Voice Recorder Reference Design

1. Introduction

The C8051F411 offers a versatile, small (5 x 5 mm), highly integrated, low-power solution for voice applications. The 12-bit ADC and DAC allow for reasonable quality sound at a 8 kHz sampling rate, and the hardware Accumulation and Burst Mode features of the ADC provide for further improvements with small processing trade-offs. The Suspend mode operating feature allows the voice recorder to "sleep" while idle, saving power in a similar fashion to the traditional 8051 Stop mode, but still allows the recorder to wake and respond to the user without a hardware reset. This document describes the solution for a telephone-quality voice recorder using the C8051F411.

This document includes the following:

- A description of the system hardware and software
- Usage notes and customization considerations
- A schematic, bill of materials, and detailed layout diagrams
- The implementation of the software showing how to sample, compress, store, and play back a voice signal

The code accompanying this application note was originally written for C8051F41x devices. It can also be ported to other devices in the Silicon Labs microcontroller range.

1.1. Key Points

- Because of its small size, versatile peripherals, and low-power features, the 'F411 readily lends itself to battery-operated voice applications.
- The system uses a DPCM (Differential Pulse Code Modulation) compression algorithm for data storage to extend the total recording time.
- The recorder takes steps to minimize power usage while active and uses the "Suspend" feature of the 'F411 to reduce power consumption when idle.

2. Overview

The system, depicted in Figure 1, consists of a microphone and speaker, input and output filtering, the microcontroller, the external Flash memory storage, and push-buttons and LEDs for user interaction. This section describes each aspect of this system.

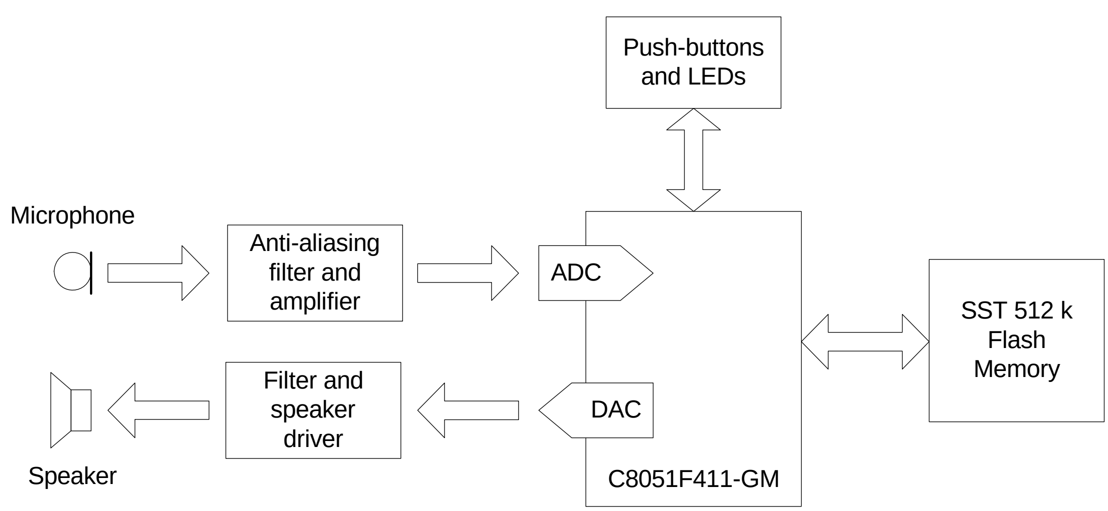


Figure 1. Voice Recorder Physical System Overview

2.1. Anti-Aliasing and Output Filtering

Both the input to the ADC and the output from the IDAC are filtered through low-pass, op-amp filters. The filters before the ADC help prevent aliasing, where sound waveforms with frequencies above half of the sampling frequency (the Nyquist frequency) "sound" much lower because the sampling is not adequately fast enough to properly reconstruct the original waveform. The filtering on the output eliminates the high-frequency content of the IDAC output and smoothes out the waveform before it is passed through the speaker driver.

2.2. Microcontroller (C8051F411)

The 'F411 samples the voice signal using the ADC, compresses the sample using DPCM (Differential Pulse Code Modulation), and sends the sample to the external Flash using the SPI. The microcontroller later retrieves the samples from the external Flash, decompresses them, and sends them to the speaker through the DAC. Figure 2 shows this dataflow path.

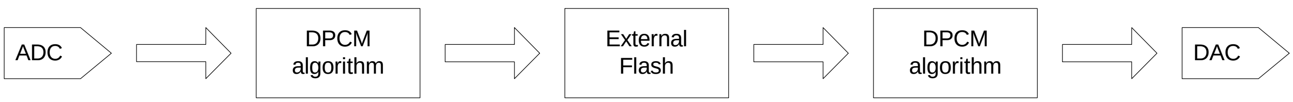


Figure 2. Voice Recorder Dataflow Path

To achieve telephone-quality sound, the microcontroller uses a sampling frequency of 8 kHz. This sampling rate can adequately reconstruct voice frequencies below 4 kHz, and still allots plenty of time between samples for the microcontroller to compress each sample and send it to external Flash memory. The microcontroller uses the hardware Burst Mode and Repeat Count features to automatically oversample and average the ADC samples, providing greater ADC accuracy. When the voice recorder is idle, the microcontroller shuts the system down using the Suspend mode feature, which minimizes the power consumption and allows the microcontroller to wake when the user presses either the Record/Play or Erase button without a hardware reset.

2.3. SPI Flash storage

With DPCM, the 'F411 compresses each 12-bit ADC sample into 6 bits, so four samples can be stored in every 3 bytes. With an 8 kHz sampling rate and 32 kB of internal Flash, the 'F411 can store approximately 5-6 seconds' worth of recordings by itself. An additional 512 kB serial (SPI) Flash memory is included on the board to extend the total storage to 1 minute 27 seconds.

2.4. Push-buttons and LEDs

The voice recorder uses simple LEDs and push-buttons to interact with the user. These LEDs indicate which function the recorder is using and whether the recorder is active or idle. In order to have the basic functionality of a voice recorder, the user needs to be able to record, play, erase, increase volume, and decrease volume. The two switches and potentiometer on the board provide these functions.

3. Hardware Description

This section includes the detailed descriptions of the hardware components for the voice recorder.

3.1. Audio Paths

The voice recorder includes an on-board microphone for mobility and ease of design, since different microphones require different biasing circuits. The microphone signal is sent through a rough low-pass filter and gain op-amp circuit to utilize the full range of the ADC, and then through a 3-pole Butterworth filter with a corner frequency of 4 kHz. The op-amps operate using 3.3 V rail-to-rail, but the ADC uses the programmed internal VREF of 2.2 V, so a voltage divider and DC-blocking capacitor provide the voltage translation from the filters to the ADC. A 5-pole Butterworth filter smoothes the output of the IDAC, which is then used by the speaker driver to output the waveform to the speaker jack.

3.2. Low-power Suspend

The 'F411 has a low power Suspend mode, during which the internal oscillator is completely dormant. An external transistor, controlled by one of the Port I/O, allows the 'F411 to disconnect the power to all of the external circuitry (op-amps, SST Flash, and speaker driver). Only the external voltage regulator and the 'F411 consume power while the system is idle.

3.3. MCU Peripherals

The voice recorder uses four of the 'F411 peripherals: Analog-Digital Converter (ADC), Serial Peripheral Interface (SPI), Current Digital-Analog Converter (IDAC), and Programmable Counter Array (PCA). The 12-bit ADC samples the voice input and provides hardware accumulation and oversampling. The SPI communicates with the external Flash memory in 4-wire master-mode to store compressed samples. The 12-bit DAC outputs the decompressed sample from memory to the speaker driver. Finally, the PCA in 8-bit PWM mode controls the brightness of the LEDs for user interaction.

3.4. Layout Considerations

This project does not include any extremely sensitive analog devices, so the main concerns during layout are size and cost. However, some care needs to be taken when routing peripherals and signals to the microcontroller Port I/O. For example, coupling can occur between the high-frequency SPI and the sensitive analog ADC and DAC peripherals, so these signals should be separated. Additionally, the DAC and VREF are only available on specific Port I/O pins. Furthermore, the SPI has higher precedence in the crossbar priority than the PCA when both are enabled, so the crossbar will route the SPI first. Careful planning of all I/O will ensure that all pins are routed correctly.

4. Software Description

The voice recorder microcontroller is responsible for checking the switches for user interaction, sampling the voice input, compressing and decompressing the samples, storing the samples in external Flash, outputting the samples to the output filters, and controlling the PWM of the LEDs. This section describes each of these functions and their implementation in detail.

4.1. Push-buttons

The 'F411 has two external interrupt pins that may be routed to any Port 0 pin. The voice recorder could successfully use these interrupts for the push-buttons, but this would limit the voice recorder design to only having two buttons. If the voice recorder is integrated into another design or if more features are added, more than two buttons would be needed.

Instead, the voice recorder uses a polling scheme, where a Timer checks the switches periodically, but the rate at which they are pressed is relatively slow compared to the other functions the voice recorder performs. The switches need to be checked quickly enough that they're adequately responsive to the user, but not so quickly that they constantly toggle before the switch is released. To account for both needs, the switches are checked every 15 ms and a 150 ms delay is added every time a switch is pressed.

4.2. Sampling Considerations

The sampling frequency for both the ADC and the DAC must be controlled as precisely as possible so that the output doesn't shift frequencies from the original input. The 'F411 uses the two 16-bit timers (Timer 2 and Timer 3) with auto reload to accomplish this. The ISR associated with each timer must be short enough that it doesn't interfere with the sampling frequency of 8 kHz, so each ISR execution must be less than 125 µs. Thus, all extraneous activities, such as the switch polling and LED control, must be executed in another, lower priority ISR that can be interrupted as necessary to meet the timing requirements of the sampling. Since these routines must communicate with one another, the 'F411 demo software uses global flags to indicate whether the sampling ISR should start, stop, or complete some other action.

4.3. DPCM Compression

The voice recorder uses a DPCM, or Differential Pulse Code Modulation, compression scheme, which is lossy because of the error incurred due to the nature of the algorithm. This scheme reduces each 12-bit sample down to a 6-bit code representing the difference between the actual sample and the predicted value of the sample. This predicted value can be calculated from sample averaging or some other complex algorithm, but, because voice samples tend to be highly correlated and the ISR needs to be short, the voice recorder simply uses the previous iteration's result as the predictor. The DPCM compression algorithm is shown in Figure 3.

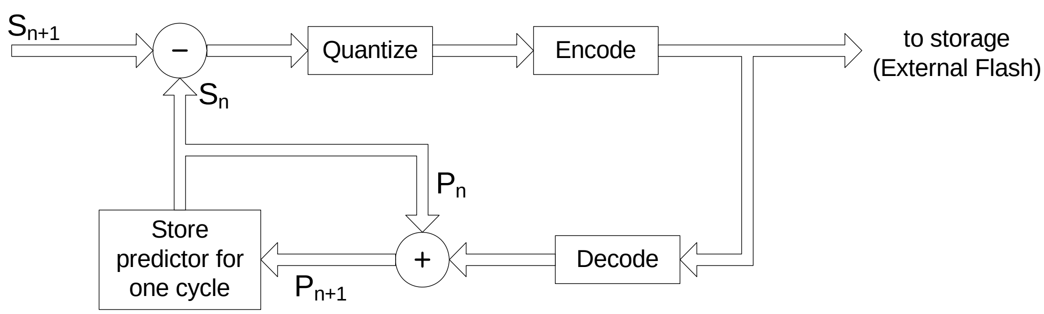


Figure 3. DPCM Compression Algorithm

The difference between the predictor and the sample is quantized, or separated into different "ranges" or "bins," and the 6-bit code represents the 64 possible ranges of values. This coded and quantized difference is then stored in memory. To calculate the new predicted value, the compression algorithm then decodes the difference and adds it to the current predicted value.

The decompression algorithm, shown in Figure 4, simply consists of matching the code in memory with the quantized difference and adding that difference to the predictor.

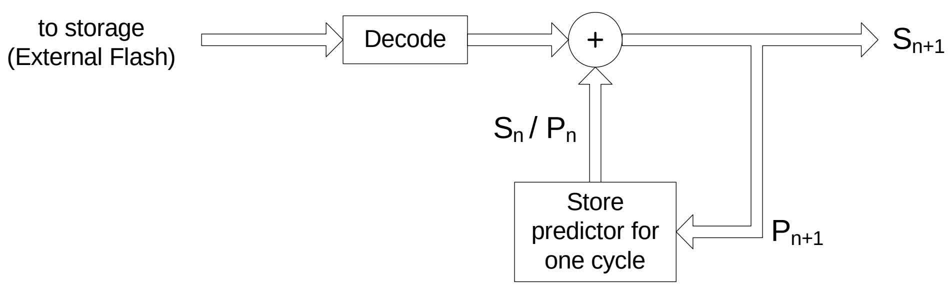


Figure 4. DPCM Decompression Algorithm

The initial predicted values from both the compression and decompression schemes should match so that the DPCM input and output are as similar as possible.

The following example uses a 4-bit (with 16 values) DPCM algorithm for an 8-bit ADC. If the first ADC sample is 0x89 (137 in unsigned decimal) and the initial predicted value is 0x80 (128 in unsigned decimal), the difference between the sample and the predicted value (sample - predicted value) is 9. When quantized, this difference of 9 falls in the "between 8 and 16" range, so the resulting DPCM code is 12. This code of 12 is then sent to storage.

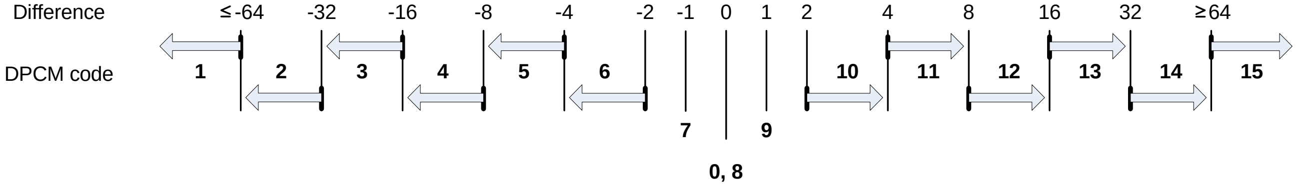


Figure 5. Example DPCM Code Quantization

To calculate the new predicted value for the second ADC sample, the DPCM code is reverted back to a difference value, which, for a DPCM code of 12, is a difference of 8. This difference is added back to the predicted value of 0x80 (128), which yields a new predicted value of 0x88 (136) for the next ADC sample. This is the same process that occurs during DPCM decompression, so that the same error is introduced on both sides. By using the decompression output (0x88) rather than the real sample (0x89), the compression and decompression schemes can include the same "DPCM" error and lessen the relative error. The predicted values in the compression scheme should match the predicted values and output of the decompression scheme exactly.

DPCM code	1	2	3	4	5	6	7	0, 8	9	10	11	12	13	14	15
Difference	-64	-32	-16	-8	-4	-2	-1	0	1	2	4	8	16	32	64

Figure 6. Example DPCM Decoding Scheme

If the second ADC value is 0x87 (135), this is compared to the predicted value of 0x88 (136), which yields a difference of –1 and a DPCM code of 7, which is sent to memory. The new predicted value is the current predicted value 0x88 (136) added to the decoded DPCM value (–1), or 0x87 (135). The predicted value of 0x87 (135) will then be compared to the next ADC sample, and so on.

With the compression quantization, any difference between 4 and 8 will yield a DPCM code of 11, a difference less than –64 will result in a DPCM code of 1, and a difference of 0 will result in a DPCM code of either 0 or 8. In the case of this DPCM algorithm implementation, the DPCM values are "snapped down" to the smaller difference value in the range bin for positive numbers and "snapped up" to the larger difference value in the range bin for negative numbers. For example, a difference of 15 uses the same code as a difference of 8. This is done to eliminate the chance of a DAC rollover upon play back.

The following table and graph continue the DPCM algorithm on a set of example ADC samples. Notice how the DPCM algorithm follows the real ADC samples and is oftentimes close to the ADC sample. The more bits the DPCM algorithm uses relative to the number of ADC bits, the more accurate the results will be, as it allows for more differences to be represented by a DPCM code.

Table 1. Example DPCM Algorithm Results

Algorithm Iteration	Example Real Samples	Difference (Sample - Predicted_Value)	DPCM Code	DPCM Decoded Difference	Compression Predicted Values (Predicted_Value + Decoded_Difference)	Decompression Output
0	—	—	—	—	128	128
1	137	9	12	8	136	136
2	135	–1	7	–1	135	135
3	138	3	10	2	137	137
4	140	3	10	2	139	139
5	132	–7	5	–4	135	135
6	120	–15	4	–8	127	127
7	100	–27	3	–16	111	111
8	107	–4	5	–4	107	107
9	111	4	11	4	111	111

Table 1. Example DPCM Algorithm Results

10	114	3	10	2	113	113
11	113	0	0	0	113	113
12	110	–3	6	–2	111	111
13	112	1	9	1	112	112

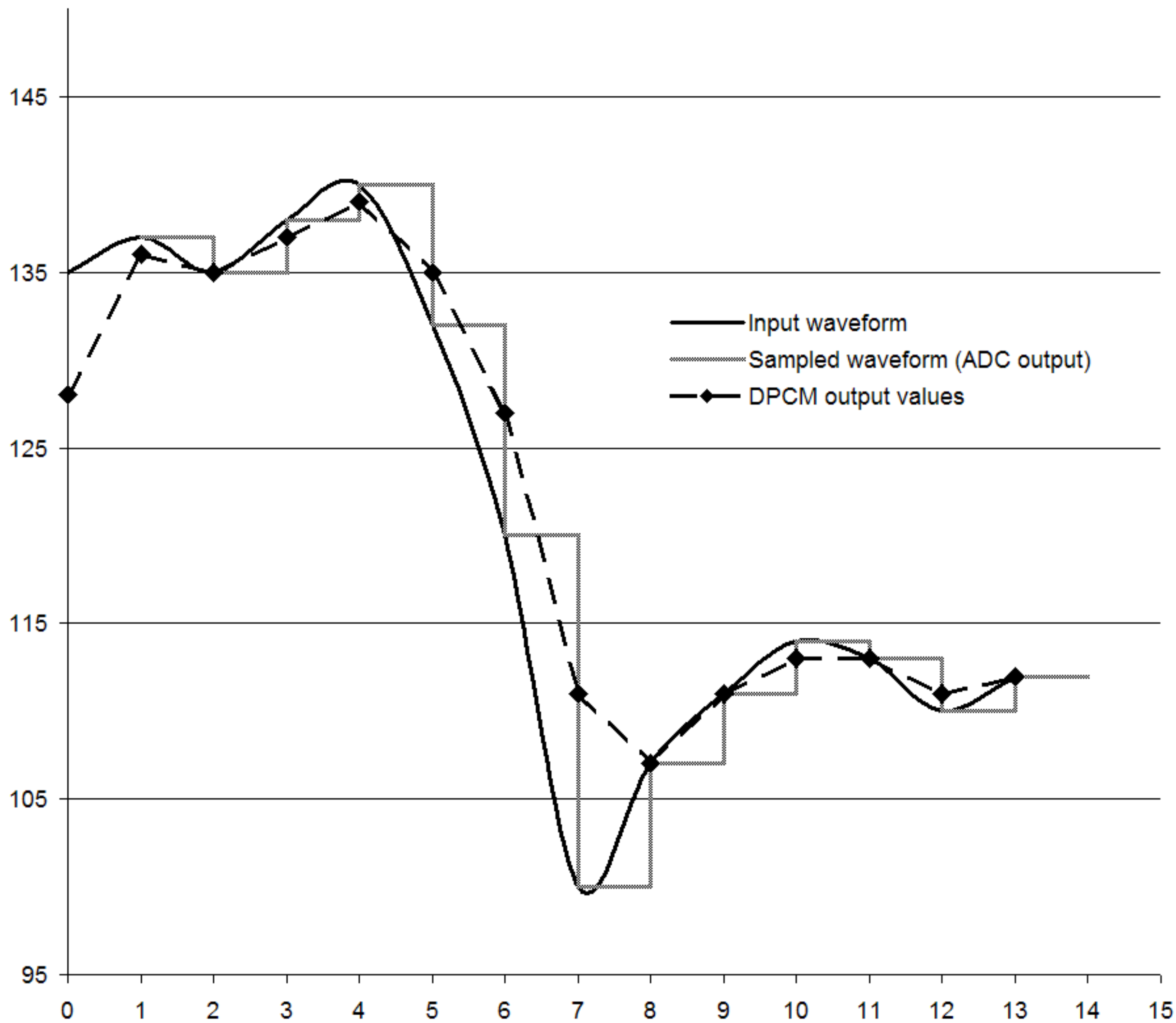


Figure 7. Example Input Waveform and DPCM Output

During the DPCM decompression process, the first DPCM code of 12 is retrieved from memory, decoded to a difference of 8, and added to the initial predicted value of 0x80 to yield the output sample of 0x88 (136). The second DPCM code of 7 is then retrieved from memory, decoded to a difference of –1, and added to the predicted value of 0x88 (136) to yield the second output sample of 0x87, and so on.

The Voice Recorder DPCM algorithm behaves the same as the above example, except the DPCM codes are 6 bits for a 12-bit ADC.

4.4. SPI Interface

The SST Flash memory used in the voice recorder has various commands that consist of a specified op-code and operands. The datasheet for the Flash memory describes each command and the timing involved in great detail. The pertinent commands for this application are the byte-program writes, status register read and enable, read, and chip erase. Upon power-up, the Flash must have the block protection bits in the status register cleared in order to write to the memory. Additionally, every write and erase operation must be preceded by a write enable command. Furthermore, the timing for the /NSS line is critical, as the Flash will abort any command it has not fully received before /NSS is disabled. The software controlling the SPI uses both the TXBMT and SPIF flags to verify that all necessary transactions between the Flash and the 'F411 occur before disabling /NSS.

4.5. Suspend Mode

The low power 'F411 includes a special Suspend mode feature that turns off the internal oscillator until a waking event occurs. The 'F411 controls a transistor that can shut off the power output from the external voltage regulator to conserve power in all other devices, as well. The lone red LED (PWR_LED) indicates whether the 'F411 has shut off power to all other devices. For the voice recorder, a port match event terminates the Suspend mode. This event can occur when the user presses one of the switches and the Port I/O, after a check with the port mask registers, mismatch the value in the port match registers. Once the 'F411 awakens, the peripherals and external SST Flash must be reinitialized after the memory's initial hold time of 10 µs. Normal voice recorder operation can then resume.

4.6. Detailed Descriptions

Figure 8 depicts the main software routine. The voice recorder completes all system initializations and begins polling on the buttons. If a specified time period passes without any interaction, the system automatically switches to Suspend mode. When the user presses a switch, the recorder acknowledges the interaction by brightening an LED and the associated action takes place. When the action completes, the LED dims.

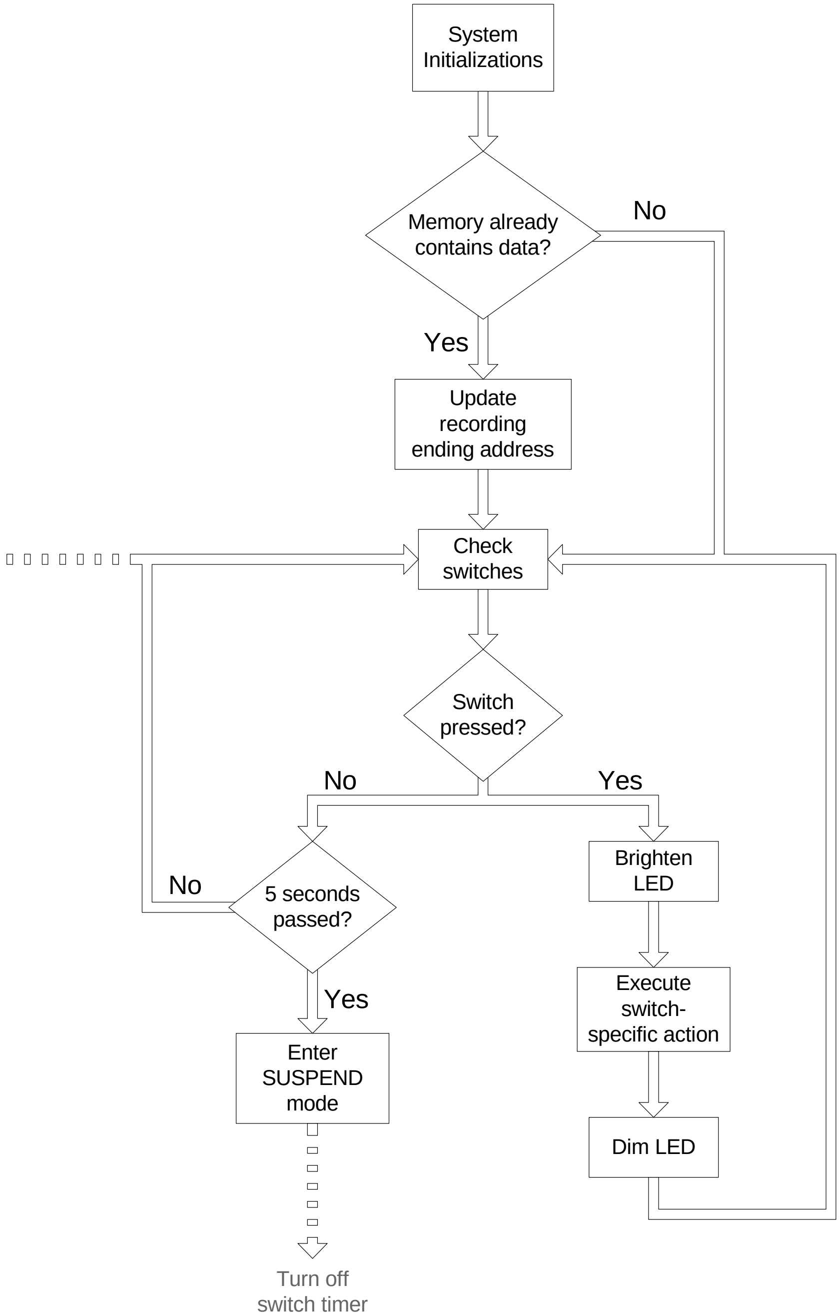


Figure 8. Initialization and Switch Polling Routine

If the user presses the Record/Play push-button, the 'F411 first determines if the recorder should start recording (button held down) or playing (button pressed and released).

Once recording, the 'F411 checks if the memory was erased and resets the beginning address of the new recording appropriately. The recorder then checks if the recording must end because the memory is full or because the user released the Record/Play push-button. If neither of these two conditions occur, then the recorder reads the sample from the ADC, executes the DPCM algorithm, and packs the 6-bit code into the byte to be sent to storage. If that byte now contains 8 bits of data, it's sent to memory; otherwise, the routine stores the byte until the next sampling time. This ISR is executed every 125 µs, or at a frequency of 8 kHz.

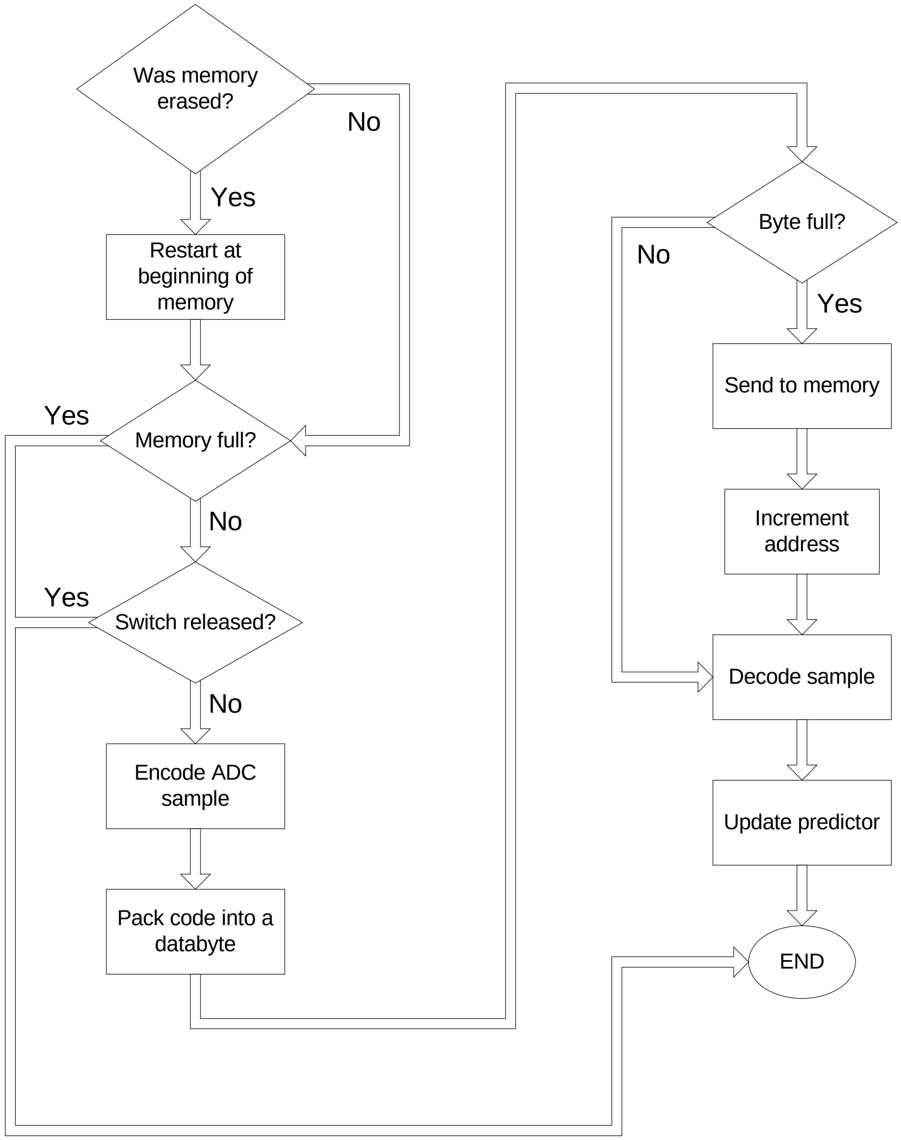


Figure 9. Record Function and Sampling Interrupt Service Routine

When the user plays the recording, as shown in Figure 10, the ISR first checks whether or not the playback should end because the user pressed the push-button again or the ISR reached the last memory address of the recording. If these checks prove to be false, the recorder unpacks a 6-bit code from the byte from memory, decodes it using the DPCM algorithm, updates the DAC output, and checks if a new byte should be fetched from memory. This ISR is also run every 125 µs.

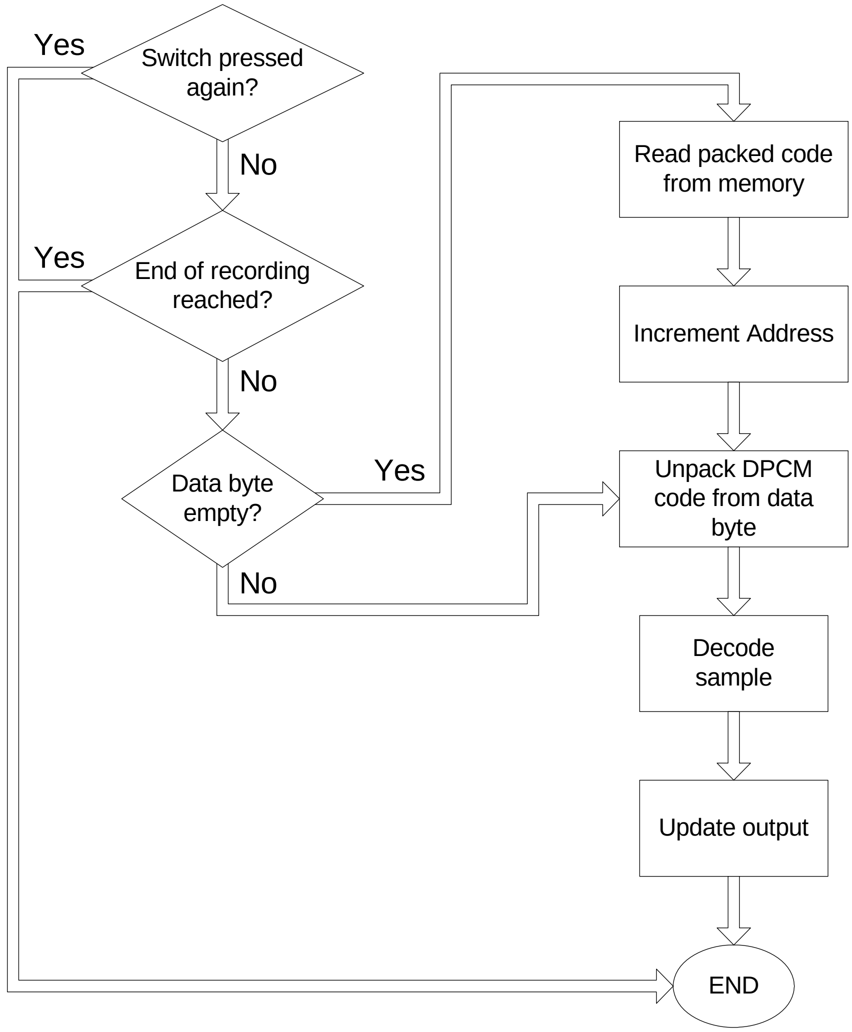


Figure 10. Playback Interrupt Service Routine

The other functions available to the recorder include recording erase and volume control. The erase uses the chip-erase command available to the SST and is controlled in the switch-polling function, as it is a lower priority than record and playback. The volume control is accomplished through a potentiometer that changes the voltage divider at the IDAC output.

Suspend mode, the final feature of the recorder, occurs after a set period of idle time. As shown in Figure 11, the recorder first turns off the timer controlling the push-button polling, for the functions associated with the switches should not be active until the 'F411 completes all re-initializations after exiting Suspend mode. The recorder also turns off all the peripherals and the power to the op-amps, external Flash memory, and speaker driver to conserve power. The C8051F411's internal oscillator halts while in Suspend mode until a waking event occurs. In this application, the relevant waking event is the port match, where the device will return to normal operation if one of the switches is pressed. After the port match event, the recorder reinitializes all peripherals, waits the power-up hold time specified for the SST Flash, initializes the SST Flash, and begins to poll the switches.

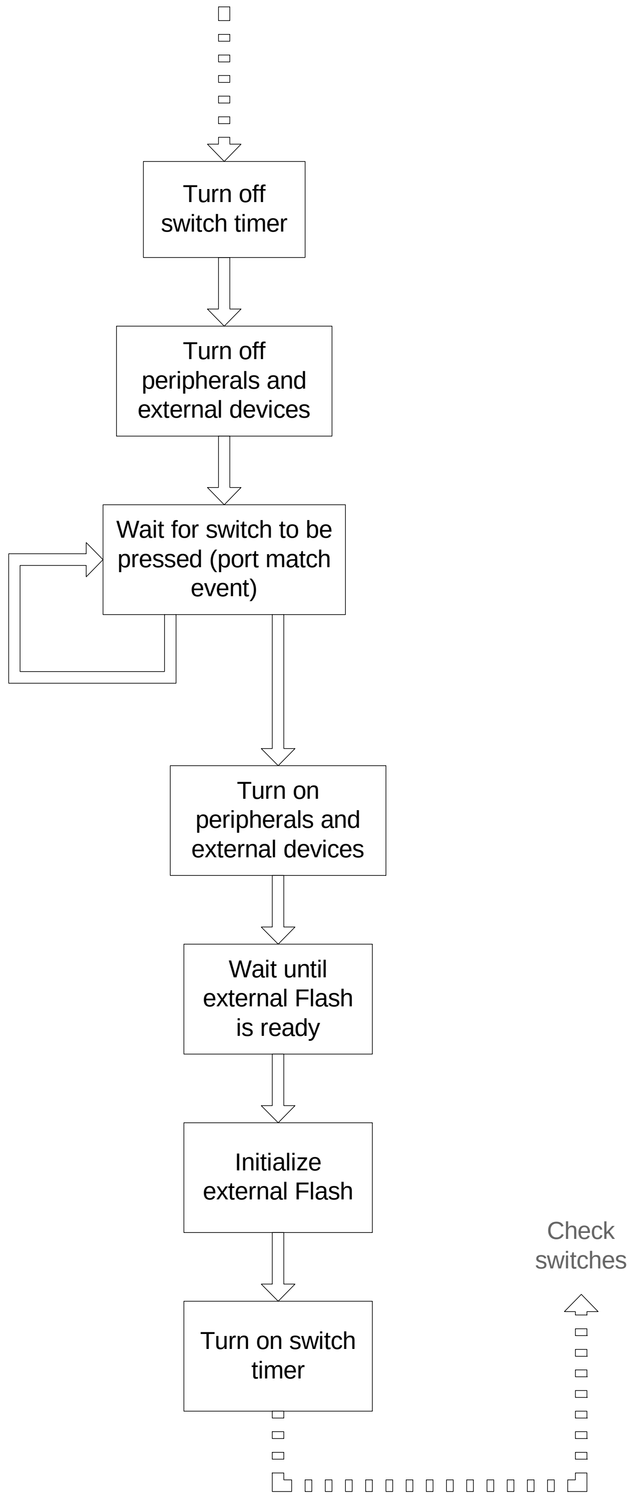


Figure 11. Suspend Mode

5. Usage Notes

The voice recorder may be powered from either a 9 V battery or a 9 V DC Power Adapter. Protection diodes will stop any mishaps from occurring if both are in place at the same time. Each button activates and deactivates the corresponding function, and the LED will indicate whether the function is active. Certain functions can only be used during certain times; for example, the erase operation can only occur if no other action is taking place. Recording multiple times without erasing appends the new recording to the end of the first. The play function always begins at the very first recording.

6. Design Customization

All of the polling, sampling periods, and DPCM quantization values are constants declared at the beginning of the software files. Any of the chosen values in this project may be changed, but take caution to observe that all requirements are met by each change. For example, changing the system clock divider may result in a sampling ISR that doesn't meet the 125 µs timing requirement, or changing the low-pass filter corner frequency may cause unwanted aliasing.

7. References

- Chipcon. "AN026: Wireless Audio using CC1010." Rev 1.0, 9/8/2004.
- Silicon Laboratories. "AN147: Wireless Digital Full-Duplex Voice Transceiver." Rev 1.1.
- SST. SST25VF040 Data Sheet. 6/04.

Appendix A—Schematic

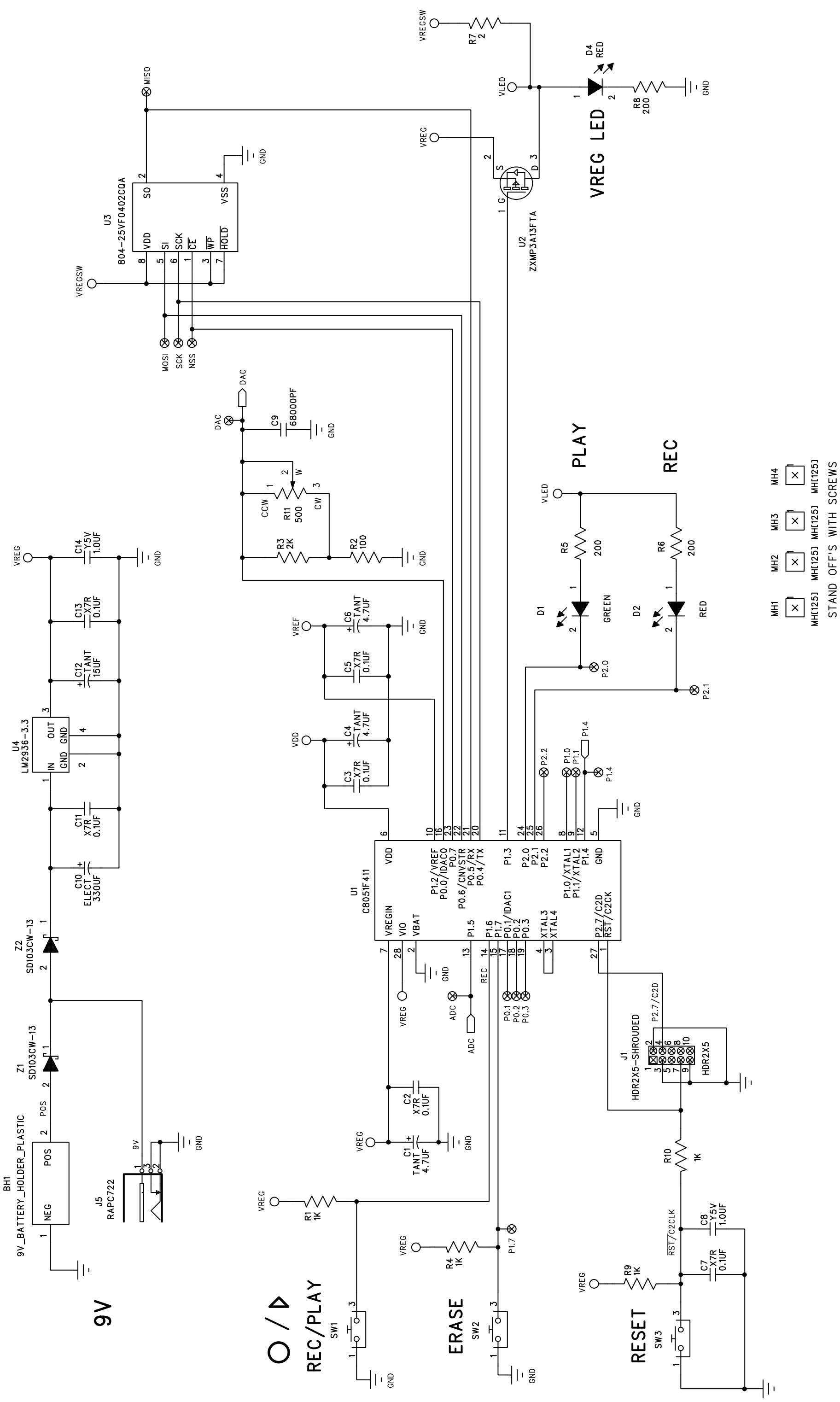


Figure 12. Voice Recorder Reference Design Schematic (Page 1 of 2)

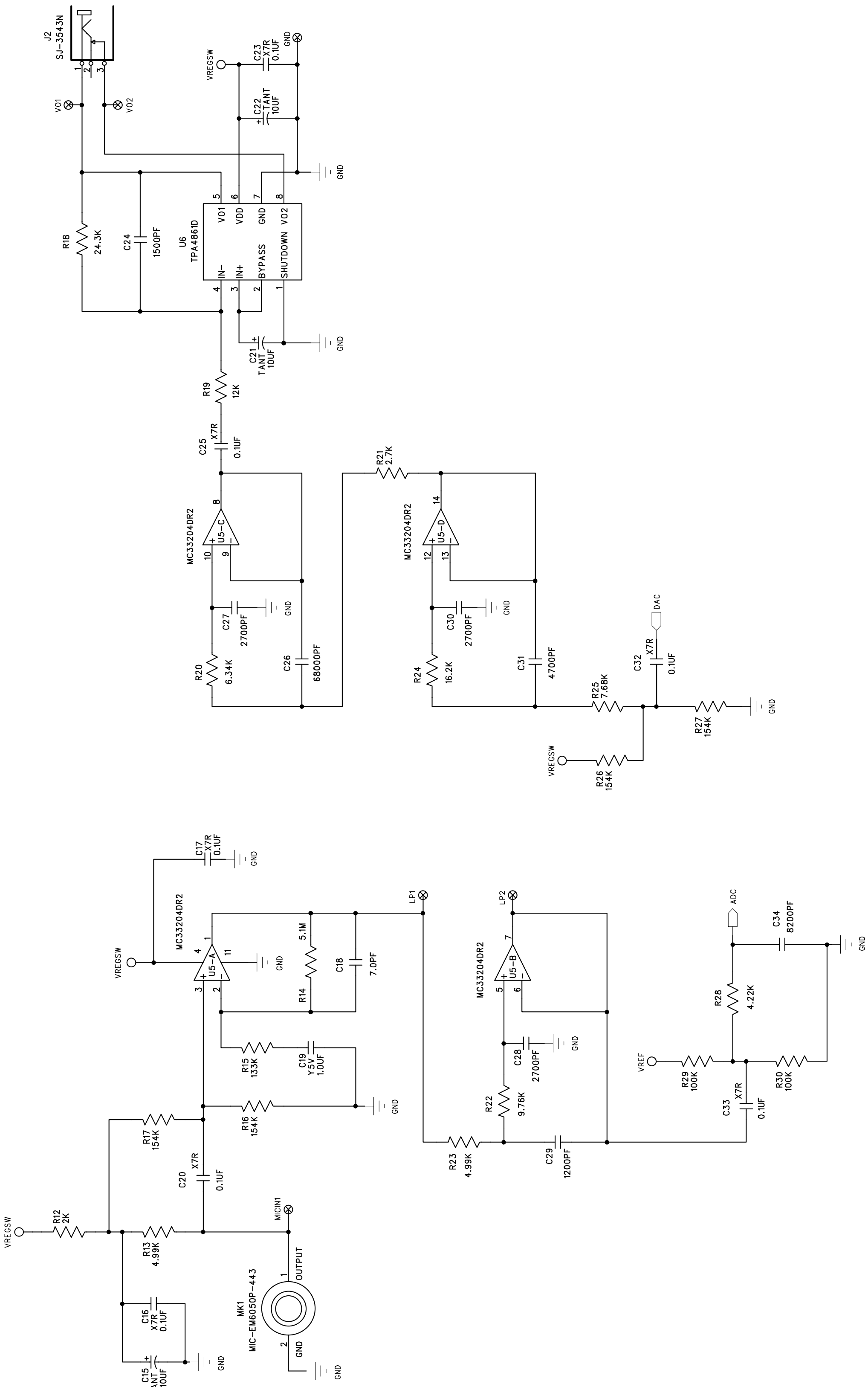


Figure 13. Voice Recorder Reference Design Schematic (Page 2 of 2)

Appendix B—Bill Of Materials

Table 2. Bill of Materials

Qty	Board Reference	Value/Part Number	Description	Manufacturer
13	C2, C3, C5, C7, C11, C13, C16, C17, C20, C23, C25, C32, C33	0.1 µF	0805	any
3	C8, C14, C19	1.0 µF	0805	any
1	C29	1200 pF	0805	any
1	C24	1500 pF	0805	any
3	C27, C28, C30	2700 pF	0805	any
1	C31	4700 pF	0805	any
2	C9, C26	68000 pF	0805	any
1	C18	7 pF	0805	any
1	C34	8200 pF	0805	any
1	C10	ECA-1EHG331	330 µF Elect.	Panasonic - ECG
3	C1, C4, C6	ECS-T0JY475R	4.7 µF Tant.	Panasonic - ECG
1	C12	T491C156K010AS	15 µF Tant.	Kemet
3	C15, C21, C22	T491B106K016AS	10 µF Tant.	Kemet
1	R2	100 Ω	0805	any
2	R29, R30	100 kΩ	0805	any
1	R19	12 kΩ	0805	any
1	R15	133 kΩ	0805	any
4	R16, R17, R26, R27	154 kΩ	0805	any
1	R24	16.2 kΩ	0805	any
4	R1, R4, R9, R10	1 kΩ	0805	any
1	R7	2 Ω	0805	any
1	R21	2.7 kΩ	0805	any
3	R5, R6, R8	200 Ω	0805	any
1	R18	24.3 kΩ	0805	any
2	R3, R12	2 kΩ	0805	any
1	R28	4.22 kΩ	0805	any
2	R13, R23	4.99 kΩ	0805	any
1	R14	5.1 MΩ	0805	any

Table 2. Bill of Materials (Continued)

Qty	Board Reference	Value/Part Number	Description	Manufacturer
1	R20	6.34 kΩ	0805	any
1	R25	7.68 kΩ	0805	any
1	R22	9.76 kΩ	0805	any
1	R11	3352T-1-501	500 Ω potentiometer	Bourns Inc.
1	BH1	1295	9 V battery holder	Keystone Electronics
1	J1	103308-1	2 x 5 shrouded	AMP/Tyco Electronics
1	J2	SJ-3543N	audio jack	CUI Inc.
1	J5	RAPC722	power jack	Switchcraft Inc.
2	Z1, Z2	SD103CW-13	Schottky diode	Diodes Inc.
1	D1	SML-LXT0805GW-TR	green LED 0805	Lumex Opto/Components Inc.
2	D2, D4	SML-LXT0805IW-TR	red LED 0805	Lumex Opto/Components Inc.
4	MH1, MH2, MH3, MH4	1902E	stand-offs	Keystone Electronics
1	MK1	EM6050P-443	Microphone	Horn Industrial Co LTD
3	SW1, SW2, SW3	EVQ-PAD04M	switches	Panasonic - ECG
1	U1	C8051F411-GM	QFN-28	Silicon Laboratories
1	U2	ZXMP3A13FTA	transistor SOT23	Zetex Inc.
1	U3	SST25VF040B-50-4C-QAF	512 kB Flash	SST
1	U4	LM2936IMP-3.3	Vol. reg SOT223	National Semiconductor
1	U5	MC33204DR2	op-amp SOIC14	ON Semiconductor
1	U6	TPA4861D	speaker driver SOIC8	Texas Instruments

Appendix C—Layout

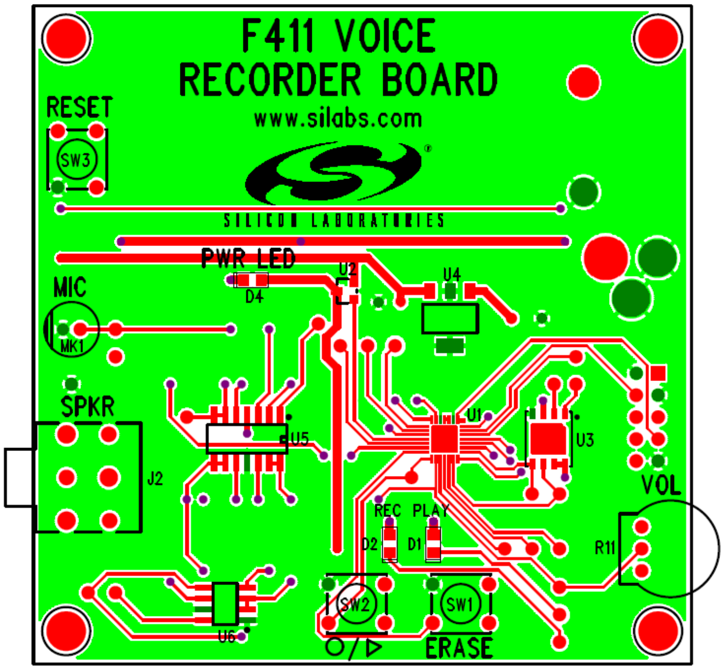


Figure 14. Top Layout and Silkscreen

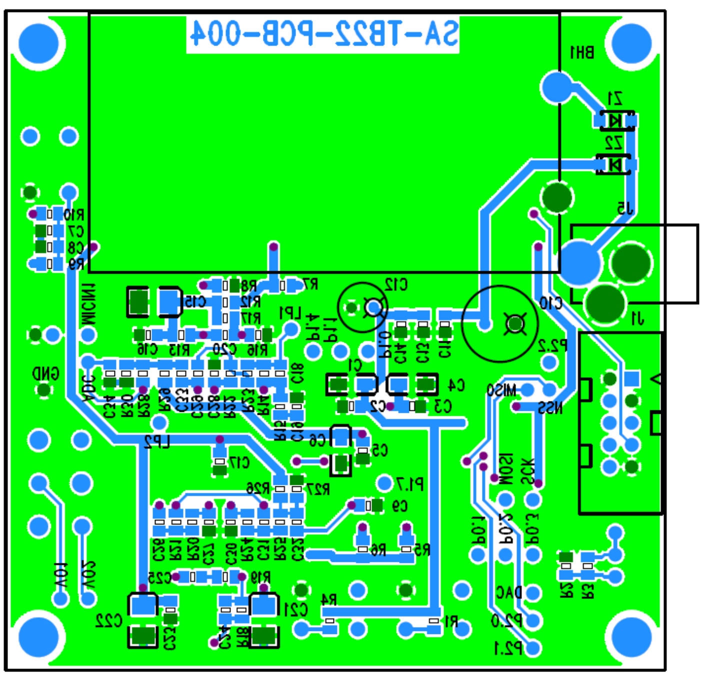


Figure 15. Bottom Layout and Silkscreen

APPENDIX D—SOFTWARE SOURCE CODE

Startup Code (Modified STARTUP.A51)

```
$NOMOD51
;------------------------------------------------------------------------------
;  This file is part of the C51 Compiler package
;  Copyright (c) 1988-2001 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.
;
;------------------------------------------------------------------------------

; Standard SFR Symbols required in XBANKING.A51
ACC      DATA     0E0H
B        DATA     0F0H
SP       DATA     81H
DPL      DATA     82H
DPH      DATA     83H
PCA0MD   DATA     0D9H

                          NAME     ?C_STARTUP


?C_C51STARTUP    SEGMENT   CODE
?STACK                    SEGMENT   IDATA

                          RSEG     ?STACK
                          DS       1

                          EXTRN CODE (?C_START)
                          PUBLIC   ?C_STARTUP

                          CSEG     AT0
?C_STARTUP:      LJMP     STARTUP1

                          RSEG     ?C_C51STARTUP

STARTUP1:

                          ANL   PCA0MD, #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      P2,#PPAGE
ENDIF
```

```
IF PDATALEN <> 0
                        MOV     R0,#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
```

Main Voice Recorder Program

```
//-----------------------------------------------------------------------------
// F411_VR.c
//-----------------------------------------------------------------------------
// Copyright 2006 Silicon Laboratories, Inc.
// http://www.silabs.com
//
// Program Description:
//
// This program uses the DPCM functions to encode voice samples and saves them
// to flash memory.  This program also interfaces with a speaker or headphones
// to play back the recorded voice.
//
// How To Use:    See Readme.txt
//
// FID:            41X000005
// Target:         C8051F411
// Tool chain:     Keil C51 7.50 / Keil EVAL C51
//                 Silicon Laboratories IDE version 2.6
// Project Name:   F411_VR
//
// Release 1.4
//    -All changes by TP
//    -06 Feb 2009
//    -Added a delay when restarting and waking up to allow the SST Flash
//     VDD to ramp up fully before initializing the Flash.
//    -Lengthened the delay before re-enabling the VDD Monitor
//
// Release 1.3
//    -All changes by TP
//    -02 Feb 2006
//    -minor changes in comments
//
// Release 1.2
//    -All changes by TP
//    -21 Nov 2005
//    -Revised for a 2-button version of the board with
//       volume wheel.
//
// Release 1.1
//    -All changes by TP
//    -16 Aug 2004
//    -project version updated, no changes to this file
//
// Release 1.0
//    -Initial Revision (TP)
//    -15 AUG 2004
//


//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------
#include <c8051f410.h>                 // SFR declarations
#include "F411_VR_DPCM.h"              // contains DPCM functions
#include "F411_VR_SSTFlash.h"          // contains functions to write to the
                                       // serial SST external 512 kb Flash
#include "F411_VR_LED.h"               // contains functions to control the
                                       // intensity of the LEDs

//-----------------------------------------------------------------------------
// 16-bit SFR Definitions for 'F411
//-----------------------------------------------------------------------------
```

```
// SFR16 Defintions (Timers, ADC, and DAC)
sfr16 TMR2RL = 0xCA;                // Timer 2 Reload address
sfr16 TMR2 = 0xCC;                  // Timer 2 Counter address
sfr16 TMR3RL = 0x92;                // Timer 3 Reload address
sfr16 TMR3 = 0x94;                  // Timer 3 Counter address
sfr16 ADC0DAT = 0xBD;               // ADC 16-bit address
sfr16 IDA0DAT = 0x96;               // IDAC 16-bit address

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

// General Constants
#define SYSCLK 6125000              // system clock in Hz (24.5 MHz / 4)

// T1 runs at SYSCLK / 48
#define POLLING 1992                // poll the switches at 64 Hz
#define PRCHANGE 20000              // wait approx. 150ms after a button is
                                    // pressed to "debounce" the switches

#define SAMP_FREQ 764               // use 8kHz sampling for both the ADC
                                    // and DAC

#define MAX_MEM_ADDR 0x0007FFFF     // 512K = 2^19 address bits
#define NEAR_END_ADDR 0x00065F55    // for every 4 samples, 3 bytes are
                                    // written to memory, so 10.67 kHz
                                    // ((8 kHz * 4)/3) writing time = 106666
                                    // addresses every 10 seconds, so give
                                    // about 10 seconds of warning

#define mid_range 2048              // middle value of 12-bit ADC and DAC

// System States
#define IDLE 0x00                   // indicates no current action
#define RECORDING 0x01              // indicates the device is recording
#define PLAYING 0x02                // indicates the device is playing
#define END_MEM 0x04                // flag used if the end of memory is
                                    // reached
#define ERASED 0x08                 // flag used if memory is erased

// Port Pin Definitions
sbit REC_PLAY = P1^7;
sbit ERASE = P1^6;
sbit TRANS = P1^3;
sbit LED0 = P2^1;
sbit LED1 = P2^0;

sbit SCK = P0^4;
sbit MISO = P0^5;
sbit MOSI = P0^6;
sbit NSS = P0^7;

//-----------------------------------------------------------------------------
// Global VARIABLES
//-----------------------------------------------------------------------------

unsigned char system_state = IDLE;  // start in idle mode
                                    // bit 3 of the system_state indicates
                                    // if the end of memory has been
                                    // reached
                                    // bit 4 of the system_state indicates
                                    // if the memory has been erased since
                                    // the last action (1 = erased)
```

```
// Ending address of the recording in memory
unsigned long rec_end_addr = 0x00000000;

// flags to communicate between the T1 ISR and the ADC/DAC ISRs for various
// termination events
bit ADC_STOP_FLAG = 0;
bit MEM_END_NEAR_FLAG = 0;
bit MEM_END_FLAG = 0;
bit REC_END_FLAG = 0;
bit DAC_STOP_FLAG = 0;
bit ENTER_SUSPEND = 0;

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
// System and peripheral initialization functions
void System_Init (void);
void VDDMon_Init (void);
void Port_Init (void);
void ADC0_Init (void);
void DAC0_Init (void);
void PCA_Init (void);
void SPI0_Init (void);
void RTC_Init (void);
void Timer0_Init (int period);
void Timer1_Init (int period);
void Timer2_Init (int period);
void Timer3_Init (int period);
void Recording_Search (void);

// Interrupt service routines
void Timer0_ISR (void);                // LED updates
void Timer1_ISR (void);                // Switch polling
void ADC0_ISR (void);                  // Recording
void Timer3_ISR (void);                // Playback

//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------
void main (void)
{
   unsigned char i;

   // Watchdog timer disabled in VR_STARTUP.A51

   System_Init ();                     // Initialize system clock
   Port_Init ();                       // Initialize crossbar and GPIO
   ADC0_Init ();                       // Initialize ADC0 (microphone)
   DAC0_Init ();                       // Initialize DAC0 (speaker)
   PCA_Init ();                        // Initialize the PCA for 8-bit PWM
                                       // in modules 0, 1, and 2
   SPI0_Init();                        // Initialize the interface to the flash

   RTC_Init ();                        // Stop the RTC from causing a wake-up
                                       // from suspend

   Timer0_Init (LED_PWM);              // Initialize timer 0 to provide a
                                       // 76 Hz interrupt rate for the LEDs

   Timer1_Init (POLLING);              // Initialize timer 1 to provide a
                                       // 64 Hz interrupt rate for the switches
```

```
   Timer2_Init (SAMP_FREQ);            // Initialize the timer to provide an
                                       // 8KHz interrupt rate for sampling

   Timer3_Init (SAMP_FREQ);            // Initialize the timer to provide an
                                       // 8KHz interrupt rate for sampling

   EA = 1;                             // enable global interrupts

   // wait for the VDD of the Flash to ramp up + 10us until the Flash
   // is ready to receive writes and reads
   for (i = 0; i < 255; i++);

   SSTFlash_Init ();                   // disable the write protection in the
                                       // external SST flash memory



   Recording_Search ();                // search for a recording already
                                       // present in memory

   TR1 = 1;                            // start polling the switches

   // loop forever
   while (1)
   {
      if (ENTER_SUSPEND == 1)          // check if no interaction has occurred
      {                                // for some time
         // disable everything to save the most power and set everything to a
         // dormant state
         ENTER_SUSPEND = 0;

         TR1 = 0;                      // stop polling the switches

         EA = 0;                       // disable all interrupts

         XBR1 = 0x40;                  // disable the PCA
         XBR0 = 0x00;                  // disable the SPI

         SCK = 0;                      // drive the SPI pins low so the
         MISO = 0;                     // external Flash won't attempt to draw
         MOSI = 0;                     // current while unpowered

         IDA0CN &= ~0x80;              // disable DAC0
         REF0CN &= ~0x01;              // disable VREF

         LED0 = 1;                     // turn the LEDs off
         LED1 = 1;

         TRANS = 1;                    // turn off the external circuitry

         RSTSRC = 0x00;                // disable missing clock detector
         VDM0CN &= ~0x80;              // disable the VDD Monitor

         OSCICN |= 0x20;               // enter suspend mode and wait
                                       // until a port match event occurs

         // re-enable and reinitialize the system
         VDDMon_Init ();

         TRANS = 0;                    // turn on the external circuitry

         REF0CN |= 0x01;               // re-enable VREF
         IDA0CN |= 0x80;               // re-enable DAC0
```

```
        XBR0 = 0x02;                    // re-enable SPI
        XBR1 = 0x42;                    // re-enable PCA0_0 and PCA0_1

        // wait for the VDD of the Flash to ramp up + 10us until the Flash
        // is ready to receive writes and reads
        for (i = 0; i < 255; i++);

        SSTFlash_Init ();               // re-initialize the SST flash

        EA = 1;                         // enable global interrupts

        // wait until the button that woke the system is released
        while ((REC_PLAY == 0) || (ERASE == 0));

        TR1 = 1;                        // begin polling the buttons again
      }
   }
}

/////////////////////////// INITIALIZATION ROUTINES ///////////////////////////

//-----------------------------------------------------------------------------
// System_Init
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// This routine initializes the system clock to use the internal 24.5MHz / 4
// oscillator as its clock source and enables the missing clock detector reset.
// Additionally, this routine sets up VREF, the internal regulator, and the
// VDD monitor.
//
void System_Init (void)
{
   OSCICN = 0x85;                       // configure internal oscillator
   RSTSRC = 0x04;                       // enable missing clock detector

   REF0CN = 0x01;                       // set up and enable VREF pin

   REG0CN = 0x10;                       // set up and enable 2.5V VDD from the
                                        // internal regulator

   VDDMon_Init ();                      // initialize the VDD Monitor
}

//-----------------------------------------------------------------------------
// VDDMon_Init
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// This routine initializes the VDD Monitor and enables it as a reset source.
//
void VDDMon_Init (void)
{
   unsigned char i;

   VDM0CN = 0x80;                       // enable the VDD monitor
   for (i = 0; i < 255; i++);           // wait for the monitor to stabilize
   RSTSRC = 0x06;                       // enable missing clock detector and
                                        // VDD monitor as reset sources
}
```

```
//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// P0.0 = DAC0 (analog, skip)
// P0.1-3 = unused (skip)
// P0.4-7 = SPI interface (digital, do not skip)
// P1.0-1 = unused (skip)
// P1.2 = VREF (analog, skip)
// P1.3 = analog power-on transistor (digital, skip)
// P1.4 = unused (skip)
// P1.5 = ADC0 (analog, skip)
// P1.6-7 = REC_PLAY and ERASE switches (digital, skip)
// P2.0-1 = LED PCA outputs (digital, do not skip)
//
void Port_Init (void)
{
   P0MDIN = 0xFE;                     // make switch and SPI pins digital
   P0MDOUT = 0xD0;                    // make SPI pins push-pull
   P1MDIN = 0xC8;                     // make trans and switches digital
   P1MDOUT = 0x08;                    // make trans pin push-pull
   P2MDIN = 0x03;                     // make PCA pins digital
   P2MDOUT = 0x03;                    // make PCA pins push-pull
   P0SKIP = 0x0F;                     // skip pins not belonging to SPI
   P1SKIP = 0xFF;                     // skip all P1 pins
   P2SKIP = 0xFC;                     // skip pins not belonging to LED PCA

   XBR0 = 0x02;                       // enable SPI
   XBR1 = 0x42;                       // enable PCA0_0 and PCA0_1

   TRANS = 0;                         // turn on the power to all analog
                                      // components

   P0MAT = 0x00;                      // the buttons will go low when pressed,
   P1MAT = 0xC0;                      // causing the port match event
   P0MASK = 0x00;                     // mask off all P0 and P1 pins except
   P1MASK = 0xC0;                     // the switches
   EIE2 = 0x00;                       // disable the port match interrupt
                                      // (not required to wake up the core)
}

//-----------------------------------------------------------------------------
// ADC0_Init
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// Configure ADC0 to update with a Timer 2 overflow using P1.5 as its positive
// input in post-tracking mode, enable burst mode, and use a repeat factor of
// 16.
//
void ADC0_Init (void)
{
   ADC0CN = 0x43;                     // ADC in low-power burst mode, use T2
                                      // overflow, right justify
   ADC0MX = 0x0D;                     // use P1.5 as the positive reference
   // set the ADC conversion about 5 MHz and use a repeat factor
   // of 16
   ADC0CF = (4 << 3) | (3 << 1);
```

```
   ADC0TK = 0xF4;                    // use post-tracking mode
   EIE1 |= 0x08;                     // enable the ADC conversion complete
                                     // interrupt
   EIP1 |= 0x08;                     // set the ADC convertion complete
                                     // interrupt to high priority
}

//-----------------------------------------------------------------------------
// DAC0_Init
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// Configure DAC0 to be right justified, update with a Timer 3 overflow, and
// use a full-scale 2 mA output current.
//
void DAC0_Init (void)
{
   IDA0CN = 0x70;                    // set the IDAC to update on a write
                                     // to IDA0DAT (initially only)
   IDA0CN |= 0x00;                   // set the IDAC to use a 0.25 mA current.
   IDA0CN |= 0x04;                   // set the IDAC to be right-justified
   IDA0CN |= 0x80;                   // enable the IDAC

   IDA0L = 0x00;                     // initialize the IDAC to be mid-scale
   IDA0H = 0x08;

   IDA0CN &= ~0x70;                  // set the IDAC to update on T3 overflow,
   IDA0CN |= 0x33;                   // and use a 2 mA full-scale current
}

//-----------------------------------------------------------------------------
// PCA_Init
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// Configure PCA0 modules 0 and 1 to 8-bit PWM mode using the system clock.
//
void PCA_Init (void)
{
   PCA0MD = 0x88;                    // set PCA to use system clock, disable
                                     // idle mode

   // PCA0 (for LED1)
   PCA0CPM0 = 0x42;                  // set PCA0 for 8-bit PWM mode
   PCA0CPH0 = 0x00;                  // set LED to off originally

   // PCA1 (for LED0)
   PCA0CPM1 = 0x42;                  // set PCA1 for 8-bit PWM mode
   PCA0CPH1 = 0x00;                  // set LED to off originally

   // add another PCA module for another LED here, if desired

   PCA0CN = 0x40;                    // turn on the PCA timer/counter
}

//-----------------------------------------------------------------------------
// SPI0_Init
//-----------------------------------------------------------------------------
//
// Return Value : None
```

```
// Parameters   : None
//
// Configure the SPI to run in 4-wire master mode at SYSCLK / 4 (1.53 MHz)
// using clock phase 0 and clock polarity 0 to interface with the SST Flash
// memory.
//
void SPI0_Init (void)
{
   SPI0CFG = 0x40;                      // set the master mode, polarity and
                                        // phase
   // set the SPI frequency to SYSCLK / 2*(1+1) = SYSCLK / 4
   SPI0CKR = 0x01;

   SPI0CN = 0x0C;                       // clear flags, turn off NSS
                                        // set the 4-wire mode
   SPIEN = 1;                           // enable the SPI
}

//-----------------------------------------------------------------------------
// Timer0_Init
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   :
//   1)  int period - number of timer counts to generate the desired period
//                    range is postive range of integer: 0 to 32767
//
// Configure Timer0 to 16-bit mode.  Timer0 is used to control the load
// time of the PCA PCA0CPHn registers, which changes the PWM intensity of the
// LEDs.
//
// The input parameter can be calculated as follows:
//   (Oscillator (Hz) / 4) / Desired_Freq (Hz) = Timer Ticks
//
void Timer0_Init (int period)
{
   TMOD |= 0x01;                        // set Timer 0 to mode 1 (16 bit)
   CKCON |= 0x04;                       // use the system clock
   ET0 = 1;                             // enable Timer 0 interrupts
   PT0 = 1;                             // set Timer 0 interrupts to high
                                        // priority (has to interrupt T1)

   TL0 = (-period) & 0x00FF;            // set the desired period
   TH0 = ((-period) & 0xFF00) >> 8;

   TR0 = 0;                             // keep Timer 0 off (LED
                                        // functions will turn it on)
}

//-----------------------------------------------------------------------------
// Timer1_Init
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   :
//   1)  int period - number of timer counts to generate the desired period
//                    range is postive range of integer: 0 to 32767
//
// Configure Timer1 to 16-bit mode.  Timer1 controls the switch polling.
//
// To calculate:
//   (Oscillator (Hz) / 4) / 48 / Desired_Freq (Hz) = Timer Ticks
//
// NOTE -  the extra 48 in this equation is present because of the settings
```

```
// in CKCON.
//
void Timer1_Init (int period)
{
   TMOD |= 0x10;                        // set Timer 1 to mode 1 (16 bit)
   CKCON |= 0x02;                       // use the system clock / 48
   ET1 = 1;                             // enable Timer 1 interrupts

   TL1 = (-period) & 0x00FF;            // set the desired period
   TH1 = ((-period) & 0xFF00) >> 8;

   TR1 = 0;                             // keep Timer 1 off until needed
}

//-----------------------------------------------------------------------------
// Timer2_Init
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   :
//   1)  int period - number of timer counts to generate the desired period
//                    range is postive range of integer: 0 to 32767
//
// Configure Timer2 to 16-bit auto reload mode.  Timer2 controls the ADC0
// start-of-conversion rate.
//
// To calculate:
//   (Oscillator (Hz) / 4) / Desired_Freq (Hz) = Timer Ticks
//
void Timer2_Init (int period)
{
   CKCON |= 0x10;                       // use the system clock
   TMR2CN = 0x00;                       // 16-bit auto-reload mode
   ET2 = 0;                             // disable T2 interrupts (use ADC
                                        // conversion complete interrupt)

   TMR2RL = -period;                    // set the desired period

   TMR2 = -period;                      // initialize the timer

   TR2 = 0;                             // keep Timer 2 off until the RECORD
                                        // function is used
}

//-----------------------------------------------------------------------------
// Timer3_Init
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   :
//   1)  int period - number of timer counts to generate the desired period
//                    range is postive range of integer: 0 to 32767
//
// Configure Timer3 to 16-bit auto reload mode.  Timer3 controls the DAC output
// rate.
//
// To calculate:
//   (Oscillator (Hz) / 4) / Desired_Freq (Hz) = Timer Ticks
//
void Timer3_Init (int period)
{
   CKCON |= 0x40;                       // use the system clock
   TMR3CN = 0x00;                       // 16-bit auto-reload mode
   EIE1 |= 0x80;                        // enable Timer 3 interrupts
```

```
   EIP1 |= 0x80;                      // set Timer 3 interrupts to high
                                      // priority

   TMR3RL = -period;                  // set the desired period

   TMR3 = -period;                    // initialize the timer

   TMR3CN = 0x00;                     // keep Timer 3 off until the PLAY
                                      // function is used
}

//-----------------------------------------------------------------------------
// RTC_Init
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// Enable the RTC so it doesn't cause a wake-up from suspend mode.
//
void RTC_Init (void)
{
   RTC0KEY = 0xA5;                    // unlock the RTC interface
   RTC0KEY = 0xF1;
   RTC0ADR = 0x06;                    // point to RTC0CN
   RTC0DAT = 0x80;                    // enable the RTC
}

//-----------------------------------------------------------------------------
// Recording_Search
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// Search for a recording already residing in memory on power-up and set the
// rec_end_addr accordingly.
//
void Recording_Search(void)
{
   unsigned long address = 0x00000000;
   bit end_flag = 0;

   // indicate to the user that the microcontroller is not ready to record
   // or playback
   LED_DCH = &LED0_DC;
   Brighten_LED ();
   LED_DCH = &LED1_DC;
   Brighten_LED ();

   // search through the SST flash until a series of 0xFF is found, indicating
   // cleared memory
   while (end_flag != 1)
   {
      if (Read_MEM_Init (address) == 0xFF)
      {
         // double-check that the 0xFF found is not just a data byte of 0xFF
         if (Read_MEM_Init (address+10) == 0xFF)
         {
            if (Read_MEM_Init (address+40) == 0xFF)
            {
               end_flag = 1;
            }
         }
```

```
      }

      address++;

      if (address == MAX_MEM_ADDR)
      {
         end_flag = 1;
      }
   }

   rec_end_addr = address-1;           // set the recording ending address

   // turn off the LEDs so the user knows the recording search has ended
   LED_DCH = &LED0_DC;
   Dim_LED ();
   LED_DCH = &LED1_DC;
   Dim_LED ();
}

////////////////////////// INTERRUPT SERVICE ROUTINES /////////////////////////

//-----------------------------------------------------------------------------
// Timer0_ISR
//-----------------------------------------------------------------------------
//
// Handle the 76Hz (13ms) Timer 0 interrupt.
//
// Timer 0 controls the rate at which the microcontroller changes the duty
// cycle of the PCA controlling the LEDs
//
// The LEDs are updated periodically, even if the LED PWM hasn't changed.
// By using the pointer (which is set before calling the LED functions) and
// updating all LEDs in the ISR every time, the same functions can be used for
// any number of LEDs.  To add an LED, simply set-up another PCA channel,
// point to that LED before calling the LED functions, and update the LED in
// the ISR.
//
void Timer0_ISR (void) interrupt 1 using 1
{
   *LED_DCH += ADJ;                    // calculate the new duty cycle based
                                       // on the values set by the LED
                                       // functions

   PCA0CPH1 = LED0_DC;                 // load all LEDs with the possibly
   PCA0CPH0 = LED1_DC;                 // updated value
   // add another LED update here, if desired

   TL0 = (-LED_PWM) & 0x00FF;          // wait the time specified by the
   TH0 = ((-LED_PWM) & 0xFF00) >> 8;   // calling LED function

   LED_PWM += LED_PWM_CHANGE;          // change the interrupt rate, if
                                       // necessary
}

//-----------------------------------------------------------------------------
// Timer1_ISR
//-----------------------------------------------------------------------------
//
// Handle the 64 Hz (15.63 ms) Timer 1 interrupt.
//
// Timer 1 controls the rate at which the microcontroller checks the switches
// for activity while in full power mode.
//
// for RECORD - press and hold REC_PLAY button, release stops recording
```

```
// for PLAYBACK - press and release REC_PLAY button, press and release again
//                    to stop
//
void Timer1_ISR (void) interrupt 3 using 0
{
   // interrupt again in 15.63 ms, unless a switch is pressed
   unsigned short reload_value = POLLING;

   static unsigned char record_counter = 0;
   static unsigned short suspend_counter = 0;

   bit switch_pressed_flag = 0;

   // REC_PLAY button pressed
   if (REC_PLAY == 0)
   {
      switch_pressed_flag = 1;         // record the user interaction

      // check if the recording time ran out, and stop any interaction
      // from the switch until the switch is released and pressed again
      if ((system_state & END_MEM) != END_MEM)
      {
         // the REC_PLAY button must be pressed and held for a period of time
         // in order to start the RECORD function
         record_counter++;

         // check if the REC_PLAY button was held down long enough to begin
         // recording (7 x 150 ms = 1.5 seconds)
         // ignore the ERASED and END_MEM state bits, check if the system is
         // idle and can start recording
         if ((record_counter > 7) && ((system_state & 0x03) == IDLE))
         {
            TR2 = 1;                   // turn on the RECORD timer
            system_state |= RECORDING; // start recording

            LED_DCH = &LED0_DC;        // point to the record LED's duty cycle
                                       // address
            Brighten_LED();            // ramp on the record LED

            record_counter = 0;        // reset the counter

            reload_value = PRCHANGE*2; // give a longer time period to check
                                       // the button (effectively debouncing)
         }

         // check if the recording time is running out (button must be held
         // to continue recording)
         if (TR2 == 1)
         {
            if (MEM_END_NEAR_FLAG == 1)
            {
               LED_DCH = &LED0_DC;
               Flutter_LED ();         // indicate to the user that time is
                                       // almost out
            }
         }
         else
         {
            // check if end of the memory has been reached
            if (MEM_END_FLAG == 1)
            {
               // stop recording
               system_state = IDLE | END_MEM;   // indicate that the end of
                                                // memory was reached
```

```
            MEM_END_FLAG = 0;

            LED_DCH = &LED0_DC;     // point to the record LED's duty cycle
                                    // address
            Dim_LED ();             // dim off the record LED
         }
      }
   }
}
else
{
   // check if the switch was pressed, but not long enough to start
   // recording
   if (record_counter > 0)
   {
      switch_pressed_flag = 1;      // record the user interaction

      // the system is currently playing - stop playing
      // ignore the ERASED and END_MEM state bits
      if ((system_state & 0x03) == PLAYING)
      {
         system_state &= ~PLAYING;  // clear the PLAYING state bit
         DAC_STOP_FLAG = 1;
         IDA0DAT = 0x0800;

         LED_DCH = &LED1_DC;        // point to the play LED's duty cycle
                                    // address
         Dim_LED ();                // dim off the play LED
      }
      else
      {
         // the system is idle - start playing
         // ignore the ERASED and END_MEM state bits
         if ((system_state & 0x03) == IDLE)
         {
            system_state |= PLAYING;
            TMR3CN = 0x04;          // start the timer controlling the DAC
            REC_END_FLAG = 0;       // reset the "end of recording" flag
            DAC_STOP_FLAG = 0;

            LED_DCH = &LED1_DC;     // point to the play LED's duty cycle
                                    // address
            Brighten_LED ();        // ramp on the play LED
         }
      }

      record_counter = 0;           // switch-press registered, reset

   }
   // the REC_PLAY switch was not pressed
   else
   {
      // clear the END_MEM recording flag after the ADC ISR has turned off
      // the ADC
      if ((system_state & END_MEM) == END_MEM)
      {
         system_state &= ~RECORDING;
      }

      // the system is currently recording - stop recording
      if (system_state == RECORDING)
      {
         system_state &= ~RECORDING;
         ADC_STOP_FLAG = 1;         // notify the ADC to stop recording
```

```
         MEM_END_NEAR_FLAG = 0;      // clear all flags
         MEM_END_FLAG = 0;

         LED_DCH = &LED0_DC;         // point to the record LED's duty cycle
                                     // address
         Dim_LED ();                 // dim off the record LED
      }

      // check if the playback has reached the end of the recording
      if (REC_END_FLAG == 1)
      {
         // stop playing
         system_state &= ~PLAYING;
         REC_END_FLAG = 0;

         LED_DCH = &LED1_DC;         // point to the play LED's duty cycle
                                     // address
         Dim_LED ();                 // dim off the play LED
      }
   }
}

// ERASE button pressed
if (ERASE == 0)
{
   // do nothing if the device is currently recording or playing
   // ignore the ERASED and END_MEM bits
   if ((system_state & 0x03) == IDLE)
   {
      // Indicate to the user that the microcontroller is busy
      LED_DCH = &LED1_DC;
      Brighten_LED ();
      LED_DCH = &LED0_DC;
      Brighten_LED ();

      rec_end_addr = 0x00000000;     // reset the counter
      system_state |= ERASED;        // set the erase bit
      Erase_MEM ();                  // erase the external SST Flash

      LED_DCH = &LED1_DC;
      Dim_LED ();
      LED_DCH = &LED0_DC;
      Dim_LED ();
   }

   switch_pressed_flag = 1;          // record the user interaction
}

if (switch_pressed_flag == 0)
{
   // check if the recorder is sitting and idle
   // ignore the ERASED and END_MEM bits
   if ((system_state & 0x03) == IDLE)
   {
      suspend_counter++;

      // if no interaction occurs in 5 seconds, enter suspend mode
      if (suspend_counter == 320)
      {
         suspend_counter = 0;
         ENTER_SUSPEND = 1;
      }
   }
```

```
   }
   else
   {
      suspend_counter = 0;             // reset the SUSPEND mode counter
                                       // if the user is interacting with the
                                       // recorder

      reload_value = PRCHANGE;         // interrupt again in 150 ms
   }

   // reload the timer for the next interrupt
   TL1 = (-reload_value) & 0x00FF;
   TH1 = ((-reload_value) & 0xFF00) >> 8;
}

//-----------------------------------------------------------------------------
// ADC0_ISR
//-----------------------------------------------------------------------------
//
// Handle the 8kHz Timer 2 interrupt.
//
// Timer 2 controls the rate at which the ADC samples the input (RECORD).
//
void ADC0_ISR (void) interrupt 10 using 2
{
   // RECORD

   // DPCM variables
   static data unsigned short predicted_value = mid_range;
   static data unsigned char packed_code = 0x00;
   data unsigned char dpcm_code = 0x00;

   // indicates how the current dpcm_code should be packed to be sent to memory
   // sample 1 dpcm_code = A, sample 2 dpcm_code = B, sample 3 dpcm_code = C
   // sample 4 dpcm_code = D, sample 5 is the same as sample 1, etc
   // [A|A|A|A|A|A|B|B] = byte 1
   // [B|B|B|B|C|C|C|C] = byte 2
   // [C|C|D|D|D|D|D|D] = byte 3
   static unsigned char state = 0;

   static short sample = 0x0000;

   AD0INT = 0;                         // clear the interrupt flag

   // check if the memory was erased
   if ((system_state & ERASED) == ERASED)
   {
      system_state &= ~ERASED;         // clear the erased bit
      predicted_value = mid_range;     // reset the dpcm predictor
      state = 0;                       // reset the packing state machine
   }

   // check for the end of memory
   if (rec_end_addr == MAX_MEM_ADDR)
   {
      TR2 = 0;                         // turn off T2
      MEM_END_NEAR_FLAG = 0;
      MEM_END_FLAG = 1;                // tell the T1 ISR to turn off the LED
      predicted_value = mid_range;     // reset the dpcm predictor
      state = 0;                       // reset the state machine
   }
   else
   {
      // check if the REC_PLAY switch was released and the recording should
```

```
// stop
if (ADC_STOP_FLAG == 1)
{
   TR2 = 0;                        // turn off T2
   ADC_STOP_FLAG = 0;              // reset the flag

   // do not reset the state or the predicted_value variables here
   // the playback ISR doesn't know when a recording starts or ends,
   // so it will also not reset the state and predicted_value
}
// take the sample, average it, compress it, and send it to memory
else
{
   // since 16 samples are automatically accumulated by the ADC,
   // average them by dividing by 16 (right shifting by 4)
   sample = (ADC0DAT >> 4) & 0x0FFF;



   // calculate the difference between the sample and the predictor
   // and compress the sample to a 6-bit DPCM code
   dpcm_code = DPCM_Encode ((sample - predicted_value));

   // pack the DPCM code into the bytes sent to the Flash memory
   switch (state)
   {
       // state machine: 0 -> 1 -> 2 -> 3
       //                  ^______________|
       case 0:

         // move the DPCM code into the 6 high bits
         // [A|A|A|A|A|A| | ] = byte 1
         packed_code = (dpcm_code << 2) & 0xFC;

         state = 1;
         break;

       case 1:

         // move the DPCM code into the 2 low bits
         // of the previously packed byte
         // [-|-|-|-|-|-|B|B] = byte 1

         packed_code |= (dpcm_code >> 4) & 0x03;

         Write_MEM (rec_end_addr, packed_code);
         rec_end_addr++;

         // move the rest of the DPCM code into the
         // 4 high bits of the next packed byte
         // [B|B|B|B| | | | ] = byte 2

         packed_code = (dpcm_code << 4) & 0xF0;

         state = 2;
         break;

       case 2:

         // move the next DPCM code into the
         // 4 low bits of the previously packed byte
         // [-|-|-|-|C|C|C|C] = byte 2

         packed_code |= (dpcm_code >> 2) & 0x0F;
```

```
            Write_MEM (rec_end_addr, packed_code);
            rec_end_addr++;

            // move the rest of the DPCM code into the
            // 2 high bits of the next packed byte
            // [C|C| | | | | | ] = byte 3

            packed_code = (dpcm_code << 6) & 0xC0;

            state = 3;
            break;

          case 3:

            // move the next DPCM code into the
            // 6 low bits of the previously packed byte
            // [-|-|D|D|D|D|D|D] = byte 3

            packed_code |= dpcm_code & 0x3F;

            Write_MEM (rec_end_addr, packed_code);
            rec_end_addr++;

            state = 0;
            break;

          default:
            state = 0;
            break;
         }

         // indicate that the T1 ISR should flutter the LED, since the end of
         // memory is close
         if (rec_end_addr == NEAR_END_ADDR)
         {
            MEM_END_NEAR_FLAG = 1;
         }

         // update the predictor for the next sample
         predicted_value += DPCM_Decode (dpcm_code);
      }
   }
}

//-----------------------------------------------------------------------------
// Timer3_ISR
//-----------------------------------------------------------------------------
//
// Handle the 8kHz Timer 3 interrupt.
//
// Timer 3 controls the rate at which the DAC outputs decompressed samples
// (PLAY).
//
void Timer3_ISR(void) interrupt 14 using 2
{
   // PLAY

   // next unwritten address
   static unsigned long current_play_addr = 0x00000000;

   // DPCM variables
   static unsigned short predicted_value = mid_range;
   static unsigned char packed_code = 0x00;
```

```
unsigned char dpcm_code = 0x00;

// indicates how the current dpcm_code should be unpacked when retrieved
// from memory
// sample 1 dpcm_code = A, sample 2 dpcm_code = B, sample 3 dpcm_code = C
// sample 4 dpcm_code = D, sample 5 is the same as sample 1, etc
// [A|A|A|A|A|A|B|B] = byte 1
// [B|B|B|B|C|C|C|C] = byte 2
// [C|C|D|D|D|D|D|D] = byte 3
static unsigned char state = 0;

TMR3CN &= 0x7F;                     // clear the T3 interrupt flag

// check if the PLAY switch was pressed and playing should stop
if (DAC_STOP_FLAG == 1)
{
   TMR3CN = 0x00;                   // turn off T3
   DAC_STOP_FLAG = 0;               // reset the flag
   current_play_addr = 0x00000000;  // start at the beginning address
   predicted_value = mid_range;     // reset the predictor
   state = 0;                       // reset the playback state machine
}
else
{
   // check for the end of the recording
   if (current_play_addr >= rec_end_addr)
   {
      TMR3CN = 0x00;                // turn off the timer
      REC_END_FLAG = 1;             // tell the T1 ISR to turn off the LED
      current_play_addr = 0x00000000;
      predicted_value = mid_range;  // reset the predictor
      state = 0;                    // reset the playback state machine
   }
   else
   {
      // unpack the DPCM code bytes retrieved from memory
      switch (state)
      {
          // state machine: 0 -> 1 -> 2 -> 3
          //                ^______________|
          case 0:

            packed_code = Read_MEM (current_play_addr);
            current_play_addr++;

            // take the DPCM code from the 6 high bits
            // [A|A|A|A|A|A| | ] = byte 1
            dpcm_code = (packed_code >> 2) & 0x3F;

            state = 1;
            break;

          case 1:

            // take the next DPCM code from the 2 low bits
            // of the previously retrieved byte
            // [-|-|-|-|-|-|B|B] = byte 1
            dpcm_code = (packed_code << 4) & 0x30;

            packed_code = Read_MEM (current_play_addr);
            current_play_addr++;

            // take the rest of the DPCM code from the
            // 4 high bits of the next retrieved byte
```

```
            // [B|B|B|B| | | | ] = byte 2

            dpcm_code |= (packed_code >> 4) & 0x0F;

            state = 2;
            break;

          case 2:

            // take the next DPCM code from the
            // 4 low bits of the previously retrieved byte
            // [-|-|-|-|C|C|C|C] = byte 2
            dpcm_code = (packed_code << 2) & 0x3C;

            packed_code = Read_MEM (current_play_addr);
            current_play_addr++;

            // take the rest of the DPCM code from the
            // 2 high bits of the next retrieved byte
            // [C|C| | | | | | ] = byte 3
            dpcm_code |= (packed_code >> 6) & 0x03;

            state = 3;
            break;

          case 3:

            // take the next DPCM code from the
            // 6 low bits of the previously retrieved byte
            // [-|-|D|D|D|D|D|D] = byte 3
            dpcm_code = packed_code & 0x3F;

            state = 0;
            break;

          default:
            state = 0;
            break;
      }

      // calculate the new predicted value
      predicted_value += DPCM_Decode (dpcm_code);

      // output the new sample to the speaker
      IDA0DAT = predicted_value;

      // overwrite the very last sample so the output is at the mid-range
      // when stopped
      // the discontinuity causes a small "clicking" sound when playback
      // starts and stops
      if (current_play_addr >= rec_end_addr)
      {
         IDA0DAT = mid_range;
      }
    }
  }
}

//-----------------------------------------------------------------------------
// End Of File
//-----------------------------------------------------------------------------
```

External Flash Access Functions

```
//-----------------------------------------------------------------------------
// F411_VR_SSTFlash.c
//-----------------------------------------------------------------------------
// Copyright 2006 Silicon Laboratories, Inc.
// http://www.silabs.com
//
// Program Description:
//
// This file contains the interfacing functions to the SST Flash, allowing the
// user to Read memory, Write memory, and Erase memory.
//
// How To Use:    See Readme.txt
//
// FID:            41X000010
// Target:         C8051F411
// Tool chain:     Keil C51 7.50 / Keil EVAL C51
//                 Silicon Laboratories IDE version 2.6
// Project Name:   F411_VR
//
// Release 1.4
//    -All changes by TP
//    -06 Feb 2009
//    -No changes to this file
//
// Release 1.3
//    -All changes by TP
//    -02 Feb 2006
//    -added Read_MEM_Init (duplicate of Read_MEM) to avoid
//       the compiler warning (multiple calls to segment)
//
// Release 1.2
//    -All changes by TP
//    -21 Nov 2005
//    -project version updated, no changes to this file.
//
// Release 1.1
//    -All changes by TP
//    -16 Aug 2004
//    -added SPIF polling while sending the WREN command
//
// Release 1.0
//    -Initial Revision (TP)
//    -15 AUG 2004
//

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------
#include <c8051f410.h>                 // SFR declarations

//-----------------------------------------------------------------------------
// Global CONSTANTS
//-----------------------------------------------------------------------------
// SST Instruction Opcodes (as shown in the datasheet)
#define EWSR 0x50                      // enable write status register
#define WRSR 0x01                      // write status register
#define RDSR 0x05                      // read status register
#define WREN 0x06                      // write enable
#define BPROG 0x02                     // byte program
#define READ 0x03                      // read
#define CERASE 0x60                    // chip erase
#define READID 0x90                    // chip ID
```

```
// Address definition
typedef union ADDRESS {                  // access an address as a
   unsigned long ULong;                  // unsigned long variable or
   unsigned char UByte[4];               // 4 unsigned byte variables
   // [0] = A31-24, [1] = A23-16, [2] = A15-8, [3] = A7-0
} ADDRESS;

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
void SSTFlash_Init (void);

void Write_MEM (unsigned long address, unsigned char data_byte);
unsigned char Read_MEM (unsigned long address);
void Erase_MEM (void);
char ReadID_MEM (void);

//-----------------------------------------------------------------------------
// SSTFlash_Init
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// Unprotect the memory so that all of memory may be written and read.
// NOTE: The SPI must be initialized before this function is called.
//
void SSTFlash_Init (void)
{
   NSSMD0 = 0;                           // enable the flash

   // send the enable write status register command
   SPI0DAT = EWSR;                       // load the XMIT register
   while (TXBMT != 1)                    // wait until EWSR command is moved into
   {                                     // the XMIT buffer
   }
   SPIF = 0;
   while (SPIF != 1)                     // wait until the SPI finishes sending
   {                                     // the EWSR command to the flash
   }
   SPIF = 0;

   NSSMD0 = 1;                           // allow the command to execute

   NSSMD0 = 0;                           // enable the flash

   // send the write status register command and clear the BP bits
   SPI0DAT = WRSR;                       // load the XMIT register
   while (TXBMT != 1)                    // wait until the XMIT register can
   {                                     // accept more data
   }
   SPI0DAT = 0x00;                       // set the block protection bits to 0
   while (TXBMT != 1)                    // wait until the data is moved into
   {                                     // the XMIT buffer
   }
   SPIF = 0;
   while (SPIF != 1)                     // wait until the SPI finishes sending
   {                                     // the data to the flash
   }
   SPIF = 0;

   NSSMD0 = 1;                           // allow the command to execute
}
```

```
//-----------------------------------------------------------------------------
// Write_MEM
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   :
//   1)  long address - address in the 512 kB external SST Flash
//                      range is postive values up to 2^19: 0 to 524287,
//                           or, 0 to 0x7FFFF
//   2)  char data_byte - the data to be written to memory
//                      range is positive range of character: 0 to 255
//
// Write one byte of data to a 24-bit address in the SST Flash Memory using
// the SPI.
//
void Write_MEM (unsigned long address, unsigned char data_byte)
{
   ADDRESS temp_addr;
   temp_addr.ULong = address;

   NSSMD0 = 0;                         // enable the flash

   // send the write enable command
   SPI0DAT = WREN;                     // load the XMIT register
   while (TXBMT != 1)                  // wait until the command is moved into
   {                                   // the XMIT buffer
   }
   SPIF = 0;
   while (SPIF != 1)                   // wait until the command reaches the
   {                                   // flash
   }
   SPIF = 0;

   NSSMD0 = 1;                         // allow the WREN to execute

   NSSMD0 = 0;                         // enable the flash

   // send the byte-program command
   SPI0DAT = BPROG;                    // load the XMIT register
   while (TXBMT != 1)                  // wait until the command is moved into
   {                                   // the XMIT buffer
   }
   SPI0DAT = temp_addr.UByte[1];       // load the high byte of the address
   while (TXBMT != 1)                  // wait until the addr is moved into
   {                                   // the XMIT buffer
   }
   SPI0DAT = temp_addr.UByte[2];       // load the middle byte of the address
   while (TXBMT != 1)                  // wait until the addr is moved into
   {                                   // the XMIT buffer
   }
   SPI0DAT = temp_addr.UByte[3];       // load the low byte of the address
   while (TXBMT != 1)                  // wait until the addr is moved into
   {                                   // the XMIT buffer
   }
   SPI0DAT = data_byte;                // load the byte of data
   while (TXBMT != 1)                  // wait until the data is moved into
   {                                   // the XMIT buffer
   }
   SPIF = 0;
   while (SPIF != 1)                   // wait until the last byte of the
   {                                   // write instruction reaches the flash
   }
   SPIF = 0;
```

```
   NSSMD0 = 1;                         // allow the WR instruction to execute
}

//-----------------------------------------------------------------------------
// Read_MEM
//-----------------------------------------------------------------------------
//
// Return Value :
//   1)  char data_byte - the data byte read from memory
//                        range is positive range of character: 0 to 255
// Parameters   :
//   1)  long address - address in the 512 kB external SST Flash
//                        range is postive values up to 2^19: 0 to 524287,
//                             or, 0 to 0x7FFFF
//
// Read one byte of data from a 24-bit address in the SST Flash Memory using
// the SPI.
//
unsigned char Read_MEM (unsigned long address)
{
   ADDRESS temp_addr;
   temp_addr.ULong = address;

   NSSMD0 = 0;                         // enable the flash

   // send the read instruction
   SPI0DAT = READ;                     // load the XMIT register
   while (TXBMT != 1)                  // wait until the command is moved into
   {                                   // the XMIT buffer
   }
   SPI0DAT = temp_addr.UByte[1];       // load the high byte of the address
   while (TXBMT != 1)                  // wait until the data is moved into
   {                                   // the XMIT buffer
   }
   SPI0DAT = temp_addr.UByte[2];       // load the middle byte of the address
   while (TXBMT != 1)                  // wait until the data is moved into
   {                                   // the XMIT buffer
   }
   SPI0DAT = temp_addr.UByte[3];       // load the low byte of the address
   while (TXBMT != 1)                  // wait until the data is moved into
   {                                   // the XMIT buffer
   }
   SPI0DAT = 0xFF;                     // load junk data in order to receive
                                       // data from the flash
   while (TXBMT != 1)                  // wait until the junk data is moved
   {                                   // into the XMIT buffer
   }
   SPIF = 0;
   while (SPIF != 1)                   // wait until the read data is received
   {
   }
   SPIF = 0;

   NSSMD0 = 1;                         // disable the flash

   return SPI0DAT;
}

//-----------------------------------------------------------------------------
// Erase_MEM
//-----------------------------------------------------------------------------
//
// Return Value : None
```

```
// Parameters   : None
//
// Erase all data from the SST flash memory.
//
void Erase_MEM (void)
{
   unsigned char mem_status = 0x01;

   NSSMD0 = 0;                         // enable the flash

   // send the write enable command
   SPI0DAT = WREN;                     // load the XMIT register
   while (TXBMT != 1)                  // wait until the command is moved into
   {                                   // the XMIT buffer
   }
   SPIF = 0;
   while (SPIF != 1)                   // wait until the command reaches the
   {                                   // flash
   }
   SPIF = 0;

   NSSMD0 = 1;                         // allow the WREN to execute

   NSSMD0 = 0;                         // enable the flash

   // send the chip erase instruction
   SPI0DAT = CERASE;                   // load the XMIT register
   while (TXBMT != 1)                  // wait until the command is moved into
   {                                   // the XMIT buffer
   }
   SPIF = 0;
   while (SPIF != 1)                   // wait until the command reaches the
   {                                   // flash
   }
   SPIF = 0;

   NSSMD0 = 1;                         // allow the erase to execute

   // poll on the busy bit in the flash until the erase operation is complete
   NSSMD0 = 0;                         // enable the flash
   SPI0DAT = RDSR;                     // send the read status register command
   while (TXBMT != 1)                  // wait until the SPI can accept more
   {                                   // data
   }
   while (mem_status == 0x01)
   {
      SPI0DAT = 0xFF;                  // send junk in order to receive data
      while (TXBMT != 1)               // wait until the junk data is moved
      {                                // into the XMIT buffer
      }
      SPIF = 0;
      while (SPIF != 1)                // wait until the read data is received
      {
      }
      SPIF = 0;
      mem_status = SPI0DAT & 0x01;     // check the BUSY bit
   }

   NSSMD0 = 1;                         // disable the flash
}

//-----------------------------------------------------------------------------
// Read_MEM_Init
//-----------------------------------------------------------------------------
```

```
//
// Return Value :
//   1)  char data_byte - the data byte read from memory
//                        range is positive range of character: 0 to 255
// Parameters   :
//   1)  long address - address in the 512 kB external SST Flash
//                        range is postive values up to 2^19: 0 to 524287,
//                             or, 0 to 0x7FFFF
//
// Read one byte of data from a 24-bit address in the SST Flash Memory using
// the SPI.  This function is called by Recording_Search in F411_VR.c
// and is a duplicate of Read_MEM to avoid a warning by the compiler.
//
unsigned char Read_MEM_Init (unsigned long address)
{
   ADDRESS temp_addr;
   temp_addr.ULong = address;

   NSSMD0 = 0;                        // enable the flash

   // send the read instruction
   SPI0DAT = READ;                    // load the XMIT register
   while (TXBMT != 1)                 // wait until the command is moved into
   {                                  // the XMIT buffer
   }
   SPI0DAT = temp_addr.UByte[1];      // load the high byte of the address
   while (TXBMT != 1)                 // wait until the data is moved into
   {                                  // the XMIT buffer
   }
   SPI0DAT = temp_addr.UByte[2];      // load the middle byte of the address
   while (TXBMT != 1)                 // wait until the data is moved into
   {                                  // the XMIT buffer
   }
   SPI0DAT = temp_addr.UByte[3];      // load the low byte of the address
   while (TXBMT != 1)                 // wait until the data is moved into
   {                                  // the XMIT buffer
   }
   SPI0DAT = 0xFF;                    // load junk data in order to receive
                                      // data from the flash
   while (TXBMT != 1)                 // wait until the junk data is moved
   {                                  // into the XMIT buffer
   }
   SPIF = 0;
   while (SPIF != 1)                  // wait until the read data is received
   {
   }
   SPIF = 0;

   NSSMD0 = 1;                        // disable the flash

   return SPI0DAT;
}

//-----------------------------------------------------------------------------
// ReadID_MEM
//-----------------------------------------------------------------------------
//
// Return Value :
//   1)  char data_byte - the device ID read from memory at address 0x000001
//                        (this address is specified in the SST Flash datasheet)
//                        range is positive range of character: 0 to 255
// Parameters   : None
//
// Read the part ID from the flash memory (used for debugging).
```

```
//
char ReadID_MEM (void)
{
   NSSMD0 = 0;                    // enable the flash

   SPI0DAT = READID;              // send the read ID instruction
   while (TXBMT != 1)             // wait until the SPI can accept more
   {                              // data
   }
   SPI0DAT = 0x00;                // send the device ID address
   while (TXBMT != 1)             // wait until the SPI can accept more
   {                              // data
   }
   SPI0DAT = 0x00;                // send the device ID address
   while (TXBMT != 1)             // wait until the SPI can accept more
   {                              // data
   }
   SPI0DAT = 0x01;                // send the device ID address
   while (TXBMT != 1)             // wait until the SPI can accept more
   {                              // data
   }
   SPI0DAT = 0xA5;                // send dummy data for shift register
   while (TXBMT != 1)             // wait until the SPI can accept more
   {                              // data
   }
   SPIF = 0;
   while (SPIF != 1)              // wait until the read data is received
   {
   }
   SPIF = 0;

   NSSMD0 = 1;                    // disable the flash

   return SPI0DAT;
}

//-----------------------------------------------------------------------------
// End Of File
//-----------------------------------------------------------------------------
```

DPCM (Differential Pulse Code Modulation) Functions

```
//-----------------------------------------------------------------------------
// F411_VR_DPCM.c
//-----------------------------------------------------------------------------
// Copyright 2006 Silicon Laboratories, Inc.
// http://www.silabs.com
//
// Program Description:
//
// This file contains the DPCM encoding and decoding functions.
//
// NOTE: For another reference for DPCM, please see Chipcon's app note an026.
//
// NOTE: The calling function must have the same register context as the DCPM
// functions, so it must either have the keyword "using 2" or all "using 2"
// keywords for the DPCM functions need to be removed
//
// How To Use:    See Readme.txt
//
// FID:            41X000006
// Target:         C8051F411
// Tool chain:     Keil C51 7.50 / Keil EVAL C51
//                 Silicon Laboratories IDE version 2.6
// Project Name:   F411_VR
//
// Release 1.4
//    -All changes by TP
//    -06 Feb 2009
//    -No changes to this file
//
// Release 1.3
//    -All changes by TP
//    -02 Feb 2006
//    -project version updated, no changes to this file
//
// Release 1.2
//    -All changes by TP
//    -21 Nov 2005
//    -expanded the 4-bit codes to 6 bits for the 12-bit ADC
//
// Release 1.1
//    -All changes by TP
//    -16 Aug 2004
//    -project version updated, no changes to this file
//
// Release 1.0
//    -Initial Revision (TP)
//    -15 AUG 2004
//

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------
#include <c8051f410.h>                 // SFR declarations

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

// 12-bit quantization codes (6 bits, so 64 codes total = 31 positive, 31
// negative, and 2 zeroes)
#define quant1       1
#define quant2       2
```

```
#define quant3       4
#define quant4       7
#define quant5       11
#define quant6       16
#define quant7       22
#define quant8       29
#define quant9       37
#define quant10      46
#define quant11      56
#define quant12      67
#define quant13      79
#define quant14      92
#define quant15      106
#define quant16      130
#define quant17      146
#define quant18      163
#define quant19      181
#define quant20      200
#define quant21      220
#define quant22      241
#define quant23      263
#define quant24      286
#define quant25      310
#define quant26      335
#define quant27      361
#define quant28      388
#define quant29      416
#define quant30      512
#define quant31      1024

// the mapping from quantization values to dpcm codes (array index)
xdata short Q_VALUES[64] = {0,               // 0
                           -quant31,         // 1
                           -quant30,         // 2
                           -quant29,         // 3
                           -quant28,         // 4
                           -quant27,         // 5
                           -quant26,         // 6
                           -quant25,         // 7
                           -quant24,         // 8
                           -quant23,         // 9
                           -quant22,         // 10
                           -quant21,         // 11
                           -quant20,         // 12
                           -quant19,         // 13
                           -quant18,         // 14
                           -quant17,         // 15
                           -quant16,         // 16 negative middle
                           -quant15,         // 17
                           -quant14,         // 18
                           -quant13,         // 19
                           -quant12,         // 20
                           -quant11,         // 21
                           -quant10,         // 22
                           -quant9,          // 23
                           -quant8,          // 24
                           -quant7,          // 25
                           -quant6,          // 26
                           -quant5,          // 27
                           -quant4,          // 28
                           -quant3,          // 29
                           -quant2,          // 30
                           -quant1,          // 31
                           0,                // 32
```

```
                                quant1,             // 33
                                quant2,             // 34
                                quant3,             // 35
                                quant4,             // 36
                                quant5,             // 37
                                quant6,             // 38
                                quant7,             // 39
                                quant8,             // 40
                                quant9,             // 41
                                quant10,            // 42
                                quant11,            // 43
                                quant12,            // 44
                                quant13,            // 45
                                quant14,            // 46
                                quant15,            // 47
                                quant16,            // 48 positive middle
                                quant17,            // 49
                                quant18,            // 50
                                quant19,            // 51
                                quant20,            // 52
                                quant21,            // 53
                                quant22,            // 54
                                quant23,            // 55
                                quant24,            // 56
                                quant25,            // 57
                                quant26,            // 58
                                quant27,            // 59
                                quant28,            // 60
                                quant29,            // 61
                                quant30,            // 62
                                quant31};           // 63

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
unsigned char DPCM_Encode (short sample_diff);
short DPCM_Decode (unsigned char dpcm_code);

//-----------------------------------------------------------------------------
// DPCM_Encode
//-----------------------------------------------------------------------------
//
// Return Value :
//   1)  char dpcm_code - the 6-bit quantized DPCM code
//                        range is positive range of 6-bit value: 0 to 63
// Parameters   :
//   1)  short sample_diff - the difference between the predicted value and
//                        the sample from the ADC
//                        range is: -4096 to 4095 (difference of 12-bit values)
//
// Encode the sample using DPCM compression.
//
// The coding uses the following scheme (0 is unused) for an 8-bit sample:
//
// code:      1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
// q value: -64 -32 -16  -8  -4  -2  -1   0   1    2   4   8  16  32  64
//
// The difference will be rounded down if positive and rounded up if
// negative (i.e. 41 => 32, and -41 => -32).
//
// NOTE: the calling function must have the same register context, so it must
// either have the keyword "using 2" or all "using 2" keywords need to be
// removed
//
```

```
unsigned char DPCM_Encode (short sample_diff) using 2
{
   short sample_diff_us;
   unsigned char dpcm_code;

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

   // narrow down which bits need to be set to use the proper quantization code
   // using a binary search algorithm (divide in halves)
   // the sign of the difference no longer matters

   // first tier
   if (sample_diff_us >= quant16)
   {
      // second tier
      if (sample_diff_us >= quant24)
      {
         // third tier
         if (sample_diff_us >= quant28)
         {
            // fourth tier
            if (sample_diff_us >= quant30)
            {
               // fifth tier
               if (sample_diff_us >= quant31)
               {
                  dpcm_code = 63;
               }
               // fifth tier
               else
               {
                  dpcm_code = 62;
               }
            }
            // fourth tier
            else
            {
               // fifth tier
               if (sample_diff_us >= quant29)
               {
                  dpcm_code = 61;
               }
               // fifth tier
               else
               {
                  dpcm_code = 60;
               }
            }
         }
         // third tier
         else
         {
            // fourth tier
            if (sample_diff_us >= quant26)
            {
               // fifth tier
```

```
            if (sample_diff_us >= quant27)
            {
               dpcm_code = 59;
            }
            // fifth tier
            else
            {
               dpcm_code = 58;
            }
         }
         // fourth tier
         else
         {
            // fifth tier
            if (sample_diff_us >= quant25)
            {
               dpcm_code = 57;
            }
            // fifth tier
            else
            {
               dpcm_code = 56;
            }
         }
      }
   }
   // second tier
   else
   {
      // third tier
      if (sample_diff_us >= quant20)
      {
         // fourth tier
         if (sample_diff_us >= quant22)
         {
            // fifth tier
            if (sample_diff_us >= quant23)
            {
               dpcm_code = 55;
            }
            // fifth tier
            else
            {
               dpcm_code = 54;
            }
         }
         // fourth tier
         else
         {
            // fifth tier
            if (sample_diff_us >= quant21)
            {
               dpcm_code = 53;
            }
            // fifth tier
            else
            {
               dpcm_code = 52;
            }
         }
      }
      // third tier
      else
      {
```

```
            // fourth tier
            if (sample_diff_us >= quant18)
            {
               // fifth tier
               if (sample_diff_us >= quant19)
               {
                  dpcm_code = 51;
               }
               // fifth tier
               else
               {
                  dpcm_code = 50;
               }
            }
            // fourth tier
            else
            {
               // fifth tier
               if (sample_diff_us >= quant17)
               {
                  dpcm_code = 49;
               }
               // fifth tier
               else
               {
                  dpcm_code = 48;
               }
            }
         }
      }
   }
   // first tier
   else
   {
      // second tier
      if (sample_diff_us >= quant8)
      {
         // third tier
         if (sample_diff_us >= quant12)
         {
            // fourth tier
            if (sample_diff_us >= quant14)
            {
               // fifth tier
               if (sample_diff_us >= quant15)
               {
                  dpcm_code = 47;
               }
               // fifth tier
               else
               {
                  dpcm_code = 46;
               }
            }
            // fourth tier
            else
            {
               // fifth tier
               if (sample_diff_us >= quant13)
               {
                  dpcm_code = 45;
               }
               // fifth tier
               else
```

```
            {
               dpcm_code = 44;
            }
         }
      }
      // third tier
      else
      {
         // fourth tier
         if (sample_diff_us >= quant10)
         {
            // fifth tier
            if (sample_diff_us >= quant11)
            {
               dpcm_code = 43;
            }
            // fifth tier
            else
            {
               dpcm_code = 42;
            }
         }
         // fourth tier
         else
         {
            // fifth tier
            if (sample_diff_us >= quant9)
            {
               dpcm_code = 41;
            }
            // fifth tier
            else
            {
               dpcm_code = 40;
            }
         }
      }
   }
   // second tier
   else
   {
      // third tier
      if (sample_diff_us >= quant4)
      {
         // fourth tier
         if (sample_diff_us >= quant6)
         {
            // fifth tier
            if (sample_diff_us >= quant7)
            {
               dpcm_code = 39;
            }
            // fifth tier
            else
            {
               dpcm_code = 38;
            }
         }
         // fourth tier
         else
         {
            // fifth tier
            if (sample_diff_us >= quant5)
            {
```

```
                dpcm_code = 37;
             }
             // fifth tier
             else
             {
                dpcm_code = 36;
             }
          }
       }
       // third tier
       else
       {
          // fourth tier
          if (sample_diff_us >= quant2)
          {
             // fifth tier
             if (sample_diff_us >= quant3)
             {
                dpcm_code = 35;
             }
             // fifth tier
             else
             {
                dpcm_code = 34;
             }
          }
          // fourth tier
          else
          {
             // fifth tier
             if (sample_diff_us >= quant1)
             {
                dpcm_code = 33;
             }
             // fifth tier
             else
             {
                dpcm_code = 32;
             }
          }
       }
    }
 }

 // convert the DPCM code to its 2's compliment if the original sample
 // difference was negative
 // For example, 41 (101001), which represents a difference of 60, 2's
 // complimented becomes 23 (010111), which represents a difference of -60
 if (sample_diff < 0)
 {
    dpcm_code = ~dpcm_code + 1;     // use the 2's compliment of the dpcm
                                    // code
    dpcm_code &= 0x3F;              // use only the 6 LSBs for the dpcm code
 }

 return dpcm_code;
}

//-----------------------------------------------------------------------------
// DPCM_Decode
//-----------------------------------------------------------------------------
//
// Return Value :
//   1)  short predicted_value - the signed and quantized difference between
```

```
//                        the predicted_value and the ADC sample, which is used
//                        create the predicted_value for the next DPCM cycle
//                        range is: -4096 to 4095 (difference of 12-bit values)
// Parameters   :
//   1)  char dpcm_code - the 6-bit code indicating the quantized difference
//                        between the old_prediction and the current sample value
//                        range is positive range of 6-bit value: 0 to 63
//
// Decode the DPCM code to a signed difference between the current predicted
// value and the next.
//
// NOTE: the calling function must have the same register context, so it must
// either have the keyword "using 2" or all "using 2" keywords need to be
// removed
short DPCM_Decode (unsigned char dpcm_code) using 2
{
   return Q_VALUES[dpcm_code];
}

//-----------------------------------------------------------------------------
// End Of File
//-----------------------------------------------------------------------------
```

LED Functions

```
//-----------------------------------------------------------------------------
// F411_VR_LED.c
//-----------------------------------------------------------------------------
// Copyright 2006 Silicon Laboratories, Inc.
// http://www.silabs.com
//
// Program Description:
//
// This file contains the functions that use the PWM to brighten, dim, and
// flutter the LEDs.
//
// These functions work by using a pointer (LED_DCH) to an "LED byte," which
// is just a byte in memory associated with each LED.  In F411_VR.c,
// one of the Timer ISRs updates the PCA PWM registers with all the LED bytes
// every interrupt period, so the LEDs that don't change are still updated,
// but visually nothing changes.
//
// When the timer interrupts, PCA0CPH1 is reloaded with the current value
// LED0_DC, which is changed by the functions (Dim, Brighten, and Flutter)
// based on the desired LED behavior.  By decrementing the time the LED is
// on in steps (based on the ADJ variable), the LED appears to "dim" off,
// and by incrementing the time the LED is on in steps, the LED appears to
// "brighten" slowly.
//
// The LED_DCH pointer must be pointed to the correct LEDx_DC byte BEFORE
// each of these functions is called.
//
// LED_DCH -> LED0_DC -> PCA0CPH1, where CEX1 (output from the PCA) is tied
// to LED0
//
// For example, the resulting dim LED waveform might look something like
// this, since the LEDs are ON when CEX1 = 0:
//
//                                     _              __              ___
// CEX1  _____________________________| |____________|  |___________|   |
//
//      |   1st step    |    2nd step   |   3rd step    |    4th step   |
//
// (continued)
//                  ____           _____          ______         _______
// CEX1  __________|    |_________|     |________|      |_______|       |
//
//      |   5th step    |    6th step   |    7th step   |    8th step   |
//
// (continued)
//              ________       _________      __________     ___________
// CEX1  ______|        |_____|         |____|          |___|           |
//
//      |   9th period  |  10th period  |   11th period   | 12th  period |
//
// (continued)
//          ____________   _____________________________________________
// CEX1  __|            |_|
//
//      |  13th period  |  14th period  |   15th period   | 16th  period |
//
// The LED has appeared to "dim" slowly off.
//
//
//
// NOTE: The calling function must have the same register context as the LED
// functions, so it must either have the keyword "using 0" or all "using 0"
```

```
// keywords for the LED functions need to be removed.
//
// How To Use:    See Readme.txt
//
// FID:            41X000008
// Target:         C8051F411
// Tool chain:     Keil C51 7.50 / Keil EVAL C51
//                 Silicon Laboratories IDE version 2.6
// Project Name:   F411_VR
//
// Release 1.4
//    -All changes by TP
//    -06 Feb 2009
//    -No changes to this file
//
// Release 1.3
//    -All changes by TP
//    -02 Feb 2006
//    -project version updated, no changes to this file
//
// Release 1.2
//    -All changes by TP
//    -21 Nov 2005
//    -project version updated, no changes to this file
//
// Release 1.1
//    -All changes by TP
//    -16 Aug 2004
//    -project version updated, no changes to this file
//
// Release 1.0
//    -Initial Revision (TP)
//    -15 AUG 2004
//

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------
#include <c8051f410.h>                 // SFR declarations

//-----------------------------------------------------------------------------
// Global Variables
//-----------------------------------------------------------------------------
unsigned char ADJ = 15;
unsigned int LED_PWM = 65535;
int LED_PWM_CHANGE = 0x0000;
unsigned char *LED_DCH;
unsigned char LED0_DC = 0x00;
unsigned char LED1_DC = 0x00;
// add another LEDx_DC variable here, if desired, and point to it with *LED_DCH
// before calling the LED functions

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
void Dim_LED (void);
void Brighten_LED (void);
void Flutter_LED (void);


//-----------------------------------------------------------------------------
// Dim_LED
//-----------------------------------------------------------------------------
//
```

```
// Return Value : None
// Parameters   : None
//
// Dim the LED using the PCA in 8-bit PWM mode.  The Timer0 ISR in
// F411_VR.c updates the value LED_DCH is pointing to.
//
// NOTE: This function requires that the LED_DCH pointer be "pointing" to the
// appropriate LED byte, as explained above.
//
void Dim_LED (void) using 0
{
   // retrieve the previous value of the duty cycle
   unsigned char duty_cycle = *LED_DCH;

   ADJ = 0xF1;                         // set the ADJ such that the LED will
                                       // get dimmer
   LED_PWM = 65535;                    // reset the Timer 0 interval
   LED_PWM_CHANGE = 0;                 // do not change the Timer 0 interval
   TCON |= 0x10;                       // start Timer 0

   // wait until the LED is fully off
   while (duty_cycle != 0x00)
   {
      duty_cycle = *LED_DCH;
   }

   TCON &= ~0x10;                      // stop Timer 0 (no more updates to the
                                       // PCA duty cycle)
}

//-----------------------------------------------------------------------------
// Brighten_LED
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// Brighten the LED using the PCA in 8-bit PWM mode.  The Timer0 ISR in
// F411_VR.c updates the value LED_DCH is pointing to.
//
// NOTE: This function requires that the LED_DCH pointer be "pointing" to the
// appropriate LED byte, as explained above.
//
void Brighten_LED (void) using 0
{
   // retrieve the previous value of the duty cycle
   unsigned char duty_cycle = *LED_DCH;

   ADJ = 0x0F;                         // set the ADJ such that the LED will
                                       // brighten
   LED_PWM = 65535;                    // reset the Timer 0 interval
   LED_PWM_CHANGE = 0;                 // do not change the Timer 0 interval
   TCON |= 0x10;                       // start Timer 0

   // wait until the LED is fully on
   while (duty_cycle != 0xFF)
   {
      duty_cycle = *LED_DCH;
   }

   TCON &= ~0x10;                      // stop Timer 0 (no more updates to the
                                       // PCA duty cycle)
}
```

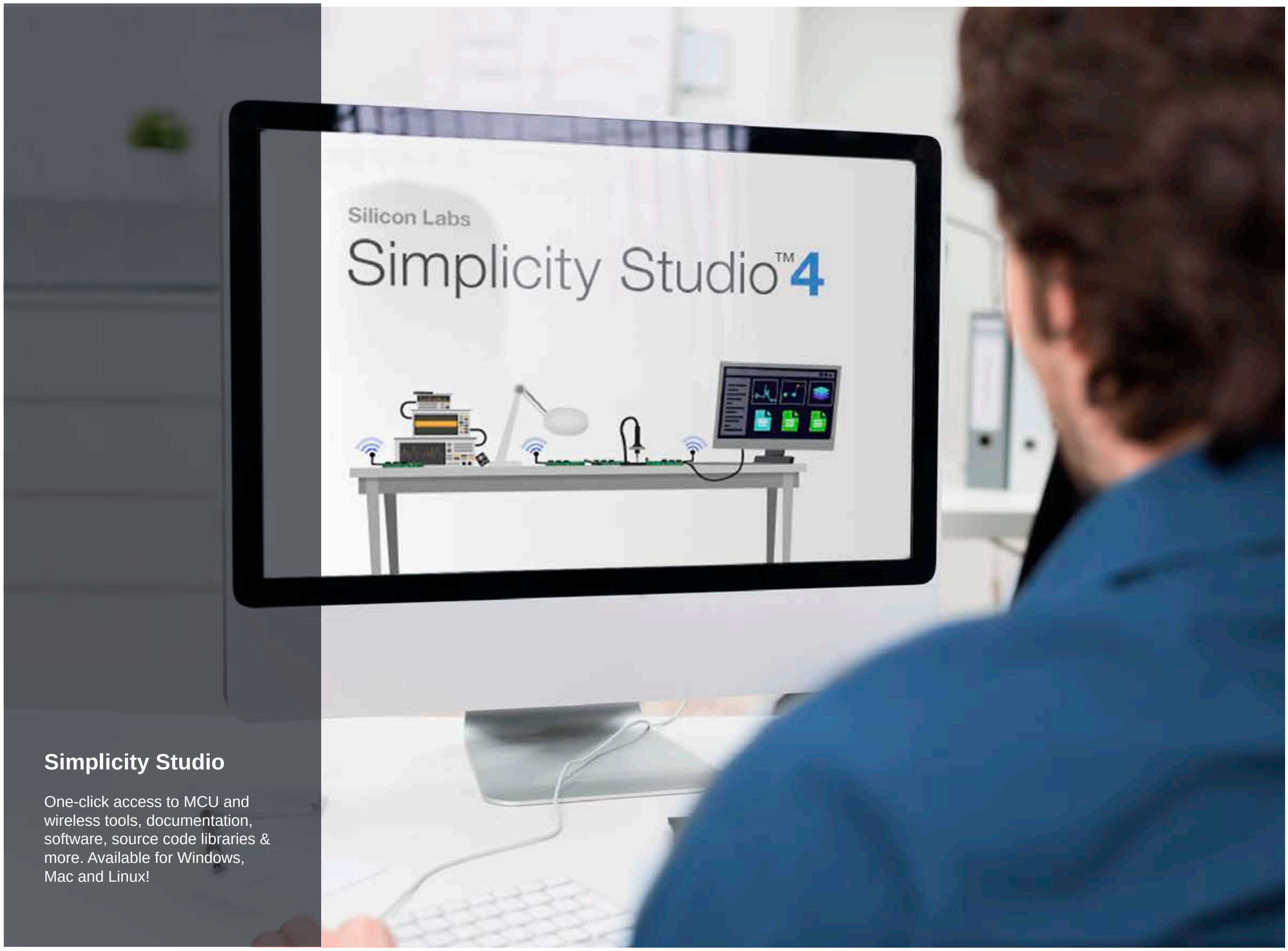
```
//-----------------------------------------------------------------------------
// Flutter_LED
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// Cause the LED to dim on and off.  The Timer0 ISR in F411_VR.c updates
// the value LED_DCH is pointing to.
//
// NOTE: This function requires that the LED_DCH pointer be "pointing" to the
// appropriate LED byte, as explained above.
//
void Flutter_LED (void) using 0
{
   // retrieve the previous value of the duty cycle
   unsigned char duty_cycle = *LED_DCH;

   // check if the LED is currently on or off
   if (duty_cycle == 0xFF)
   {
      ADJ = 0xF1;
   }
   else
   {
      ADJ = 0x0F;
   }
   LED_PWM = 65535;                    // reset the Timer 0 interval
   LED_PWM_CHANGE = -200;              // change the Timer 0 interval each
                                       // interrupt cycle so the LED has a
                                       // "fluttering" effect
   TCON |= 0x10;                       // start Timer 0

   // Wait for a flutter cycle to finish
   while (LED_PWM > 17000)
   {
   }

   TCON &= ~0x10;                      // stop Timer 0 (no more updates to the
                                       // PCA duty cycle)
}

//-----------------------------------------------------------------------------
// End Of File
//-----------------------------------------------------------------------------
```



IoT Portfolio	SW/HW	Quality	Support and Community
www.silabs.com/IoT	*www.silabs.com/simplicity*	*www.silabs.com/quality*	*community.silabs.com*

Disclaimer

Silicon Labs intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Labs products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Labs reserves the right to make changes without further notice and limitation to product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Silicon Labs shall have no liability for the consequences of use of the information supplied herein. This document does not imply or express copyright licenses granted hereunder to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any Life Support System without the specific written consent of Silicon Labs. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Labs products are not designed or authorized for military applications. Silicon Labs products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons.

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com