has been designated as the Slave Mode unit, then the microcontroller synchronizes *SPI_Timer*. This process is discussed in the section titled RX/TX State Machine Clock Synchronization. After *SPI_Timer* reaches its terminal value, the system enters the *RXTX_TimeOut* state.

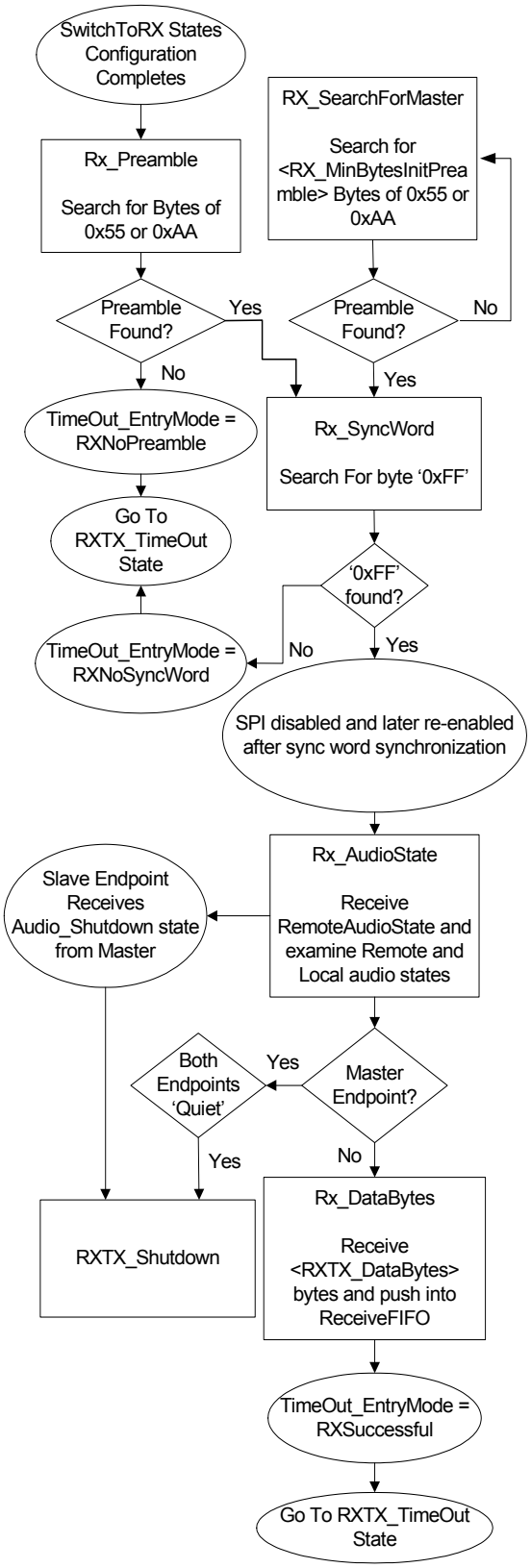


Figure 4. Receive State Machine

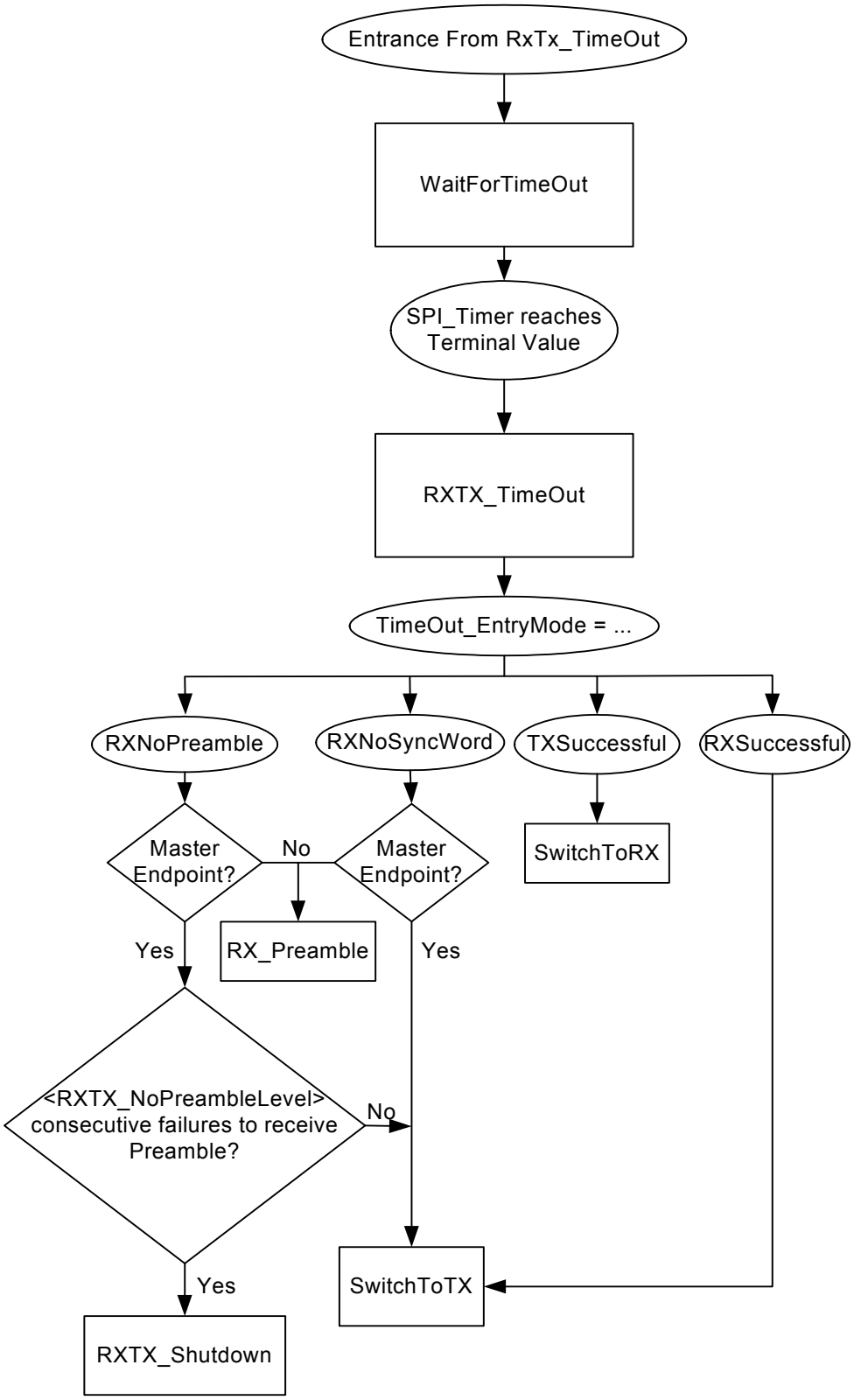


Figure 5. RX/TX Packet Time Out Decision Process

4.9. End-of-Packet Decision Process

Figure 5 shows the decision process that takes place inside the *RXTX_TimeOutState*. This state is entered after the allotted window of time for reception or transmission of a packet has expired.

A variable named *Timeout_EntryMode*, which has been set by the state machine before *RXTX_TimeOutState* was entered, indicates whether the just-completed transmission or reception was successful or unsuccessful, along with information on what caused the failure. *RXTX_TimeOutState* makes decisions on whether to transmit or receive a packet during the next time-slot based on *TimeOut_EntryMode*. Additionally, the master endpoint can decide to terminate the Communication Channel based on the value of *Timeout_EntryMode*.

4.10. RX/TX State Machine Clock Synchronization

While the Communication Channel is active, progress through the RX/TX state machine on both the master and slave endpoints is gated by a single clock source controlled by the master endpoint. The clock's value is stored in the variable *SPI_Timer*. When the clock source has been enabled, *SPI_Timer* will be incremented once each SPI interrupt. When *SPI_Timer* reaches the terminal value defined by the constant *SPI_PacketTime*, the *SPI_TimeOutEvent* flag is set.

SPI_Timer clock synchronization occurs at the conclusion of the slave endpoint's reception of a data packet. By this point, the slave endpoint has determined that a valid packet is being transmitted and has received the packet. Setting *SPI_Timer* to the value that the master endpoint's *SPI_Timer* is known to be at this point in reception eliminates drift in the slave endpoint's clock. The slave endpoint's clock may have drifted from this defined value because of issues including system clock discrepancies, Sync Word detection, and missed transmitted data packets.

4.11. Buffer Management

From compression at one endpoint to decompression at the other endpoint, an audio signal passes through four distinct data buffers. The ADCRXFIFO buffer stores raw 10-bit samples from the audio input, the TransmitFIFO buffer stores compressed bytes ready to be transmitted across the RF link, the ReceiveFIFO buffer stores the compressed bytes received through the RF link, and DACTXFIFO stores decompressed 10-bit audio that is ready to be output through the DAC.

The function *FIFO_ManagementRoutine()*, executed continuously inside main(void), calls compression and decompression routines. These routines shuttle data from the ADC buffer to the Transmit buffer and from the Receive Buffer to the DAC buffer. Additionally, *FIFO_ManagementRoutine()* pulls the oldest value from any buffer marked as full to prevent buffer overflow.

Small differences in microcontroller system clock and RF transceiver DCLK frequencies causes audio sampling rates for one endpoint to differ slightly from the other endpoint. If samples are compressed faster than they can be decompressed and output, audio information can be lost by data overflow or underflow.

This loss can be avoided by adjusting sampling rates during real-time packet transmission and reception. Immediately after a data packet has been transmitted, the ADC's sampling rate is adjusted by comparing the TransmitFIFO buffer's size to a target size defined by *UTXFIFO_TARGET*. Examining the buffer of collected and compressed ADC samples at a time governed by the RF transceiver-sourced SPI clock allows the system to dynamically compensate for timing discrepancies between the microcontroller and the RF transceiver. If the buffer contains too many samples, the system slows the sampling rate to reduce the speed at which new data is pushed to the buffer. Conversely, if the buffer is nearly empty, the sampling rate is increased so that data is placed on the buffer faster.

The receive path of the state machine also adjusts sampling rates to avoid buffer corruption. Immediately after a data packet has been received, the DAC output rate is adjusted according to how close the ReceiveFIFO buffer's size is to a target size defined by *URXFIFO_TARGET*. This comparison allows the system to compensate for the differences in clock frequencies between the two endpoints.

4.12. Communication Channel Shutdown Process

Communication across the RF link can be terminated if both endpoints are transmitting audio with amplitude below an audible threshold or if an endpoint has failed to receive a packet of audio from the other endpoint for a defined period of time.

4.12.1. Shutdown Due to Audio Silence

The Audio State variable is set inside the ADC ISR after audio has been sampled and the amplitude of the signal has been determined. *Audio_LocalState* will be set to either *Audio_Loud* or *Audio_Quiet* depending on the amplitude of the audio signal present at the ADC input. Samples are compared to a value defined by *Audio_QuietThreshold* to determine whether the signal is loud or quiet. The response time for this determination can be increased or decreased by adjusting the constants *AudioStateQuietToLoud* and *AudioStateLoudToQuiet*.

RX/TX State Machine decisions are made in states where *Audio_LocalState* is transmitted and received. Unless both endpoints' states are set to *Audio_Quiet*, transmission will continue. Only the master endpoint is allowed to terminate communication, and only after a special Audio State specified as *Audio_ChannelShutdown* has been transmitted. After this special state has been transmitted, the master transmitter can power down the transceiver. Simultaneously, the slave receiver will begin its own shutdown procedure after reception of the Audio_Channel Shutdown command.

4.12.2. Shutdown Due To RF Failure

Inside *RXTX_TimeOutState*, an endpoint terminates the Communication Channel after it has failed to receive <*RXTX_NoPreambleLevel*> consecutive packets from the other endpoint. These failures could be caused by a loss of power on the remote endpoint or the act of an endpoint moving out of transmission range.

4.13. Audio Compression

The bit rate of the RF data stream is 153.6 kbps, so 19.2 kB can be transmitted in a second. Since each endpoint will be transmitting 50% of the time, the effective byte rate per transceiver reduces to 9.6 kB per second. 10-bit ADC samples are taken at a frequency of 8 kHz, which equals a bit rate of about 80 kbps, or 10 kB per second.

Since the data stream can only support up to a 9.6 kB rate, the data bytes must be compressed before they can be sent across the link. In this design, differential pulse code modulation (DPCM) is utilized to compress the audio data. Using this scheme, each 10-bit sample is reduced to a 4-bit value.

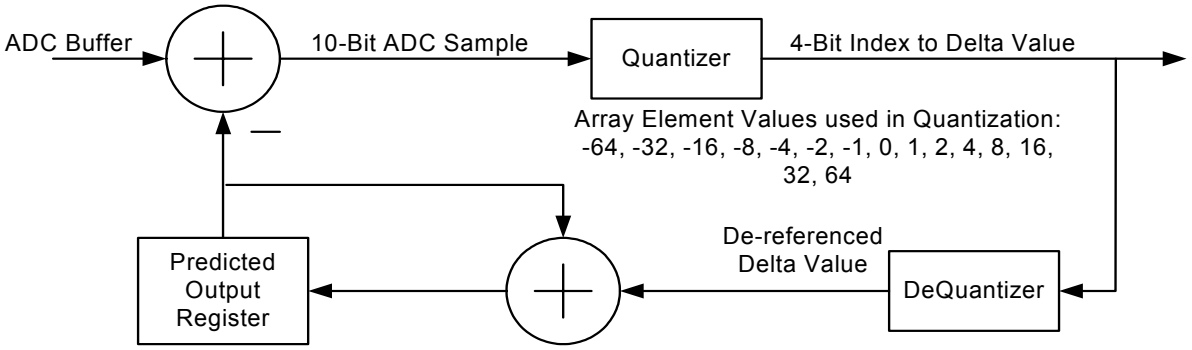


Figure 6. DPCM Encoder Signal Flow Diagram

4.13.1. DPCM Encoding

A signal flow diagram for the DPCM encoder can found in Figure 6. DPCM works by measuring the difference in amplitude between successive samples, and transmitting only an encoded delta value across the link. The encoded difference value is represented using an array of delta values that increase exponentially in size. This allows the algorithm to trace small changes in audio closely, but also respond to sudden, larger changes effectively as well.

The feedback path of the encoder reconstructs the ADC value using the delta array. First, the index is de-referenced to obtain the predicted delta value. Then, this positive or negative value is added to the predicted output to generate the updated output value.

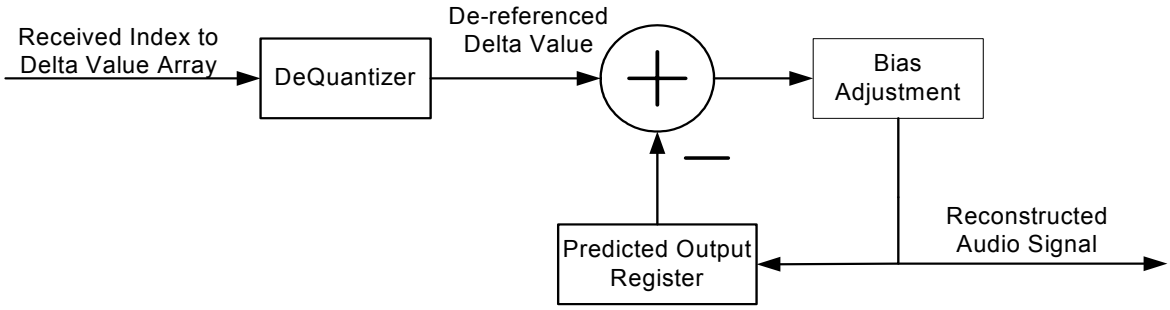


Figure 7. DPCM Decoder Signal Flow Diagram

4.13.2. DPCM Decoding

A signal flow diagram for the DPCM decoder is shown in Figure 7. The decoder functions identically to the feedback path of the encoder. A de-quantizer retrieves the delta value from the array using the received index, and the predicted value is updated by summing it with the signed delta value. The predicted output is then shifted so that it tends toward a mid-rail value. This bias adjustment is necessary to compensate for errors in the bit stream that cause indexes into the delta value array to be received incorrectly.

5. Usage Notes

The hardware system presented in this reference design needs only a microphone, a set of headphones, a serial adapter, and a power source to run properly. The right angle power jack is compatible with the ac adapters that ship with all Silicon Laboratories target boards. Power the board and connect the serial adapter to the board's 10-pin connector. The serial adapter draws its power from the reference design's board.

Connect to the IDE and download the AN147 software described in this documentation and included with this reference design. Plug in the microphone and headphones to the 3.5 mm jacks with the jacks labeled "Mic" and "Headphone," respectively. Repeat these steps on a second board, and the two programmed boards are ready to communicate with each other.

The green LED labeled P16 (for bit 6 of Port 1) is turned on when the transceiver is configured to receive, and turns off when the transceiver is set to transmit. The LED labeled P15 turns on whenever the local audio level is above the configured audible threshold. Pressing the button labeled SW2 initiates the establishment of a Communication Channel and transmits a low frequency sine wave. The button labeled SW1 resets the microcontroller, causing any communications to terminate.

6. Custom Configuration

The software system's constants and configuration values are declared with the "#define" preprocessor directives so that the system can be easily customizable. Buffer sizes, bytes sent per data packet, and tolerance values can be adjusted by changing the appropriate values listed in the "#define" section of code.

APPENDIX A—SCHEMATICS

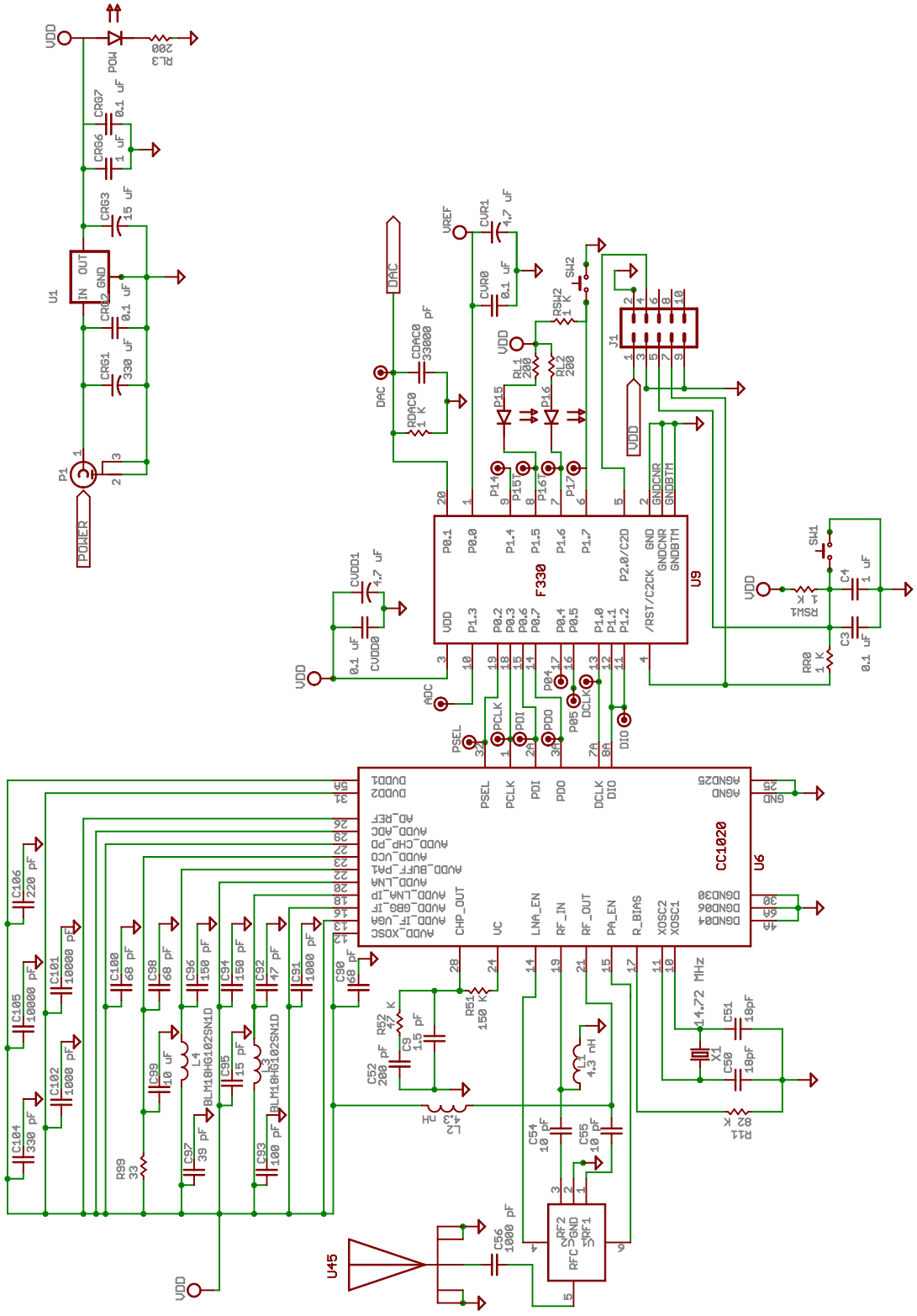


Figure 8. RF, MCU, and Power Supply

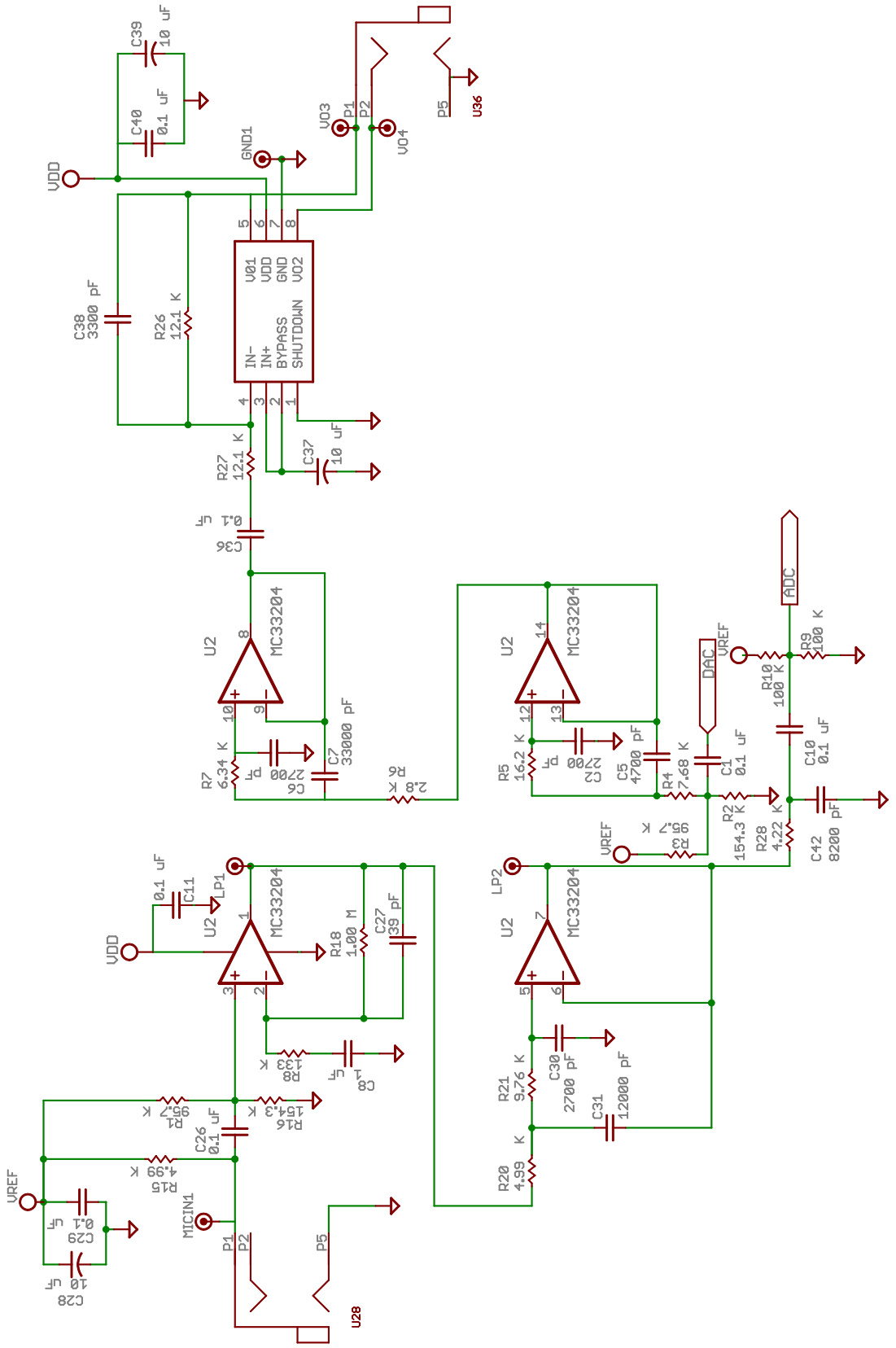


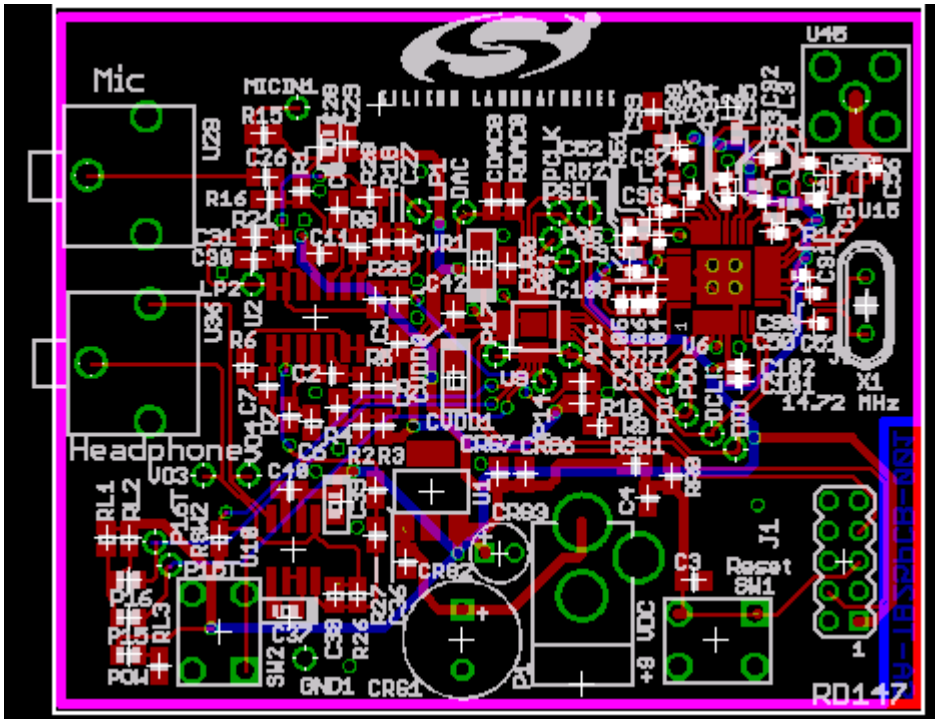
Figure 9. Audio Circuits

APPENDIX B—BILL OF MATERIALS

SA-TB25PCB-001 Bill Of Materials
Reference Design for AN147

Quantity	Value	Units / Description	Package Size/Part Number	Component
12	0.1	uF	C805	C1, C3, C10, C11, C26, C29, C36, C40, CRG2, CRG7, CVDD0, CVR0
4	1	K	R805	RDAC0, RR0, RSW1, RSW2
3	1	uF	C805	C4, C8, CRG6
1	1	Mohm	R805	R18
1	1.5	pF	C603	C9
1	2.8	K	R805	R6
2	4.3	nH	L603	L1, L2
2	4.7	uF (tant)	C3216	CVDD1, CVR1
1	4.22	K	R805	R28
2	4.99	K	R805	R15, R20
1	6.34	K	R805	R7
1	7.68	K	R805	R4
1	9.76	K	R805	R21
2	10	pF	C603	C54, C55
1	10	uF	C805	C99
3	10	uF	C805	C28, C37, C39
2	12.1	K	R805	R26, R27
1	15	pF	C603	C95
1	15	uF	Thru Hole	CRG3
1	16.2	K	R805	R5
2	18	pF	C603	C50, C51
1	33	Ohm	R603	R99
1	39	pF	C603	C97
1	39	pF	C805	C27
1	47	K	R603	R52
1	47	pF	C603	C92
3	68	pF	C603	C90, C98, C100
1	82	K	R603	R11
2	95.7	K	R805	R1, R3
2	100	K	R805	R9, R10
1	100	pF	C603	C93
1	133	K	R805	R8
1	150	K	R603	R51
2	150	pF	C603	C94, C96
2	154.3	K	R805	R2, R16
3	200	Ohm	R805	RL1,RL2, RL3
1	200	pF	C603	C52
1	220	pF	C603	C106
1	330	pF	C603	C104
1	330	uF	Thru Hole	CRG1
3	1000	pF	C603	C56, C91, C102
3	2700	pF	C805	C2, C6, C30
1	3300	pF	C805	C38
1	4700	pF	C805	C5
1	8200	pF	C805	C42
2	10000	pF	C603	C101, C105
1	12000	pF	C805	C31
2	33000	pF	C805	C7, CDAC0
2		GREEN LED	805	P15, P16
1		RED LED	805	POW
2	1000 OHM 100MA Ferrite Bead		L603 (Mn#:BLM18HG102SN1D)	L3, L4
1		RF Transceiver	Mn#:CC1020	U6
1	Silicon Laboratories Microcontroller		Mn#:C8051F330	U9
1		Quad Op Amp	Mn#:MC33204DR2	U2
2		3.5mm Audio Jack	Mn#:SJ-3543N	U28, U36
1		Right Angle Power Jack	Mn#:RAPC722	P1
1		SMA Jack	Mn#:CONREVSMA001	U45
1	M/A-COM GaAs RF Switch		Mn#:SW-438	U15
1		Speaker Driver	Mn#:TPA4861D	U10
1	3.3 Volt Voltage Regulator		Mn#:LM2937-3.3	U1
1	14.7456MHZ Crystal HC-49/US		Mn#:ECS-147.4-20-4	X1
1		916 MHz Antenna	Mn#:ANT-916-CW-RH	(screws into SMA connector)
2		Push Button		SW1, SW2
1		2x5 Pin JTAG Header		J1
1		Static-free Bag		
4		Rubber Feet		

APPENDIX C—PCB LAYOUT



APPENDIX D—FIRMWARE LISTING

Startup Code (Modified STARTUP.A51)

```
$NOMOD51
;------------------------------------------------------------------------------
;  This file is part of the C51 Compiler package
;  Copyright (c) 1988-2002 Keil Elektronik GmbH and Keil Software, Inc.
;------------------------------------------------------------------------------
;  STARTUP.A51:  This code is executed after processor reset.
;
;  To translate this file use A51 with the following invocation:
;
;     A51 STARTUP.A51
;
;  To link the modified STARTUP.OBJ file to your application use the following
;  BL51 invocation:
;
;     BL51 <your object file list>, STARTUP.OBJ <controls>
;
;------------------------------------------------------------------------------
;
;  User-defined Power-On Initialization of Memory
;
;  With the following EQU statements the initialization of memory
;  at processor reset can be defined:
;
;               ; the absolute start-address of IDATA memory is always 0
IDATALEN        EQU     80H     ; the length of IDATA memory in bytes.
;
XDATASTART      EQU     0H      ; the absolute start-address of XDATA memory
XDATALEN        EQU     0H      ; the length of XDATA memory in bytes.
;
PDATASTART      EQU     0H      ; the absolute start-address of PDATA memory
PDATALEN        EQU     0H      ; the length of PDATA memory in bytes.
;
;  Notes:  The IDATA space overlaps physically the DATA and BIT areas of the
;          8051 CPU. At minimum the memory space occupied from the C51
;          run-time routines must be set to zero.
;------------------------------------------------------------------------------
;
;  Reentrant Stack Initilization
```

```
;
;  The following EQU statements define the stack pointer for reentrant
;  functions and initialized it:
;
;  Stack Space for reentrant functions in the SMALL model.
IBPSTACK        EQU     0       ; set to 1 if small reentrant is used.
IBPSTACKTOP     EQU     0FFH+1  ; set top of stack to highest location+1.
;
;  Stack Space for reentrant functions in the LARGE model.
XBPSTACK        EQU     0       ; set to 1 if large reentrant is used.
XBPSTACKTOP     EQU     0FFFFH+1; set top of stack to highest location+1.
;
;  Stack Space for reentrant functions in the COMPACT model.
PBPSTACK        EQU     0       ; set to 1 if compact reentrant is used.
PBPSTACKTOP     EQU     0FFFFH+1; set top of stack to highest location+1.
;
;------------------------------------------------------------------------------
;
;  Page Definition for Using the Compact Model with 64 KByte xdata RAM
;
;  The following EQU statements define the xdata page used for pdata
;  variables. The EQU PPAGE must conform with the PPAGE control used
;  in the linker invocation.
;
PPAGEENABLE     EQU     0       ; set to 1 if pdata object are used.
;
PPAGE           EQU     0       ; define PPAGE number.
;
PPAGE_SFR       DATA    0A0H    ; SFR that supplies uppermost address byte
;               (most 8051 variants use P2 as uppermost address byte)
;
;------------------------------------------------------------------------------

; Standard SFR Symbols
ACC     DATA    0E0H
B       DATA    0F0H
SP      DATA    81H
DPL     DATA    82H
DPH     DATA    83H

                NAME    ?C_STARTUP
```

```
?C_C51STARTUP   SEGMENT   CODE
?STACK          SEGMENT   IDATA

                RSEG    ?STACK
                DS      1

                EXTRN CODE (?C_START)
                PUBLIC  ?C_STARTUP

                CSEG    AT      0




?C_STARTUP:     LJMP    STARTUP1

                RSEG    ?C_C51STARTUP

STARTUP1:

ANL 0D9H,#0BFH

IF IDATALEN <> 0
                MOV     R0,#IDATALEN - 1
                CLR     A
IDATALOOP:      MOV     @R0,A
                DJNZ    R0,IDATALOOP
ENDIF

IF XDATALEN <> 0
                MOV     DPTR,#XDATASTART
                MOV     R7,#LOW (XDATALEN)
  IF (LOW (XDATALEN)) <> 0
                MOV     R6,#(HIGH (XDATALEN)) +1
  ELSE
                MOV     R6,#HIGH (XDATALEN)
  ENDIF
                CLR     A
XDATALOOP:      MOVX    @DPTR,A
                INC     DPTR
```

```
                DJNZ    R7,XDATALOOP
                DJNZ    R6,XDATALOOP
ENDIF

IF PPAGEENABLE <> 0
                MOV     PPAGE_SFR,#PPAGE
ENDIF

IF PDATALEN <> 0
                MOV     R0,#LOW (PDATASTART)
                MOV     R7,#LOW (PDATALEN)
                CLR     A
PDATALOOP:      MOVX    @R0,A
                INC     R0
                DJNZ    R7,PDATALOOP
ENDIF

IF IBPSTACK <> 0
EXTRN DATA (?C_IBP)

                MOV     ?C_IBP,#LOW IBPSTACKTOP
ENDIF

IF XBPSTACK <> 0
EXTRN DATA (?C_XBP)

                MOV     ?C_XBP,#HIGH XBPSTACKTOP
                MOV     ?C_XBP+1,#LOW XBPSTACKTOP
ENDIF

IF PBPSTACK <> 0
EXTRN DATA (?C_PBP)
                MOV     ?C_PBP,#LOW PBPSTACKTOP
ENDIF

                MOV     SP,#?STACK-1
; This code is required if you use L51_BANK.A51 with Banking Mode 4
; EXTRN CODE (?B_SWITCH0)
;               CALL    ?B_SWITCH0      ; init bank mechanism to code bank 0
                LJMP    ?C_START

                END
```

AN147 Firmware

```
//-----------------------------------------------------------------------------
// Wireless Voice Transmitter/Receiver
//-----------------------------------------------------------------------------
// Copyright 2005 by Silicon Laboratories, Inc.
// AUTH: PD
// DATE: 1/23/05
//
// This software system is written to control the PCB designed with AN 147.
//

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------
#include <c8051f330.h>                  // SFR declarations
#include <intrins.h>                    // includes the _nop_() command

//-----------------------------------------------------------------------------
// Global CONSTANTS
//-----------------------------------------------------------------------------


#define SYSCLK 24500000                 // speed of internal clock
#define SAMPLE_RATE 8000                // sampling rate for the ADC
#define SPI_DATARATE 153600             // used for timing in SPI state machine

#define TransmitFIFO_FIFOSIZE   236     // sets the size of data buffers
#define ReceiveFIFO_FIFOSIZE   236
#define ADCRXFIFO_FIFOSIZE 10
#define DACTXFIFO_FIFOSIZE 10


#define DAC_UPDATERATE 8000              // rate of DAC sample output

// Transmission-Related Constants
#define RXTX_BytesOfData 200            // number of bytes of compressed audio
```

```
                                    // data to be transmitted per data
                                    // packet

#define TransmitFIFO_TARGET 200     // target size of TransmitFIFO, used to make
                                    // sampling rate adjustments to avoid
                                    // FIFO over- and underflows

#define ReceiveFIFO_TARGET 10       // target size of ReceiveFIFO, used to make
                                    // sampling rate adjustments to avoid
                                    // FIFO over- and underflows

#define RXTX_NoPreambleLevel 20     // consecutive number of data packets
                                    // that will be missed before the
                                    // RF signal is considered lost and
                                    // transmission shuts down

#define I_TX 0x02                   // this value is written to the RF
                                    // transceiver's Interface register when
                                    // the RF switch is to be placed in the
                                    // TX position
#define I_RX 0x01                   // this value is written to the RF
                                    // transceiver's Interface register when
                                    // the RF switch is to be placed in the
                                    // RX position

#define Audio_QuietThreshold 2      // difference threshold in ADC codes
                                    // between two consecutive samples that
                                    // is used to determine whether a
                                    // signal is "loud" enough to transmit
```

```
#define AudioStateQuietToLoud 10        // number of instances where
                                        // the difference between consecutive
                                        // ADC samples is GREATER than
                                        // <Audio_QuietThreshold> that must be
                                        // accumulated before a "quiet"
                                        // audio signal should be considered
                                        // "loud"

#define AudioStateLoudToQuiet 400       // number of instances where
                                        // the difference between consecutive
                                        // ADC samples is LESS than
                                        // <Audio_QuietThreshold> that must
                                        // CONSECUTIVELY occur before a "loud"
                                        // audio  signal should be considered
                                        // "quiet"

#define RX_MinBytesInitPreamble 5       // number of consecutively received
                                        // preamble bytes that must be received
                                        // before the incoming data for the
                                        // first received packet can be
                                        // considered valid

#define TX_NumBytesInitPreamble 50      // number of preamble bytes sent during
                                        // a RXTX_Master's first data packet
                                        // transmission

#define TX_NumBytesPreamble 8           // number of preamble bytes transmitted
                                        // before each data packet, excluding
                                        // the RXTX_Master's first packet

#define RX_MinBytesPreamble 2          // minimum number of preamble bytes
                                        // the must be received before incoming
                                        // data can be considered valid,
                                        // excluding the reception of the
                                        // RXTX_Master's first packet
```

```
#define RXTX_SyncWordSize 3            // used to re-sync the slave <SPI_Timer>

#define Audio_StateSize   1            // used to re-sync the slave <SPI_Timer>



#define DPCM_MULTIPLIER 2              // used to amplify the decompressed
                                       // audio samples


// 10-bit DPCM quantization codes
#define low_low 1
#define low_mid 2
#define low_high 4
#define middle 8
#define high_low 16
#define high_mid 32
#define high_high 64


//-----------------------------------------------------------------------------
// CC1020 Registers
//-----------------------------------------------------------------------------
#define  MAIN            0x00
#define  INTERFACE       0x01
#define  RESETT          0x02
#define  SEQUENCING      0x03
#define  FREQ_2A         0x04
#define  FREQ_1A         0x05
#define  FREQ_0A         0x06
#define  CLOCK_A         0x07
#define  FREQ_2B         0x08
#define  FREQ_1B         0x09
#define  FREQ_0B         0x0A
```

```
#define  CLOCK_B         0x0B
#define  VCO             0x0C
#define  MODEM           0x0D
#define  DEVIATION       0x0E
#define  AFC_CONTROL     0x0F
#define  FILTER          0x10
#define  VGA1            0x11
#define  VGA2            0x12
#define  VGA3            0x13
#define  VGA4            0x14
#define  LOCK            0x15
#define  FRONTEND        0x16
#define  ANALOG          0x17
#define  BUFF_SWING      0x18
#define  BUFF_CURRENT    0x19
#define  PLL_BW          0x1A
#define  CALIBRATE       0x1B
#define  PA_POWER        0x1C
#define  MATCH           0x1D
#define  PHASE_COMP      0x1E
#define  GAIN_COMP       0x1F
#define  POWERDOWN       0x20
#define  TEST1           0x21
#define  TEST2           0x22
#define  TEST3           0x23
#define  TEST4           0x24
#define  TEST5           0x25
#define  TEST6           0x26
#define  TEST7           0x27
#define  STATUS          0x40
#define  RESET_DONE      0x41
#define  RSS             0x42
#define  AFC             0x43
#define  GAUSS_FILTER    0x44
#define  STATUS1         0x45
#define  STATUS2         0x46
```

```
#define  STATUS3         0x47
#define  STATUS4         0x48
#define  STATUS5         0x49
#define  STATUS6         0x4A
#define  STATUS7         0x4B

#define  TEST_NFC        0x28


//-----------------------------------------------------------------------------
// 16-bit SFR Definitions for 'F33x
//-----------------------------------------------------------------------------

sfr16 DP       = 0x82;                // data pointer
sfr16 TMR3RL   = 0x92;                // Timer3 reload value
sfr16 TMR3     = 0x94;                // Timer3 counter
sfr16 IDA0     = 0x96;                // IDAC0 data
sfr16 ADC0     = 0xbd;                // ADC0 data
sfr16 ADC0GT   = 0xc3;                // ADC0 Greater-Than
sfr16 ADC0LT   = 0xc5;                // ADC0 Less-Than
sfr16 TMR2RL   = 0xca;                // Timer2 reload value
sfr16 TMR2     = 0xcc;                // Timer2 counter
sfr16 PCA0CP1  = 0xe9;                // PCA0 Module 1 Capture/Compare
sfr16 PCA0CP2  = 0xeb;                // PCA0 Module 2 Capture/Compare
sfr16 PCA0     = 0xf9;                // PCA0 counter
sfr16 PCA0CP0  = 0xfb;                // PCA0 Module 0 Capture/Compare



//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
void SYSCLK_Init (void);              // initialize system clock to 24.5 MHz
void PORT_Init (void);                // initialize crossbar
void ADC0_Init (void);                // ADC captures on Tmr2 overflows
                                      // interrupts enabled
```

```
void Timer0_Init(void);                 // used for  WaitUS()
void Timer2_Init (unsigned int);        // ADC start-of-conversion clock source
void Timer3_Init(unsigned int);         // sets DAC output rate
void IDAC0_Init(void);                  // enables IDAC output on P0.1
void SPI_Init(void);                    // enable 3-wire Slave SPI for RF trans.
                                        // DATA interface
void Variables_Init(void);              // set global variables to reset values

void PCA0_Init(void);                   // configure PCA0 to edge-triggered
                                        // capture mode


void Timer3_ISR (void);                 // updates the DAC
void UART0_ISR (void);                  // RX and TX with RF transceiver
void ADC0_ISR (void);                   // checks for a quiet signal,
                                        // pushes samples onto ADCRXFIFO
// FIFO Routines
unsigned char TransmitFIFO_Pull();      // pulls compressed sample to be TXed
unsigned char ReceiveFIFO_Pull();       // pulls RXed compressed sample
unsigned short ADCRXFIFO_Pull();        // pulls ADC sample
unsigned short DACTXFIFO_Pull();        // pulls decompressed sample
void TransmitFIFO_Push(unsigned char); // pushes compressed sample to be TXed
void ReceiveFIFO_Push(unsigned char);  // pushes RXed compressed sample
void ADCRXFIFO_Push(unsigned short);   // pushes ADC sample
void DACTXFIFO_Push(unsigned short);   // pushes decompressed sample
unsigned short ADCRXFIFO_Newest(void); // returns the value most recently
                                        // pushed onto the ADC FIFO, but does
                                        // not alter any of the index values


void CLEAR_FIFOS(void);                 // resets all FIFOs to default values

// DPCM Compression Algorithms
void DPCM_Decompress(void);             // Compresses an ADC sample
void DPCM_Compress(void);               // Decompresses received samples

// RF Transceiver Register Routines
void SETREG(unsigned char, unsigned char);
```

```
                                        // updates register to value parameter

unsigned char READREG(unsigned char);   // return register value
unsigned char CC1020_Init(void);        // initializes RF transceiver
void CC1020_SwitchToRX(void);           // sets RF transceiver settings to RX
void CC1020_SwitchToTX(void);           // sets RF transceiver settings to TX
void C1020_POWERDOWN(void);             // shuts down components of the RF
                                        // transceiver to conserve power

void ASLEEP(void);                      // function that runs when the
                                        // communications channel is idle
void WaitUS(unsigned int);              // function will delay controller
                                        // for a set number of microseconds
void WaitMS(unsigned int);

void FIFO_ManagementRoutine(void);      // encapsulates all foreground function
                                        // calls concerning buffers, and
                                        // all checks and tests on buffer
                                        // sizes

void RXTX_InitMaster(void);
void RXTX_InitSlave(void);


//-------------------------------------------------------------------------
// User-Defined Types
//-------------------------------------------------------------------------


typedef union USHORT {                  // access a short variable as two
      unsigned short S;                 // 8-bit integer values
      unsigned char C[2];
   } USHORT;

enum Logic{ FALSE, TRUE};
```

```
enum RXTX_Classes{
   RXTX_Master,
   // designates the endpoint that initiates the creation of the Communication
   // Channel

   RXTX_Slave,
   // designates the endpoint that joins an initiated Communication Channel

   RXTX_Searching
   // designates an endpoint that is searching for a master

};

typedef enum RXTX_StateMachineStates {

   RX_SearchForMaster,
   // Search for <RX_MinBytesInitPreamble> consecutive preamble bytes in
   // the RF transceiver's output data stream

   TX_InitPreamble,
   // Transmit <TX_NumBytesInitPreamble> preamble bytes before transmitting
   // the first data packet

   TX_Preamble,
   // Transmit <TX_NumBytesPreamble> preamble bytes

   TX_SyncWord,
   // Transmit sync word 0xFFFFEE

   TX_SwitchTime,
   // Delay for a time period specified in the SPI ISR's Transmission State
   // Machine, using the timer variable <SPI_Timer> as the time base

   TX_AudioState,
   // Transmit local shutdown state
```

```
TX_Data,
// Transmit <RXTX_BytesOfData> bytes from the TransmitFIFO

RX_Preamble,
// Search for <RX_MinBytesPreamble> consecutive bytes of received preamble

RX_SyncWord,
// Search for and synchronize receiver to the sync word 0xFFFFEE

RX_AudioState,
// Receive the remote (transmitter's) Shutdown State

RX_Data,
// Receive <RXTX_BytesOfData> bytes and push each to the ReceiveFIFO

RXTX_TimeOut,
// state is entered when <SPI_TIMER> times out.  Variable
// <TimeOut_EntryMode> should be set before entering state <RXTX_TimeOut>



Switch_ToTX01,
Switch_ToTX02,
Switch_ToTX03,
Switch_ToTX04,
Switch_ToTX05,
Switch_ToTX06,
Switch_ToTX07,
// These states configure the RF Transceiver to its transmit mode

Switch_ToRX01,
Switch_ToRX02,
Switch_ToRX03,
Switch_ToRX04,
Switch_ToRX05,
Switch_ToRX06,
// These states configure the RF Transceiver to its receive mode
```

```
   RXTX_WaitForTimeOut,
   // this state will pause state machine progress until the
   // RXTX_StateMachine's time base <SPI_Timer> reaches its time-out value

   RXTX_Shutdown,
   // Entered when the Communication Channel is to be terminated

   Uninitialized
   // Reset state of the RF state machine

} RXTX_StateMachineStates;

typedef enum TimeOut_EntryModes{
   TimeOut_RXSuccessful,
   // indicates that the state machine has just successfully completed the
   // reception of a data packet

   TimeOut_TXSuccessful,
   // indicates that the state machine has just successfully completed the
   // transmission of a data packet

   TimeOut_RXNoPreamble,
   // state machine failed to find the minimum number of bytes of preamble
   // needed to determine that a valid data packet has been received before
   // SPI_Timer reached its terminal value

   TimeOut_RXNoSyncWord
   // state machine failed to find a valid Sync Word before SPI_Timer reached
   // its terminal value

} TimeOut_EntryModes;



typedef enum Audio_States{
```

```
   Audio_Quiet = 0x10,
   // state indicates that audio is below the audible threshold

   Audio_Loud = 0x30,
   // audio is above audible threshold

   Audio_ChannelShutdown = 0x40
   // special state transmitted after both Endpoints' Audio States have
   // been set to

} Audio_States;


//-------------------------------------------------------------------------------
// Global Variables
//-------------------------------------------------------------------------------


unsigned char code RegValue[] = {
//      CLOCK_A, (this register is configured separately inside CC1020_Init() )
//      0x25,

//    CLOCK_B,
      0x25,

//    VCO,
      0x44,

//    MODEM,
      0x50,

//    DEVIATION,
      0x5A,

//    AFC_CONTROL,
      0xCC,
```

```
//    FILTER,
      0x80,

//    VGA1,
      0x65,

//    VGA2,
      0x57,

//    VGA3,
      0x34,

//    VGA4,
      0x3E,

//    LOCK,
      0x20,

//    FRONTEND,
      0x76,

//    ANALOG,
      0x86,

//    BUFF_SWING,
      0x50,

//    BUFF_CURRENT,
      0x25,

//    PLL_BW,
      0xAE,

//    CALIBRATE,
      0x35,
```

```
//    PA_POWER,
      0xFF,

//    MATCH,
      0x22,

//    PHASE_COMP,
      0x00,

//    GAIN_COMP,
      0x00,

//    POWERDOWN,
      0x00,

//    TEST1,
      0x4D,

//    TEST2,
      0x10,

//    TEST3,
      0x06,

//    TEST4,
      0x00,

//    TEST5,
      0x40,

//    TEST6,
      0x00,

//    TEST7,
      0x00
```

```
};


// 1/2 Period Sine Wave Table
static char code Audio_SineTable[16] = {
   0x00,0x18,0x30,0x47,0x5a,0x6a,0x76,0x7d,
   0x7f,0x7d,0x76,0x6a,0x5a,0x47,0x30,0x18
};



// the mapping from DPCM quantization values to dpcm codes (array index)
short code Q_VALUES[16] = {0,
                     -high_high,
                     -high_mid,
                     -high_low,
                     -middle,
                     -low_high,
                     -low_mid,
                     -low_low,
                     0,
                     low_low,
                     low_mid,
                     low_high,
                     middle,
                     high_low,
                     high_mid,
                     high_high};

// CC1020 Interface Bits
sbit PALE = P0^2;
sbit DIO = P1^2;
sbit DCLK = P1^0;
sbit PCLK = P0^3;
sbit PDI = P0^6;
```

```
sbit PDO = P0^7;

// AN147 PCB Pins
sbit DEBUG= P1^4;
sbit LED1 = P1^5;
sbit LED2 = P1^6;
sbit SW2  = P1^7;


// RX/TX State Machine variables, flags
RXTX_StateMachineStates RXTX_StateMachine;
void CLEAR_FIFOS(void);                // resets all FIFOs to default values

bit RXTX_ResetVariables;               // indicates to RF state machines that
                                       // variables should be re-initialized

unsigned char RXTX_MasterSelect;       // determines whether RF state machine
                                       // behaves as slave or master across
                                       // the RF link

bit RXTX_RunInitSlave;                 // signals RF state machine to run
                                       // as slave

bit SPI_TimeOutEvent;                  // set when SPI_Timer reaches its
                                       // terminal value

unsigned int SPI_Timer;                // time base for RF state machine

unsigned char SPI_DataBytes;           // counts number of data bytes
                                       // transmitted or received during
                                       // current packet time

int DAC_Error;                         // used to measure how far buffers are
                                       // from defined ideal value

unsigned char RXTX_NoPreambleCount;    // counts number of consecutive packet
```

```
                                        // receptions that have failed due to
                                        // finding no preamble bytes

bit SPI_TimerEnable;                    // signals that SPI_Timer should
                                        // be incremented

TimeOut_EntryModes TimeOut_EntryMode;   // indicates whether packet
                                        // reception/transmission was successful
                                        // or unsuccessful

Audio_States Audio_LocalState;          // shows whether audio signal is
Audio_States Audio_RemoteState;         // "quiet" or "loud"

bit RXTX_Indicator;                     // shows whether RF state machine is in
                                        // transmit or receive mode

bit OutputByteReady;                    // toggles to indicate whether both
                                        // nibbles of OutputByte inside
                                        // DPCM_Compress contain valid compressed
                                        // data

// TransmitFIFO Variables
unsigned char TransmitFIFO_COUNT;
bit TransmitFIFO_EMPTY;
bit TransmitFIFO_OF;
bit TransmitFIFO_UF;
bit TransmitFIFO_FULL;
bit CC1020_StartUpCall;
unsigned char idata TransmitFIFO_FIRST;
unsigned char idata TransmitFIFO_LAST;
unsigned char xdata TransmitFIFO_FIFO[TransmitFIFO_FIFOSIZE];


// ReceiveFIFO Variables
unsigned char ReceiveFIFO_COUNT;
bit ReceiveFIFO_EMPTY;
```

```
bit ReceiveFIFO_OF;
bit ReceiveFIFO_UF;
bit ReceiveFIFO_FULL;
unsigned char idata ReceiveFIFO_FIRST;
unsigned char idata ReceiveFIFO_LAST;
unsigned char xdata ReceiveFIFO_FIFO[ReceiveFIFO_FIFOSIZE];



// ADCRXFIFO Variables
unsigned char ADCRXFIFO_COUNT;
bit ADCRXFIFO_EMPTY;
bit ADCRXFIFO_OF;
bit ADCRXFIFO_UF;
bit ADCRXFIFO_FULL;
unsigned char idata ADCRXFIFO_FIRST;
unsigned char idata ADCRXFIFO_LAST;
int xdata ADCRXFIFO_FIFO[ADCRXFIFO_FIFOSIZE];


// DACTXFIFO Variables
unsigned char DACTXFIFO_COUNT;
bit DACTXFIFO_EMPTY;
bit DACTXFIFO_OF;
bit DACTXFIFO_UF;
bit DACTXFIFO_FULL;
bit DACTXFIFO_DECOMPRESS_HALT;
unsigned char idata DACTXFIFO_FIRST;
unsigned char idata DACTXFIFO_LAST;
unsigned int xdata DACTXFIFO_FIFO[DACTXFIFO_FIFOSIZE];



//-----------------------------------------------------------------------------
// Macros
//-----------------------------------------------------------------------------
```

```
// configures Cross Bar to route SPI to port pins
#define RouteSPI() P1SKIP &= ~0x07;  P1MDOUT |= 0x10;  XBR1 = 0x41;  XBR0 = 0x02

// configures Cross Bar to route PCA0 module 0 to port pins
#define RoutePCA() P1SKIP &= ~0x07;  P1MDOUT &=~0x10;  P1SKIP |=  0x03;  XBR0 = 0x00;
XBR1 = 0x41

// used in RF state machine to determine SPI_Timer timing thresholds
#define SPI_ms(x) (x * SPI_DATARATE / 8) / 1000
#define SPI_us(x) ((x * SPI_DATARATE / 8) / 1000) / 1000

#define SPI_SlopTimeOut SPI_ms(1) + SPI_us(500)

#define SPI_CalibrationWaitTime SPI_ms(1)

#define SPI_PacketTime SPI_ms(17)

#define SPI_TX()  P1MDIN |=  0x02; P1MDOUT |=  0x02
#define SPI_RX()  P1MDIN &= ~0x02; P1MDOUT &= ~0x02

//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------
void main (void) {

   PCA0MD &= ~0x40;                    // disable watchdog timer

   PORT_Init();                        // initialize and enable the Crossbar
   SYSCLK_Init();                      // initialize oscillator
   Timer2_Init(SYSCLK/SAMPLE_RATE);    // initialize timer to overflow
                                       // at SAMPLE_RATE

   ADC0_Init();                        // ADC samples on Timer 2 interrupts
   SPI_Init();                         // init
   Timer3_Init(SYSCLK/DAC_UPDATERATE); // initialize timer 3 to overflow at
```

```
                                    // DACUPDATERATE
PCA0_Init();                        // initialize PCA0 module 0 for
                                    // edge-triggered interrupts
IDAC0_Init();                       // enable DAC outputs at P0.1

Variables_Init();

WaitMS(500);


CC1020_StartUpCall = TRUE;          // forces CC1020_Init to turn on all
                                    // RF transceiver components

while(!CC1020_Init()){}             // initalize the RF transceiver

CC1020_StartUpCall = FALSE;

WaitMS(200);

RXTX_InitSlave();                   // configure RF state machine to slave
                                    // mode, where it will search for a
                                    // Communication Channel


AD0EN = 1;                          // enables ADC0
EA = 1;                             // enables all interrupts

while (1)
{

   FIFO_ManagementRoutine();        // compresses and decompresses
                                    // audio samples

   // if audile signal present is present at the audio input and the
   // system is not already communicating across the RF line,
   // then initiate a Communication Channel
```

```
      if((Audio_LocalState == Audio_Loud) &&
         (RXTX_MasterSelect == RXTX_Searching) &&
         (RXTX_RunInitSlave != TRUE))
      {
         EA = 0;                        // disable interrupts
         RXTX_InitMaster();             // configure to be master endpoint
         EA = 1;                        // re-enable interrupts
      }


      if(Audio_LocalState == Audio_Loud) LED1 = 0; else LED1 = 1;


      // only run the following conditional if SPIBSY == 0, that is
      // if the Audio_ShutdownChannel Audio_Level command has been transmitted
      // so that the receiving slave will no to end transmission
      if((RXTX_RunInitSlave == TRUE) && !(SPI0CFG & 0x80))
      {
         EA = 0;                        // disable interrupts
         RXTX_InitSlave();              // configure to be slave endpoint
                                        // and search for a transmitting master
         EA = 1;                        // re-enable interrupts
      }

  } // end while(1)

} // end void main()


//-----------------------------------------------------------------------------
// Initialization Functions
//-----------------------------------------------------------------------------
//

//-----------------------------------------------------------------------------
// SYSCLK_Init
```

```
//-----------------------------------------------------------------------------
//
// This routine initializes the system clock to use the internal 24.5MHz
// oscillator as its clock source.  Also enables missing clock detector
// reset and enables the VDD monitor as a reset source.
//
void SYSCLK_Init (void)
{
   OSCICN |= 0x03;                  // set clock to 24.5 MHz
   RSTSRC  = 0x06;                  // enable missing clock detector and VDD monitor
}

//-----------------------------------------------------------------------------
// ADC0_Init
//-----------------------------------------------------------------------------
//
// Configure ADC0 to use Timer2 overflows as conversion source, and to
// generate an interrupt on conversion complete.
// Enables ADC end of conversion interrupt. Leaves ADC disabled.
//
void ADC0_Init(void)
{
   REF0CN = 0x03;                       // set VREF pin as voltage reference,
                                        // enable internal bias generator and
                                        // internal reference buffer

   ADC0CN = 0x02;                       // overflow on Timer 2 starts
                                        // conversion
                                        // ADC0 disabled and in normal
                                        // tracking mode
   AMX0P  = 0x0B;                       // select P1.3 as positive conv. source
   AMX0N  = 0x11;                       // set ADC to single-ended mode
   ADC0CF = (SYSCLK/3000000) << 3;      // ADC Conversion clock = 3 MHz
   ADC0CF&= ~0x04;                      // ADC readings are right-justified
   EIE1  |= 0x08;                       // enable ADC0 EOC interrupt
}
```

```
//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// P0.0 - VREF
// P0.1 - IDAC0 Output
// P0.2 - PSEL
// P0.3 - PCLK
// P0.4 -
// P0.5 -
// P0.6 - PDI
// P0.7 - PDO
// P1.0 - SCK (DCLK)
// P1.1 - MISO (DIO)
// P1.2 - MOSI (DIO)
// P1.3 - Audio Input
// P1.4 - Test Point
// P1.5 - LED1
// P1.6 - LED2
// P1.7 - Switch

void PORT_Init (void)
{

   P0SKIP  = 0xFF;                     // skip all port 0 pins
   P1SKIP = 0x08;                      // skip ADC input P1.3
   P1MDIN &= ~0x08;                    // set P0.0 to analog input
   P0MDOUT |= 0x4C;                    // set P0.2, P0.3, P0.6 to push-pull
   P1MDOUT |= 0x68;                    // Audio input, debug pins
   RouteSPI();                         // routes SPI pins to port pins
   XBR1   |= 0x40;                     // Enable crossbar, CEX0 at port pin
}

//-----------------------------------------------------------------------------
// SPI_Init
```

```
//-----------------------------------------------------------------------------
//
// Set SPI to master, CKPHA = 0, CKPOL = 1.  Set SPI to 3 wire mode, and
// enable SPI.  SPI0CKR = 11, SCLK = 24.5Mhz / 12 = 1.021 MHz.
//
void SPI_Init(void)
{
   SPI0CFG = 0x00;                     // Master disable, CKPOL = 0
   SPI0CN  = 0;                        // clear all flags
   IE |= 0x40;                         // enable SPI interrupts
   SPIEN = 0;                          // leave SPI disabled
   IP |= 0x40;                         // make SPI ISR high priority
}


//-----------------------------------------------------------------------------
// IDAC0_Init
//-----------------------------------------------------------------------------
//
// Configure IDAC to update with every Timer 3 overflow, using 2.0 mA
// full-scale output current.
//

void IDAC0_Init(void)
{
   IDA0CN &= ~0x70;                    // Clear Update Source Select Bits
   IDA0CN |=  0x30;                    // Set DAC to update on Tmr 3 Overflows
   IDA0CN |=  0x80;                    // Enable DAC

}



//
//-----------------------------------------------------------------------------
// PCA0_Init
//-----------------------------------------------------------------------------
```

```
//
// The PCA is used to watch for the edges of the Sync Word at the SPI MOSI pin.
// Module 0 is configured for edge-triggered captures, and count sysclocks.
// PCA interrupts are enabled.  Leaves PCA disabled.
//
void PCA0_Init(void)
{
   PCA0CPM0 = 0x42;                    // set PCA0 to edge-triggered capture
                                       // mode
   PCA0MD |= 4<<1;                     // Set PCA0 to count sys clocks
   CR = 1;                             // enable PCA0 timer
}

//-----------------------------------------------------------------------------
// Timer2_Init
//-----------------------------------------------------------------------------
//
// Timer 2 is used as the start of conversion clock source for the ADC,
// and controls the audio sampling rate.
//
void Timer2_Init(unsigned int counts)
{
   TMR2CN = 0x00;                      // resets Timer 2, sets to 16 bit mode
   CKCON |= 0x10;                      // use system clock
   TMR2RL = -counts;                   // Initial reload value
   TMR2 = -counts;                     // init timer
   ET2 = 0;                            // disable Timer 2 interrupts
   TR2 = 1;                            // start Timer 2
}




//-----------------------------------------------------------------------------
// Timer3_Init
//-----------------------------------------------------------------------------
//
```

```
// Configure Timer3 to auto-reload at interval specified by <counts>
// using SYSCLK as its time base.  Interrupts are enabled.  Timer 3 controls
// the DAC output rate.
//
void Timer3_Init(unsigned int counts)
{
   TMR3CN  = 0x00;                     // resets Timer 3, sets to 16 bit mode
   CKCON  |= 0x40;                     // use system clock
   TMR3RL  = -counts;                  // Initial reload value

   TMR3    = -counts;                  // init timer
   EIE1   |= 0x80;                     // enable Timer 3 interrupts
   TMR3CN  = 0x04;                     // start Timer 3
}


//-----------------------------------------------------------------------------
// Variables_Init
//-----------------------------------------------------------------------------
//
void Variables_Init(void)
{
   Audio_LocalState = Audio_Quiet;
   Audio_RemoteState = Audio_Quiet;

   CLEAR_FIFOS();
}


//-----------------------------------------------------------------------------
// Interrupt Service Routines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// ADC0_ISR
//-----------------------------------------------------------------------------
```

```
// This routine captures and saves an audio input sample, and then
// compares it to past samples to find out whether or not the audio
// stream is quiet.
//

void ADC0_ISR(void) interrupt 10
{
   signed short SampleDifference;      // stores difference between current
                                       // and previous ADC0 values
   static short OldADCValue;           // previous ADC value
   static short NewADCValue;           // newest ADC value
   static unsigned short  AudioStateThreshold;
                                       // stores number of consecutive audio
                                       // bytes that have fallen outside
                                       // the defined threshold used to
                                       // determine if the audio state should
                                       // switch from "quiet" to "loud" or
                                       // "loud" to "quiet"
   static unsigned char TableIndex = 0;// index into sine table

   AD0INT = 0;                         // acknowledge end-of-conversion
                                       // interrupt

   OldADCValue = NewADCValue;          // save old ADC value
   NewADCValue = ADC0 - 512;           // add bias to new ADC value

   // if the button on AN147's PCB has been pressed, add a sine wave to the
   // the audio input signal
   if(!SW2)
   {
      TableIndex += 1;                 // increment sine table index value
      if(TableIndex >= 32) TableIndex = 0;
                                       // step through 16-element 1/2 period
                                       // sine table twice to produce a sine
                                       // wave's complete period
```

```
   // conditional adjusts the sine wave so that the first cycle
   // through the sine table produces the half of the period with values
   // higher than 0, and the second cycle produces values less than 0
   // Also, amplifies the sine wave by shifting the index value to the
   // left, which is equivalent to a gain of 2
   if(TableIndex < 16)
      NewADCValue += (unsigned char)(Audio_SineTable[TableIndex] << 1);
   else
      NewADCValue -= (unsigned char)(Audio_SineTable[TableIndex - 16]) << 1;
}


// This section of the ADC0 ISR determines whether the signal present at the
// audio input should be considered "loud" or "quiet"
SampleDifference = (signed)OldADCValue - (signed)NewADCValue;

// measure whether difference between the current sample and the
// last sample is greater than the threshold value <Audio_QuietThreshold>
if( (SampleDifference > (signed)Audio_QuietThreshold) ||
   (SampleDifference < (-(signed)Audio_QuietThreshold)) )
{
   if(Audio_LocalState == Audio_Quiet)
   {
      // add one to accumulator and check to see if
      // the critical accumulator value has been reached
      if(AudioStateThreshold++ == AudioStateQuietToLoud)
      {
         Audio_LocalState = Audio_Loud;
         AudioStateThreshold = 0;
      }
   }
   else if(Audio_LocalState == Audio_Loud)
   {
      // reset accumulator
      AudioStateThreshold = 0;
   }
```

```
   }

   // else <SampleDifference> is below the threshold value
   else
   {
      if(Audio_LocalState == Audio_Quiet)
      {
         if(AudioStateThreshold > 0)
         {
            AudioStateThreshold--;
         }
      }
      else if(Audio_LocalState == Audio_Loud)
      {
         // add one to accumulator and check to see if
         // the critical accumulator value has been reached
         if(AudioStateThreshold++ == AudioStateLoudToQuiet)
         {
            Audio_LocalState = Audio_Quiet;
            AudioStateThreshold = 0;
         }
      }
   }

   ADCRXFIFO_Push(NewADCValue);        // save ADC value for compression


}



//-----------------------------------------------------------------------------
// Timer3_ISR
//-----------------------------------------------------------------------------
// This ISR updates the DAC output at a rate of DACUPDATERATE.  It also
```

```
// fetches the most recently captured local ADC sample, attenuates the sample,
// and adds the signal to the DAC output to provide local side-tone.
//

void TIMER3_ISR(void) interrupt 14
{
      static unsigned short new_value;
      USHORT tempvalue;
      TMR3CN &= ~0xC0;                 // acknowledge interrupt

      if (!DACTXFIFO_EMPTY)            // if new DAC data is available,
      {                                // update output information
         new_value = DACTXFIFO_Pull();

         // only play received audio if the remote endpoint has determined
         // that the audio is of audible amplitude
         if(Audio_RemoteState == Audio_Loud)
         {

            // DAC output must be left-justified, and loaded
            // low byte first
            tempvalue.S = new_value;
//            tempvalue.S = tempvalue.S + ADCRXFIFO_Newest()>>4;
            tempvalue.S = tempvalue.S << 6;

            IDA0L = tempvalue.C[1];
            IDA0H = tempvalue.C[0];

         }
      }

}

//-----------------------------------------------------------------------------
// PCA0_ISR
//-----------------------------------------------------------------------------
```

```
//
//  The PCA ISR is used to synchronize with the data stream from
//  the CC1020 along byte boundaries using the Sync Word 0xFFFFEE.
//  The PCA is set to interrupt on the falling edge of the DIO line.
//  When the interrupt gets requested, the PCA ISR will spin while it
//  looks for another rising edge.  Upon finding the second rising edge,
//  the PCA will disable itself, route SPI back to the port pins, and
//  enable the SPI interface.
//
void PCA0_ISR(void) interrupt 11
{



   if(CCF0)
   {
      // acknowledge interrupt
      CCF0 = 0;

      // wait until rising edge after first received '0' in Sync Word stream
      while(!DIO);
      _nop_();

      // wait until falling edge of second '0' in stream
      while(DIO);
      _nop_();

      // wait for rising edge in DCLK
      while(!DCLK);
      _nop_();

      // wait for falling edge
      while(DCLK);

      // route SPI back to port pins
      RouteSPI();
```

```
      EIE1 &= ~0x10;                    // Disable PCA0 interrupts
      PCA0CPM0 = 0x42;                  // set to 8-bit PWM output

      SPIF = 0;                         // clear pending interrupt flag
      SPIEN = 1;                        // re-enable SPI

   }
   else if(CCF1)
   {
    CCF1 = 0
   }
   else if(CCF2)
   {
    CCF2 = 0
   }

}


//-----------------------------------------------------------------------------
// SPI_ISR
//-----------------------------------------------------------------------------
//
//  SPI ISR contains the RF state machine.  An interrupt will be requested
//  after 8 bits of data have been shifted out to or in from the RF
//  transceiver.  Progress through the state machine is gated by byte counters
//  that increment once per interrupt service.
//
void SPI0_ISR(void) interrupt 6
{
   static unsigned char PreambleByte;  // counts Preamble bytes transmitted
   static unsigned char SyncWordByte;  // counts Sync Word bytes transmitted
   unsigned char SPI_Input;            // saves the received byte acquired
                                       // from SPI0DAT

   SPI_Input = SPI0DAT;
```

```
// test for all SPI error conditions
if(SPI0CN & 0x70)
{
   SPI0CN &= ~0x70;                 // acknowledge error conditions, clear
                                    // flags
}

// bit is set inside RF state machine initialization routines
if(RXTX_ResetVariables)
{
   RXTX_ResetVariables = FALSE;     // clear flag
   RXTX_NoPreambleCount = 0;        // reset all counters
   PreambleByte = 0;
   SyncWordByte = 0;
}

if(SPIF)
{
   SPIF = 0;                        // acknowledge interrupt

   switch(RXTX_StateMachine)
   {

      case(RX_SearchForMaster):

         // if received byte is Preamble, take appropriate action
         if((SPI0DAT == 0x55) || (SPI0DAT == 0xAA))
         {
            if(++PreambleByte == RX_MinBytesInitPreamble)
            {
               RXTX_StateMachine = RX_SyncWord;

               // timer will synchronize with master's at the end
               // of the first received data packet
               SPI_TimerEnable = TRUE;
```

```
            SPI_Timer = SPI_SlopTimeOut + TX_NumBytesPreamble;

            RXTX_MasterSelect = RXTX_Slave;

            // reset counter
            PreambleByte = 0;
         }
      }
      else // received byte was not Preamble
      {
         // reset counter
         PreambleByte = 0;
      }

   break;

   case(RX_Preamble):
   {

      if(!SPI_TimeOutEvent)
      {
         // of byte received is Preamble, increment counter
         if((SPI0DAT == 0x55) || (SPI0DAT == 0xAA))
         {
            // if minimum number of Preamble bytes has been received,
            // begin searching for the Sync word
            if(PreambleByte++ >= RX_MinBytesPreamble)
            {
               RXTX_StateMachine = RX_SyncWord;
               PreambleByte = 0;
               SPI_DataBytes = 0;
            }
         }
         else
         {
            // received byte was not Preamble, reset counter
```

```
            PreambleByte = 0;
         }
      }
      // if time allotted to finding preamble has expired, enter
      // Time Out state with failure information
      else  // (SPI_TimeOutEvent)
      {
         TimeOut_EntryMode = TimeOut_RXNoPreamble;
         RXTX_StateMachine = RXTX_TimeOut;
         PreambleByte = 0;
      }
   }

   break;

   case(RX_SyncWord):
   {
      // only acknowledge time-outs during data reception

      //Sync Word not found before SPI TimeOut Event
      if(SPI_TimeOutEvent == TRUE)
      {
         TimeOut_EntryMode = TimeOut_RXNoSyncWord;
         RXTX_StateMachine = RXTX_TimeOut;
      }

      else if(SPI_Input == 0xFF)
      {
         // configure external interrupt to trigger on falling edge
         // of MOSI data line

         SPIEN = 0;              // disable SPI, re-enabled in PCA ISR

         PCA0CPM0 = 3<<4;        // Set PCA to edge-capture on rising
                                 // and falling edges
         PCA0CPM0 |= 1;          // enable CCF0 interrupts
```

```
            // route PCA CEX0 instead of SPI to RF transceiver data line
            RoutePCA();

            // enable PCA to edge-capture mode on P1.1

            EIE1 |= 0x10;           // enable PCA0 interrupts
            CCF0 = 0;               // clear pending interrupt flag

            RXTX_StateMachine = RX_AudioState;

         }


      }

      break;

      case(RX_AudioState):
      {
         Audio_RemoteState = SPI0DAT;

         // if local endpoint is designated as slave, and the shutdown
         // command has been received, configure the RF state machine
         // to shutdown
         if((RXTX_MasterSelect == RXTX_Slave) &&
            (Audio_RemoteState == Audio_ChannelShutdown))
         {
            RXTX_StateMachine = RXTX_Shutdown;
         }
         // else receive a packet of data
         else
         {
            RXTX_StateMachine = RX_Data;
         }
      }
```

```
break;


case(RX_Data):
{

   ReceiveFIFO_Push(SPI0DAT);     // store received byte

   SPI_DataBytes++;           // increment received bytes counter

   // once all packet's bytes have been received, exit state
   if(SPI_DataBytes == RXTX_BytesOfData)
   {
      // configure state machine to enter time out state
      // after SPI_Timer reaches terminal value
      TimeOut_EntryMode = TimeOut_RXSuccessful;
      RXTX_StateMachine = RXTX_WaitForTimeOut;

      // reset the SPI timer to the value it should be during
      // this point in packet reception
      if(RXTX_MasterSelect == RXTX_Slave)
      {
         // The slave should re-sync the SPI clock to the
         // reception of the last transmitted master CRC
         SPI_Timer = SPI_SlopTimeOut + TX_NumBytesPreamble
                     + RXTX_SyncWordSize + Audio_StateSize
                     + RXTX_BytesOfData;
      }


      // This code will adjust the output rate of the DAC
      // in order to avoid data overflows and underflows
      DAC_Error = (signed int)ReceiveFIFO_COUNT -
                  (signed int)ReceiveFIFO_TARGET;
      DAC_Error *= 5;
```

```
      TMR3RL = -((SYSCLK/DAC_UPDATERATE) - DAC_Error);

   }

}
break;

case(TX_SwitchTime):
{
   // spin until defined time period has passed
   if(SPI_Timer <= SPI_SlopTimeOut)
   {
      SPI0DAT = 0xFF;
   }
   else
   {
      SPI0DAT = 0xFF;
      RXTX_StateMachine = TX_Preamble;
   }
}

break;

case (TX_InitPreamble):

   if(PreambleByte++ != TX_NumBytesInitPreamble - 1)
   {
      // only write to SPI0DAt if the transmit buffer is empty
      if(TXBMT)
      {
         SPI0DAT = 0x55;
      }
      _nop_();

      if(TXBMT)
      {
```

```
            // conditional should execute only once
            SPI0DAT = 0x55;
         }
      }

      else // transmit last byte of Preamble
      {
         SPI0DAT = 0x55;

         SPI_TimerEnable = TRUE;

         // set timer to position it would be in at this
         // point during a normal data transmission
         SPI_Timer = SPI_SlopTimeOut + TX_NumBytesPreamble;

         RXTX_StateMachine = TX_SyncWord;
         PreambleByte = 0;
         SPI_DataBytes = 0;
      }

   break;

   case(TX_Preamble):

      if(PreambleByte++ != TX_NumBytesPreamble - 1)
      {
         if(TXBMT)
         {
            SPI0DAT = 0x55;
         }
         _nop_();

         if(TXBMT)
         {
            // conditional should execute only once
            SPI0DAT = 0x55;
```

```
            }
        }

        else
        {
            SPI0DAT = 0x55;
            RXTX_StateMachine = TX_SyncWord;
            PreambleByte = 0;
            SPI_DataBytes = 0;
        }

    break;

    case(TX_SyncWord):
    {
        // State Transmits 0xFFFFEE
         if(SyncWordByte < 2)
        {
            SPI0DAT = 0xFF;
            ++SyncWordByte;
        }
        else
        {
            SPI0DAT = 0xEE;
            RXTX_StateMachine = TX_AudioState;
            SyncWordByte = 0;
        }
     break;
     }

    case(TX_AudioState):
    {

        // if both endpoints' audio signals are "quiet" and the
        // local endpoint is designated as the master, transmit
        // the Shutdown command to terminate the Communication Channel
```

```
   if((RXTX_MasterSelect == RXTX_Master) &&
      (Audio_RemoteState == Audio_Quiet)
   && (Audio_LocalState == Audio_Quiet))
   {
      SPI0DAT = Audio_ChannelShutdown;
      RXTX_StateMachine = RXTX_Shutdown;
   }
   else
   {
      SPI0DAT = Audio_LocalState;
      RXTX_StateMachine = TX_Data;
   }
}
break;

case(TX_Data):
{

  SPI0DAT = TransmitFIFO_Pull();   // pull compressed byte and transmit


  SPI_DataBytes++;            // increment byte counter

  // once defined number of bytes has been transmitted, exit state
  if(SPI_DataBytes == RXTX_BytesOfData)
  {
      SPI_DataBytes = 0;
      SPI_DataBytes = 0;
      TimeOut_EntryMode = TimeOut_TXSuccessful;
      RXTX_StateMachine = RXTX_WaitForTimeOut;

      // This code adjusts the sampling rate of the ADC
      // in order to avoid data overflows or underflows
      DAC_Error = (signed int)TransmitFIFO_COUNT -
                  (signed int)TransmitFIFO_TARGET;
      DAC_Error *= 5;
```

```
      // adjust ADC sample rate
      TMR2RL = -((SYSCLK/DAC_UPDATERATE) + DAC_Error);
    }

}

break;

case(RXTX_WaitForTimeOut):
{
   // state machine will exit state after SPI_Timer has reached its
   // terminal value
   if(TXBMT)
   {
      SPI0DAT = 0x00;
   }
   if(SPI_TimeOutEvent == TRUE)
   {
      RXTX_StateMachine = RXTX_TimeOut;
   }
}

break;


case(RXTX_TimeOut):


   SPI_TimeOutEvent = FALSE;  // clear flag

   // reset RX failure counter if receive was successful
   if(TimeOut_EntryMode == TimeOut_RXSuccessful)
   {
      RXTX_NoPreambleCount = 0;
   }
```

```
switch(TimeOut_EntryMode)
{
   case(TimeOut_RXSuccessful):
      // toggle LED on to indicate that RF is
      // entering transmit mode
      LED2 = 0;

      RXTX_StateMachine = Switch_ToTX01;

   break;

   case(TimeOut_TXSuccessful):
      // toggle LED off to indicate that RF is
      // entering receive mode
      LED2 = 1;

      RXTX_StateMachine = Switch_ToRX01;

   break;

   case(TimeOut_RXNoPreamble):

      LED2 = 0;

      // count consecutive failed preamble detections,
      // make decision based on count

      if(++RXTX_NoPreambleCount == RXTX_NoPreambleLevel)
      {
         RXTX_NoPreambleCount = 0;
         RXTX_StateMachine = RXTX_Shutdown;
      }
      // else if endpoint is master, transmit a packet
      else if(RXTX_MasterSelect == RXTX_Master)
      {
         RXTX_StateMachine = Switch_ToTX01;
```

```
            }
            // if endpoint is slave, continue trying to receive a packet
            else
            {
               RXTX_StateMachine = RX_Preamble;
            }

         break;

         case(TimeOut_RXNoSyncWord):
            // if master, transmit a packet
            if(RXTX_MasterSelect == RXTX_Master)
            {
               RXTX_StateMachine = Switch_ToTX01;
            }
            // if slave, try to receive a packet again
            else
            {
               RXTX_StateMachine = RX_Preamble;
            }

         break;
      } // switch(TimeOut_EntryMode)

   break;


   case(RXTX_Shutdown):

      // set system to search for a Communication Channel
      RXTX_MasterSelect = RXTX_Searching;
      RXTX_RunInitSlave = TRUE;

      SPI_TimerEnable = FALSE;

      RXTX_StateMachine = Uninitialized;
```

```
break;

case(Uninitialized):

break;


case(Switch_ToTX01):

   SPI0DAT = 0xFF;
   _nop_();
   if(TXBMT) SPI0DAT = 0xFF;

   // route RF transceiver's output through the RF switch
   SETREG(INTERFACE,I_TX);

   // configure transceiver to TX mode
   SETREG(MAIN,0xC1);
   RXTX_StateMachine = Switch_ToTX02;

 break;

 case(Switch_ToTX02):

   SPI0DAT = 0xFF;
   if(SPI_Timer >= SPI_CalibrationWaitTime)
   {
      if((READREG(STATUS) & 0x10) != 0)
      {
          SETREG(PA_POWER,0xFF);

          RXTX_StateMachine = TX_SwitchTime;

          SPI_TX();           // set MISO pin to push-pull
      }
```

```
      else
      {
         RXTX_StateMachine = Switch_ToTX02;
      }
}
break;

case(Switch_ToRX01):

   SPI0DAT = 0xFF;
   _nop_();

   if(TXBMT) SPI0DAT = 0xFF;

   // set RF transceiver to lowest output power setting
   SETREG(PA_POWER, 0x00);

   // configure RF transceiver to RX mode
   SETREG(MAIN,0x01);
   RXTX_StateMachine = Switch_ToRX02;

break;

case(Switch_ToRX02):

  SPI0DAT = 0xFF;
  if(SPI_Timer >= SPI_CalibrationWaitTime)
  {
     if((READREG(STATUS) & 0x10) != 0)
     {
        // route antenna to the input pin of the RF transceiver
        SETREG(INTERFACE,I_RX);

        RXTX_StateMachine = RX_Preamble;

        SPI_RX();            // disconnect MISO pin by configuring it
```

```
                                        // to analog input
               }
            }
            else
            {
               RXTX_StateMachine = Switch_ToRX02;
            }

            break;

      }// end switch(RXTX_StateMachine)

   }// end if(SPIF)



   // this section of code increments the SPI_Timer, if enabled
   if(SPI_TimerEnable == TRUE)
   {
      SPI_Timer++;

      if(SPI_Timer == SPI_PacketTime)
      {
        SPI_TimeOutEvent = TRUE;
        SPI_Timer = 0;
      }
   }
   else
   {
      SPI_Timer = 0;
      SPI_TimeOutEvent = FALSE;
   }
}

//-------------------------------------------------------------------------
// RF State Machine Functions
```

```
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// RXTX_InitMaster(void)
//-----------------------------------------------------------------------------
//
// This function switches the RF transceiver to transmit and sets variables
// to indicate that this endpoint is designated as master.
//
void RXTX_InitMaster(void)
{

   SPIEN = 0;                           // disable SPI
   SPIF = 0;                            // clear any pending interrupt flag

   CC1020_SwitchToTX();                 // configure system to transmit

   RXTX_ResetVariables = TRUE;          // signal RF state machine to
                                        // reset variables

   RXTX_StateMachine = TX_InitPreamble;// set state machine to transmit
                                        // initial preamble

   RXTX_MasterSelect = RXTX_Master;     // designate endpoint as master

   SPIEN = 1;                           // re-enable SPI

   // pad SPI0DAT with known values to be shifted out
   SPI0DAT = 0x00;

   _nop_();

   if(TXBMT) SPI0DAT = 0x00;


}
```

```
//-----------------------------------------------------------------------------
// RXTX_InitSlave(void)
//-----------------------------------------------------------------------------
//
// This function switches the RF transceiver to receive and sets variables
// to indicate that the endpoint is designated as slave.
//

void RXTX_InitSlave(void)
{

   SPIEN = 0;                           // disable SPI
   SPIF = 0;                            // clear any pending interrupt flags

   CC1020_SwitchToRX();                 // configure system to receive

   RXTX_RunInitSlave = FALSE;           // clear signal flag

   RXTX_ResetVariables = TRUE;          // signal state machine to reset
                                        // variables

   // set RF state machine to search for a transmitting master endpoint
   RXTX_StateMachine = RX_SearchForMaster;

   RXTX_MasterSelect = RXTX_Searching; // designate endpoint as neither master
                                        // or slave

   SPIEN = 1;                           // re-enable SPI

   SPI0DAT = 0x00;                      // pad SPI0DAT with known values

   _nop_();

   if(TXBMT) SPI0DAT = 0x00;
```

```
}


//-----------------------------------------------------------------------------
// CC1020 Functions
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// SETREG
//-----------------------------------------------------------------------------
// Writing registers in the CC1020 transceiver is accomplished using a bit-based
// interface.  Two bytes are sent, an address byte and a data byte.
// First PALE is driven low.  The address byte contains the 7-bit address,
// and then a logic '1', signifying "write mode".  Then PALE is brought
// high and the data byte is transmitted.
//
void SETREG(unsigned char address, unsigned char value)
{
   unsigned char output,count;


   output = address << 1 | 0x01;       // 7 address bits + WRITE

   PCLK = 0;

   PALE = 0;                           // selects the RF transceiver


   // bit-banged interface sends the register address to the RF transceiver
   for (count = 0;count<8;count++)
   {
      PCLK = 0;                        // data is clocked into the RF
                                       // transceiver on the rising edge
                                       // of the clock

      if(output & 0x80) PDI = 1;       // set or clear the config data line
```

```
      else PDI = 0;                    // to transmit data MSB first

      PCLK = 1;                        // clock in the data bit
      output = output<<1;              // shift to read the next data bit
   }

   output = value;                     // prepare to shift out new register
                                       // value

   for (count = 0;count<8;count++)
   {
      PCLK = 0;

      if(output & 0x80) PDI = 1;       // set or clear config data line
      else PDI = 0;                    // with data out MSB first

      PCLK = 1;                        // clock out data

      output = output<<1;              // shift to prepare next bit's output
   }

   PCLK = 0;


   PALE = 1;                           // deselect the RF transceiver

}


//-----------------------------------------------------------------------------
// READREG
//-----------------------------------------------------------------------------
// Reading registers is accomplished by sending the address of the
// register to be read and then receiving a data byte in return.
// PALE is driven low, and a byte consisting of the 7-bit address
// plus a bit set to '0' signifying "read mode" is sent.  PALE is
```

```
// then raised high and a data byte is read in serially.
//
unsigned char READREG(unsigned char address)
{

   unsigned char output,input,count;

   output = address<<1;                // 7 address bits + READ

   PCLK = 0;

   PALE = 0;                           // select the RF transceiver

   for(count = 0;count<8;count++)
   {
      PCLK = 0;

      if(output & 0x80) PDI = 1;       // shift out register address
      else PDI = 0;                    // MSB first


      PCLK = 1;

      output = output<<1;              // prepare next bit to shift out
   }

   for(count = 0;count<8;count++)
   {
      PCLK = 0;

      _nop_();
      _nop_();
      _nop_();
      _nop_();

      PCLK = 1;                        // clock in register value MSB first
```

```
      // save bit of register value
      if(PDO == 1)
      {
         input = (input<<1) | 1;
      }
      else
      {
         input<<=1;
      }
   }

   PCLK = 0;

   PALE = 1;                              // deselect the RF transceiver

   return input;

}

//-----------------------------------------------------------------------------
// CC1020_Init
//-----------------------------------------------------------------------------
// This function will initialize the transceiver and calibrate it to
// transmit and receive on a defined frequency.
//
unsigned char cal_complete;

unsigned int idata count;

unsigned char CC1020_Init(void)
{
   unsigned char temp_reg;
   unsigned char RegAddress;

   if(CC1020_StartUpCall)
```

```
{
   SETREG(MAIN,0x0E);
   SETREG(MAIN,0x0F);
   SETREG(INTERFACE,I_RX);
   SETREG(RESETT,0xFF);
   SETREG(SEQUENCING,0x8F);



}

   // Configure receive frequency of 908 MHz
   SETREG(FREQ_2A,0x3c);
   SETREG(FREQ_1A,0xCE);
   SETREG(FREQ_0A,0x8E);

   // Configure transmit frequency of 908 MHz
   SETREG(FREQ_2B,0x3c);
   SETREG(FREQ_1B,0xd3);
   SETREG(FREQ_0B,0xe3);

if(CC1020_StartUpCall)
{

   // CLOCK_A is not located in a space contiguous with the
   // other registers to be programmed
   SETREG(CLOCK_A,0x25);

   // this loop writes to each register of the CC1020 a
   // corresponding value found in RegValue
   for(RegAddress = CLOCK_B;RegAddress != TEST7 + 1;)
   {
      temp_reg = RegValue[RegAddress - CLOCK_B];
      SETREG(RegAddress,RegValue[RegAddress - CLOCK_B]);
      RegAddress++;
   }
```

```
   SETREG(MAIN,0x0B);
   WaitMS (5*200);                  // wait for oscillator to stabilize
   SETREG(MAIN,0x09);
   WaitMS (1*200);                  // wait for bias generator to stabilize
   SETREG(MAIN,0x01);


}


// Calibration Routine
count = 0;                          // reset counter
cal_complete = 0;                   // clear temporary storage variable
SETREG(MAIN,0x11);
SETREG(CALIBRATE,0x85);

// check for calibration complete
do
{
   count++;
   cal_complete = READREG(STATUS);
} while (!(cal_complete & 0x80) && (count != 0xFFFF));

// if the count is too high or too low, an error occurred during
// calibration, exit the initialization and re-run
if(!(cal_complete & 0x80))
{
   return 0;
}

// now check for lock
count = 0;                          // reset counter
do
{
   count++;                         // increment counter
   cal_complete = READREG (STATUS);
```

```
   WaitUS(1);
   // spin until a lock is detected
} while (!(cal_complete & 0x10) && (count != 0xFFFE));


// if count timed out, exit function
if (!(cal_complete & 0x10))
{
   return 0;
}



count = 0;                          // reset counter
SETREG(MAIN,0xD1);

SETREG(CALIBRATE,0x85);
do
{
   count++;          // increment counter
   cal_complete = READREG(STATUS);
} while (!(cal_complete & 0x80) && (count != 0xFFFF));

// if the count is too high or too low, an error occurred during
// calibration, exit the initialization and re-run
if((count < 10) || (!(cal_complete & 0x80)))
{
   return 0;
}


// now check for lock
count = 0;                          // reset counter
do
{
   count++;                         // increment counter
```

```
      cal_complete = READREG (STATUS);
      // spin until lock detected
   } while (!(cal_complete & 0x10) && (count != 0xFFFF));


   // if count timed out, exit initialization and re-run
   if (!(cal_complete & 0x10))
   {
      return 0;
   }

   WaitMS(3);

   // Configure RF transceiver to receive by default
   SETREG(MAIN,0x01);

   return 1;


}


//-------------------------------------------------------------------------------
// CC1020_SwitchToRX(void)
//-------------------------------------------------------------------------------
//
// Function configures the RF transceiver's power amplifier to its lowest
// output setting, routes the antenna to the RF input pin using the RF switch,
// and configures the transceiver to receive.
//
void CC1020_SwitchToRX(void)
{
   SETREG(PA_POWER,0x00);              // configure PA to lowest setting

   SETREG(INTERFACE,I_RX);             // configure RF switch to RX path
```

```
   SETREG(MAIN,0x01);                  // configure transceiver to receive

   WaitMS(1);

   // wait for the PLL to lock
   while(!(READREG(STATUS) & 0x10)){};

   SPI_RX();                           // configure MISO as analog input
                                       // to disconnect pin from DIO trace
                                       // and avoid contention

}

//-------------------------------------------------------------------------
// CC1020_SwitchToTX(void)
//-------------------------------------------------------------------------
//
// Function configures PA to lowest setting,
//
void CC1020_SwitchToTX(void)
{

   SETREG(PA_POWER,0x00);              // configure internal PA to lowest
                                       // setting

   SETREG(MAIN,0xC1);                  // set RF transceiver to transmit

   WaitMS(1);

   // wait for the PLL to lock
   while(!(READREG(STATUS) & 0x10)){};

   SETREG(PA_POWER,0xFF);              // set PA to maximum transmit setting

   SETREG(INTERFACE,I_TX);             // route antenna to TX pin using RF
                                       // switch
```

```
   SPI_TX();                           // set MISO to push-pull

}




//-----------------------------------------------------------------------------
// FIFO Routines
//-----------------------------------------------------------------------------
// All FIFO functions pass a pointer to the fifo.  Pull functions return
// either a short or a char, and push functions have the data to be pushed
// as an additional parameter.
// Pushes and pulls update EMPTY, COUNT, and OF variables.
//
unsigned char TransmitFIFO_Pull(void)
{
   unsigned char output;

   if(TransmitFIFO_EMPTY)
   {
      // if buffer is empty, set the underflow flag and exit function
      TransmitFIFO_UF = 1;
      return 0;
   }
   else
   {
      TransmitFIFO_FULL = 0;               // if byte is pulled, buffer can no
                                        // longer be full

      // wrap FIFO pointer if necessary
      if (TransmitFIFO_FIRST==TransmitFIFO_FIFOSIZE-1) TransmitFIFO_FIRST = 0;
      else (TransmitFIFO_FIRST)++;

      // pull value from fifo
```

```
      output = TransmitFIFO_FIFO[TransmitFIFO_FIRST];

      // set empty indicator if necessary
      if (--(TransmitFIFO_COUNT) == 0) TransmitFIFO_EMPTY = 1;

      return output;
   }
}

void TransmitFIFO_Push(unsigned char num)
{
   TransmitFIFO_EMPTY = 0;                  // buffer is no longer empty


   if (!TransmitFIFO_FULL) {
      (TransmitFIFO_LAST)++;                // increment, wrap if necessary
      if (TransmitFIFO_LAST == TransmitFIFO_FIFOSIZE)
      {
         TransmitFIFO_LAST = 0;
      }

      TransmitFIFO_FIFO[TransmitFIFO_LAST]=num;  // push value to FIFO


      TransmitFIFO_COUNT++;                 // increment count

      // check for full buffer
      if ( TransmitFIFO_COUNT == TransmitFIFO_FIFOSIZE)
      {
         TransmitFIFO_FULL = 1;
      }
   }
   else // buffer is full
   {
      // if buffer is full, do not push byte, just set overflow flag
      // and exit
```

```
      TransmitFIFO_OF = 1;
   }
}

unsigned char ReceiveFIFO_Pull(void)
{
   unsigned char output;

   if(ReceiveFIFO_EMPTY)
   {
      // if buffer is empty, set underflow flag and exit
      ReceiveFIFO_UF = 1;

      return 0;
   }
   else
   {
      ReceiveFIFO_FULL = 0;                // buffer is no longer full

      // wrap FIFO pointer if necessary
      if (ReceiveFIFO_FIRST==ReceiveFIFO_FIFOSIZE-1) ReceiveFIFO_FIRST = 0;
      else (ReceiveFIFO_FIRST)++;

      // pull value from fifo
      output = ReceiveFIFO_FIFO[ReceiveFIFO_FIRST];

      // set empty indicator if necessary
      if (--(ReceiveFIFO_COUNT) == 0) ReceiveFIFO_EMPTY = 1;

      return output;
   }
}


void ReceiveFIFO_Push(unsigned char num)
{
```

```
   ReceiveFIFO_EMPTY = 0;                  // FIFO is no longer empty
   if (!ReceiveFIFO_FULL) {

      (ReceiveFIFO_LAST)++;                // increment, wrap if necessary
      if (ReceiveFIFO_LAST == ReceiveFIFO_FIFOSIZE)
      {
         ReceiveFIFO_LAST = 0;
      }

      ReceiveFIFO_FIFO[ReceiveFIFO_LAST]=num;  // push value to FIFO

      ReceiveFIFO_COUNT++;

      if ( ReceiveFIFO_COUNT == ReceiveFIFO_FIFOSIZE)
      {
         ReceiveFIFO_FULL = 1;
      }
   // buffer is full
   } else {
      // only set overflow flag and exit
      ReceiveFIFO_OF = 1;
   }
}

void ADCRXFIFO_Push(unsigned short num)
{
   ADCRXFIFO_EMPTY = 0;                // buffer is no longer empty

   if (!ADCRXFIFO_FULL) {
      (ADCRXFIFO_LAST)++;              // increment, wrap around if necessary
      if (ADCRXFIFO_LAST == ADCRXFIFO_FIFOSIZE)
      {
         ADCRXFIFO_LAST = 0;
      }
```

```
      ADCRXFIFO_FIFO[ADCRXFIFO_LAST]=num;

      // update count, check for an overflow
      ++ADCRXFIFO_COUNT;

      if(ADCRXFIFO_COUNT == ADCRXFIFO_FIFOSIZE)
      {
         ADCRXFIFO_FULL = 1;
      }
   } else {
      ADCRXFIFO_OF = 1;
   }
}

unsigned short ADCRXFIFO_Pull(void)
{
   unsigned short output;

   // if a function tries to pull a value from an empty buffer, set the
   // underflow flag and exit the function
   if(ADCRXFIFO_EMPTY)
   {
      ADCRXFIFO_UF = 1;
      return 0;
   }

   // else the buffer is not empty, pull the value and return it
   else
   {
      ADCRXFIFO_FULL = 0;
      // update the first pointer, wrap around if necessary
      if (ADCRXFIFO_FIRST==ADCRXFIFO_FIFOSIZE-1) ADCRXFIFO_FIRST = 0;
      else (ADCRXFIFO_FIRST)++;

      // save output value
```

```
      output = ADCRXFIFO_FIFO[ADCRXFIFO_FIRST];

      // decrement buffer size
      if (--(ADCRXFIFO_COUNT) == 0) ADCRXFIFO_EMPTY = 1;

      return output;
   }
}


void DACTXFIFO_Push(unsigned short num)
{
   DACTXFIFO_EMPTY = 0;                // buffer is no longer empty

   if (!DACTXFIFO_FULL) {
      (DACTXFIFO_LAST)++;              // increment, wrap around if necessary
      if (DACTXFIFO_LAST == DACTXFIFO_FIFOSIZE)
      {
         DACTXFIFO_LAST = 0;
      }

      DACTXFIFO_FIFO[DACTXFIFO_LAST]=num;



      // update count, check for an overflow
      ++DACTXFIFO_COUNT;
      if (DACTXFIFO_COUNT == DACTXFIFO_FIFOSIZE - 1)
      {
         // the decompression routine will push two decompressed
         // audio samples per call to the routine, and
         // DACTXFIFO_DECOMPRESS_HALT is set when the buffer can only
         // store 1 more decompressed sample.
         // The flag will be cleared when at least one buffer byte is pulled
         DACTXFIFO_DECOMPRESS_HALT = 1;
      }
```

```
      else if (DACTXFIFO_COUNT == DACTXFIFO_FIFOSIZE)
      {
         DACTXFIFO_DECOMPRESS_HALT = 1;
         DACTXFIFO_FULL = 1;
      }
      else if ( DACTXFIFO_COUNT == DACTXFIFO_FIFOSIZE+1)
      {
         DACTXFIFO_DECOMPRESS_HALT = 1;
         DACTXFIFO_FULL = 1;
         DACTXFIFO_OF = 1;
      }
   } else {
      DACTXFIFO_OF = 1;
   }
}

unsigned short DACTXFIFO_Pull(void)
{
   unsigned short output;

   // if a function tries to pull a value from an empty buffer, set the
   // underflow flag and exit the function
   if(DACTXFIFO_EMPTY)
   {
      DACTXFIFO_UF = 1;
      return 0;
   }

   // else the buffer is not empty, pull the value and return it
   else
   {
      DACTXFIFO_FULL = 0;
      // update the first pointer, wrap around if necessary
      if (DACTXFIFO_FIRST==DACTXFIFO_FIFOSIZE-1) DACTXFIFO_FIRST = 0;
      else (DACTXFIFO_FIRST)++;
```

```
      // save output value
      output = DACTXFIFO_FIFO[DACTXFIFO_FIRST];

      // decrement buffer size
      if (--(DACTXFIFO_COUNT) == 0) DACTXFIFO_EMPTY = 1;

      if((DACTXFIFO_DECOMPRESS_HALT) &&
         (DACTXFIFO_COUNT <= DACTXFIFO_FIFOSIZE - 2))
      {
         DACTXFIFO_DECOMPRESS_HALT = 0;
      }

      return output;
   }
}

//-----------------------------------------------------------------------------
// ADCRXFIFO_Newest()
//-----------------------------------------------------------------------------
//
// This function returns the newest received ADC sample, which can be used
// to loop back the local audio signal during communication.
//
unsigned short ADCRXFIFO_Newest(void)
{
   return ADCRXFIFO_FIFO[ADCRXFIFO_LAST];
}

//-----------------------------------------------------------------------------
// CLEAR_FIFOS()
//-----------------------------------------------------------------------------
//
// Routine resets all buffers; index values and flags to initial values.
//
void CLEAR_FIFOS(void)
{
```

```
   // reset the ADCRX FIFO
   ADCRXFIFO_EMPTY = 1;
   ADCRXFIFO_FIRST = 0;
   ADCRXFIFO_LAST = 0;
   ADCRXFIFO_COUNT = 0;
   ADCRXFIFO_OF = 0;
   ADCRXFIFO_FULL = 0;

   // reset the DACTX FIFO
   DACTXFIFO_EMPTY = 1;
   DACTXFIFO_FIRST = 0;
   DACTXFIFO_LAST = 0;
   DACTXFIFO_COUNT = 0;
   DACTXFIFO_OF = 0;
   DACTXFIFO_FULL = 0;

   // reset the UART TX FIFO
   TransmitFIFO_EMPTY = 1;
   TransmitFIFO_FIRST = 0;
   TransmitFIFO_LAST = 0;
   TransmitFIFO_COUNT = 0;
   TransmitFIFO_OF = 0;
   TransmitFIFO_FULL = 0;

   // reset the UART RX FIFO
   ReceiveFIFO_EMPTY = 1;
   ReceiveFIFO_FIRST = 0;
   ReceiveFIFO_LAST = 0;
   ReceiveFIFO_COUNT = 0;
   ReceiveFIFO_OF = 0;
   ReceiveFIFO_FULL = 0;

}

//-----------------------------------------------------------------------------
// FIFO_ManagementRoutine()
```

```
//-----------------------------------------------------------------------------
//
//
//
void FIFO_ManagementRoutine(void)
{
   bit EAState;

   EAState = EA;
   EA = 0;



   PCA0CPH0 = ReceiveFIFO_COUNT;

   if (!ADCRXFIFO_EMPTY)               // The compression algorithm should run
      DPCM_Compress();                 // any time samples exist to compress
   EA = EAState;

   EAState = EA;
   EA = 0;
   if ((!ReceiveFIFO_EMPTY) && (!DACTXFIFO_DECOMPRESS_HALT))
   {
                                       // Decompress received samples when
      DPCM_Decompress();               // available
   }
   EA = EAState;

   EAState = EA;
   EA = 0;
   if(ADCRXFIFO_FULL)
   {
      ADCRXFIFO_Pull();
   }
   EA = EAState;
```

```
   EAState = EA;
   EA = 0;
    if(ReceiveFIFO_FULL)
   {

      ReceiveFIFO_Pull();
   }
   EA = EAState;

   EAState = EA;
   EA = 0;
   if(TransmitFIFO_FULL)
   {
      TransmitFIFO_Pull();
   }
   EA = EAState;

   EAState = EA;
   EA = 0;
   if(DACTXFIFO_OF)
   {
      DACTXFIFO_Pull();
   }
   EA = EAState;


}

//-----------------------------------------------------------------------------
// DPCM_Compress
//-----------------------------------------------------------------------------
//
// Encode the 8-bit (MSBs of 10-bit sample) sample using DPCM compression.
// INPUTS:
//    sample_diff -  8-bit difference between the predicted value and the 8
//                   MSBs of the sample from the ADC
```

```
// OUTPUTS:
//    dpcm_code - the 4-bit quantized DPCM code
//
//    The difference will be rounded down if positive and rounded up if
//    negative (i.e. 41 => 32, and -41 => -32).
//
   volatile short data encoderpredsample;
void DPCM_Compress (void)
{
   short isr_count1 = TMR2;
   short sample_diff;
   unsigned short sample_diff_us;
   unsigned char dpcm_code;
   static unsigned char OutputByte;

   bit EAstate;

   EAstate = EA;
   EA = 0;
   sample_diff = ADCRXFIFO_Pull();
   EA = EAstate;

   sample_diff -= encoderpredsample;

   // determine if the difference is positive or negative
   if (sample_diff < 0)
   {
      sample_diff_us = -sample_diff;   // use the absolute value
   }
   else
   {
      sample_diff_us = sample_diff;
   }

   // narrow down which bits need to be set to use the proper
   // quantization code using a binary search algorithm (divide in halves)
```

```
// the sign of the difference no longer matters, so only use the
// lower 8 bits
if (sample_diff_us >= middle)
{
   if (sample_diff_us >= high_mid)
   {
      if (sample_diff_us >= high_high)
      {
         dpcm_code = 15;
      }
      else
      {
         dpcm_code = 14;
      }
   }
   else
   {
      if (sample_diff_us >= high_low)
      {
         dpcm_code = 13;
      }
      else
      {
         dpcm_code = 12;
      }
   }
}
else
{
   if (sample_diff_us >= low_mid)
   {
      if (sample_diff_us >= low_high)
      {
         dpcm_code = 11;
      }
      else
```

```
            {
               dpcm_code = 10;
            }
         }
         else
         {
            if (sample_diff_us >= low_low)
            {
               dpcm_code = 9;
            }
            else
            {
               dpcm_code = 8;
            }
         }
      }


      if (sample_diff < 0)
      {
         dpcm_code = ~dpcm_code + 1;       // use the 2's compliment of
                                           // the dpcm code
         dpcm_code &= 0x0F;                // use only the 4 LSBs for the
                                           // dpcm code
      }

      isr_count1 = Q_VALUES[dpcm_code];
      isr_count1 = isr_count1;
      encoderpredsample = encoderpredsample + Q_VALUES[dpcm_code];

      if(!OutputByteReady)
      {
    OutputByteReady = TRUE;
    OutputByte = dpcm_code;
         OutputByte<<=4;
      }
```

```
   else
   {
      OutputByteReady = FALSE;
      OutputByte |= dpcm_code & 0x0F;
      EAstate = EA;
      EA = 0;
      TransmitFIFO_Push(OutputByte);
      EA = EAstate;
   }

}

//-----------------------------------------------------------------------------
// DPCM_Decompress
//-----------------------------------------------------------------------------
//
// decode the 8-bit (MSBs of 10-bit sample) sample using DPCM compression.
// INPUTS:
//    old_prediction - the 8-bit unsigned predicted value from the current
//                     cycle which will be used to calculate the new predicted
//                     value
//    dpcm_code -   the 4-bit code indicating the quantized difference between
//                  the old_prediction and the current sample values
//                  (see DPCM_Encode for the dpcm_code values)
// OUTPUTS:
//    new_prediction - the 8-bit unsigned predicted value for the next cycle
//
void DPCM_Decompress (void)
{
unsigned char CompressedByte, CompressedSample;
static signed short UncompressedWord;
   bit EAstate;

   EAstate = EA;
   EA = 0;
CompressedByte = ReceiveFIFO_Pull();
```

```
EA = EAstate;

CompressedSample = CompressedByte>>4;

UncompressedWord = UncompressedWord + Q_VALUES[CompressedSample];

   if (UncompressedWord > 3)
   {
      UncompressedWord--;
   }
   else
   if (UncompressedWord < 3)
   {
      UncompressedWord++;
   }

   if (UncompressedWord > 511)
   {
      UncompressedWord = 511;
   }
   else
   if (UncompressedWord < -512)
   {
      UncompressedWord = -512;
   }

   EAstate = EA;
   EA = 0;
   DACTXFIFO_Push(UncompressedWord + 512);
   EA = EAstate;

CompressedSample = CompressedByte & 0x0F;

UncompressedWord = UncompressedWord + Q_VALUES[CompressedSample];

   if (UncompressedWord > 3)
```

```
   {
      UncompressedWord--;
   }
   else
   if (UncompressedWord < 3)
   {
      UncompressedWord++;
   }

   if (UncompressedWord > 511)
   {
      UncompressedWord = 511;
   }
   else
   if (UncompressedWord < -512)
   {
      UncompressedWord = -512;
   }

EAstate = EA;
EA = 0;
   DACTXFIFO_Push(UncompressedWord + 512);
   EA = EAstate;

}
//-------------------------------------------------------------------------
// Wait Functions
//-------------------------------------------------------------------------
// These functions cause the micro controller to spin for a time defined by the
// functions' input parameter.
//

void WaitUS (unsigned int count)
{
   TCON &= ~0x30;                      // Stop Timer0; clear TF0
   TMOD &= ~0x0F;                      // Timer0 in 8-bit autoreload mode
```

```
   TMOD |=  0x03;
   CKCON |= 0x04;                       // Timer0 counts SYSCLKs

   TH0 = (-SYSCLK / 10000000);          // overflow in 1us
   TL0 = TH0;

   TR0 = 1;                             // Start Timer0
   do {
      while (!TF0);                     // wait for overflow
      TF0 = 0;                          // clear overflow flag
      count--;                          // update counter
   } while (count != 0);                // continue for <count> us

   TR0 = 0;                             // Stop Timer0
}

void WaitMS (unsigned int count)
{

   TCON &= ~0x30;                       // Stop Timer0; clear TF0
   TMOD &= ~0x0F;                       // Timer0 in 16-bit mode
   TMOD |=  0x01;
   CKCON |= 0x04;                       // Timer0 counts SYSCLKs

   TH0 = (-SYSCLK / 1000) >> 8;         // overflow in 1ms
   TL0 = (-SYSCLK / 1000);

   TR0 = 1;                             // Start Timer0
   do {

      while (!TF0);                     // wait for overflow
      TR0 = 0;                          // Stop Timer0
      TF0 = 0;                          // clear overflow flag

      TH0 = (-SYSCLK / 1000) >> 8;      // overflow in 1ms
      TL0 = (-SYSCLK / 1000);
```

```
      TR0 = 1;                          // Start Timer0

      count--;                          // update counter
   } while (count != 0);                // continue for <count> ms

   TR0 = 0;                             // Stop Timer0
}
```

DOCUMENT CHANGE LIST

Revision 1.1 to Revision 1.2

- Updated RF transceiver from CC1000 to CC1020.
- Changed compression algorithm from CVSD to DPCM.
- Updated RF state machines to improve robustness of link.

CONTACT INFORMATION

Silicon Laboratories Inc.
4635 Boston Lane
Austin, TX 78735
Tel: 1+(512) 416-8500
Fax: 1+(512) 416-9669
Toll Free: 1+(877) 444-3032

Email: MCUinfo@silabs.com
Internet: www.silabs.com

SILICON LABORATORIES