Analog – Sine Wave Generation with PSoC® 1 (Demonstration with CTCSS)

Author: Jeff Dahlin

Associated Project: Yes
Associated Part Family: CY8C23x33, CY8C24x23A, CY8C24x94 CY8C27x43, CY8C28xxx, CY8C29x66
Software: PSoC Designer 5.4
Related Application Notes: None
Help us improve! To send feedback: click here

AN2025 demonstrates how to generate a sine wave using two methods, lookup table and filtering, with PSoC® 1. The document also shows how to implement a Continuous Tone Coded Squelch System (CTCSS) carrier generator in PSoC 1. There are three projects associated with this document. The first two show how to generate a sine wave using the lookup table and filtering methods, and the third project demonstrates CTCSS implementation.

Introduction

AN2025 discusses two methods to generate sine waves in PSoC 1: DAC and filtering. In the DAC method, a counter is used to generate periodic interrupts. In the interrupt service routine (ISR), the DAC is updated from a lookup table (LUT), which has the values of a sine wave sampled at equal intervals. The periodic updating of the DAC with sine wave values causes the output of the DAC to be a sine wave.

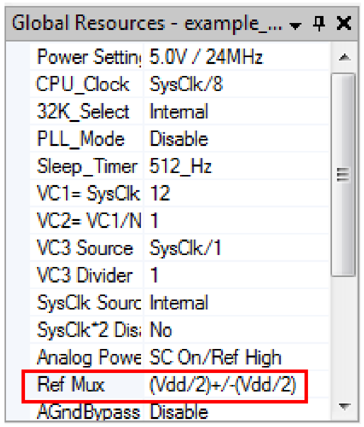
Another method is by filtering a square wave to get a sine wave. A square wave with frequency equal to the desired sine wave frequency is generated using timers. This square wave is then passed through a band pass filter (BPF) to eliminate the higher-order harmonics. Only the fundamental frequency, which is the desired sine wave, is untouched.

The application note then discusses how to implement a Continuous Tone Coded Squelch System (CTCSS) by using the DAC method. This particular example uses a 256-byte sine wave lookup table in ROM and a 6-bit Voltage Output Digital to Analog Convertor (DAC6) User Module (UM).

Sine Wave Generation using DAC

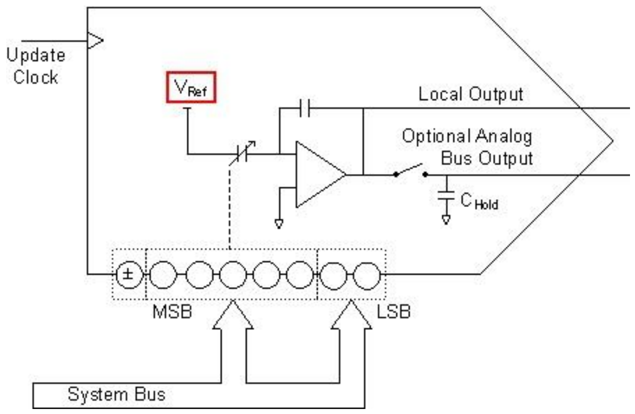
The DAC8 User Module translates digital codes into output voltages. The input digital codes can be represented as numbers in 2's complement form, ranging from −127 to +127 or, alternatively, in offset-binary form, as numbers ranging from 0 to 254. Several output voltage ranges are possible, depending on the value selected for a system-level parameter called RefMux. See the Global Resources parameter window of PSoC Designer (see Figure 1). RefMux sets the Vref of DAC, as shown in Figure 2.

Figure 1. Global Resources Parameter Window


Global Resources - example_...

Power Settin	5.0V / 24MHz
CPU_Clock	SysClk/8
32K_Select	Internal
PLL_Mode	Disable
Sleep_Timer	512_Hz
VC1= SysClk	12
VC2= VC1/N	1
VC3 Source	SysClk/1
VC3 Divider	1
SysClk Sourc	Internal
SysClk*2 Dis	No
Analog Powe	SC On/Ref High
Ref Mux	(Vdd/2)+/-(Vdd/2)
AGndBypass	Disable

Figure 2. DAC8 Block Diagram



In the example project associated with this application note, RefMux is set to (VDD/2) ± (VDD/2). Say VDD = 5 V.

Then Vref = 2.5 V and the output will swing from 0 V (2.5 V to 2.5 V) to 5 V (2.5 V to +2.5 V). The input data format is set to offset binary (0 to 254).

Table 1. DAC Input/Output Table

SNo	DAC Input	Output Voltage(volts)
1	0x00 (0)	0V
2	0x7F (127)	2.5V
3	0xFE (254)	5V

For a sampled system, the sample rate required to produce the desired output frequency is given by the following formula:

$$Sample\ Rate = Output\ Frequency\ x\ No. of\ Samples$$

Let us consider an example to generate a 60-Hz sine wave using 64 samples. For an output frequency of 60 Hz and 64 samples, the desired sample rate is 3.84 ksps.

Using the formula, the desired output frequency may be generated by varying the sample rate or the number of samples. Note that reducing the number of samples increases the error in the output sine wave.

A 64-point LUT is created, which resembles a sine wave. The formula to create the samples of the LUT is:

$$Value[n] = \left(\sin\left[n\ x\ \frac{2\pi}{64}\right] x\ Full\ Scale\right) + \ Zero$$

Where,

n = Sample number (0 to 63)

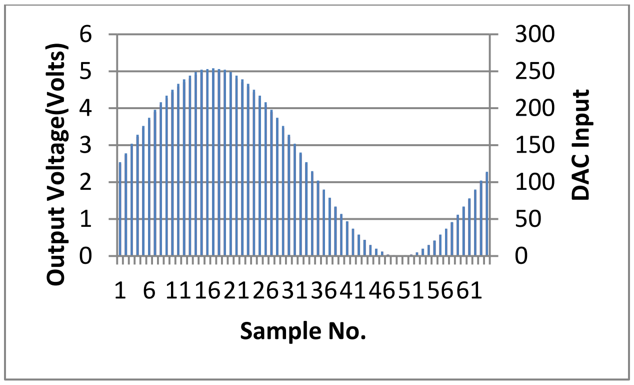
$Full\ Scale$ = $\pm$Full scale count for the DAC wrt Vref(2.5V), which is 127 for DAC8

$Zero$ = Value at which DAC produces a zero cross voltage of sine wave(2.5 V), which is 127 for DAC8

Using the formula given earlier, the following LUT is created and stored as a ROM array in *main.c*.

```
const char SineTable64[] = {
127,  139,  152,  164,  176,  187,  198,  208,
217, 225, 233, 239, 244, 249, 252, 253,
254,  253,  252,  249,  244,  239,  233,  225,
217, 208, 198, 187, 176, 164, 152, 139,
127, 115, 102,  90,  78,  67,  56,  46,
 37,  29,  21,  15,  10,   5,   2,   1,
  0,   1,   2,   5,  10,  15,  21,  29,
 37,  46,  56,  67,  78,  90, 102,  115
};  /* 64 samples store in ROM */
```

Figure 3. Output Waveform



Calculate the maximum output frequency achievable using the 8-bit DAC as follows:

$$Max\ Out\ Frequency = \frac{Max\ sample\ rate}{Number\ of\ samples}$$

Maximum Sample rate for an 8-bit DAC = 125 Kbps

Maximum No. of samples = 256

Maximum Output Frequency achievable = 488 Hz

Firmware

After reset, all hardware settings from the device configuration are loaded into the device, and *main.c* is executed. The following operations are performed in *main.c*:

- Counter16 is started.
- Counter16 interrupt is enabled.
- DAC8 is started at high power.
- Global interrupts are enabled. Any interrupt in the code will be serviced only if the global interrupt is enabled.
- The pointer to the LUT is initialized to 0.
- An infinite loop is entered. After that, all the operations take place inside the Counter's ISR.

Reading from the LUT and writing to the DAC8 take place inside the Counter's ISR. The function Counter_ISR in *main.c* performs the operations. This function is declared as an interrupt handler by using the following code.

```
#pragma interrupt_handler Counter_ISR;

The #pragma interrupt_handler is use to
make a "C" function as an ISR function.

The ISR will be:
```

```
void Counter_ISR(void)
void Counter_ISR(void)
{
       /* Update the DAC with the value in
       * lookup table pointed the variable
       * Pointer */

DAC8_1_WriteBlind(SineTable64[Pointer]);

    /* Increment pointer */
    Pointer++;
    /* If the Pointer is incremented by
    * 2, the effective number of samples
    * will be 32 and the output frequency
    * will be doubled.If Pointer is
    * incremented by 4,number of samples
    * will be 16 and the output frequency
    * will be quadrupled. */


    /* Reset Pointer if greater than or
equal to 64 */
    if (Pointer >= 64) Pointer = 0;
}
```

To execute the function on interrupt, a jump to this function must be done inside the ISR. This is done by placing the following code inside the _Counter16_1_ISR inside the *Counter16_1INT.asm* file.

```
_Counter16_1_ISR:

   ;@PSoC_UserCode_BODY@ (Do not change
   ;this line.)
   ;---------------------------------------
   ; Insert your custom code below this
    ; banner
   ;---------------------------------------
   ; NOTE: interrupt service routines
    ; must preserve the values of the A
and
    ; X CPU registers.

   ljmp _Counter_ISR
   ;---------------------------------------
   ; Insert your custom code above this
   ; banner
   ;---------------------------------------
   ;@PSoC_UserCode_END@ (Do not change this
    ;line.)

   reti
```

When the counter interrupt occurs, the *boot.asm* redirects the interrupt to the _Counter16_1_ISR. From there, the control is transferred to the interrupt handler in *main.c*. The following operations take place inside the ISR:

- The value from the LUT that corresponds to the pointer is read
- The DAC is updated with this value
- The Pointer is incremented
- If Pointer = 64, Pointer is reset to 0, so that the next cycle starts from the first sample in the LUT.

Figure 4. DAC Update Routine

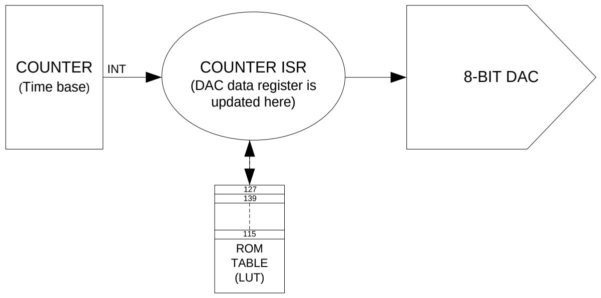
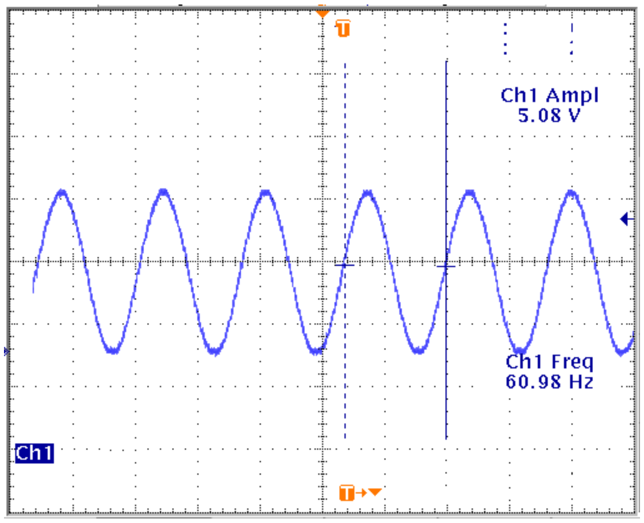


Figure 5. Scope Capture of Output Observed on P0[3]



Design Considerations

- The same 64-point LUT can be used for smaller tables, such as ones with 32 samples, 16 samples, and 8 samples. For 32 samples, inside the ISR, increment the Pointer by 2 instead of 1. For 16 samples, increment the Pointer by 4.
- The higher the number of samples, the smoother the output wave and the less the harmonics. But as the desired output frequency becomes higher, the sample rate also increases. When the time taken to execute the Counter's ISR exceeds the sample time, the CPU load becomes 100%. At this point, the number of samples must be reduced to further increase the output frequency.

- To smooth the output signal when samples per cycle decreases, add an RC Filter with corner frequency (fc) above the fundamental frequency and below the sampling frequency.

  Example: Fundamental Frequency = 60 Hz, Sampling Freq = 120 Hz. Consider a corner frequency = 80 Hz

Then, we can design an RC low pass filter (LPF) with R= 2 kΩ and C = 1 uf.

Sine Wave Generation Using Filter

This method involves filtering techniques to obtain a sine wave from a square wave. This method uses the hardware resources of PSoC 1, leaving the CPU free to implement other functions. It is easier to generate a square wave of the required frequency using the PWM UM in PSoC 1. According to Fourier analysis, this square wave consists of the fundamental frequency (sine wave) and its odd harmonics of diminishing amplitudes.

The amplitude of the nth harmonic is (1/n) times the amplitude of the fundamental frequency. This square wave is then passed through a BPF, whose cutoff frequencies are selected such that only the fundamental frequency is passed un-attenuated and the odd harmonics are attenuated significantly. The output of the BPF is the fundamental frequency, which is the required sine wave.

Implementation

Let us see how this is implemented in PSoC 1. The project consists of the following UMs:

- **Center_Frequency:** A 16-bit PWM UM used to generate the required square wave. In this case, it is set to generate a 5-kHz sine wave.
- **BPF4_1 :** A four-pole BFP (BPF4 UM) is used to achieve the necessary filtering.
- **PGA_1:** A PGA UM is used for amplifying or attenuating the input signal to BPF4_1.
- **Column_Clock:** A 16-bit PWM UM is used to generate the column clock required for the BPF4_1 UM (per BPF4_1 UM wizard).

Figure 6. UM for Generating Centre Frequency and Clocking Signal

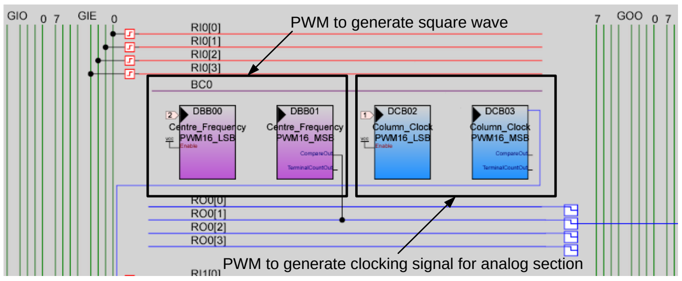
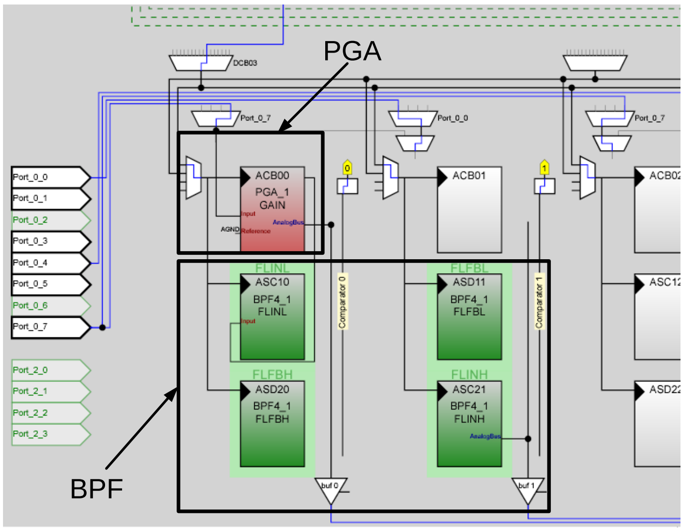


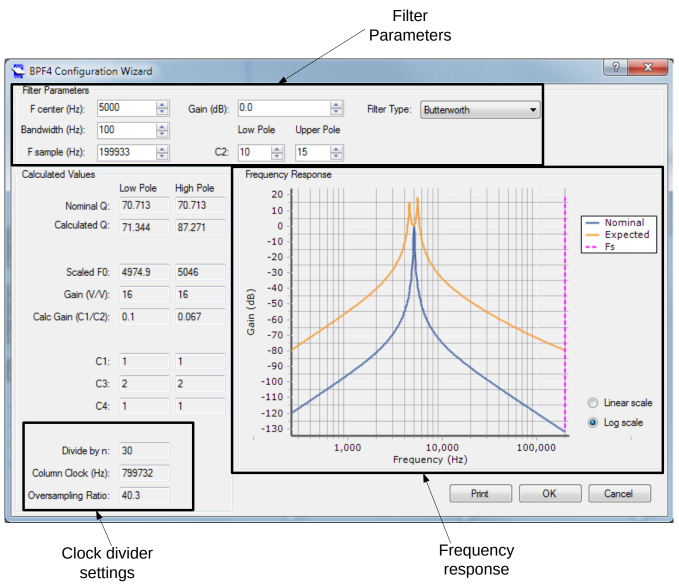
Figure 7. PGA and BPF to Obtain Sine Wave from Square Wave



Configuring the BPF UM

The BPF4 User Module implements a four-pole BPF. The center frequency and Q (ratio of center frequency to bandwidth) are functions of the clock frequency and the ratios of the capacitor values chosen. You can set or adjust the center frequency by controlling the sample rate clock. Any of the classical all-pole filter configurations (Butterworth, Gaussian, Bessel, and Chebyshev) can be implemented. The filter output can drive the analog output bus.

Figure 8. BPF4 Configuration Wizard



Filter Parameters

F center (Hz):	5000	Gain (dB):	0.0	Filter Type:	Butterworth
Bandwidth (Hz):	100		Low Pole	Upper Pole	
F sample (Hz):	199933	C2:	10	15	

Calculated Values

	Low Pole	High Pole
Nominal Q:	70.713	70.713
Calculated Q:	71.344	87.271
Scaled F0:	4974.9	5046
Gain (V/V):	16	16
Calc Gain (C1/C2):	0.1	0.067
C1:	1	1
C3:	2	2
C4:	1	1

Divide by n:	30
Column Clock (Hz):	799732
Oversampling Ratio:	40.3

Linear scale [✓ Log scale]

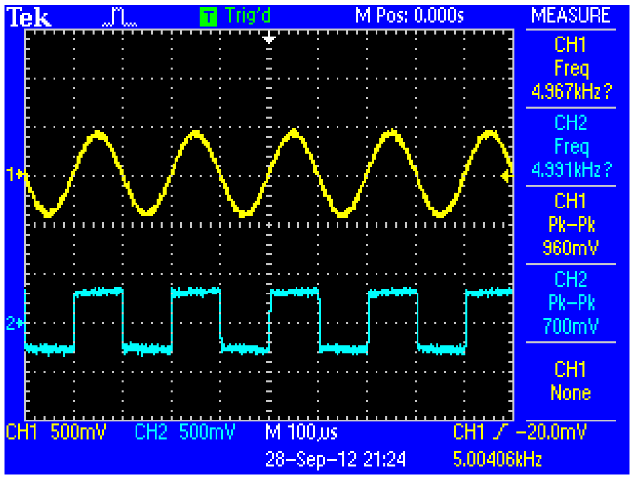
Figure 8 shows the BPF4 configuration wizard. Based on the design requirements, you can set filter parameters such as centre frequency, bandwidth, sampling frequency, gain of the filter, and filter type.

The clock divider setting shows the clock parameters that must be set for the analog column to obtain the expected frequency response. The frequency response plots show the nominal frequency response and the expected frequency response.

Because of the discrete nature of the capacitance, there is a difference between the nominal frequency response and the expected frequency response. The filter parameter setting should be set such that the expected frequency response matches the design requirements. In the current example project, which is designed to obtain a 5-kHz sine wave, the filter parameter settings satisfy this condition. The slight gain provided by the filter is compensated by using same amount of attenuation in the PGA.

After all of the UMs are configured, the firmware simply consists of starting all the UMs. Connect P0[1] to P0[7] externally, and a 5-kHz sine wave is obtained on P0[5], as shown in Figure 9. Change the output frequency by changing frequency of center frequency PWM. Also, change the center frequency in the BPF UM wizard.

Figure 9. 5-kHz Square Wave and the Filtered Sine Wave



Advantages

1) Implementation is purely in hardware. The CPU can be used to implement any other functionality.
2) Obtain higher-frequency sine waves that cannot be obtained by the DAC method because of the limitation of the maximum DAC update rate and CPU overhead.

Disadvantages

1) Because of the discrete values of the capacitance, you may have to use a trial-and-error method to achieve the right expected filter response.

Continuous Tone Coded Squelch System (CTCSS)

Continuous Tone Coded Squelch System (CTCSS) or similar mechanisms are used in most handheld radios to allow multiple users to share one carrier frequency. The goal of CTCSS is to allow the receiving radio to suppress (or squelch) signals that are not intended for its user. If a pair of radios is set to the same CTCSS tone, then the audio transmitted by one radio is received on the other radio's speaker.

CTCSS works by mixing a single tone with the transmitted voice audio at all times. A radio receiving a signal checks if the CTCSS tone selected for that radio is present. If the tone is present, the receiving radio outputs the voice audio. If the tone is not present, any signal received is not sent to the speaker. This allows multiple radios to coexist on the same carrier frequency in the same area without users having to listen to everyone.

CTCSS Carrier Frequencies

CTCSS uses 38 frequencies between 67.0 Hz and 250.3 Hz as the selection tones. Table 2 lists the frequency associated with each of the tones. Some radios use additional "nonstandard" frequencies for CTCSS functionality. This application uses only the standard CTCSS frequencies.

Table 2. Standard CTCSS Frequencies

Tone	Frequency	Tone	Frequency
1	67.0 Hz	20	131.8 Hz
2	71.9 Hz	21	136.5 Hz
3	74.4 Hz	22	141.3 Hz
4	77.0 Hz	23	146.2 Hz
5	79.7 Hz	24	151.4 Hz
6	82.5 Hz	25	156.7 Hz
7	85.4 Hz	26	162.2 Hz
8	88.5 Hz	27	167.9 Hz
9	91.5 Hz	28	173.8 Hz
10	94.8 Hz	29	179.9 Hz
11	97.4 Hz	30	186.2 Hz
12	100.0 Hz	31	192.8 Hz
13	103.5 Hz	32	203.5 Hz
14	107.2 Hz	33	210.7 Hz
15	110.9 Hz	34	218.1 Hz
16	114.8 Hz	35	225.7 Hz
17	118.8 Hz	36	233.6 Hz
18	123.0 Hz	37	241.8 Hz
19	127.3 Hz	38	250.3 Hz

Frequency Generation – Implementation

In the project accompanying this application note, the CTCSS frequencies are generated by transferring data from a 256-entry, 8-bit LUT into a DAC6 UM at a fixed rate. The DAC is updated at a constant rate in an ISR that is controlled by an 8-bit timer (Timer8 UM). The frequency of the CTCSS tone is adjusted by varying the step size through the LUT.

The 256-byte LUT contains data that represents one cycle of a sine wave. This data is stored in sign and magnitude format. Sign and magnitude is the native format of the DAC data register. Writing to the DAC is more efficient in this format.

In the example project, the Timer8 is set to approximately 10 kHz. The frequency of the output waveform is determined by the step size of the index through the LUT.

For example, for a 10-kHz update rate and a step size of '1', the resulting sine wave has a frequency of about 39 Hz (10,000/256). If a step size of '2' is used, the resulting sine wave has a frequency of twice that, at about 78 Hz. The Excel file used to generate the LUT is provided along with this application note. It contains the options for the clock frequency and the number of steps in LUT, and generates the LUT based on them.

The 256-byte LUT is stored in ROM and is accessed using the MCU's `INDEX` instruction. The `INDEX` instruction uses a base address, which is hardcoded as the operand in the instruction and an index, which is the value in the Accumulator at the start of the command. When the `INDEX` instruction is executed, the table entry pointed to by the sum of the base address and the index is loaded into the Accumulator.

The 8-bit index for the LUT comes from the upper byte of a 16-bit index (`iCTCSSFreqIndex` in the project). The upper byte can be considered as the integer portion of the index and the lower byte as the fractional portion of the index. To step through the LUT, a 16-bit index increment (`iCTCSSFreqInc` in the project), also having an integer portion and a fractional portion, is added to the 16-bit accumulated index. This allows the use of fractional increments, which creates a more accurate frequency than is possible with an integer increment.

In the example described earlier, where a step size of '2' resulted in an output frequency of 78 Hz, the index increment is 0x0200. To get an output frequency of 67 Hz (CTCSS tone 1), a step size of 1.72 is needed. The index increment in this case is 0x01b3.

Frequency Selection

In this project, a function is provided that sets the index increment value for the desired frequency. This function, `SetCTCSSFreq()`, is passed a 1-byte argument that is the CTCSS tone number. The tone number is manipulated (subtract 1 and multiply by 2) to convert it into an index for a 38-entry, 16-bit LUT, which contains index increment values.

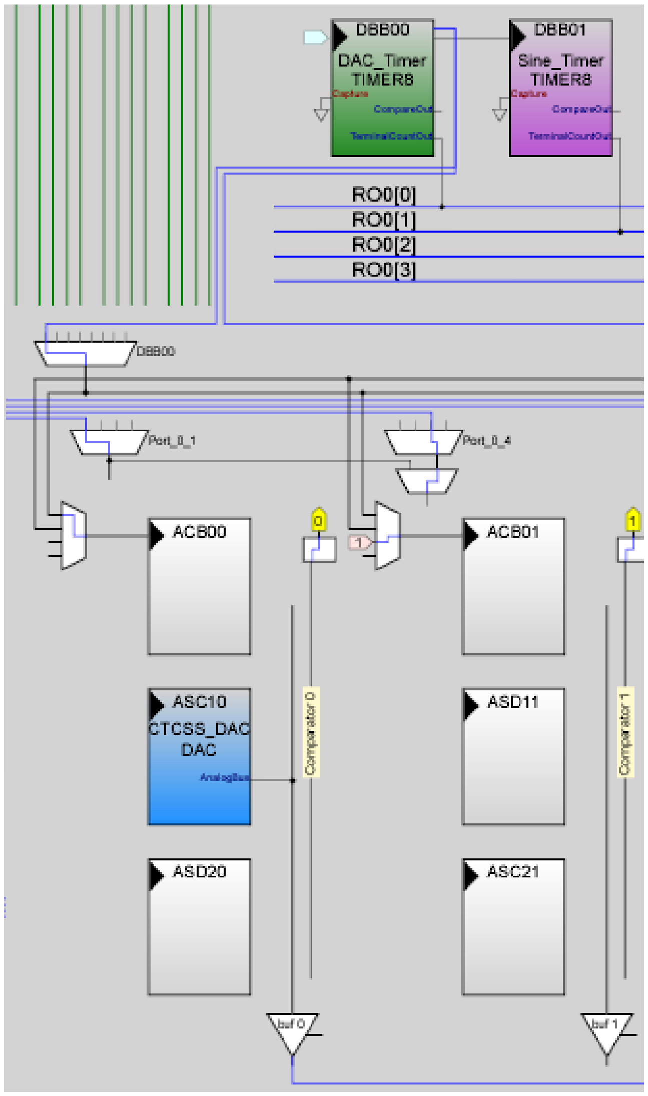
Phase Coherence

A side effect of the way in which the output frequency is changed (the accumulated index variable is not cleared) is that the sine wave exhibits no discontinuity when the frequency changes. Although this is not critical for CTCSS generation, it may be required in other waveform applications.

Hardware Configuration

Figure 10 shows the placement of the three user modules used in this project. CTCSS_DAC, a DAC6, outputs the analog signal that is the sine wave. CTCSS_DAC is configured to use a SignAndMagnitude data format. The output to the analog column bus is enabled and Buf0 is enabled to output the analog signal to Port0[3].

Figure 10. CTCSS User Module Placement



DAC_Timer, a Timer8 configured to divide the 48M clock by 61, is used to generate the column clock for the DAC6. This results in a DAC update rate of 197 kHz. A high DAC update rate is required for this application. The DAC update frequency appears in the spectrum of the output. The higher the DAC update rate, the more effectively it is removed with a simple R-C LPF.

Sine_Timer, a Timer8 configured to divide the output of DAC_Timer by 78, generates an output frequency of 10.088 kHz.

Port0[3] (pin 3 on a 28-pin package) is set to AnalogOutBuf0 with a drive mode of High Z to enable the analog signal to be output.

The frequency accuracy requirement for CTCSS is usually better than the ±2.5% accuracy of the internal main oscillator (IMO) in the PSoC microcontroller; therefore an external crystal oscillator is required. The Port1[0] and Port1[1] pins are set to High Z and a 32.678 kHz crystal is connected to the part as specified in the Cypress application note AN2027, *Using the PSoC Microcontroller External Crystal Oscillator.*

Note To check the working of the project without an external oscillator, the PLL_Mode in global parameters must be set to “Disable” and the 32K_Select must be set to “Internal”. If a crystal is added, but the selection for the 32K_Select is external, the ECO circuit continues to oscillate. The frequency of this oscillation is far from 32 kHz. With these settings, the project outputs at an unexpected frequency.

ISR Overhead

If the PSoC microprocessor also has other tasks, the amount of CPU overhead used by the ISR while generating the CTCSS output is important. Three factors influence the ISR overhead: the DAC6 analog column clock speed, CPU clock speed, and Sine_Timer clock rate.

The Sine_Timer ISR involves a write stall when updating the CTCSS_DAC. A write stall pauses the CPU until the start of the rising edge of the Phi1 clock in the analog column. This is done to prevent glitches on the DAC output. (See the Analog Synchronization Interface section in the Analog Interface chapter of the Technical Reference Manual (TRM) for more details on write stall.) The worst-case stall period for the DAC_Timer period used in this project is 5.1 μsec. Apart from the write stall, the ISR takes an additional 93 CPU clock cycles to execute.

With a DAC update rate of 10.088 kHz, the worst-case ISR overhead is 15%. Table 3 shows the ISR overhead for different CPU clock rates, assuming an analog column clock of 197 kHz and a DAC update rate of 10.088 kHz.

Table 3. CTCSS ISR Overhead

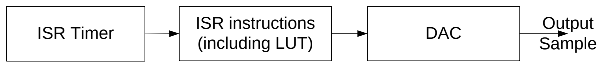
CPU_Clock	ISR Overhead
24 MHz	15%
12 MHz	24%
6 MHz	44%
3 MHz	82%
This project cannot be run below 3 MHz	

If different timing is used for this application, the resulting ISR overhead differs from those listed in Table 3.

Frequency Limitation

Multiple clocks are required to implement this technique. The maximum frequency obtainable from this implementation thus depends on the clocks and on the user module interdependencies. Consider this block diagram.

Figure 11. Block Diagram to Highlight the Clocks Required



The ISR timer triggers the ISR, in which the value from the LUT is loaded into the DAC. The slowest of these three blocks will determine the maximum frequency of the output. The ISR timer cannot be faster than the maximum update rate of the DAC. The maximum update rate of the DAC is 750 kHz.

Thus, the output of the timer is limited to the same value as the DAC update rate, The ISR overhead for the highest frequency of the DAC is the sum of CPU cycles for DAC stall and 93 CPU clock cycles for the rest of the ISR, as explained in the section earlier. The CPU cycles for the DAC write stall, for an update rate of 750 kHz and CPU clock speed of 24 MHz is 32 cycles.

Thus, the total delay due is 24 MHz/(93+32) = 192 kHz. The ISR instruction implementation is the slowest of the three blocks. Therefore, the maximum frequency possible for one output sample is 192 kHz. At least four samples are required to create a sine wave.

Therefore, the absolute maximum frequency possible for a sine wave output with this implementation is 48 kHz. Also note that the accuracy is higher at lower output frequencies with higher steps into the LUT, as explained in the Trade-offs and Performance sections.

Trade-offs

Trade-offs that must be made when designing a waveform generator.

ISR overhead versus DAC update rate is one trade-off that must be considered. More steps in a wavelength results in a lower harmonic distortion of the waveform. Fewer steps in a wavelength reduce the ISR overhead, leaving more of the CPU available for other tasks.

Waveform distortion versus output-filter complexity is another part of the same trade-off. If the DAC update rate is much higher than the frequency of the output waveform, the major contributor to distortion is at the DAC update frequency. In this case a simple R-C low pass filter on the output (often called a reconstruction filter) can acceptably reduce the waveform distortion. If the DAC update rate is too close to the frequency of the output waveform, the harmonics of the output frequency is the major contributor to distortion. In this case something more than a simple R-C low pass filter is needed to clean up the waveform. A Low Pass Filter User Module can be used.

In some designs, due to limited resources, the source for the DAC's analog column clock must be shared with another function. For this, a slower clock must be used. This affects the ISR overhead; it increases the maximum stall time, which in turn increases the waveform distortion by moving the DAC clock frequency closer to the output frequency. Ideally, a frequency close to the DAC6 maximum update rate is used.

In this project, a 6-bit DAC is selected over an 8-bit DAC. The 8-bit DAC results in more precise steps, but it uses more analog SoC blocks. Experiments show that the number of steps per waveform has a much greater impact on the accuracy of the output waveform than the DAC resolution does.

Performance

After configuring the timers as described in the previous sections and using an external crystal, the data was collected to determine the frequency accuracy and the waveform distortion of the output.

The output frequencies were measured for all 48 CTCSS tones using a Fluke 87 Digital Multimeter. The results of these measurements are shown in Table 4. The range of the frequency error is +0.10% to 0.13%.

The waveform distortions for selected CTCSS tones (1, 7, 13, 19, 26, 32, and 38) are calculated based on measurements taken with a Hewlett-Packard 3585A spectrum analyzer. The measurements are taken directly from the output of the PSoC microcontroller with no reconstruction filter. The results are graphed in the Waveform Distortion table below. The maximum distortion, 3.8%, occurred at the highest output frequency, as expected.

Additional testing was run using a DAC update rate (Sine_Timer) of 6.5 kHz. The maximum distortion increased to 6.3%.

Figure 12. Waveform Distortion

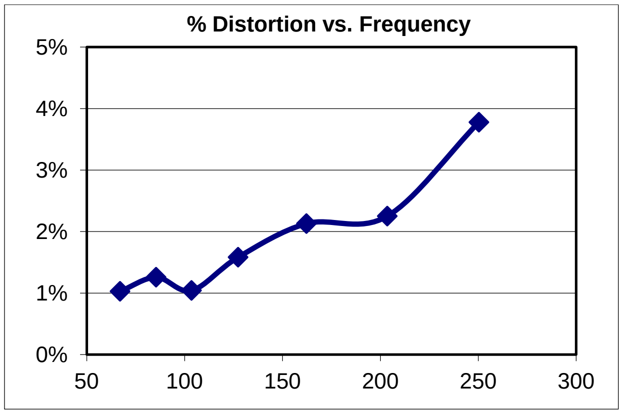


Table 4. Frequency Accuracy

Tone #	Frequency	Measured	Error	Tone #	Frequency	Measured	Error
1	67.0	67.09	-0.13%	20	131.8	131.83	-0.02%
2	71.9	71.83	0.10%	21	136.5	136.47	0.02%
3	74.4	74.45	-0.07%	22	141.3	141.20	0.07%
4	77.0	77.08	-0.10%	23	146.2	146.16	0.03%
5	79.7	79.71	-0.01%	24	151.4	151.37	0.02%
6	82.5	82.45	0.06%	25	156.7	156.79	-0.06%
7	85.4	85.39	0.01%	26	162.2	162.15	0.03%
8	88.5	88.50	0.00%	27	167.9	167.87	0.02%
9	91.5	91.54	-0.04%	28	173.8	173.83	-0.02%
10	94.8	94.81	-0.01%	29	179.9	179.81	0.05%
11	97.4	97.38	0.02%	30	186.2	186.18	0.01%
12	100.0	100.01	-0.01%	31	192.8	192.72	0.04%
13	103.5	103.51	-0.01%	32	203.5	203.40	0.05%
14	107.2	107.22	-0.02%	33	210.7	210.70	0.00%
15	110.9	110.93	-0.03%	34	218.1	218.10	0.00%
16	114.8	114.78	0.02%	35	225.7	225.60	0.04%
17	118.8	118.83	-0.03%	36	233.6	233.50	0.04%
18	123.0	123.12	-0.10%	37	241.8	242.10	-0.12%
19	127.3	127.22	0.06%	38	250.3	250.30	0.00%

Figure 13 and Figure 14 show the waveforms for Tone 1 and Tone 38, respectively. They are captured with a Tektronix TDS 3034 Digital Storage Oscilloscope. Notice that the individual DAC increments are more apparent at the higher frequency. This is because there are fewer steps per cycle at higher frequencies. The result is greater waveform distortion.

Figure 13. Tone 1 (67.0-Hz) Waveform

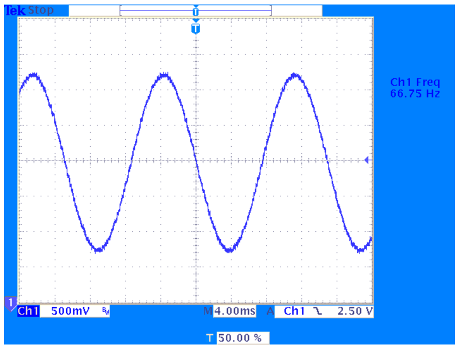
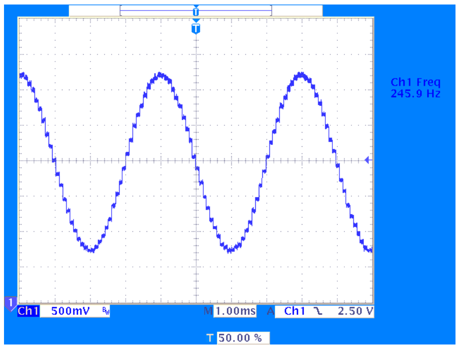


Figure 14. Tone 38 (250.3-Hz) Waveform



Document History

Document Title: AN2025 – Analog – Sine Wave Generation with PSoC® 1 (Demonstration with CTCSS)

Document Number: 001-40881

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	1536344	JVY	See ECN	New application note
*A	2793434	YARA	10/27/09	Updated part number from CY8C25xxx to CY8C27xxx. Updated accompanying projects from PSoC Designer version 2.1 to 5.0. Updated document text where appropriate.
*B	3188204	YARA	03/04/2011	Added ‘Frequency limitation’ section and a figure to show the maximum frequency possible.
*C	3825120	DESH	11/29/2012	Modified Abstract and Introduction. Added sections Sine Wave Generation using DAC, Firmware, and Some Design Considerations. Sine Wave Generation using filter.
*D	4102333	TEJU	08/22/2013	Modified the Section Sine Wave generation using DAC. Added more information about DAC8 and 3 new figures and a table. Associated projects have been updated to PD5.4
*E	5702260	AESATMP8	04/19/2017	Updated logo and Copyright.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer’s representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709