# Wireless Digital Full-Duplex Voice Transceiver

## Relevant Devices

This note applies to the following devices:C8051F330

## 1. Introduction

With its small size (4x4 mm) and high levels of integration (on-chip ADC, DAC, and 25 MIPS peak CPU), the C8051F330 lends itself to wireless voice applications. This reference design demonstrates how the F330 can enable wireless, digital, full-duplex voice transmission and reception.

This document includes:

- 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

## 2. Key Points

- The 'F330's on-chip ADC, DAC, SPI, UART, calibrated internal oscillator, and small 4x4 package size allow designers to decrease PCB size in wireless voice applications
- Because of RF bandwidth limitations, the voice signal sampled by the ADC is compressed before transmission using the high-speed 25 MIPS peak core
- To enable full-duplex communication in the half duplex RF channel, synchronized RX/TX switching between endpoints is implemented
- To minimize power consumption, the system only transmits and receives data when the audio signals are of audible volume levels
- Careful component placement during board layout is required to ensure optimal RF performance

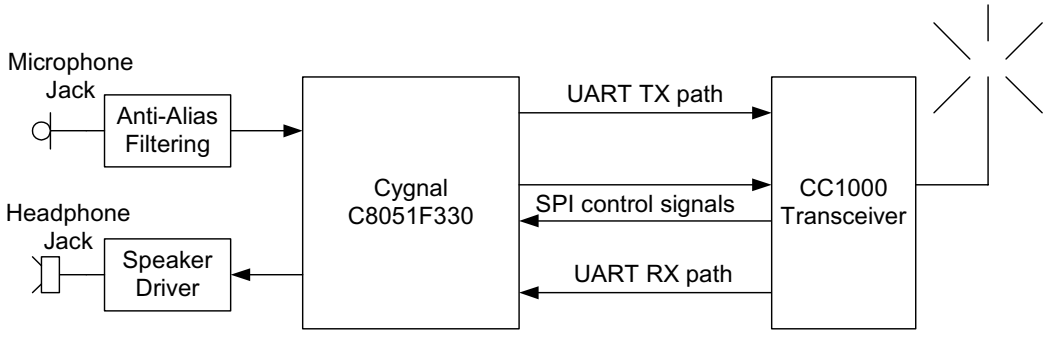


**Figure 1. System Block Diagram**



*CYGNAL Integrated Products, Inc.*
4301 Westbank Drive
Suite B-100
Austin, TX 78746
www.cygnal.com

# 3. Overview

Figure 1 illustrates the path travelled by a voice signal as it is filtered, sampled, compressed, transmitted, received, decompressed, and output by the microcontroller and the transceiver. Each stage of the process is discussed below.

## 3.1. Input Signal Filtering

The voice signal enters the board through a microphone jack. Before the signal reaches the ADC, an op-amp is used to amplify the signal to a full-scale level. A low pass filter with a bandwidth of 4 kHz is implemented to prevent aliasing.

## 3.2. Signal Path in the Microcontroller

The 10-bit ADC samples at a rate of 16 kHz, pushing each sample onto an ADC FIFO. The compression algorithm pulls these samples. Using a Continuously Varying Slope Delta compression scheme, each 10-bit sample can be compressed to a single bit. Eight consecutive samples are combined to form a frame of data that can be transmitted by the controller's UART.

Compressed voice data is pushed onto a UART Transmit FIFO. The UART is used to transmit data to and receive data from the RF transceiver. The SPI peripheral is used to control transceiver settings during transmission and reception. The transmitting and receiving RF transceivers are synchronized so that two transceivers do not try to transmit simultaneously. This can be achieved by implementing transmission state machines in software.

After RF transmission, the controller receiving the compressed voice data stores bytes in a UART RX FIFO. Data is pulled from this FIFO by a voice decompression algorithm that converts each byte of received data to eight 10-bit DAC output values. Each pair of output values is averaged to minimize noise, so the decompression algorithm outputs four DAC samples per UART data byte. Recovered voice samples are pushed onto the DAC output FIFO, and the DAC pulls samples from this FIFO at a rate of 8 kHz.

## 3.3. Signal Output

The DAC output is connected to a speaker driver. This component boosts the signal current so that sound will be audible at the headphone jack.

# 4. Hardware Description

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

## 4.1. Microphone Input

In order to take advantage of the ADC's full-scale input voltage of 2.43 Volts, the audio signal should be amplified. A gain of 10 provides adequate amplification while avoiding clipping high-amplitude audio signals.

When sampling audio at a given frequency, signals above half the sampling frequency will appear in the sample as an aliased signal, causing the audio to become distorted. Placing a low-pass filter with a corner frequency at half the sampling frequency can remove these aliased signals. In this design, a 7-pole Butterworth filter was used. However, the corner frequency for the filter is set to 4 kHz (for telephone-quality audio) even though the sampling rate is 16 kHz. This is done because sampling at 16 kHz helps to maintain audio fidelity through the compression process.

## 4.2. Peripherals

### 4.2.1. 10-bit ADC

The 10-bit ADC is set to sample at 16 kHz by configuring the peripheral to start conversions on Timer 2 overflows. The ADC uses the on-chip voltage reference as its reference voltage. If this configuration is used, a 0.1 µF capacitor and a 4.7 µF capacitor should be placed in parallel from the VREF pin to ground.

### 4.2.2. SPI Interface

Register changes for the RF transceiver used in this design are made through a 3-wire SPI interface with one pin devoted to the SPI clock, one pin to select between register reads and writes, and one pin shared for data input and data output. To connect an 'F330's SPI to a part with a single pin for MISO and MOSI, the 'F330's MISO and MOSI lines can be tied together and connected to VDD through a 1Kohm pull-up resistor. In this configuration, the output modes of MOSI and MISO should be configured to "open-drain".

### 4.2.3. UART Interface

The RF transceiver used in this design can send and receive data through a UART protocol, with one pin for reception and one pin for transmission. The 'F330's two UART pins can be connected to the RF transceiver, and UART interrupts can be used to send and receive data.

### 4.2.4. 10-Bit DAC

The 'F330's Current DAC outputs samples at a rate of 8 kHz using Timer 3 interrupts. A 1 Kohm resistor and a 33,000 pF capacitor are connected in parallel between the IDAC output pin and ground.

### 4.2.5. PCA

The software system requires that bit transition widths (the time between high-to-low and low-to-high transitions) on the UART RX data line be measured in order to detect valid data on the RX line.

One of the PCA's modules, set to edge-triggered capture mode and with its input pin tied to the UART RX pin, is used to measure these incoming bit widths. The PCA module used is configured to interrupt on rising and falling edges, and set to count system clocks.

At every 0-to-1 and 1-to-0 edge in the PCA ISR, a width is measured by taking the PCA capture value read during the last transition and subtracting it from the current PCA capture value. When valid data bits are received by the transceiver, all widths should be multiples of the baud rate's bit size, which is equal to the system clock divided by the set baud rate.

## 4.3. PCB Design Considerations

Following some simple guidelines in board layout can optimize audio and RF performance. When placing components, power supply decoupling capacitors should be located close to their associated power and ground pins. Whenever possible, analog and digital data traces should be separated to prevent digital noise from coupling into nearby voice and RF analog traces. Vias should be avoided when routing traces which carry high frequency data signals to minimize radiated noise.

If trace routing causes areas of copper to be isolated from the ground plane, vias should be used to connect those floating areas to the ground plane.

## 4.4. RF Transceiver Placement

When designing PCBs with RF components, special care must be taken to insure optimal operation. Chipcon provides a recommended layout that was followed closely in this design[1].

This design layout shows the implementation of an antenna using an SMA connector as well as an embedded PCB antenna. If the SMA antenna is to be used, the trace between the embedded PCB antenna and the transceiver should be cut.

When designing an embedded antenna, the antenna should be located at least 5 millimeters away from the ground plane. The trace length depends on the frequency at which the transceiver is operating. In the case of this design, which operates at 868 MHz, the trace should be about 3.2 inches long[2].

# 5. Software Description

The goal of the software system of this reference design is to transmit and receive intelligible voice signals while conserving power whenever possible. Transmitting and receiving signals requires capturing, compressing, and transmitting at one endpoint and receiving, decompressing, and outputting at the other end point.

The system uses interrupts to ensure that data reliably moves through the system at defined frequencies. Power consumption is minimized by recognizing the difference between an idle or "quiet" channel verses an active or "loud" channel. If no one is speaking into either microphone, sound does not need to be transmitted, so RF operation terminates.

The sections below discuss each part of the software system, including "loud" channel detection, RX/TX synchronization methods, and an example compression algorithm. All code described in the section below can be found in Appendix D on page 20.

## 5.1. Idle Channel

When neither end point is sampling an audio signal that is loud enough to warrant transmission, the communications channel can be considered Idle, and the microcontrollers conserve power by shutting down the RF transmitters. Each controller then watches for one of two events: a local voice signal to be transmitted, or a remote voice signal that should be received. When one of these two events occurs, the system then enters Transmit Mode or Receive Mode, respectively. Figure 2 on page 6 shows the Transmit and Receive state diagram.

## 5.2. Starting in Transmit Mode

When the local voice signal amplitude rises above a certain threshold (AUDIO_THRESHOLD in code), the audio signal is considered loud enough to transmit, and the software system is set to Transmit Mode. Transmit Mode switches the RF transceiver to enable transmission and sets the UART to transmit compressed data once DATA_BYTES_PER_TRANSMISSION have been pushed onto the UART TX FIFO by the compression algorithm.

After an RF calibration sequence (discussed in "RF Transceiver Calibration" on page 7), the system enters its first Transmit Session (discussed in"A Transmit Session" on page 8).

## 5.3. Starting in Receive Mode

Checking for a remote audio signal requires that the controller watch for a transmission from the remote RF transceiver. The Received Signal Strength Indicator (RSSI) analog output pin will indicate the presence of a strong signal at a set frequency by changing the pin's output voltage level.

The RSSI pin on the RF transceiver used in this design outputs a voltage that is inversely proportional to the signal strength. If the RSSI pin's voltage drops below a defined threshold (RSSI_THRESHOLD in code), then RF reception should begin, and the software system is set to Receive Mode. In Receive Mode, the RF transceiver is set to enable reception and the PCA interrupts are enabled.

After an RF calibration sequence, the system enters its first Receive Session (discussed in "A Receive Session" on page 8).

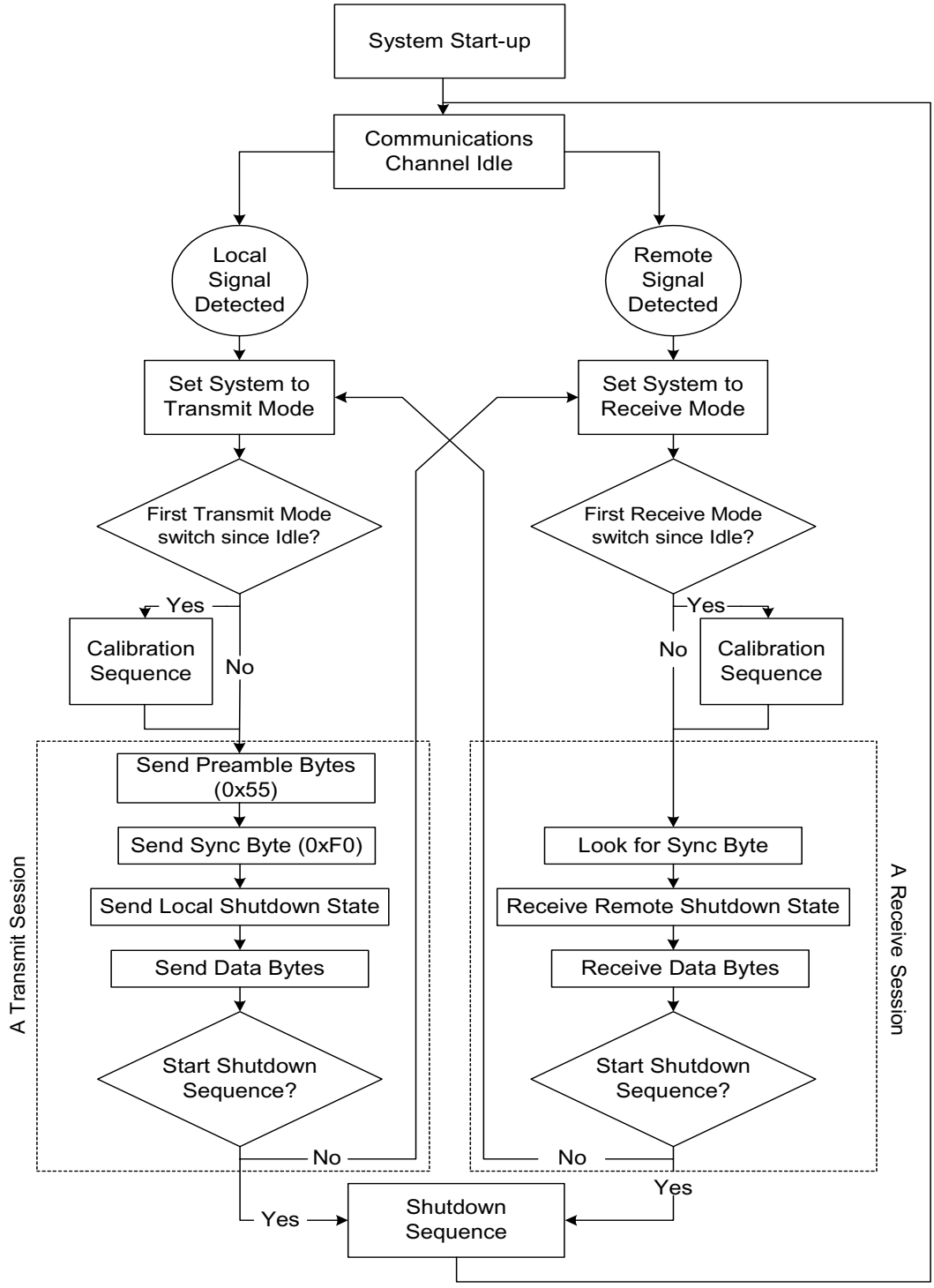


**Figure 2. Transmit and Receive State Diagram**

## 5.4. RF Transceiver Calibration

RF transceivers must be calibrated in order to optimize operation. An RF transceiver is considered calibrated if, upon data reception, it can accurately distinguish between transmitted 1's and 0's. In this design, calibration takes place every time the communication channel changes from Idle to Active and the software system switches an endpoint to Transmit Mode for the first time since the channel became active. Therefore, calibration occurs twice for every Idle-to-Active transition, since both end points' RF transceivers must be calibrated.

During calibration, the transmitter sends a number of preamble bytes, which consist of patterns of alternating 1's and 0's, such as 0x55 or 0xAA. Preamble bytes are used by the receiving RF transceiver to calibrate itself to distinguish between 1's and 0's using an Averaging Filter.

The Averaging Filter sets the slicing level for the internal data slicer. As a pattern such as 0x55 is received, the free-running Averaging Filter will drift to a mid-level value. After the controller manually locks the Averaging Filter, the slicing level will be held at this level and data will be received with optimal accuracy.

## 5.5. Disabling The Watchdog Timer

On microcontroller reset, the Watchdog Timer is automatically enabled. On the 'F330, a reset will be issued if the timer overflows, which occurs in about 1 ms. In most programs, the Watchdog Timer can be disabled at the beginning of the *main* function without any problems. However, if a software system spends too much time in code that is run before *main*, the WDT might overflow before it can be disabled.

In this software system, many variables are initialized in XDATA space, and XDATA space is initialized in code that is run before *main*. To avoid this reset, the file STARTUP.A51 must be modified so that the watchdog timer is disabled prior to variable initialization.

## 5.6. Synchronization Method

The RF transceiver used in this design can transmit and receive, but it cannot do both simultaneously. Software TX/RX synchronization between RF transceivers is necessary to implement a full duplex communication channel.

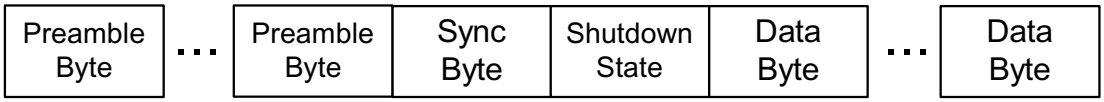


**Figure 3. Transmitted Data Frame**

## 5.7. A Transmit Session

Figure 3 shows the transmitted data frame used by the RF transceivers. After DATA_BYTES_PER_TRANSMISSION bytes have accumulated inside the UART TX FIFO, transmission begins. Preamble bytes are transmitted first and provide "padding" to allow for slack time in TX/RX transitions between the transceivers. After TX_BYTES_OF_PREAMBLE bytes have been sent, a Sync Byte (0xF0) is transmitted and used by the receiver to synchronize its UART. Since the UART transmits least significant bit first, the bits of 0xF0 will be received as "00001111".

Following the Sync Byte is the RF transceiver's LOCAL_SHUTDOWN_STATE (See "Shutdown Procedure" on page 10.). This byte of data indicates whether the local voice signal is quiet enough to allow for communication termination.

Next, data bytes are transmitted. Once DATA_BYTES_PER_TRANSMISSION have been sent, the system switches to Receive Mode.

## 5.8. A Receive Session

Receive Sessions begin in the PCA ISR, measuring bit widths in search of the Sync byte. The PCA ISR looks for a bit transition width of 5 bit times, or 5 * (SYSCLK / BAUDRATE), because the first nibble of the Sync byte will appear as "1000001", including the Stop bit from the last frame, the Start bit of this frame, and the first '1' of the Sync byte's second nibble.

When the ISR detects this width, it disables PCA interrupts and enables UART reception. At this point, the next 5 bits received will be the second nibble of the Sync byte and the UART frame's Stop bit, which will look like '11111'. UART reception will then begin when the next frame's Start bit (logic level '0') is received.

The first data byte to be received is the transmitter's local shutdown state, saved as REMOTE_SHUTDOWN_STATE. This byte tells the receiver if the other endpoint is ready to shutdown. Next, data bytes are received and stored in a UART RX FIFO. A counter increments at every received byte until it equals DATA_BYTES_PER_TRANSMISSION. At this point, the system switches to Transmit Mode.

# 6. Voice Compression

The RF transceiver used in this design can transmit at a maximum baud rate of 76.8 Kbd. Counting one Start Bit and one Stop bit for each UART frame, 7,680 bytes of data can be transmitted or received per second. However, because a full-duplex channel is needed, data must be transmitted by both microcontrollers, thus limiting the average number of bytes transmitted per endpoint to 3,840 bytes per second. Even if the 10-bit ADC samples were shortened to 8-bits, 16,000 bytes per transmitter per second would need to be sent to support telephone-quality audio fidelity.

Fortunately, a compression scheme called Differential Pulse Code Modulation (DPCM) has been developed that encodes and transmits the change between samples rather than each discrete sample. One form of DPCM compresses each sample down to a single bit, where '1' means to raise the amplitude by a delta value and '0' means to lower the amplitude by that same delta value.

## *6.1. Continuously Variable Slope Delta Modulation*

Continuously Variable Slope Delta (CVSD) Modulation[3], used in this design, improves on basic DPCM by adjusting the delta value according to the rate of change of the amplitude in the input signal. As the rate of change increases, the delta is increased so that the original signal can be traced more accurately.

A brief description of CVSD encoding follows. For a more thorough explanation, consult the tutorial at *http://www.eetkorea.com/ARTICLES/2001SEP/2001SEP05_AMD_MSD_AN1.PDF*.

### 6.1.1. Encoding

Figure 4 shows the CVSD encoder. $I_1$ and $I_2$ are integrators whose time constants are set to be approximately 4 ms and 1 ms, respectively. Q is a quantizer that outputs '1' if the difference between the current amplitude x(n) and $x_p(n)$ is positive, and outputs a '0' otherwise. The two tap delays ($z^{-1}$) allow the current output of the quantizer and the two previous outputs to be examined. Delta values can

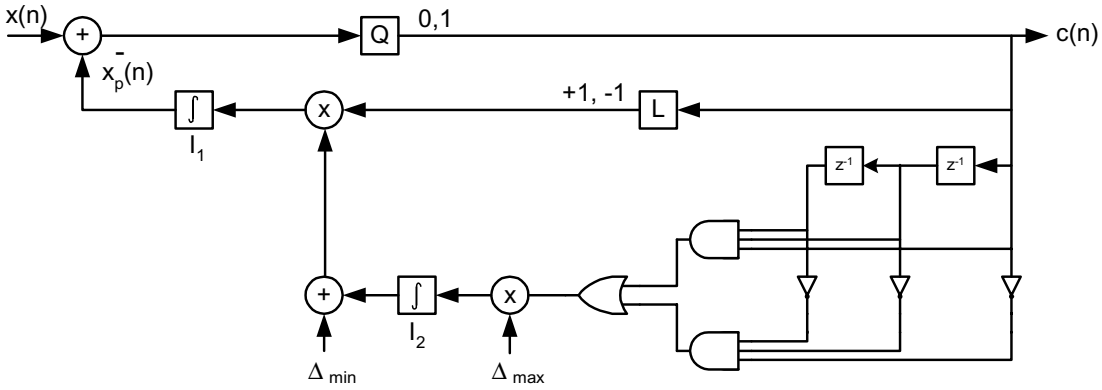


**Figure 4. CVSD Encoder Block Diagram [3]**

dynamically change because the output of $I_2$ will grow when consecutive '1's or '0's appear. L is a level converter that outputs '1' for input '1' and outputs '-1' for input '0'.

### 6.1.2. Decoding

Figure 5 shows the CVSD decoder. As in the encoder, tap delays and boolean logic are used to examine the last 3 quantizer values, and adjust the delta size according to the frequency of consecutive 1's or 0's. A level converter adjusts the quantized values so that either a positive or negative value is entered into the $I_1$ integrator.

Because the audio signal will increase or decrease by a calculated delta amount, every iteration of the decoder introduces high frequency noise. Adding a low pass filter after decoding can improve audio quality. In this reference design, the decompression algorithm averages each pair of samples to provide a digital low pass filter, and the output speaker driver implements an analog low pass filter as well.

## 6.2. Communication Shutdown

Data is transmitted between RF transceivers until both audio input signals are considered to be quiet. If no one has spoken into either microphone input for a determined length of time, then transmission is terminated in order to conserve power.

### 6.2.1. Shutdown Procedure

A quiet audio signal is one that does not change in amplitude, within an upper and lower threshold, over a set amount of time. To determine whether the local audio signal is "quiet", samples from the ADC must be examined.

The ADC ISR checks to see whether the current ADC input is within a certain window of values. If it is, a counter increases until SHUTDOWN_TOLERANCE is reached. If the signal goes outside of the window, then the counter is reset, and the current value is set to be the center of the new window of values.

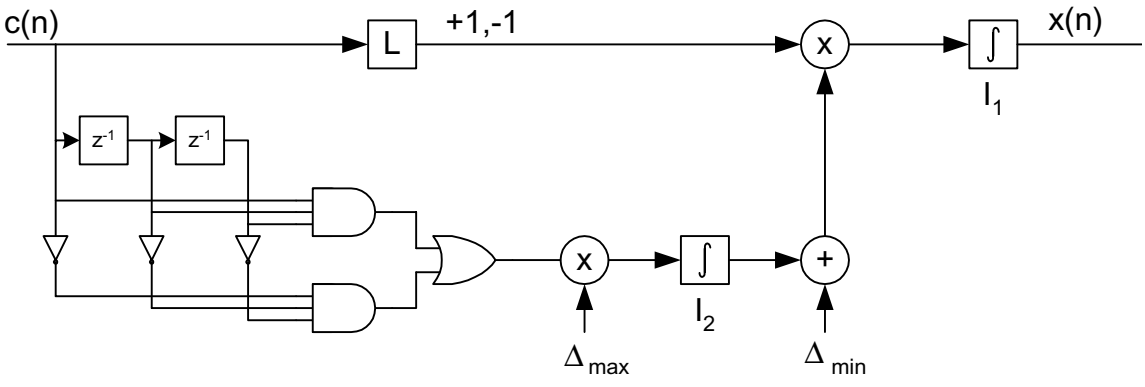


**Figure 5. CSVD Decoder Block Diagram [3]**

### 6.2.2. Shutdown State Machine

The system uses an RF transceiver shutdown state machine to ensure proper, synchronized shutdown between RF transceivers. The state machine is controlled by the variables LOCAL_SHUTDOWN_STATE and REMOTE_SHUTDOWN_STATE. The state machine updates once every receive session and once every transmit session. Figure 6 shows the state machine.

Once every transmit session, the audio signal is checked to see if it has become quiet by comparing ADC_LOW to SHUTDOWN_TOLERANCE. Once both the LOCAL_SHUTDOWN_STATE and the REMOTE_SHUTDOWN_STATE have reached the state ONE_ENDPOINT_QUIET, transmission can shut down.

## 6.3. RF Transceiver Settings

Chipcon provides a configuration wizard that calculates optimal register values for use with

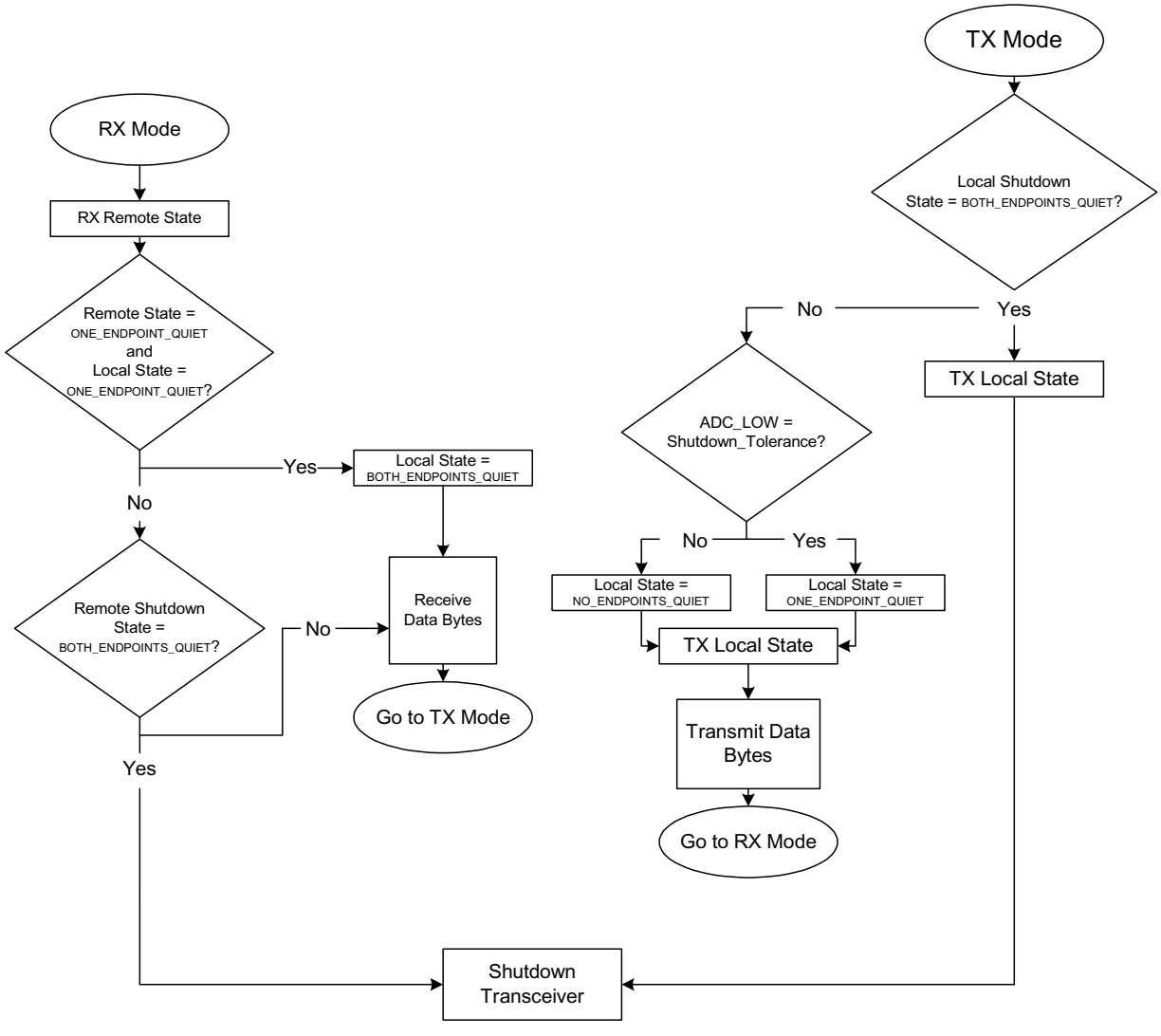


**Figure 6. Transmission Shutdown State Machine**

its transceivers. Values used in this design were found using the configuration wizard.

The RF transceiver has three basic modes of operation: Synchronous Manchester mode, Synchronous NRZ mode, and Transparent Asynchronous UART mode. For this design, Asynchrnoous UART mode is used. For more information on the RF transceiver's register settings, consult the CC1000 data sheet[4].

# 7. Usage Notes

To use the example, an external power supply, a microphone, and a set of headphones will be needed for each end point. Connect the power connector, headphone jack, and microphone jack into the correct places on the board. After connecting all external components, speaking into one microphone will cause the system to begin transmission.

# 8. Custom Configuration

Using the preprocessor *#define* directive allows users to adjust settings of the system easily. In this system, *#define* directives at the beginning of the code enable customization of tolerance levels, transmission speeds, etc.

Care must be taken when altering these values, however. For example, if the DATA_BYTES_PER_TRANSMISSION definition is set too high, transmit and receive buffers may overflow, causing a loss of data. If the UART's baud rate is set too low, data will be transmitted too slowly, also causing FIFO overflows.

Memory limitations must be considered before changing FIFO sizes. While RX_TX_FIFOSIZE is an array of 1-byte numbers, ADC_DAC_FIFOSIZE is an array of 2-byte numbers. So, setting ADC_DAC_FIFOSIZE to 100 will allocate 400 bytes of data space for the ADC and DAC FIFOs.

Changing threshold and tolerance values will effectively adjust the sensitivity of the system. If SHUTDOWN_TOLERANCE is set to a higher value, the system will continue to transmit for a longer period of time before shutting down. If CHANGE_IN_VOLUME is changed, the system's sensitivity to quiet signals changes as well.

# 9. References

[1]Chipcon. *CC1000PP Plug and Play User's Guide*. Rev 1.22, 2-3-2003.

[2]Telecontrolli. *Antenna Design Considerations*. Application note. http://www.telecontrolli.com/pdf/Antenna%20Design%20Considerations.pdf

[3] MX-Com, INC. *Continuously Variable Slope Delta Modulation: A Tutorial*. 1998.

[4]Chipcon. *CC1000 Data Sheet*. Rev 2.1, 4-19-2002.

# Appendix A - Schematics

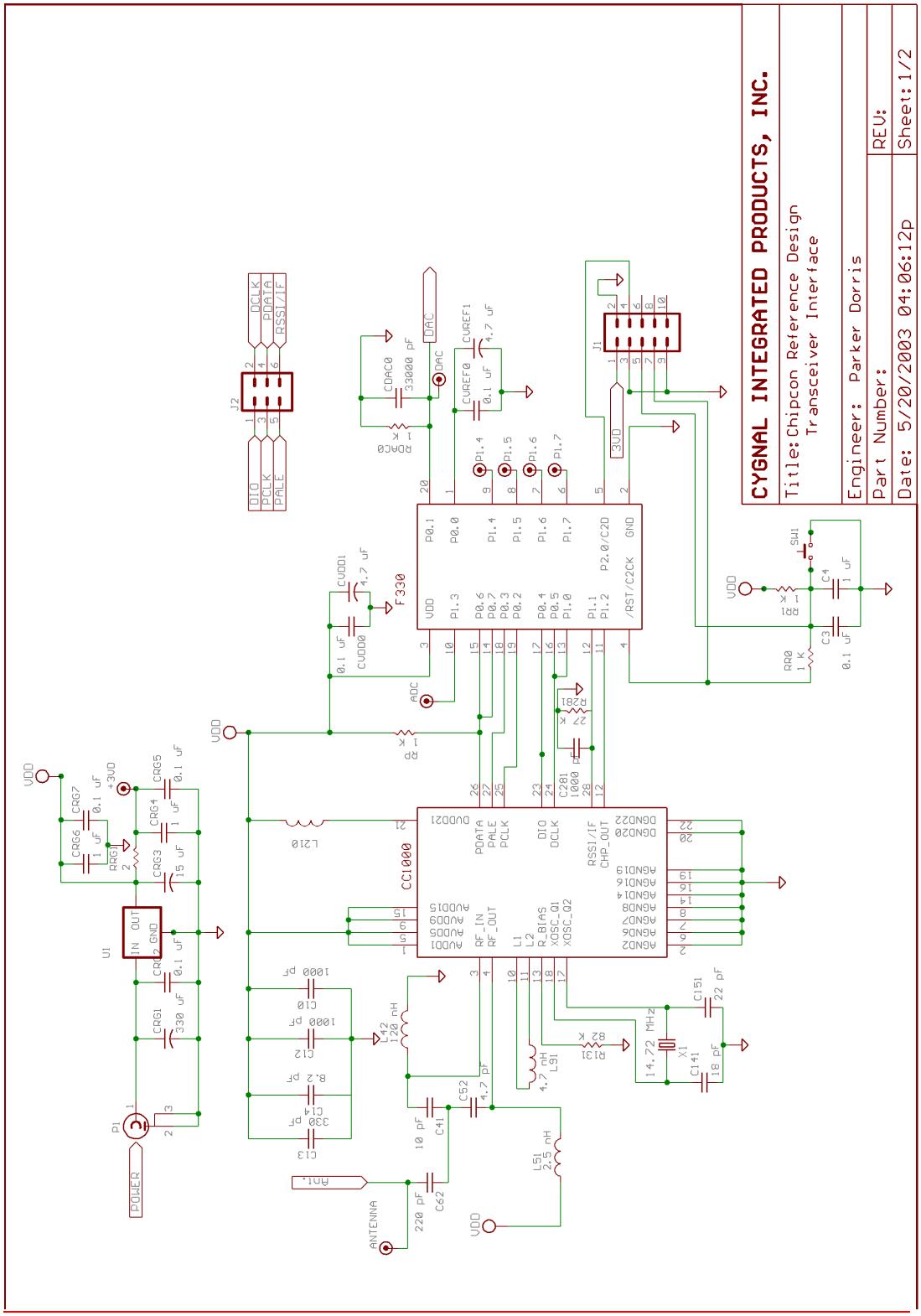


Figure 7. Transceiver / Microcontroller Interface

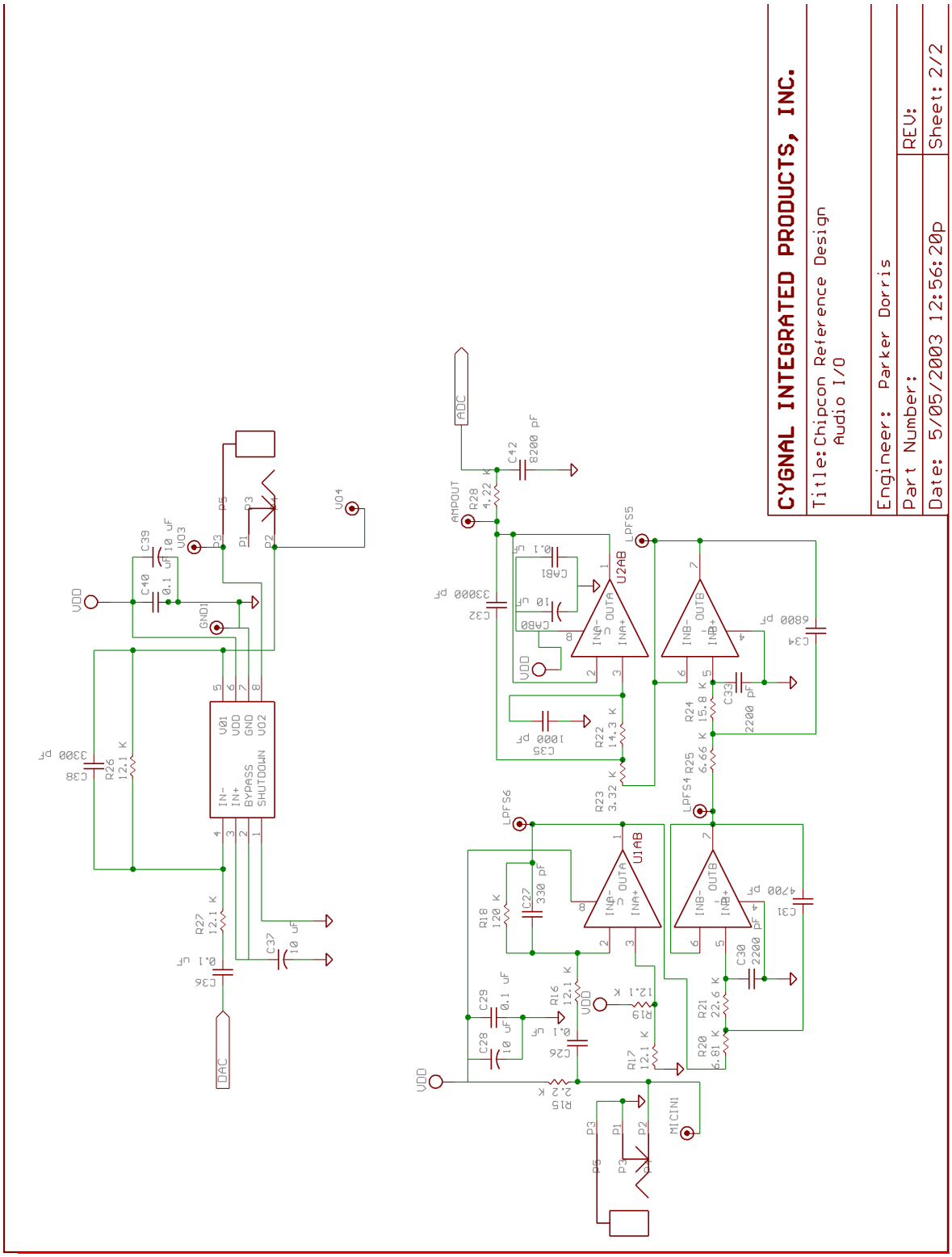


**Figure 8. Audio Amplification, Filtering, and Biasing**

# Appendix B - Bill of Materials

| Part | Value | Part Description | Comments |
|---|---|---|---|
| C3 | 0.1 uF | 805 | |
| C4 | 1 uF | 805 | |
| C10 | 1000 pF | 805 | |
| C12 | 1000 pF | 805 | |
| C13 | 330 pF | 805 | |
| C14 | 8.2 pF | 1206 | |
| C26 | 0.1 uF | 805 | |
| C27 | 330 pF | 805 | |
| C28 | 10 uF | 805 | |
| C29 | 0.1 uF | 805 | |
| C30 | 2200 pF | 805 | |
| C31 | 4700 pF | 805 | |
| C32 | 33000 pF | 805 | |
| C33 | 2200 pF | 805 | |
| C34 | 6800 pF | 805 | |
| C35 | 1000 pF | 805 | |
| C36 | 0.1 uF | 805 | |
| C37 | 10 uF | 805 | |
| C38 | 3300 pF | 805 | |
| C39 | 10 uF | 805 | |
| C40 | 0.1 uF | 805 | |
| C41 | 10 pF | 805 | |
| C42 | 8200 pF | 805 | |

| | | | |
|---|---|---|---|
| C52 | 4.7 pF | 1206 | |
| C62 | 220 pF | 805 | |
| C141 | 18 pF | 805 | |
| C151 | 22 pF | 805 | |
| C281 | 1000 pF | 805 | |
| CAB0 | 10 uF | 805 | |
| CAB1 | 0.1 uF | 805 | |
| CC1000 | | CC1000 | |
| CDAC0 | 33000 pF | 805 | |
| CRG1 | 330 uF | CR_5,0 | |
| CRG2 | 0.1 uF | 805 | |
| CRG3 | 15 uF | CR_2,5 | |
| CRG4 | 1 uF | 805 | |
| CRG5 | 0.1 uF | 805 | |
| CRG6 | 1 uF | 805 | |
| CRG7 | 0.1 uF | 805 | |
| CVDD0 | 0.1 uF | 805 | |
| CVDD1 | 4.7 uF | 3216 | |
| CVREF0 | 0.1 uF | 805 | |
| CVREF1 | 4.7 uF | 3216 | |
| F330 | | F330 | |
| J1 | | 5x2 Conn. | |
| L42 | 120 nH | 805 | |
| L51 | 2.5 nH | 805 | Coilcraft 0805HQ-2N5X_BB |
| L91 | 4.7 nH | 805 | |
| L210 | EMI filter bead | 603 | Murata BLM18HG102SN1D |
| R15 | 2.2 K | 805 | |
| R16 | 12.1 K | 805 | |

| | | | |
|---|---|---|---|
| R17 | 12.1 K | 805 | |
| R18 | 120 K | 805 | |
| R19 | 12.1 K | 805 | |
| R20 | 6.81K | 805 | |
| R21 | 22.6 K | 805 | |
| R22 | 14.3 K | 805 | |
| R23 | 3.32 K | 805 | |
| R24 | 15.8 K | 805 | |
| R25 | 6.66 K | 805 | |
| R26 | 12.1 K | 805 | |
| R27 | 12.1 K | 805 | |
| R28 | 4.22 K | 805 | |
| R131 | 82 K | 805 | |
| R281 | 27 K | 805 | |
| RDAC0 | 1 K | 805 | |
| RP | 1 K | 805 | |
| RR0 | 1 K | 805 | |
| RR1 | 1 K | 805 | |
| RRG1 | 2 Ohm | 2512B | |
| SW1 | SW_PB_6MM | SW_6MM | |
| TEXAS1 | TPA4861D | SO-08 | Texas Instruments TPA4861D |
| U$21 | SMAPLUG | SMAPLUG | AMP/Tyco Electronics 221789-1 |
| U$28 | | SMTAUDIO | CUI, Inc MJ-3510-SM |
| U$36 | | SMTAUDIO | CUI, Inc MJ-3510-SM |
| U1 | | SOT223 | Voltage Regulator 3.3 Volts |
| U1AB | LT1810 | SOIC8 | Maxim MAX492CSA |
| U2AB | LT1810 | SOIC8 | Maxim MAX492CSA |
| X1 | 14.72 MHz | HC49/S | Citizen America Corporation HC49US14.7456MABJ |

# Appendix C - Board Layout

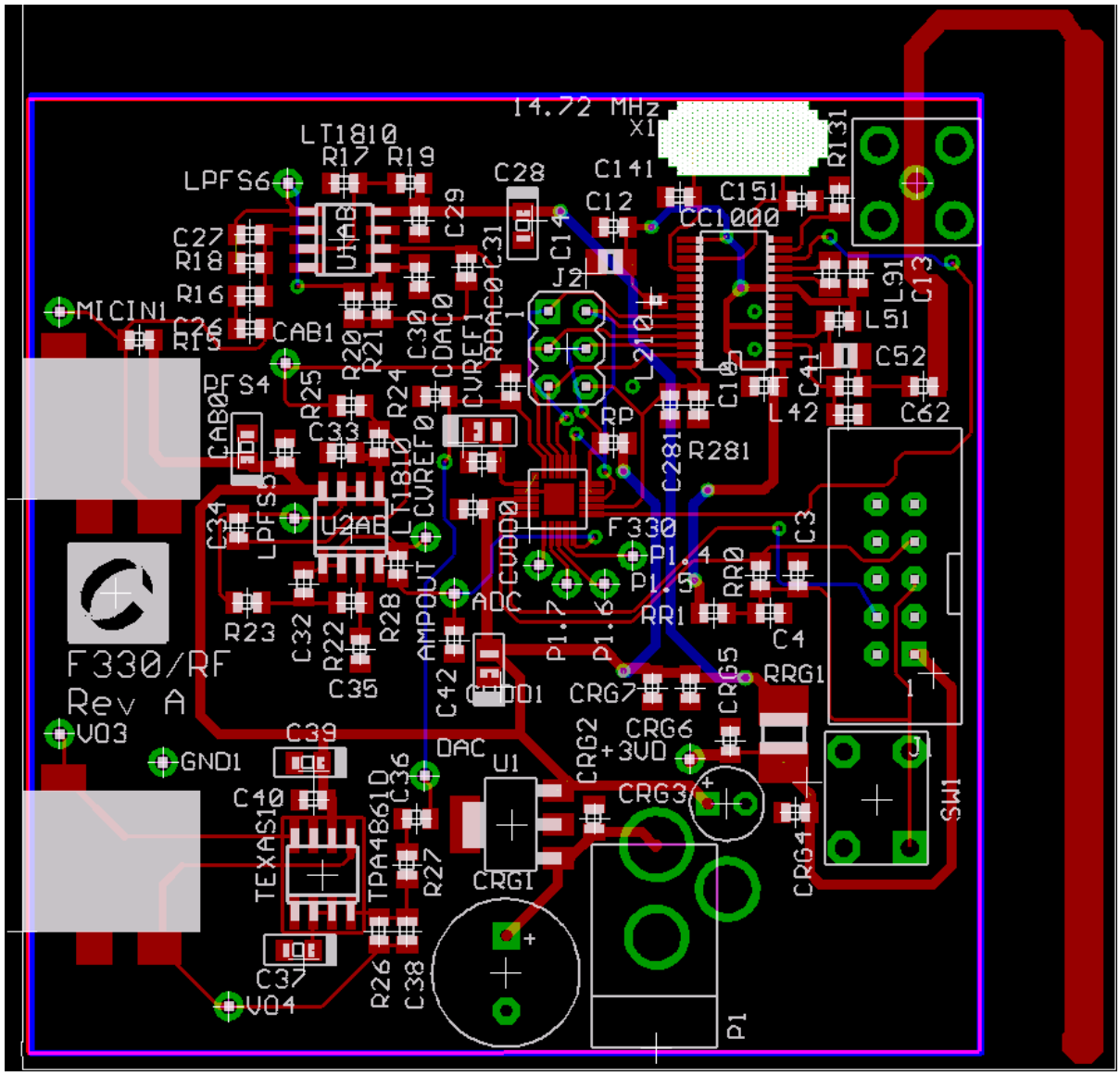


**Figure 9. Board Layout**

# Appendix D - Firmware Listing

```
//-----------------------------------------------------------------------------
// Wireless Voice Transmitter/Receiver
//-----------------------------------------------------------------------------
//
// AUTH: PD
// DATE: 28 Apr 03


//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------
#include <c8051f330.h>                 // SFR declarations


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


#define SYSCLK 24500000                // speed of internal clock
#define SAMPLE_RATE 16000              // sampling rate for the ADC
#define BAUDRATE  76800                // Baud rate of UART in bps
#define ADC_DAC_FIFOSIZE 90            // Size of ADC/DAC FIFOs
#define RX_TX_FIFOSIZE  60             // Size of UART FIFOs
#define DACUPDATERATE 8000             // rate of DAC sample output

// Transmission-Related Constants
#define DATA_BYTES_PER_TRANSMISSION 20 // bytes TX per Transmit Session
#define TX_BYTES_OF_PREAMBLE    7      // bytes of 0x55 TX per Transmit Session
#define TX_WAKEUP_BYTES 20             // for first TX for RX detection
                                       // and calibration
#define RX_MODE   0                    // used to set UART_MODE to RX
#define TX_MODE   1                    // used to set UART_MODE to TX
#define RX_MINIMUM_PREAMBLE_BAUDS 40   // number of preamble bauds that
                                       // must be detected before the
                                       // averaging filter locks
#define EDGES_TO_DISREGARD 20          // used by PCA interrupt during TX-to-RX
                                       // transitions to avoid invalid data
#define BAUDTOLERANCE 9                // used for baud measurements:
                                       // baudtol./10 < width < 20-baudtol./10
#define STARTUP_SILENCE 30             // to avoid audbile "pop" at beginning
                                       // transmission:
                                       // STARTUP_SILENCE * DACUPDATERATE
                                       // seconds of silence

// Wake up Constants
#define RSSI_THRESHOLD 80              // wake-up amplitude level for RSSI
                                       // in ADC codes
#define AUDIO_THRESHOLD 720            // wake-up amp. level for audio input
                                       // in ADC codes

// Shutdown Constants
#define CHANGE_IN_VOLUME 8             // sets window for quiet signal
                                       // detection in ADC codes.  sample is
                                       // compared to center saved
```

```
#define SHUTDOWN_TOLERANCE  4000        // value +/- C_I_V sets number of
                                        // consecutive samples that must fall
                                        // within window before shutdown
                                        // sequence begins
#define TRANSITION_COMPLETE 4           // used to test for 250 us elapsed time
                                        // during RX-to-TX and TX-to_RX
                                        // transitions.
                                        // TRANSITION COMPLETE =
                                        // (SAMPLE_RATE * 25) / 100000



// CSVD Constants
#define DELTAMAX 200                    // adjusts the magnitude of delta value
                                        // change
#define DELTAMIN 10                     // sets minimum delta value
#define I1_TIMECONST 4                  // sets speed of output change
                                        // I1 = I1_TIMECONST/SAMPLE_RATE sec
#define I2_TIMECONST 5                  // sets speed of delta changes
                                        // I2 = I2_TIMECONST/SAMPLE_RATE sec

//-----------------------------------------------------------------------------
// CC1000 Registers
//-----------------------------------------------------------------------------
#define MAIN          0x00
#define FREQ_2A       0x01
#define FREQ_1A       0x02
#define FREQ_0A       0x03
#define FREQ_2B       0x04
#define FREQ_1B       0x05
#define FREQ_0B       0x06
#define FSEP1         0x07
#define FSEP0         0x08
#define CURRENT       0x09
#define FRONT_END     0x0A
#define PA_POW        0x0B
#define PLL           0x0C
#define LOCK          0x0D
#define CAL           0x0E
#define MODEM2        0x0F
#define MODEM1        0x10
#define MODEM0        0x11
#define MATCH         0x12
#define FSCTRL        0x03
#define PRESCALER     0x1C
#define TEST4         0x42


//-----------------------------------------------------------------------------
// 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 PCA0_Init(void);                   // edge-triggered interrupts
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 UART0_Init (void);                 // set to BAUDRATE, interrupts enabled
void SPI_Init(void);                    // enable 3-wire Master SPI
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 UTXFIFO_Pull();           // pulls compressed sample to be TXed
unsigned char URXFIFO_Pull();           // pulls RXed compressed sample
unsigned short ADCRXFIFO_Pull();        // pulls ADC sample
unsigned short DACTXFIFO_Pull();        // pulls decompressed sample
void UTXFIFO_Push(unsigned char);       // pushes compressed sample to be TXed
void URXFIFO_Push(unsigned char);       // pushes RXed compressed sample
void ADCRXFIFO_Push(unsigned short);    // pushes ADC sample
void DACTXFIFO_Push(unsigned short);    // pushes decompressed sample
void CLEAR_FIFOS(void);                 // resets all FIFOs to default values

// CVSD Compression Algorithms
void RXalgorithm(void);                 // Compresses an ADC sample
void TXalgorithm(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
void C1000_LOCK_FILTER(void);           // locks Averaging Filter
void C1000_UNLOCK_FILTER(void);         // sets Avg. Filter to free-running
void C1000_Init(void);                  // initializes RF transceiver
void C1000_RX_MODE(void);               // sets RF transceiver settings to RX
void C1000_TX_MODE(void);               // sets RF transceiver settings to TX
void C1000_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



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


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

// This struct is used for the 8-bit data frame UART fifos
struct CHAR_FIFO_ST
{
   unsigned char EMPTY;                 // empty = 1 means the fifo is empty
   unsigned char FIRST;                 // index to element to be pulled next
   unsigned char LAST;                  // index to elmnt most recently pushed
   signed short COUNT;                  // saves number of elements in FIFO
   unsigned char OF;                    // overflow indicator
   unsigned char FIFO[RX_TX_FIFOSIZE]; // array to store elements
};

// This struct is used for ADC and DAC data, which require 16-bit precision
struct SHORT_FIFO_ST
{
   unsigned char EMPTY;                 // empty = 1 means the fifo is empty
   unsigned char FIRST;                 // index to element to be pulled next
   unsigned char LAST;                  // index to elmnt most recently pushed
   signed short COUNT;                  // saves number of elements in FIFO
   unsigned char OF;                    // overflow indicator
   unsigned short FIFO[ADC_DAC_FIFOSIZE];
                                        // array to store elements
};

/*
RX State Machine States

RX_UNCALIBRATED
   Averaging Filter has not been locked

RX_SEARCH_FOR_SYNCBYTE
   inside PCA ISR measuring bit widths

RX_DATA
   state spent in UART ISR receiving data bytes

*/
enum RX_STATES {RX_UNCALIBRATED, RX_SEARCH_FOR_SYNCBYTE, RX_DATA};

/*
TX State Machine States
```

```
TX_UNCALIBRATED
   state where wake-up bytes are sent

TX_PREAMBLE_BAUDS
   transmit preamble bauds

TX_SYNCBYTE
   transmit Sync byte

TX_DATA
   transmit data
*/
enum TX_STATES {TX_UNCALIBRATED, TX_PREAMBLE_BAUDS, TX_SYNCBYTE, TX_DATA};


/*
Shutdown State Machine States

NO_ENDPOINTS_QUIET
   transmission should continue, channel is active

ONE_ENDPOINT_QUIET
   either remote or local endpoint's voice signal is considered quiet, and
   is ready to shutdown

BOTH_ENDPOINTS_QUIET
   both endpoints' voice signals are quiet, power down
   RF transceivers

POWERING_DOWN
   transmission should immediately terminate
*/

enum SHUTDOWN_STATES {NO_ENDPOINTS_QUIET, ONE_ENDPOINT_QUIET,
   BOTH_ENDPOINTS_QUIET, POWERING_DOWN};



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

sbit PALE = P0^3;                     // indicates to RF transceiver whether
                                      // SPI data is a register read or write


// FIFOs
// FIFOs are stored in 512-byte XDATA space
// All FIFOs are initialized to be empty
struct SHORT_FIFO_ST xdata DACTXFIFO = {{1}};
struct SHORT_FIFO_ST xdata ADCRXFIFO = {{1}};
struct CHAR_FIFO_ST xdata URXFIFO = {{1}};
struct CHAR_FIFO_ST xdata UTXFIFO = {{1}};

// RX/TX State Machine variables, flags
unsigned char RX_STATEMACHINE;        // used in PCA ISR and UART ISR
```

 

```
unsigned char TX_STATEMACHINE;          // used in UART ISR
unsigned short EDGES_THIS_SESSION;      // counts edges at the beginning of
                                        // an RX session
bit UART_MODE;                          // indicates RX/TX: TX = 1, RX = 0
bit RUN_ASLEEP = 1;                     // if set, channel is idle and
                                        // ASLEEP() should run
bit TX_IN_PROGRESS;                     // indicates whether or not a TX
                                        // session has started
bit RX_IN_PROGRESS;
unsigned short ADC_LOW;                 // counts consecutive ADC samples
                                        // that fall within quiet signal window
unsigned short CENTER_OF_WINDOW;            // saves center value for ADC window
                                        // calculations
unsigned char LOCAL_SHUTDOWN_STATE;     // saves micro's shutdown state
unsigned char REMOTE_SHUTDOWN_STATE;    // saves other micro's shutdown state,
                                        // received from UART
unsigned char TRANSITIONCOUNT;          // measures time between RX/TX
                                        // register calls to update and when
                                        // the transition will take effect
bit TRANSITION_IN_PROGRESS;             // indicates RX/TX transition
bit TO_TX_INDICATOR;                    // equals 1 for RX->TX and 0 for TX->RX


// RX Algorithm Variables
bit INIT_RX_ALGORITHM;
// TX Algorithm Variables
bit INIT_TX_ALGORITHM;

short accumulate;                       // used to create a LPF on DAC output

// DAC ISR variable
unsigned short DACstartupTimer;         // counts first STARTUP_SILENCE
                                        // interrupts



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

   PORT_Init();                         // initialize and enable the Crossbar
   SYSCLK_Init();                       // initialize oscillator
   Timer0_Init();
   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/DACUPDATERATE);   // 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

   C1000_Init();                        // initalize the transceiver
```

```
   UART0_Init();                        // initialize UART


   while (1)
   {
      while (RUN_ASLEEP){ASLEEP();}     // RUN_ASLEEP is set on start-up
                                        // and when the communications channel
                                        // switches from active to idle during
                                        // the shutdown procedure

      if (!ADCRXFIFO.EMPTY)             // The compression algorithm should run
         RXalgorithm();                 // any time samples exist to compress

      if ((!URXFIFO.EMPTY))             // Decompress received samples when
         TXalgorithm();                 // available
   }

}


//-----------------------------------------------------------------------------
// 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
}

//-----------------------------------------------------------------------------
// 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 - SCLK
// P0.3 - PALE (Digital RF Control Signal)
// P0.4 - UART TX
// P0.5 - UART RX
// P0.6 - MISO
// P0.7 - MOSI
// P1.0 - CEX0
// P1.1 - RSSI Analog Input
// P1.2 - CHP_OUT (Open Drain)
// P1.3 - Analog Input (Incoming Signal)
// P1.4 -

void PORT_Init (void)
{
   XBR0  = 0x03;                        // enable UART at P0.4 and P0.5
                                        // and enable SPI
   P0SKIP  = 0x0B;                      // Skip VREF (P0.0), IDAC0(P0.1),
                                        // and P0.3
   P1SKIP = 0x5E;                       // Skip CEX0(P1.0), P1.1, P1.2,
                                        // P1.3, P1.4, P1.6
   XBR1   |= 0x41;                      // Enable crossbar, CEX0 at port pin
   P0MDIN &= ~0x01;                     // set P0.0 to analog input
   P0MDOUT |= 0x1C;                     // set P0.3 (PALE), UTX, SCLK to
                                        // push-pull
   P1MDIN &= ~0x0A;                     // Set P1.3, P1.1 as analog input

}

//-----------------------------------------------------------------------------
// 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 = 0x50;                      // Master enable, CKPOL = 1
   SPI0CKR = (SYSCLK / (2 * 1000000)) - 1;
                                        // sets SPI clock to 1 MHz
   SPI0CN  = 0;                         // clear all flags
   SPIEN = 1;                           // enable SPI
}
```

```
//-----------------------------------------------------------------------------
// 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 2 Overflows
   IDA0CN |=  0x80;                    // Enable DAC

}

//-----------------------------------------------------------------------------
// UART0_Init
//-----------------------------------------------------------------------------
//
// Configure the UART0 using Timer1, for <BAUDRATE> and 8-N-1.  UART interrupt
// is enabled.
//
void UART0_Init (void)
{
   SCON0 = 0x00;                       // SCON0: 8-bit variable bit rate
                                       //        level of STOP bit is ignored
                                       //        RX disabled
                                       //        ninth bits are zeros
                                       //        clear RI0 and TI0 bits

   if (SYSCLK/BAUDRATE/2/256 < 1) {
      TH1 = -(SYSCLK/BAUDRATE/2);
      CKCON &= ~0x0B;                  // T1M = 1; SCA1:0 = xx
      CKCON |=  0x08;
   } else if (SYSCLK/BAUDRATE/2/256 < 4) {
      TH1 = -(SYSCLK/BAUDRATE/2/4);
      CKCON &= ~0x0B;                  // T1M = 0; SCA1:0 = 01
      CKCON |=  0x01;
   } else if (SYSCLK/BAUDRATE/2/256 < 12) {
      TH1 = -(SYSCLK/BAUDRATE/2/12);
      CKCON &= ~0x0B;                  // T1M = 0; SCA1:0 = 00
   } else {
      TH1 = -(SYSCLK/BAUDRATE/2/48);
      CKCON &= ~0x0B;                  // T1M = 0; SCA1:0 = 10
      CKCON |=  0x02;
   }

   TL1 = TH1;                          // init Timer1
   TMOD &= ~0xf0;                      // TMOD: timer 1 in 8-bit autoreload
   TMOD |=  0x20;
   TR1 = 1;                            // START Timer1

   IE |= 0x10;                         // Enable UART Interrupts
}
```

```
//
//-----------------------------------------------------------------------------
// PCA0_Init
//-----------------------------------------------------------------------------
//
// The PCA is used to check for valid data at the UART RX pin.  Module 0
// is configured for edge-triggered captures, and count sysclocks.
// PCA interrupts are enabled.  Leaves PCA disabled.
//
void PCA0_Init(void)
{
   PCA0CPM0 |= 3<<4;                  // Set PCA to edge-capture on rising
                                      // and falling edges
   PCA0CPM0 |= 1;                     // enable CCF0 interrupts
   PCA0MD |= 4<<1;                    // Set PCA0 to count sys clocks
   EIE1 |= 0x10;                      // Enable PCA0 interrupts
}

//-----------------------------------------------------------------------------
// Timer0_Init
//-----------------------------------------------------------------------------
//
// Timer 0 is used by WaituS() to delay by a reliable amount of time.
// Timer 0 is set to overflow every 1us and is left disabled.
void Timer0_Init(void)
{
   CKCON |= 0x04;                     // Timer 0 counts sysclocks
   TMOD |= 0x02;                      // set Timer 0 to 8-bit counter with
                                      // auto-reload
   TL0 = -(SYSCLK/1000000);           // to overflow at rate of 1us
   TH0 = -(SYSCLK/1000000);           // set reload value


}



//-----------------------------------------------------------------------------
// Timer2_Init
//-----------------------------------------------------------------------------
//
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.
//
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
}


//-----------------------------------------------------------------------------
// 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.  The ISR also acts as a timer to measure the 250us
// delay when issuing an RX->TX or TX->RX command to the RF transceiver.
//

void ADC0_ISR(void) interrupt 10
{
   unsigned short value;
   AD0INT = 0;

   value = ADC0;

   // Checks for a quiet audio input.  If signal falls within the
   // CENTER_OF_WINDOW +/- CHANGEE_IN_VOLUME window, increment
   // ADC_LOW up to SHUTDOWN_TOLERANCE.  If it does not,
   // reset ADC_LOW to 0.
   if (((value > (CENTER_OF_WINDOW - CHANGE_IN_VOLUME)) &&
      (value < (CENTER_OF_WINDOW + CHANGE_IN_VOLUME))) )
   {
      // ADC_LOW is used to determine whether LOCAL_SHUTDOWN_STATE
      // should change from NO_ENDPOINTS_QUIET to ONE_ENDPOINT_QUIET
      if (ADC_LOW != SHUTDOWN_TOLERANCE)
         ADC_LOW++;
   }
   else
   {
      CENTER_OF_WINDOW = value;        // current value outside of
                                       // window should be set as the
                                       // new center of the allowable
                                       // values
      ADC_LOW = 0;                     // reset counter because signal's
                                       // amplitude has changed too much
```

 

```
                                        // to be considered quiet
   }

   ADCRXFIFO_Push(ADC0);                // save ADC value for compression

   // TRANSITION_IN_PROGRESS is set at the end of an RX or TX session
   // by C1000_RX_MODE or C1000_TX_MODE, after the CURRENT register is
   // updated, because a 250 us delay is needed.
   if(TRANSITION_IN_PROGRESS)
   {
      if(++TRANSITIONCOUNT == TRANSITION_COMPLETE)
      {
         // TO_TX_INDICATOR = 1 if RX->TX transition
         if(TO_TX_INDICATOR)
         {
            UART_MODE = TX_MODE;
            SETREG(PA_POW,0x80);       // sets the RF TX output power
         }
         else                          // TX->RX transition
         {
            UART_MODE = RX_MODE;
         }
         TRANSITIONCOUNT = 0;          // reset counter
         TRANSITION_IN_PROGRESS = 0;   // reset flag
      }
   }

}

//-----------------------------------------------------------------------------
// PCA0_ISR
//-----------------------------------------------------------------------------
// This ISR measures widths between level transitions on the UART RX pin
// and updates the RX_STATEMACHINE depending on these transition widths.
// In certain states, the ISR can lock the averaging filter or
// enable UART reception.
//

void PCA0_ISR(void) interrupt 11
{
   static unsigned short newvalue = 0; // saves newest PCA0CP0 value
   static unsigned short oldvalue = 0; // saves second most recent value
   unsigned short width;               // saves calculated bit width
   // used to count the number of preamble bits found
   static unsigned char preamble_count = 0;


   CCF0 = 0;                           // Acknowledge interrupt

   oldvalue = newvalue;                // save value from last ISR
   newvalue = PCA0CP0;                 // save newest value
   width = newvalue - oldvalue;        // calculate transition width


   // This code looks for the sync byte in received data by
   // watching for a 5-byte-wide transition width corresponding
   // to the first (low logic level) nibble of 0x0F
```

```
// Remember that UART transmits bytes LSB first
if (RX_STATEMACHINE == RX_SEARCH_FOR_SYNCBYTE)
{
   // Edges are counted at the beginning of the RX session
   // to prevent the state machine from getting confused
   // by noise output as data from the RF transceiver
   if (++EDGES_THIS_SESSION > EDGES_TO_DISREGARD)
   {
      if( (width > 5 * (BAUDTOLERANCE * SYSCLK / BAUDRATE / 10))
       && (width < 5 * ( (20 - BAUDTOLERANCE) * SYSCLK / (BAUDRATE * 10) )))
      {
         ES0 = 1;                    // Enables UART interrupts
         REN0 = 1;                   // Enables UART receives
         CR = 0;                     // Disables PCA0 counter
         EIE1 &= ~0x10;              // Disable PCA0 interrupts
         CCF0 = 0;

         // These statements reset counter variables and registers
         oldvalue = 0;
         PCA0CP0 = 0;
         EDGES_THIS_SESSION = 0;
         preamble_count = 0;
      }
   }
}

else // RX_STATEMACHINE == UNCALIBRATED

// This code runs once, after ASLEEP exits, the first time the system
// gets set to RX mode.  It looks for a preamble and sets the
// transceiver's averaging filter after enough preamble bits have been
// received.
{
   if ( ((width > BAUDTOLERANCE *(SYSCLK / BAUDRATE) / 10)
     && (width < (20 - BAUDTOLERANCE) * (SYSCLK / BAUDRATE) / 10))
     || ((width > 2 * BAUDTOLERANCE *(SYSCLK / BAUDRATE) / 10)
     && (width < 2 * (20 - BAUDTOLERANCE) * (SYSCLK / BAUDRATE) / 10) ))
   {
      ++preamble_count;

      // if most of the bits received are preamble bits, then
      // lock the Averaging Filter and look for the Sync byte
      if (preamble_count == RX_MINIMUM_PREAMBLE_BAUDS)
      {
         RX_STATEMACHINE = RX_SEARCH_FOR_SYNCBYTE;
         C1000_LOCK_FILTER();
      }
   }
   else
   {
      // if the bit is not a preamble, decrement the count unless
      // it already equals 0
      if(preamble_count != 0)
      {
         preamble_count--;
      }
   }
```

```
   }
}

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

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();

         // DACstartupTimer prevents the output of the
         // start-up audio "thump" caused by the compression algorithm
         if(DACstartupTimer == STARTUP_SILENCE)
         {
            // DAC output must be left-justified, and loaded
            // low byte first
            tempvalue.S = new_value << 6;
            IDA0L = tempvalue.C[1];
            IDA0H = tempvalue.C[0];
         }
         else
         {
            DACstartupTimer++;
         }

      }

}




//-------------------------------------------------------------------------
// UART_ISR
//-------------------------------------------------------------------------
//
// This interrupt checks to see whether TIO or RIO is set.  If TIO is set,
// then the next byte of information is transmitted.  If RIO is set,
// the received byte of information is stored in a FIFO.
// The ISR also updates the shutdown state machine according to
// REMOTE_SHUTDOWN_STATE and LOCAL_SHUTDOWN_STATE.
//
void UART0_ISR(void) interrupt 4
{
   static unsigned char Preamblecount = 0;
   static unsigned char TXcount = 0;
   static unsigned char RXcount = 0;
```

```
if (UART_MODE == TX_MODE)           // System is running a Transmit Session
{
   if (TI0 == 1)
   {
      TI0 = 0;                      // acknowledge interrupt

      if(TX_STATEMACHINE == TX_PREAMBLE_BAUDS)
      {
         SBUF0 = 0x55;              // 0x55 = 01010101
         ++Preamblecount;           // count measures number of preamble
                                    // bytes transmitted this session
         if(Preamblecount == TX_BYTES_OF_PREAMBLE)
         {
            Preamblecount = 0;      // reset counter for next TX session
            TX_STATEMACHINE = TX_SYNCBYTE;
                                    // transmit Sync Byte next
         }
      }
      else if (TX_STATEMACHINE == TX_SYNCBYTE)
      {
         SBUF0 = 0xF0;              // Sync byte = "11110000" but is
                                    // received LSB first.
         TX_STATEMACHINE = TX_DATA; // transmit data next
      }

      else if (TX_STATEMACHINE == TX_DATA)
      {
         // The first transmitted byte of data is the
         // LOCAL_SHUTDOWN_STATE.  At this time the shutdown state
         // machine is updated and, potentially, the RF transceiver
         // is powered down.
         if (TXcount == 0)
         {
            if (LOCAL_SHUTDOWN_STATE == BOTH_ENDPOINTS_QUIET)
            {
               // channel is idle, power down
               CR = 0;              // disable PCA clock
               RUN_ASLEEP = 1;      // set flag to run Asleep()
               TXcount = 0;         // reset counters
               RXcount = 0;
               SBUF0 = LOCAL_SHUTDOWN_STATE;
                                    // TX shudown state before
                                    // powering down
               REN0 = 0;            // disable UART reception
               LOCAL_SHUTDOWN_STATE = POWERING_DOWN;
            }
            else if (ADC_LOW == SHUTDOWN_TOLERANCE)
            {
               // the local voice stream can be considered quiet
               LOCAL_SHUTDOWN_STATE = ONE_ENDPOINT_QUIET;
               SBUF0 = LOCAL_SHUTDOWN_STATE;
            }

            else if(LOCAL_SHUTDOWN_STATE == POWERING_DOWN)
            {
               // LOCAL_SHUTDOWN_STATE = BOTH_ENDPOINTS_QUIET
```

```
                // has been transmitted, power down now
                EA = 0;              // disable interrupts
                C1000_POWERDOWN();

             }
             else
             {
                // local voice stream is not quiet
                LOCAL_SHUTDOWN_STATE = NO_ENDPOINTS_QUIET;
                SBUF0 = LOCAL_SHUTDOWN_STATE;
             }

          }

          else if (TXcount != DATA_BYTES_PER_TRANSMISSION)
          {
                SBUF0 = UTXFIFO_Pull();
                                  // output compressed voice stream

          }

          if (TXcount++ == DATA_BYTES_PER_TRANSMISSION)
          {
             TXcount = 0;             // Resets the data byte counter
             EIE1 |= 0x10;            // Enable PCA0 interrupts
             ES0 = 0;                 // Disable UART

             TX_IN_PROGRESS = 0;      // indicate that TX session has
                                      // completed
             CR = 1;                  // disable PCA clock
             TX_STATEMACHINE = TX_PREAMBLE_BAUDS;
             C1000_RX_MODE();         // switch transceiver to RX mode

          }
       }
       // This is the wake-up signal transmission state.  It only
       // runs once after the ASLEEP function resets the
       // TX_STATEMACHINE
       else if (TX_STATEMACHINE == TX_UNCALIBRATED)
       {
          // Transmit Preamble
          SBUF0 = 0x55;
          if (++TXcount == TX_WAKEUP_BYTES)
          {
             TXcount = 0;             // reset counter for next TX
             TX_STATEMACHINE = TX_PREAMBLE_BAUDS;
                                      // begin standard TX session
          }
       }

    }

    if (RI0 == 1)
    {
       RI0 = 0;
    }
 }
```

```
   else                                // System is running a Receive Session
   {

      if(RI0 == 1)
      {
         RI0 = 0;                      // acknowledge interrupt

         // The first received byte of data is the remote transceiver's
         // shutdown state.  This variable, and the local transceiver's
         // shutdown state, determine the next state of the shutdown
         // state machine.
         if(RXcount == 0)
         {
            REMOTE_SHUTDOWN_STATE = SBUF0;
            // if both endpoints are quiet, update the state machine
            if ((REMOTE_SHUTDOWN_STATE == ONE_ENDPOINT_QUIET)
               && (LOCAL_SHUTDOWN_STATE == ONE_ENDPOINT_QUIET))
            {
               LOCAL_SHUTDOWN_STATE = BOTH_ENDPOINTS_QUIET;
            }
            // if the state machine has already been set to
            // its BOTH_ENDPOINTS_QUIET state, then both endpoints
            // are ready to shutdown.
            else if(REMOTE_SHUTDOWN_STATE == BOTH_ENDPOINTS_QUIET)
            {
               EA = 0;                   // disable interrupts
               CR = 0;                   // disable PCA clock
               RUN_ASLEEP = 1;           // set flag to run asleep
               TXcount = 0;              // reset counters
               RXcount = 0;
               REN0 = 0;                 // disable UART reception
               C1000_POWERDOWN();        // power down transceiver
            }
         }

         else                            // byte received is not the
                                         // remote shutdown state, save
                                         // it as a data byte
         {
            URXFIFO_Push(SBUF0);
         }

         // if the last byte of the session has been received,
         // switch the system to Transmit Mode
         if (++RXcount == DATA_BYTES_PER_TRANSMISSION)
         {
            RXcount = 0;                 // reset counters
            REN0 = 0;                    // disable UART reception
            RX_STATEMACHINE = RX_SEARCH_FOR_SYNCBYTE;
            C1000_TX_MODE();             // switch RF transceiver to TX
         }
      }
   }
}

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

```
// C1000 Functions
//-------------------------------------------------------------------------


//-------------------------------------------------------------------------
// C1000 Functions
//-------------------------------------------------------------------------


//-------------------------------------------------------------------------
// SETREG
//-------------------------------------------------------------------------
// Writing registers in the CC1000 transceiver is accomplished using the
// SPI 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)
{
   PALE = 0;
   SPIF = 0;
   SPI0DAT = (address<<1) + 1;         // Shift out 6 bit address
                                       // plus Write mode bit
   while(!SPIF);
   SPIF = 0;
   PALE = 1;
   SPI0DAT = value;
   while(!SPIF);
}


//-------------------------------------------------------------------------
// 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 through SPI.
//
unsigned char READREG(unsigned char address)
{
   unsigned char ad;
   PALE = 0;
   SPIF = 0;
   ad = address;
   // shift 7 bit address to upper 7 bits of <address>,
   // and leave the lowest bit at value 0 to indicate a
   // register read
   SPI0DAT = ad << 1;

   while(!SPIF);
   SPIF = 0;
   PALE = 1;
   SPI0DAT = 0xFF;
   while(!SPIF);
```

```
   ad = SPI0DAT;
   return SPI0DAT;
}

//-----------------------------------------------------------------------------
// C1000_LOCK_FILTER
//-----------------------------------------------------------------------------
// This function sets the LOCK_AVG_IN bit in the MODEM1 register.
//
void C1000_LOCK_FILTER(void)
{
   SETREG(MODEM1, 0x7B);
}

//-----------------------------------------------------------------------------
// C1000_UNLOCK_FILTER
//-----------------------------------------------------------------------------
// This function clears the LOCK_AVG_IN bit in the MODEM1 register.
//
void C1000_UNLOCK_FILTER(void)
{
   SETREG(MODEM1, 0x6B);
}

//-----------------------------------------------------------------------------
// C1000_Init
//-----------------------------------------------------------------------------
// This function is called at startup and when the transceiver needs
// to be reset.
//
void C1000_Init(void)
{

   unsigned char cal_complete;
   // MAIN:
   // RXTX = 0, F_REG = 0, RX_PD = 1, TX_PD = 1, FS_PD = 1, CORE_PD = 0,
   // BIAS_PD = 1, RESET_N = 0
   SETREG(MAIN,0x3A);

   // SET RESET_N = 1
   SETREG(MAIN,0x3B);


   // 1 iteration = 9 clock cycles
   // 9 clock cycles = 367.3 ns
   // 5444 iterations = 2 ms

   // Wait for at least 2 ms
   WaitUS(2000);                        // wait 2 ms

   // writes 24-bit frequency value used for TX to 868 MHz.
   // Check CC1000 data sheet for equations used to find
   // 0x583000.
   SETREG(FREQ_2A, 0x58);
   SETREG(FREQ_1A, 0x30);
   SETREG(FREQ_0A, 0x00);
```

```
// writes 24-bit frequency value used for RX to 868 MHz.
// Check CC1000 data sheet for equations used to find
// 0x583313.
SETREG(FREQ_2B, 0x58);
SETREG(FREQ_1B, 0x33);
SETREG(FREQ_0B, 0x13);
// sets frequency separation between 0 value and 1 value
SETREG(FSEP1, 0x01);
SETREG(FSEP0, 0xAB);
// sets some internal current levels, and enables RSSI output to pin
SETREG(FRONT_END,0x02);
// sets the PLL reference divider to divide by 6
SETREG(PLL,0x30);
// sets continuous lock indicator to output on the CHP_OUT pin
SETREG(LOCK,0x10);

// sets threshold level for peak detector (not used in this design)
SETREG(MODEM2, 0xC1);
// sets the averaging filter to free-running and controlled by writes
// to bit 4 of this register.
// Sets averaging filter sensitivity to .6dB worst-case loss of sensitivity
SETREG(MODEM1, 0x6B);
// baud rate to 76.8, Transparent Asyn. UART, and crystal freq. to 14 MHz
SETREG(MODEM0, 0x58);
// sets capacitor array values for RX and TX
SETREG(MATCH, 0x10);
// disables dithering and data shaping
SETREG(FSCTRL, 0x01);
// sets prescaling to nominal values
SETREG(PRESCALER,0);
// sets charge pump current scaling factor, which determines the bandwidth
// of the PLL.
SETREG(TEST4,0x3F);

// Calibration Process
// RX Calibration
// set transceiver to RX mode
SETREG(MAIN,0x11);
SETREG(CURRENT,0x88);               // RX mode current
SETREG(CAL,0xA6);                   // begin calibration
cal_complete = 0;
do
{
   cal_complete = READREG(CAL);
} while (!(cal_complete & 0x08));   // spin until calibration is complete

SETREG(CAL,0x26);

// TX Calibration
SETREG(MAIN,0xE1);                  // set to TX mode
SETREG(CURRENT,0xF3);               // TX mode current
SETREG(PA_POW,0x00);                // set output power to 0 during
                                    // calibration
SETREG(CAL,0xA6);                   // begin calibration
cal_complete = 0;
do
{
```

```
      cal_complete = READREG(CAL);
   } while (!(cal_complete & 0x08));   // spin until finished
   SETREG(CAL,0x26);

}

//---------------------------------------------------------------------------
// C1000_POWERDOWN
//---------------------------------------------------------------------------
// This function powers down all components of the transceiver,
// including all RX, TX, Oscillator, and bias components.
//
void C1000_POWERDOWN(void)
{
   SETREG(MAIN,0xFE);                  // shutdown RF transceiver
   SETREG(PA_POW,0x00);                // set output power to 0
}


//---------------------------------------------------------------------------
// C1000_RSSI
//---------------------------------------------------------------------------
// C1000_RSSI functions almost like C1000_RX_MODE, except
// that it spins for 250 microseconds instead of using
// the ADC ISR as a counter.  This method is used because
// ADC interrupts are not enabled when C1000_RSSI is called.
//
void C1000_RSSI(void)
{
   SETREG(MAIN, 0x11);                 // set to RX mode
   SETREG(PA_POW,0x00);                // no output power needed
   SETREG(CURRENT,0x88);               // set RX current level
   // wait 250 microseconds
   WaitUS(250);
}

//---------------------------------------------------------------------------
// C1000_RX_MODE
//---------------------------------------------------------------------------
// C1000_RX_MODE sets the transceiver to RX mode and
// then initializes the 250 microsecond counter
// that gets incremented in the ADC ISR.
//
void C1000_RX_MODE(void)
{
   SETREG(MAIN, 0x11);                 // set to RX mode
   SETREG(PA_POW,0x00);                // no output power needed
   SETREG(CURRENT,0x88);               // set RX current level
   TRANSITION_IN_PROGRESS = 1;         // begin counter in ADC interrupt
   TO_TX_INDICATOR = 0;                // set flag to TX->RX
}

//---------------------------------------------------------------------------
// C1000_TX_MODE
//---------------------------------------------------------------------------
// C1000_TX_MODE sets the transceiver to TX mode and
// then initializes the 250 microsecond counter
```

 

```
// found in the ADC interrupt.
//
void C1000_TX_MODE(void)
{
   SETREG(PA_POW,0x00);                // set output power to 0
   SETREG(MAIN,0xE1);                  // set to TX mode
   SETREG(CURRENT,0xF3);               // TX current level
   TRANSITION_IN_PROGRESS = 1;         // begin counter in ADC interrupt
   TO_TX_INDICATOR = 1;                // set flag to RX->TX
}



//-------------------------------------------------------------------------
// 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.
//

//-------------------------------------------------------------------------
// Pull Functions
//-------------------------------------------------------------------------
// Pull functions save the value at element FIRST in the fifo array, then
// increment FIRST by one element.  This increment wraps around the
// array when FIRST equals the highest element of the FIFO before the
// increment.
// COUNT is decremented, and if the new value of COUNT = 0, the EMPTY
// flag for the FIFO is set.
//

unsigned char UTXFIFO_Pull(void)
{
   unsigned char output;
   // wrap FIFO pointer if necessary
   if (UTXFIFO.FIRST==RX_TX_FIFOSIZE-1) UTXFIFO.FIRST = 0;
   else (UTXFIFO.FIRST)++;

   // pull value from fifo
   output = UTXFIFO.FIFO[UTXFIFO.FIRST];

   // set empty indicator if necessary
   if (--(UTXFIFO.COUNT) == 0) UTXFIFO.EMPTY = 1;

   return output;
}

unsigned char URXFIFO_Pull(void)
{
   unsigned char output;
   if (URXFIFO.FIRST==RX_TX_FIFOSIZE-1) URXFIFO.FIRST = 0;
   else (URXFIFO.FIRST)++;

   output = URXFIFO.FIFO[URXFIFO.FIRST];
```

```
   if (--(URXFIFO.COUNT) == 0) URXFIFO.EMPTY = 1;

   return output;
}

unsigned short ADCRXFIFO_Pull(void)
{
   unsigned short output;
   if (ADCRXFIFO.FIRST==ADC_DAC_FIFOSIZE-1) ADCRXFIFO.FIRST = 0;
   else (ADCRXFIFO.FIRST)++;

   output = ADCRXFIFO.FIFO[ADCRXFIFO.FIRST];

   if (--(ADCRXFIFO.COUNT) == 0) ADCRXFIFO.EMPTY = 1;

   return output;
}

unsigned short DACTXFIFO_Pull(void)
{
   unsigned short output;
   if (DACTXFIFO.FIRST==ADC_DAC_FIFOSIZE-1) DACTXFIFO.FIRST = 0;
   else (DACTXFIFO.FIRST)++;

   output = DACTXFIFO.FIFO[DACTXFIFO.FIRST];

   if (--(DACTXFIFO.COUNT) == 0) DACTXFIFO.EMPTY = 1;

   return output;
}

//-------------------------------------------------------------------------
// Push Functions
//-------------------------------------------------------------------------
// Push functions always set EMPTY = 0.  They increment LAST,
// taking into account the possibility of wrap-around.  Then
// the value num is written into the LAST element of the FIFO.
// COUNT is incremented, and if COUNT's new value is greater
// than the size of the FIFO, then an overflow error has
// occurred.  The OF flag is thenset and will remain set on
// the FIFO until reset.
//

void UTXFIFO_Push(unsigned char num)
{
   UTXFIFO.EMPTY = 0;                  // FIFO is no longer empty
   (UTXFIFO.LAST)++;                   // increment, wrap if necessary
   if (UTXFIFO.LAST == RX_TX_FIFOSIZE)
   {
      UTXFIFO.LAST = 0;
   }

   UTXFIFO.FIFO[UTXFIFO.LAST]=num;     // push value to FIFO
   // test for overflows

   if(!(UTXFIFO.OF)) ++UTXFIFO.COUNT;
```

```
   if ( UTXFIFO.COUNT == RX_TX_FIFOSIZE) UTXFIFO.OF = 1;
}

void URXFIFO_Push(unsigned char num)
{
   URXFIFO.EMPTY = 0;
   (URXFIFO.LAST)++;
   if (URXFIFO.LAST == RX_TX_FIFOSIZE)
   {
      URXFIFO.LAST = 0;
   }

   URXFIFO.FIFO[URXFIFO.LAST]=num;

   if(!(URXFIFO.OF)) ++URXFIFO.COUNT;

   if ( URXFIFO.COUNT == RX_TX_FIFOSIZE) URXFIFO.OF = 1;
}


void ADCRXFIFO_Push(unsigned short num)
{
   ADCRXFIFO.EMPTY = 0;
   (ADCRXFIFO.LAST)++;
   if (ADCRXFIFO.LAST == ADC_DAC_FIFOSIZE)
   {
      ADCRXFIFO.LAST = 0;
   }

   ADCRXFIFO.FIFO[ADCRXFIFO.LAST]=num;


   if(!(ADCRXFIFO.OF)) ++ADCRXFIFO.COUNT;

   if ( ADCRXFIFO.COUNT == ADC_DAC_FIFOSIZE) ADCRXFIFO.OF = 1;

}

void DACTXFIFO_Push(unsigned short num)
{
   DACTXFIFO.EMPTY = 0;
   (DACTXFIFO.LAST)++;
   if (DACTXFIFO.LAST == ADC_DAC_FIFOSIZE)
   {
      DACTXFIFO.LAST = 0;
   }

   DACTXFIFO.FIFO[DACTXFIFO.LAST]=num;

   if(!(DACTXFIFO.OF)) ++DACTXFIFO.COUNT;

   if ( DACTXFIFO.COUNT == ADC_DAC_FIFOSIZE) DACTXFIFO.OF = 1;
}

//-------------------------------------------------------------------------
// CLEAR_FIFOS
//-------------------------------------------------------------------------
```

```
// CLEAR FIFOs resets all FIFOs to their microcontroller reset values.
//

void CLEAR_FIFOS(void)
{
   int x;
   // reset DAC FIFO
   DACTXFIFO.EMPTY = 1;
   DACTXFIFO.FIRST = 0;
   DACTXFIFO.LAST = 0;
   DACTXFIFO.COUNT = 0;
   DACTXFIFO.OF = 0;
   for(x = 0;x<ADC_DAC_FIFOSIZE; x++) DACTXFIFO.FIFO[x] = 0;

   // reset the UART RX FIFO
   URXFIFO.EMPTY = 1;
   URXFIFO.FIRST = 0;
   URXFIFO.LAST = 0;
   URXFIFO.COUNT = 0;
   URXFIFO.OF = 0;
   for(x = 0;x<RX_TX_FIFOSIZE;x++) URXFIFO.FIFO[x] = 0;

   // reset the ADC RX FIFO
   ADCRXFIFO.EMPTY = 1;
   ADCRXFIFO.FIRST = 0;
   ADCRXFIFO.LAST = 0;
   ADCRXFIFO.COUNT = 0;
   ADCRXFIFO.OF = 0;
   for(x = 0;x<ADC_DAC_FIFOSIZE; x++) ADCRXFIFO.FIFO[x] = 0;

   // reset the UART TX FIFO
   UTXFIFO.EMPTY = 1;
   UTXFIFO.FIRST = 0;
   UTXFIFO.LAST = 0;
   UTXFIFO.COUNT = 0;
   UTXFIFO.OF = 0;
   for(x = 0;x<RX_TX_FIFOSIZE;x++) UTXFIFO.FIFO[x] = 0;

}

//-------------------------------------------------------------------------
// Compression Algorithms
//-------------------------------------------------------------------------


//-------------------------------------------------------------------------
// RXalgorithm
//-------------------------------------------------------------------------
// RXalgorithm pulls 1 ADC sample and pushes 1 bit of a UART TX byte per
// iteration.
//

void RXalgorithm(void)
{
   static short RXI1;                  // integrator I1 for RXAlgorithm
   static short RXI2;                  // integrator I2 for RXAlgorithm
   static unsigned char output_byte;   // stores 8 compressed ADC samples
```

```
// used to update bits of output_byte
static unsigned char output_byte_mask;
// saves last three output bit values
static unsigned char last_three_outputs;

bit EAstate;
USHORT value;



// This flag is set in Asleep() and is used to re-initialize static
// variables the first time RXalgorithm() runs after the communications
// channel changes from idle to active
if(INIT_RX_ALGORITHM)
{
   RXI1 = 0;
   RXI2 = 0;
   output_byte = 0;
   output_byte_mask = 0x80;
   last_three_outputs = 1;
   INIT_RX_ALGORITHM = 0;
}

EAstate=EA;
EA = 0;                              // disable interrupts while compressing

value.S = ADCRXFIFO_Pull();          // pull the next ADC value

EA = EAstate;

last_three_outputs<<=1;              // only the lower 3 bytes are used,
                                     // LSB is set to zero with a shift

// This conditional performs a level conversion.  If the current ADC
// value is greater than the value in RXI1, a 1 is saved as the UTX
// byte and a 1 is saved as the last output (which is used in the
// next conditional).

// input - I1 = positive number
if (((short)value.S - (RXI1>>I1_TIMECONST)) > 0)
{
   output_byte |= output_byte_mask; // sets the output bit
                                     // samples are compressed so that
                                     // the LSB of output_byte is the most
                                     // recently compressed sample
   last_three_outputs |= 0x01;       // saves output by setting LSB
}

// Conditional checks for 3 consecutive 1's or 0's, which indicates that
// the sigma size needs to be increased in order for the compression
// algorithm to track the sample properly
if ( ((last_three_outputs & 0x07) == 0x07) ||
   ((last_three_outputs & 0x07) == 0x00))
{
      RXI2 += DELTAMAX;

}
```

```
   else
   {
      if (RXI2 != 0) // guards against RXI2 becoming negative
         RXI2 -= DELTAMAX;
   }

   // This conditional adjusts RXI1 using the new value of RXI2.
   if ((last_three_outputs & 0x01) == 1)
   {
      RXI1 += ((RXI2>>I2_TIMECONST) + DELTAMIN);
   }
   else
      RXI1 -= ((RXI2>>I2_TIMECONST) + DELTAMIN);

   // If output_byte_mask = 1, a TX UART byte is ready to be transmitted
   // because 8 left shifts on 0x80 have occurred.
   if (output_byte_mask == 1)
   {
      output_byte_mask = 0x80;          // reset mask for next sample

      EAstate=EA;
      EA = 0;                           // disable interrupts while compressing
      UTXFIFO_Push(output_byte);        // push byte of compressed data
                                        // to UART
      EA = EAstate;

      // This conditional begins a UART transmission if the UART is in
      // transmit mode, a TX is not already in progress, a transition
      // between RX and TX or TX and RX is not in progress, and there
      // are at least DATA_BYTES_PER_TRANSMISSION bytes in the URX FIFO.
      if ((UART_MODE == TX_MODE) && !(TX_IN_PROGRESS) &&
         !(TRANSITION_IN_PROGRESS) &&
         (UTXFIFO.COUNT >= DATA_BYTES_PER_TRANSMISSION))
      {
         TX_IN_PROGRESS = 1;            // set flag so this conditional
                                        // only executes once per TX session
         TI0 = 1;
      }

      output_byte = 0;                  // reset output_byte
   }
   else
   {
      output_byte_mask >>= 1;           // shift mask for next sample
   }

   EA = EAstate;
}

//-----------------------------------------------------------------------------
// TXalgorithm
//-----------------------------------------------------------------------------
// TXalgorithm pulls 1 received UART byte per 8 iterations, and
// pushes one DAC sample every 2 iterations.
//

void TXalgorithm(void)
```

```
{
   bit EAstate;
   static short TXI1;
   static unsigned short TXI2;
   static unsigned char top_of_FIFO;   // saves newest pull from RXFIFO
   // used to read from top_of_FIFO
   static unsigned char top_of_FIFO_mask;
   // saves last three input bit values
   static unsigned char last_three_inputs;



   if(INIT_TX_ALGORITHM)
   {
      top_of_FIFO = 0;
      top_of_FIFO_mask = 0;
      last_three_inputs = 1;
      TXI1 = 0;
      TXI2 = 0;
      accumulate = 0;
      DACstartupTimer = 0;
      INIT_TX_ALGORITHM = 0;
   }
   // This conditional checks to see if 8 bits have been
   // read from the top_of_FIFO variable.  If so, another
   // byte from the URXFIFO must be pulled before the
   // algorithm can continue.
   if (top_of_FIFO_mask == 0)
   {
      EAstate=EA;
      EA = 0;
      top_of_FIFO = URXFIFO_Pull();    // pull another compressed byte
      EA = EAstate;

      top_of_FIFO_mask = 0x80;         // reset mask
      }

      last_three_inputs<<=1;           // only last 3 bits are used,
                                       // LSB is cleared

      // This conditional reads a bit from the byte pulled
      // from URXFIFO and saves it in last_three_inputs.
      if((top_of_FIFO & top_of_FIFO_mask) != 0)
      {
         last_three_inputs |= 0x01;    // set LSB
      }
      top_of_FIFO_mask>>=1;            // shift mask for next sample

      // checks for 3 consecutive 0's or 1's and increases
      // TXI2 if that pattern occurs.
      if (((last_three_inputs & 0x07) == 0x07) ||
         ((last_three_inputs & 0x07) == 0x00))
      {
         TXI2+=DELTAMAX;
      }
      else
      {
```

```
         if (TXI2 != 0)
            TXI2-=DELTAMAX;
      }

      // Adjusts TXI1 with new value of TXI2
      if(last_three_inputs & 0x01)
      {
         TXI1 += ((TXI2>>I2_TIMECONST) + DELTAMIN);
      }
      else
      {
         TXI1 -= ((TXI2>>I2_TIMECONST) + DELTAMIN);
      }

      // This code averages every pair of samples to minimize
      // noise at the DAC output.

      if(!accumulate)
      {
         // divide TXI1 to account for I1's time constant
         accumulate = (TXI1>>I1_TIMECONST);
      }
      else
      {
         // add second value, dividing for I1's time constant
         accumulate = accumulate + (TXI1>>I1_TIMECONST);
         accumulate = accumulate >> 1; // divide sum by to average total

         EAstate=EA;
         EA = 0;
         DACTXFIFO_Push(accumulate);   // push result
         EA = EAstate;

         accumulate = 0;               // reset accumulator
      }

   EA = EAstate;
}
//-------------------------------------------------------------------------
// Power Consumption Functions
//-------------------------------------------------------------------------


//-------------------------------------------------------------------------
// ASLEEP
//-------------------------------------------------------------------------
//
// This function runs when audio levels on both transceivers are below
// AUDIO_THRESHOLD.  The ADC samples the audio input and the RSSI/IF
// pin of the RF transceiver.  If the audio input goes above
// AUDIO_THRESHOLD or the RSSI/IF pin's value drops below SIGNAL_THRESHOLD,
// a synchronization sequence begins.
//
void ASLEEP(void)
{
   unsigned int audio_value;
   unsigned int RSSI_value;
```

```
EA = 0;                             // disable global interrupts
CR = 0;

RUN_ASLEEP = 0;

CLEAR_FIFOS();                      // removes any old audio samples

ADC0CN &=~0x07;
ADC0CN |=0x00;                      // Set ADC to sample when AD0BUSY = 1

PALE = 1;                           // PALE should be high as default
C1000_Init();                       // initialize the transceiver
C1000_RSSI();                       // set it to output received
                                    // signal strength

UART_MODE = RX_MODE;                // set initial UART mode to RX


AD0EN = 1;
do{
   // check audio input (amplitude) voltage level
   AMX0P = 0x0B;                    // set to P1.3 (audio input)

   // wait for ADC MUX to settle
   WaitUS(60);                      // wait 60 microseconds
   AD0INT = 0;                      // clear flag
   AD0BUSY = 1;                     // begin capture
   while(!AD0INT);                  // spin until ADC capture is complete
   audio_value = ADC0;              // save value

   // check RSSI pin voltage level
   AMX0P = 0x09;                    // set to P1.1 (RSSI input)
   // wait for ADC MUX to settle
   WaitUS(60);                      // wait 60 microseconds
   AD0INT = 0;                      // clear flag
   AD0BUSY = 1;                     // begin capture
   while(!AD0INT);                  // spin until ADC capture is complete
   RSSI_value = ADC0;               // save value

}while((audio_value < AUDIO_THRESHOLD)
 && (RSSI_value > RSSI_THRESHOLD));

ADCRXFIFO_Push(audio_value);


RX_STATEMACHINE = RX_UNCALIBRATED;  // reset receive state machine
TX_STATEMACHINE = TX_UNCALIBRATED;  // reset transmit state machine

AMX0P = 0x0B;                       // set ADC to P1.3 (audio input)
ADC0CN &=~0x07;
ADC0CN |= 0x02;                     // set ADC to sample on Tmr 2 overflows


// RX Algorithm Variable Initialization
INIT_RX_ALGORITHM = 1;
```

```
   // TX Algorithm Variable Initialization
   INIT_TX_ALGORITHM = 1;

   // Shutdown state machine initialization
   REMOTE_SHUTDOWN_STATE = 0;
   LOCAL_SHUTDOWN_STATE = 0;
   ADC_LOW = 0;

   // Clear UART flags and interrupts
   RI0 = 0;
   TI0 = 0;
   TX_IN_PROGRESS = 0;
   RX_IN_PROGRESS = 0;

   C1000_UNLOCK_FILTER();              // unlocks RF transceiver's Averaging
                                       // filter, will be locked inside
                                       // PCA ISR during calibration sequence

   // if received signal strength indicates that the other endpoint is
   // transmitting, set to Receive Mode before exiting Asleep()
   if (RSSI_value <= RSSI_THRESHOLD)
   {
      SETREG(MAIN, 0x11);              // set to RX Mode
      SETREG(PA_POW,0x00);             // set output power to 0
      SETREG(CURRENT,0x88);            // set to RX current
      CR = 1;                          // enable PCA timer
      EIE1 |= 0x10;                    // Enable PCA0 interrupts
      UART_MODE = RX_MODE;             // set initial UART mode to RX

   }

   // if the local signal is loud enough to transmit, set the system
   // to TX Mode before exiting Asleep()
   else
   {

      SETREG(PA_POW,0x00);             // set output power to 0
      SETREG(MAIN,0xE1);               // set to TX Mode
      SETREG(CURRENT,0xF3);            // set TX current
      // wait 250 microseconds
      WaitUS(250);
      SETREG(PA_POW,0x80);
      //wait for 20 microseconds
      WaitUS(20);
      UART_MODE = TX_MODE;             // set initial UART mode to TX

   }

   AD0EN = 1;                          // enable ADC
   EA = 1;                             // enable global interrupts

}

void WaitUS(unsigned int count)
{
   unsigned int x;
```

 

```
   TR0 = 1;
   // each iteration of the for loop lasts approximately 1 us
   for(x = 0; x < count; x++)
   {
      TF0 = 0;
      while(!TF0);
   }
   TF0 = 0;
   TR0 = 0;
}
```

# Modified STARTUP.A51 file

```
$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
```