

USING THE DAC AS A FUNCTION GENERATOR

Relevant Devices

This application note applies to the following devices: C8051F020, C8051F021, C8051F022, and C8051F023.

Introduction

This document describes how to implement an interrupt driven multifunction generator on C8051 devices using the on-chip digital-to-analog converter (DAC).

Features

- Four different waveforms expandable to any periodic function defined in a table.
 - Sine Wave (Table Defined)
 - Square Wave (Calculated)
 - Triangle Wave (Calculated)
 - Saw Tooth Wave (Calculated)
- Allows selection of the frequency and amplitude of waveform at run time.
- An interactive interface with a PC using the serial communications port and HyperTerminal or an equivalent program.

Key Points

- Output waveforms have 16-bit frequency resolution using the phase accumulator approach.
- The on-chip DAC's can support waveform generation up to 50 kHz.
- By using a 16-bit lookup table with a 12-bit DAC, error in the amplitude is virtually eliminated.

Implementation

The main routine of this program is a command interpreter that sets parameters for the Timer 4 interrupt service routine (ISR) which manages the DAC updates. The Timer 4 interrupts occur at a predetermined rate set at compile time. In the included software example, this value is stored in the constant `<SAMPLE_RATE_DAC>`. The Timer 4 ISR updates the DAC and calculates or looks up the next output value based on the waveform settings.

Setting up the DAC

Any free DAC, referred to as DACn, may be used to generate waveforms. In this example DACn is used in left-justified mode with output scheduling based on Timer 4 overflows. Refer to the data sheet for specific information on how to set the DACnCN register to specify DACn modes.

When the DAC is configured to left-justified mode, 16-bit data can be written to the 12-bit data register with no shifting required. In this case, the 4 least significant bits are ignored.

In this example, DACn updates occur on Timer 4 overflows, meaning writes to DACnH and DACnL have no immediate effect on the DAC output, but instead are held until the next Timer 4 overflow.

Another important note is that the internal voltage reference must be enabled by setting the appropriate bits in the REFnCN register before the DAC can be used.

Sampling Rate

The sampling rate is configured by initializing the Timer 4 reload value with the number of SYSCLK

cycles between interrupts. This number is negative because C8051 timers are up-counters and can be calculated using the following formula:

$$\text{Timer 4 Reload} = \frac{(-\text{SYSCLK})}{\text{SAMPLE_RATE_DAC}}$$

The maximum sampling rate allowed by the DAC is approximately 100 kHz, given by the 10 μ s output settling time. However, use caution when selecting the DAC sampling rate because all instructions in the longest path of the ISR must be executed before the next Timer 4 interrupt, or the output frequency will be affected. For example, using a SYSCLK of 22.1 MHz and a DAC update rate of 80 kHz allows 276 SYSCLK cycles for the ISR to finish execution. The main trade-off is between the sampling rate and the execution time of the Timer 4 ISR. One way execution time of the ISR can be reduced to achieve a higher sampling rate is by removing the gain adjustment stage. Also note that the maximum output frequency is limited to no more than one half the sampling rate (Nyquist theorem).

Waveform Generation

Waveform generation occurs entirely in the Timer 4 ISR and is implemented in three stages.

The 2D playing field, shown in Figure 1, is used to define one period of any periodic function. It has two 16-bit axes, a horizontal phase axis ranging

from 0 to 65535, and a vertical 2's complement amplitude axis ranging from -32768 to 32767.

All waveforms generated use a 16-bit phase accumulator which keeps track of where the output waveform is on the horizontal axis. This phase accumulator provides a frequency resolution of 1.2 Hz, given a DAC update rate of 80 kHz. Based on waveform settings, the first stage of Timer 4 ISR either calculates or looks up the next DAC output level corresponding to the phase accumulator. The phase accumulator is incremented by the variable <phase_add> every time the Timer 4 ISR is called. The magnitude of <phase_add> is determined by the desired output frequency based on this formula:

$$\text{phase_add} = \text{frequency} \times \frac{\text{PHASE_PRECISION}}{\text{SAMPLE_RATE_DAC}}$$

where PHASE_PRECISION = 65536

The entries in the lookup table and the results of the initial calculations are full-scale values. The second stage of the Timer 4 ISR scales the output level according to the <amplitude> parameter specified at the command prompt.

The final processing stage converts the scaled 2's complement value to an unsigned unipolar value prior to delivery to the DAC. This is accomplished by adding 32768 to the 2's complement

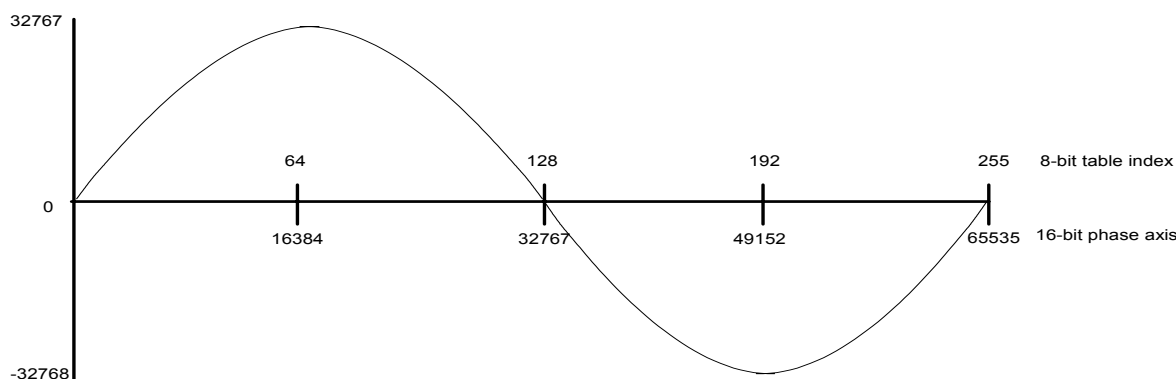


Figure 1. One Period of a Table Defined Sine Wave

value. An efficient way to implement this operation is to XOR the 2's complement value with 0x8000.

Table Defined Waveforms

As mentioned above, waveform generation consists of three stages before samples are written to the DAC. The output of the first stage, which determines the full scale output value, can either result from a calculation or a table lookup. A lookup table can be used if the output is not quickly or easily calculated. The main trade-off is sampling speed vs. code size.

Phase Error

Figure 1 shows one period of a sine wave. A lookup table containing 256 samples of this waveform is used to approximate a true sine wave. Keep in mind that the lookup table can approximate any other periodic waveform. If the output is set to "sine wave" at the command prompt, the Timer 4 ISR performs a lookup to obtain the output, using the eight most significant bits of the phase accumulator as the table index. The truncation to 8-bits introduces an error which can be interpreted as an instantaneous phase error or a slight error in the waveform amplitude. The frequency resolution, which is determined by the 16-bit accumulator, is not affected by the truncation because the error is not accumulated across multiple samples.

Amplitude Error

Amplitude error can be introduced from two sources, a low resolution amplitude or phase axis. Since the DAC has a 12-bit output resolution, error resulting from the amplitude axis can be eliminated by storing 16-bit values in the lookup table. Amplitude error that results from the phase axis can only be corrected by increasing the number of entries in the lookup table. Increasing the number of table entries will stabilize the instantaneous frequency by reducing the phase error, at the expense of increased code size.

Calculated Waveforms

Stage one of the Timer 4 ISR calculates the full scale output value of the waveform corresponding to the 16-bit phase accumulator. Since using the full 16-bit precision of the phase accumulator in the calculation does not require many clock cycles, both the amplitude and phase error are less than in table-defined waveforms.

Square Wave

The algorithm used to calculate the output value of the square wave is quite simple. As shown in Figure 2, if the phase accumulator is in the first half of the cycle, then the output is set to the maximum value of +32767. Otherwise, the output is set to the minimum value (-32768). The most significant bit of the phase accumulator contains enough information to determine the output value of the square wave.

Triangle Wave

The calculation of a triangle wave involves the equation of 2 lines with opposite slope. From Figure 3, the slope is +2 in the first half and -2 in the second half.

Saw Tooth Wave

The equation of a saw tooth wave is a straight line with a slope of 1. Figure 4 shows one period of a full scale saw tooth wave.

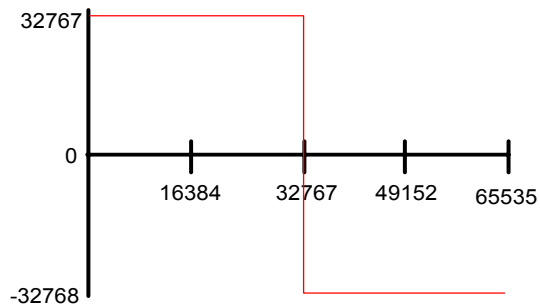


Figure 2. One period of a calculated square wave

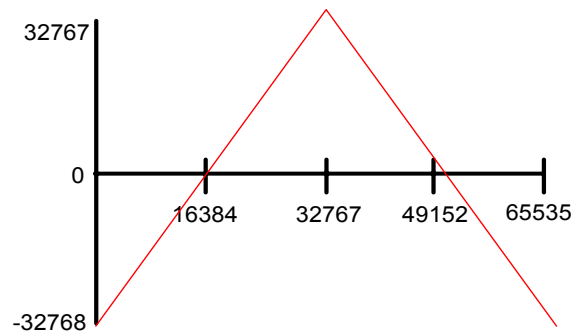


Figure 3. One period of a calculated triangle wave

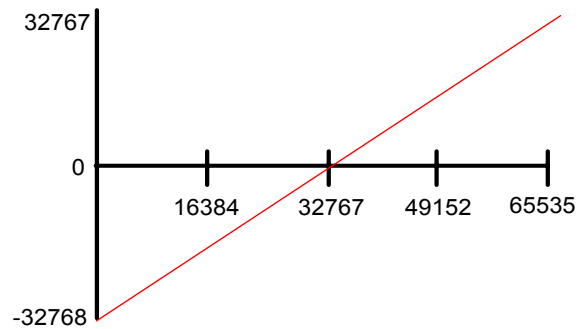


Figure 4. One period of a calculated saw tooth wave

Software Example

```
//-----
// DAC1_fgen1.c
//-----
//
// AUTH: BW,FB
// DATE: 2 OCT 01
//
// Target: C8051F02x
// Tool chain: KEIL C51
//
// Description:
//   Example source code which outputs waveforms on DAC1. DAC1's output is
//   scheduled to update at a rate determined by the constant
//   <SAMPLE_RATE_DAC>, managed and timed by Timer4.
//
//   Implements a 256-entry full-cycle sine table of 16-bit precision. Other
//   waveforms supported are square, triangle, and saw tooth.
//
//   The output frequency is determined by a 16-bit phase adder.
//   At each DAC update cycle, the phase adder value is added to a running
//   phase accumulator, <phase_accumulator>, the upper bits of which are used
//   to access the sine lookup table.
//
//   The program is controlled through UART using HyperTerminal running on a
//   PC. All commands are two characters in length and have optional
//   frequency and amplitude arguments. Note that the amplitude parameter
//   cannot be specified unless the frequency is also specified.
//
// Command Format:
//
//   XX [frequency] [amplitude]
//
//   where XX denotes the command
//
// Command List:
//
//   SQ - Square Wave
//   SI - Sine Wave
//   TR - Triangle Wave
//   SA - Saw Tooth Wave
//   OF - Output OFF
//   ?? - Help
//
//-----
// Includes
//-----

#include <c8051f020.h>           // SFR declarations
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

//-----
// 16-bit SFR Definitions for 'F02x
//-----
```

```
sfr16 DP          = 0x82;          // data pointer
sfr16 TMR3RL      = 0x92;          // Timer3 reload value
sfr16 TMR3        = 0x94;          // Timer3 counter
sfr16 ADC0        = 0xbe;          // ADC0 data
sfr16 ADC0GT      = 0xc4;          // ADC0 greater than window
sfr16 ADC0LT      = 0xc6;          // ADC0 less than window
sfr16 RCAP2       = 0xca;          // Timer2 capture/reload
sfr16 T2          = 0xcc;          // Timer2
sfr16 RCAP4       = 0xe4;          // Timer4 capture/reload
sfr16 T4          = 0xf4;          // Timer4
sfr16 DAC0        = 0xd2;          // DAC0 data
sfr16 DAC1        = 0xd5;          // DAC1 data

//-----
// Function PROTOTYPES
//-----

void main (void);
void SYSCLK_Init (void);
void PORT_Init (void);
void UART0_Init (void);

void Timer4_Init (int counts);
void Timer4_ISR (void);
long pow(int x, int y);
void Print_Command_List(void);

void Sine(void);
void Square(void);
void Triangle(void);
void Saw(void);
void Off(void);
void Help(void);
void Error(void);

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

#define SYSCLK          22118400      // SYSCLK frequency in Hz

#define BAUDRATE        9600          // Baud rate of UART in bps

#define SAMPLE_RATE_DAC 80000L        // DAC sampling rate in Hz

#define PHASE_PRECISION 65536         // range of phase accumulator

#define command_length 2              // command length is 2 characters
#define command_size 3               // command size is 3 bytes

typedef struct Command_Table_Type {    // when a command is entered, it is
    char command[command_size];       // compared to the command field of
    void (*function_ptr)(void);       // of the table. If there is a match
}Command_Table_Type;                 // then the the function located at
                                    // function_ptr will be executed

typedef enum Waveform {               // the different possible output
    SQUARE,                          // waveforms
```

```

    SINE,
    TRIANGLE,
    SAW,
    OFF
}Waveform;

typedef union lng {
    long Long;
    int Int[2];
} lng;

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

unsigned long frequency = 1000;          // frequency of output in Hz,
                                         // defaults to 1000 Hz

unsigned int phase_add = 1000 * PHASE_PRECISION / SAMPLE_RATE_DAC;

unsigned int amplitude = 100 * 655;      // 655 is a scaling factor
                                         // see the Timer 4 ISR

Waveform output_waveform = OFF;

char input_str[16]= "";

#define num_commands 6
Command_Table_Type code function_table[num_commands + 1] = {
    {"SQ", Square},
    {"SI", Sine},
    {"TR", Triangle},
    {"SA", Saw},
    {"OF", Off},
    {"??", Help},
    {"", Error}
};

// a full cycle, 16-bit, 2's complement sine wave lookup table
int code SINE_TABLE[256] = {

    0x0000, 0x0324, 0x0647, 0x096a, 0x0c8b, 0x0fab, 0x12c8, 0x15e2,
    0x18f8, 0x1c0b, 0x1f19, 0x2223, 0x2528, 0x2826, 0x2b1f, 0x2e11,
    0x30fb, 0x33de, 0x36ba, 0x398c, 0x3c56, 0x3f17, 0x41ce, 0x447a,
    0x471c, 0x49b4, 0x4c3f, 0x4ebf, 0x5133, 0x539b, 0x55f5, 0x5842,
    0x5a82, 0x5cb4, 0x5ed7, 0x60ec, 0x62f2, 0x64e8, 0x66cf, 0x68a6,
    0x6a6d, 0x6c24, 0x6dca, 0x6f5f, 0x70e2, 0x7255, 0x73b5, 0x7504,
    0x7641, 0x776c, 0x7884, 0x798a, 0x7a7d, 0x7b5d, 0x7c29, 0x7ce3,
    0x7d8a, 0x7e1d, 0x7e9d, 0x7f09, 0x7f62, 0x7fa7, 0x7fd8, 0x7ff6,
    0x7fff, 0x7ff6, 0x7fd8, 0x7fa7, 0x7f62, 0x7f09, 0x7e9d, 0x7e1d,
    0x7d8a, 0x7ce3, 0x7c29, 0x7b5d, 0x7a7d, 0x798a, 0x7884, 0x776c,
    0x7641, 0x7504, 0x73b5, 0x7255, 0x70e2, 0x6f5f, 0x6dca, 0x6c24,
    0x6a6d, 0x68a6, 0x66cf, 0x64e8, 0x62f2, 0x60ec, 0x5ed7, 0x5cb4,
    0x5a82, 0x5842, 0x55f5, 0x539b, 0x5133, 0x4ebf, 0x4c3f, 0x49b4,
    0x471c, 0x447a, 0x41ce, 0x3f17, 0x3c56, 0x398c, 0x36ba, 0x33de,
    0x30fb, 0x2e11, 0x2b1f, 0x2826, 0x2528, 0x2223, 0x1f19, 0x1c0b,

```

```
0x18f8, 0x15e2, 0x12c8, 0x0fab, 0x0c8b, 0x096a, 0x0647, 0x0324,
0x0000, 0xfcdc, 0xf9b9, 0xf696, 0xf375, 0xf055, 0xed38, 0xeale,
0xe708, 0xe3f5, 0xe0e7, 0xdddd, 0xdad8, 0xd7da, 0xd4e1, 0xd1ef,
0xcf05, 0xcc22, 0xc946, 0xc674, 0xc3aa, 0xc0e9, 0xbe32, 0xbb86,
0xb8e4, 0xb64c, 0xb3c1, 0xb141, 0xaecd, 0xac65, 0xaa0b, 0xa7be,
0xa57e, 0xa34c, 0xa129, 0x9f14, 0x9d0e, 0x9b18, 0x9931, 0x975a,
0x9593, 0x93dc, 0x9236, 0x90a1, 0x8f1e, 0x8dab, 0x8c4b, 0x8afc,
0x89bf, 0x8894, 0x877c, 0x8676, 0x8583, 0x84a3, 0x83d7, 0x831d,
0x8276, 0x81e3, 0x8163, 0x80f7, 0x809e, 0x8059, 0x8028, 0x800a,
0x8000, 0x800a, 0x8028, 0x8059, 0x809e, 0x80f7, 0x8163, 0x81e3,
0x8276, 0x831d, 0x83d7, 0x84a3, 0x8583, 0x8676, 0x877c, 0x8894,
0x89bf, 0x8afc, 0x8c4b, 0x8dab, 0x8f1e, 0x90a1, 0x9236, 0x93dc,
0x9593, 0x975a, 0x9931, 0x9b18, 0x9d0e, 0x9f14, 0xa129, 0xa34c,
0xa57e, 0xa7be, 0xaa0b, 0xac65, 0xaecd, 0xb141, 0xb3c1, 0xb64c,
0xb8e4, 0xbb86, 0xbe32, 0xc0e9, 0xc3aa, 0xc674, 0xc946, 0xcc22,
0xcf05, 0xd1ef, 0xd4e1, 0xd7da, 0xdad8, 0xdddd, 0xe0e7, 0xe3f5,
0xe708, 0xeale, 0xed38, 0xf055, 0xf375, 0xf696, 0xf9b9, 0xfcdc,
};

code char string0[] = "\n\n*** OUTPUT IS NOW A ";
code char string1[] = "\n\n-----\n\n";

//-----
// MAIN Routine
//-----

void main (void) {

    char i;                                // counting variable
    char* arg_ptr1;                        // pointers to command line parameters
    char* arg_ptr2;

    long temp_frequency;                   // used to hold the values input from the
    int temp_amplitude;                    // keyboard while they are error checked

    int printed_amplitude = 100;           // a separate copy of amplitude because
                                           // temp_amplitude is written over

    void (*f)(void);                       // function pointer used to call the proper
                                           // function from the command table

    WDTCN = 0xde;                          // Disable watchdog timer
    WDTCN = 0xad;

    SYSCLK_Init ();
    PORT_Init ();

    // initializations for wave generation
    REF0CN = 0x03;                         // enable internal VREF generator
    DAC1CN = 0x97;                         // enable DAC1 in left-justified mode

    Timer4_Init(SYSCLK/SAMPLE_RATE_DAC);
                                           // using Timer4 as update scheduler
                                           // initialize T4 to update DAC1
                                           // after (SYSCLK cycles)/sample have
                                           // passed.

    // initialization for command input
```



```
UART0_Init ();

EA = 1;                                // Enable global interrupts

Print_Command_List();

while(1){

    // get user input
    printf ("ENTER A COMMAND:>");
    gets(input_str,sizeof(input_str));    // wait for input
    input_str[0] = toupper(input_str[0]); // convert the two characters
    input_str[1] = toupper(input_str[1]); // in the command to uppercase

    // Parse the command
    for (i = 0; i < num_commands; i++){

        // strcmp() returns 0 if the first two arguments are the same string
        // set <i> for the command that matched
        if (0 == strcmp(input_str, function_table[i].command, command_length)){

            arg_ptr1 = strchr (input_str, ' ');
            arg_ptr1++;                // point to the frequency

            arg_ptr2 = strchr(arg_ptr1, ' ');
            arg_ptr2++;                // point to amplitude

            temp_frequency = atol(arg_ptr1);
            temp_amplitude = atol(arg_ptr2);

            // check to make sure entered frequency is valid
            if (temp_frequency) {

                frequency = temp_frequency;

            } else {

                printf("\n** Frequency will not change\n");
            }

            // check to make sure entered amplitude is valid
            if ((temp_amplitude > 0) && (temp_amplitude <=100)){

                // multiply by 655 to be divided by 65535 (16-bit shift) in the
                // ISR; this is an optimization to reduce the number of
                // instructions executed in the ISR
                amplitude = temp_amplitude * 655;

                printed_amplitude = temp_amplitude;

            } else {

                printf("\n** Amplitude will not change\n");
            }

        }

    }

}
```

```
printf("\nFREQUENCY: %ld Hz", frequency);
printf("\nAMPLITUDE: %d %% of VREF/2", printed_amplitude);

EA = 0;                                // Disable Interrupts to avoid
                                        // contention between the ISR
                                        // and the following code.

// set the frequency
phase_add = frequency * PHASE_PRECISION / SAMPLE_RATE_DAC;

    break;
} // end if

} // end for

// call the associated function
f = (void *) function_table[i].function_ptr;
f();

EA = 1;                                // re-enable interrupts

} // end while(1)

} // end main

//-----
// Init Routines
//-----

//-----
// SYSCLK_Init
//-----
//
// This routine initializes the system clock to use a 22.1184MHz crystal
// as its clock source.
//
void SYSCLK_Init (void)
{
    int i;                                // delay counter

    OSCXCN = 0x67;                        // start external oscillator with
                                        // 22.1184MHz crystal

    for (i=0; i < 256; i++) ;            // Wait for osc. to start up

    while (!(OSCXCN & 0x80)) ;            // Wait for crystal osc. to settle

    OSCICN = 0x88;                        // select external oscillator as SYSCLK
                                        // source and enable missing clock
                                        // detector
}

//-----
// PORT_Init
//-----
//
// Configure the Crossbar and GPIO ports
```

```

//
void PORT_Init (void)
{
    XBR0      = 0x04;          // Enable UART0
    XBR1      = 0x00;
    XBR2      = 0x40;          // Enable crossbar and weak pull-up
    POMDOUT |= 0x01;          // Set TX0 pin to push-pull
}

//-----
// Timer4_Init
//-----
// This routine initializes Timer4 in auto-reload mode to generate interrupts
// at intervals specified in <counts>.
//
void Timer4_Init (int counts)
{
    T4CON = 0;                // STOP timer; set to auto-reload mode
    CKCON |= 0x40;            // T4M = '1'; Timer4 counts SYSCLKs
    RCAP4 = -counts;          // set reload value
    T4 = RCAP4;
    EIE2 |= 0x04;            // enable Timer4 interrupts
    T4CON |= 0x04;            // start Timer4
}

//-----
// UART0_Init
//-----
//
// Configure the UART0 using Timer1, for <baudrate> and 8-N-1.
//
void UART0_Init (void)
{
    SCON0 = 0x50;            // SCON0: mode 1, 8-bit UART, enable RX
    TMOD = 0x20;             // TMOD: timer 1, mode 2, 8-bit reload
    TH1 = -(SYSCLK/BAUDRATE/16); // set Timer1 reload value for baudrate
    TR1 = 1;                 // start Timer1
    CKCON |= 0x10;           // Timer1 uses SYSCLK as time base
    PCON |= 0x80;            // SMOD0 = 1
    TI0 = 1;                 // Indicate TX0 ready
}

//-----
// Print_Command_List
//-----
//
// Prints the command list to the standard output.
//
void Print_Command_List (void)
{
    printf ("\n\
SQ - Square Wave\n\
SI - Sine Wave\n\
TR - Triangle Wave\n\
SA - Saw Tooth Wave\n\
OF - Output OFF\n\
?? - Help\n\n");
}

```

```
//-----  
// Sine  
//-----  
//  
// Sets output to a sine wave.  
//  
void Sine (void)  
{  
    output_waveform = SINE;  
    // print this message: *** OUTPUT IS NOW A SINE WAVE  
    printf ("%sSINE WAVE%s",string0,string1);  
    Print_Command_List();  
  
}  
  
//-----  
// Square  
//-----  
//  
// Sets output to a square wave.  
//  
void Square (void)  
{  
    output_waveform = SQUARE;  
    // print this message: *** OUTPUT IS NOW A SQUARE WAVE  
    printf ("%sSQUARE WAVE%s",string0,string1);  
    Print_Command_List();  
  
}  
  
//-----  
// Triangle  
//-----  
//  
// Sets output to a triangle wave.  
//  
void Triangle (void)  
{  
    output_waveform = TRIANGLE;  
    // print this message: *** OUTPUT IS NOW A TRIANGLE WAVE  
    printf ("%sTRIANGLE WAVE%s",string0,string1);  
    Print_Command_List();  
  
}  
  
//-----  
// Saw  
//-----  
//  
// Sets output to a saw tooth wave.  
//  
void Saw (void)  
{  
    output_waveform = SAW;  
    // print this message: *** OUTPUT IS NOW A SAW TOOTH WAVE  
    printf ("%sSAW TOOTH WAVE",string0,string1);  
    Print_Command_List();  
  
}
```

```

//-----
// Off
//-----
//
// Sets output to zero volts DC.
//
void Off (void)
{
    printf ("\n\n*** OUTPUT OFF",string1);
    output_waveform = OFF;
    Print_Command_List();
}

//-----
// Help
//-----
//
// Prints the command list.
//
void Help (void)
{
    Print_Command_List();
}

//-----
// Error
//-----
//
// Indicates that an invalid command was entered at the command prompt.
//
void Error(void)
{
    printf ("    ***INVALID INPUT = %s\n", input_str);
}

//*****
// Interrupt Handlers
//*****

//-----
// Timer4_ISR -- Wave Generator
//-----
//
// This ISR is called on Timer4 overflows. Timer4 is set to auto-reload mode
// and is used to schedule the DAC output sample rate in this example.
// Note that the value that is written to DAC1 during this ISR call is
// actually transferred to DAC1 at the next Timer4 overflow.
//
void Timer4_ISR (void) interrupt 16 using 3
{
    static unsigned phase_acc = 0; // holds phase accumulator

    int temp1;                    // the temporary value that passes
                                // through 3 stages before being written
                                // to DAC1

```

```
int code *table_ptr;           // pointer to the lookup table

lng temporary_long;           // holds the result of a 16-bit multiply

T4CON &= ~0x80;               // clear T4 overflow flag

table_ptr = SINE_TABLE;

phase_acc += phase_add;       // increment phase accumulator

// set the value of <temp1> to the next output of DAC1 at full-scale
// amplitude; the rails are +32767, -32768
switch (output_waveform){

    case SINE:

        // read the table value
        temp1 = *(table_ptr + (phase_acc >> 8));

        break;

    case SQUARE:

        // if in the first half-period, then high
        if ( (phase_acc & 0x8000) == 0 ) {

            temp1 = 32767;

        } else {

            temp1 = -32768;

        }

        break;

    case TRIANGLE:

        // in first half-period, then y = mx + b
        if ( (phase_acc & 0x8000) == 0 ) {

            temp1 = (phase_acc << 1) - 32768;

        } else, in the second half of period
        } else {

            temp1 = -(phase_acc << 1) + 32767;

        }

        break;

    case SAW:

        temp1 = phase_acc - 32768;
        break;

    case OFF:
```

```
        templ = -32768;
        break;

    default:
        while(1);
}

// Adjust the Gain
temporary_long.Long = (long) ((long)templ * (long)amplitude);

templ = temporary_long.Int[0]; // same as temporary_long >> 16

// Add a DC bias to make the rails 0 to 65535
// Note: the XOR with 0x8000 translates the bipolar quantity into
// a unipolar quantity.
DAC1 = 0x8000 ^ templ;
}
```

Contact Information

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

The information in this document is believed to be accurate in all respects at the time of publication but is subject to change without notice. Silicon Laboratories assumes no responsibility for errors and omissions, and disclaims responsibility for any consequences resulting from the use of information included herein. Additionally, Silicon Laboratories assumes no responsibility for the functioning of undescribed features or parameters. Silicon Laboratories reserves the right to make changes without further notice. Silicon Laboratories makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Silicon Laboratories assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. Silicon Laboratories products are not designed, intended, or authorized for use in applications intended to support or sustain life, or for any other application in which the failure of the Silicon Laboratories product could create a situation where personal injury or death may occur. Should Buyer purchase or use Silicon Laboratories products for any such unintended or unauthorized application, Buyer shall indemnify and hold Silicon Laboratories harmless against all claims and damages.

Silicon Laboratories and Silicon Labs are trademarks of Silicon Laboratories Inc.

Other products or brandnames mentioned herein are trademarks or registered trademarks of their respective holders.