AN69133

PSoC® 3 / PSoC 5LP Easy Waveform Generation with the WaveDAC8 Component

Author: Mark Hastings

Associated Project: Yes

Associated Part Family: All PSoC 3 and PSoC 5LP parts

Software Version: PSoC Creator™ 4.0

More code examples? We heard you.

To access an ever-growing list of hundreds of PSoC code examples, please visit our code examples web page. You can also explore the PSoC video library here.

AN69133 describes how the WaveDAC8 Component works and how to use it to generate either predefined or custom waveforms. The WaveDAC8 uses DMA to generate continuous waveforms that require no CPU overhead. Several example projects are included to show simple waveform generation, frequency shift keying (FSK) modulation, and DTMF tone generation with minimal hardware and user code.

Contents

1 Introduction....1
2 Configuring WaveDAC8....2
3 The WaveDAC8 Core....3
4 Example Projects....5
4.1 1_WaveDAC8_SimpleSine Project....5
4.2 2_WaveDAC8_TwoWaves Project....6
4.3 3_WaveDAC8_UART_FSK Project....7
4.4 4_WaveDAC8_DTMF_EN Project....9
5 Over Sampling and Filters....13
6 Summary....17
A Code Listing for 4_WaveDAC8_DTMF_EN Project....18

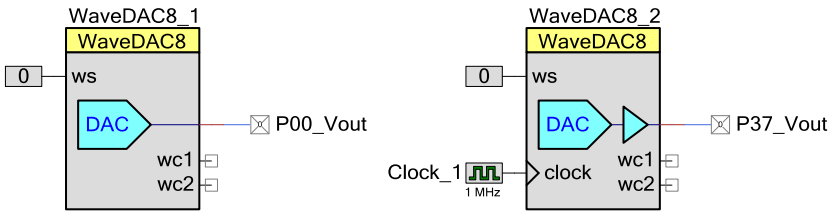
1 Introduction

The WaveDAC8 component provides a simple and fast solution for periodic waveform generation. WaveDAC8 has the following features:

- Standard and arbitrary waveform generation
- Output may be voltage, current sink, or current source
- Hardware selection between two waveforms
- External clock input may be used to change the output waveform frequency
- Waveform tables can be up to 4000 points
- Predefined sine, triangle, square, and sawtooth waveforms are included
- Allows you to change waveform arrays during run time
- Single line of C code required to initiate waveform output

Figure 1 shows two WaveDAC8s in a schematic, the one on the right shows options enabled for an external clock and an optional internal voltage buffer. An internal clock is usually selected when a fixed digital-to-analog converter (DAC) sample rate is used. An external clock provides an additional way to modulate the output or to synchronize the WaveDAC8 with additional logic. The internal buffer uses one of the on board opamps in the follower mode to drive up to a 25 mA load.

Figure 1. Internal and External Clock Configurations



Internally, the WaveDAC8 contains a voltage or current DAC, two direct memory access (DMA) channels, optional opamp follower and a clock when the internal clock option is selected. The user interface and the API handle configuration of the DAC, DMA, and wave table generation. No knowledge of the DAC or DMA is required to take full advantage of the WaveDAC8 component. For more detailed documentation of all the WaveDAC8 parameters and terminal definitions, see the datasheet included with the component.

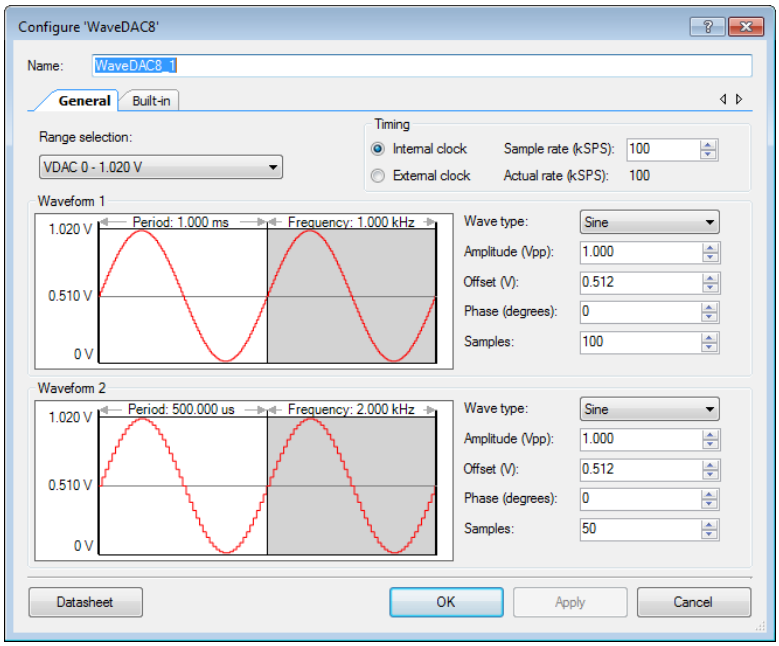
2 Configuring WaveDAC8

The graphical user interface (GUI) of the WaveDAC8 configuration dialog makes it easy to configure the waveform shape and frequency. To invoke the WaveDAC8 user interface, simply double-click on the component after it is placed in the schematic. The other option is to right-click on the symbol and select **Configure** from the menu window. See Figure 2 for an example of the configuration dialog.

The configuration dialog is divided into three sections. The top section determines the DAC range and mode, the sampling rate, and whether the clock is internal or external. These settings affect both waveforms configured in the other two sections, Waveform 1 and Waveform 2. You can select from ten output range options, four voltage ranges (2 buffered and 2 unbuffered), and six current ranges with the **Range Selection** parameter in the upper-left corner of the dialog box.

In the upper-right corner of the dialog, you can select either an internal or an external clock. If you select an internal clock, you can also configure the DAC sample rate. When you select an internal clock source, the top of the waveform areas display the waveform frequency and period. If you select an external clock, note the addition of a clock terminal on the WaveDAC8 schematic symbol.

Figure 2. WaveDAC8 Configuration Dialog



The **Waveform 1** and **Waveform 2** sections allow you to define two separate waveforms that share sample rate and DAC output range. The state of the “ws” input terminal determines which waveform is generated. A logic low on this terminal selects Waveform 1 and a logic high selects Waveform 2.

One of the four predefined waveforms may be selected: Sine, Square, Triangle, or Sawtooth. There are two options to generate a custom or arbitrary waveform. The user can either draw a waveform right on the screen by selecting the “Arbitrary (draw)” option or select “Arbitrary (from file)” to import a file. To draw a waveform, move the cursor over the displayed waveform area and press the left mouse button to draw the waveform.

If a waveform must be generated during runtime, there are two provided functions that allow the programmer to use any memory array to generate the waveform. See the WaveDAC8_Wave1Setup() or WaveDAC8_Wave2Setup() function descriptions in the WaveDAC8 component datasheet.

The output signal frequency is a function of the Sample Rate (SPS) and the **Samples** parameter for each of the waveforms, as shown in Equation 1. The **Samples** parameter defines how many values in the lookup table will be used to create each waveform period.

$$frequency = \frac{SampleRate}{Samples} \qquad \text{Equation 1}$$

Because each waveform has its own **Samples** parameter, two waveforms with the same sample rate may have a different output frequency. For example, if the sample rate is set to 100 kHz, the **Samples** parameter for Waveform 1 is set to 100 and 50 for Waveform 2, and then each waveform will have a different output frequency. Waveform 1 will have an output frequency of 1 kHz and Waveform 2 will have a frequency of 2 kHz, as shown in Equations 2 and 3. Using this configuration, frequency shift keying (FSK) can be achieved by changing the state of the “ws” input.

$$freq_{Waveform1} = \frac{100{,}000}{100} = 1000Hz \qquad \text{Equation 2}$$

The output frequency for Waveform 2 will be:

$$freq_{Waveform2} = \frac{100{,}000}{50} = 2000Hz \qquad \text{Equation 3}$$

Refer to the Over Sampling and Filters section when determining how many samples per waveform is right for your application. It provides an overview of using a sampled system and options to minimize output waveform distortion.

The **Amplitude** and **Offset** parameters allow you to scale and add a voltage (or current) offset to the waveform.

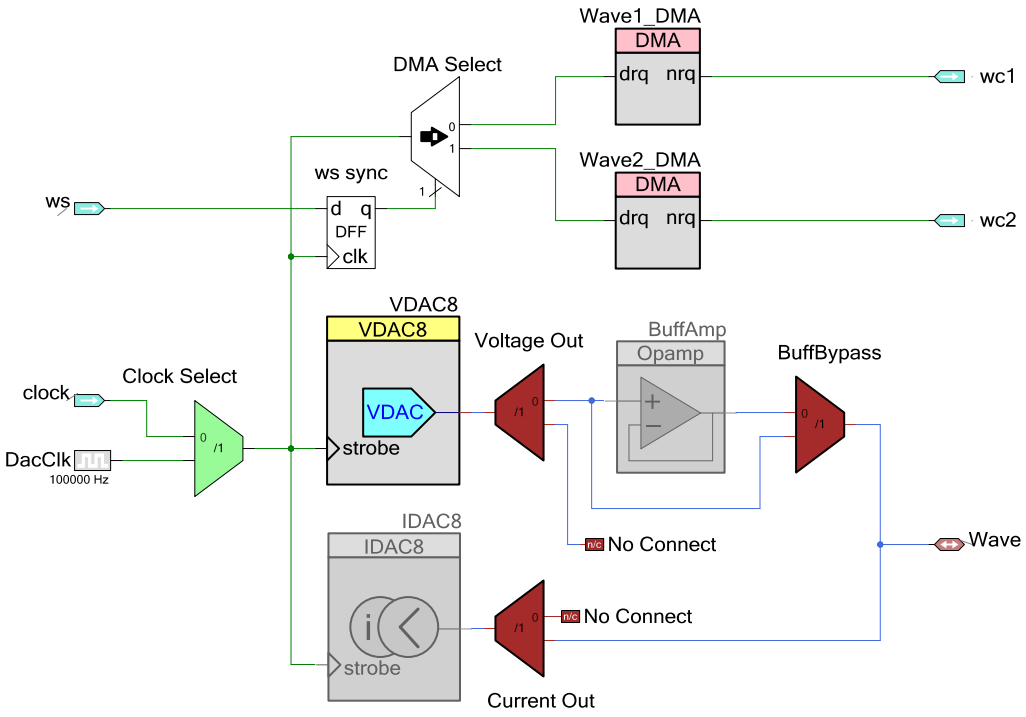
The user interface provides a simple interface to quickly select the desired waveform. Other than that to start the component in the user’s code, there is no other configuration or functions required to generate a waveform.

3 The WaveDAC8 Core

The core of WaveDAC8 is the DAC. It will either be a standard VDAC8 (voltage DAC) or IDAC8 (current DAC) depending on the range selected. Therefore, the WaveDAC8 analog electrical specifications are identical to those of the standard PSoC Creator VDAC8 and IDAC8 components. This is a good example of how PSoC Creator allows the use of standard components to build more complex ones, without reinventing the functionality and APIs of the original components.

The two DMA channels, Wave1_DMA and Wave2_DMA, are used to transfer the waveform array data in memory to either the IDAC or VDAC. When you configure a waveform with the user interface, the component automatically configures each of the DMA channels to transfer the data. Both of these DMA channels transfer data to the DAC, but only one can operate at a time. The wave select “ws” input selects which of these DMA channels is triggered by the clock, using the demultiplexer “DMA Select”, to route the signal to the corresponding DMA channel. The two wave complete outputs “wc1” and “wc2” can be used to signal that the DMA channel transferred the last value in the waveform table, or that one full waveform is complete. Figure 3 shows the internal WaveDAC8 component schematic.

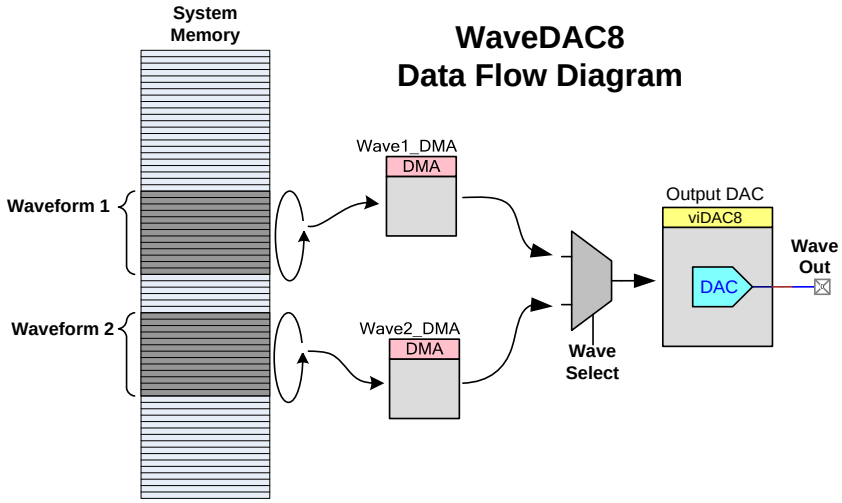
Figure 3. Internal Schematic of WaveDAC8



The multiplexer labeled “Clock Select” is a virtual mux that is set during configuration and removed automatically during the build process. It is used to select whether an internal or external clock is used. For example, if the external clock option is selected, the internal clock “DacClk” will be removed from the project and the virtual mux will connect the clock input terminal “clk” to the VDAC8 and IDAC8 strobe inputs and the DMA Select demultiplexer. If the internal clock option is selected, the “clk” terminal will be removed and the “DacClk” clock output will be connected to the IDAC8, VDAC8, and DMA Select multiplexer instead.

The two analog demultiplexers labeled “Voltage Out” and “Current Out” is also virtual. They are configured such that one DAC output will be connected to the output terminal “Wave” and the other will not be connected. The DAC that is not used, based on the current or voltage range, will be removed. The BuffBypass virtual multiplexer is used enable or disable the BuffAmp (opamp).

Figure 4. WaveDAC8 Data Flow



Two important parts of the WaveDAC8 component are the waveform arrays and the firmware that forms the API. The two arrays or tables contain the data that is continuously copied to the DAC with DMA to create the waveforms. Figure 4 show the internal data flow from the arrays to the DAC. The firmware takes care of all the initialization of the IDAC8 or VDAC8 and the DMA channels. This is what makes this component so easy to use.

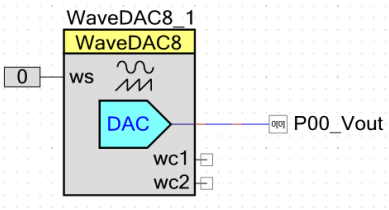
4 Example Projects

The following four example projects show how easily you can configure and use WaveDAC8. The examples include generation of a simple waveform, FSK modulation, and a very simple DTMF phone dialer. A workspace with all four projects and the WaveDAC8 library are included with this application note. Although the projects are configured for the PSoC 3, they will work equally well with the PSoC 5LP by just choosing the PSoC 5LP used in your design with the Device Selector in the Project menu.

4.1 1_WaveDAC8_SimpleSine Project

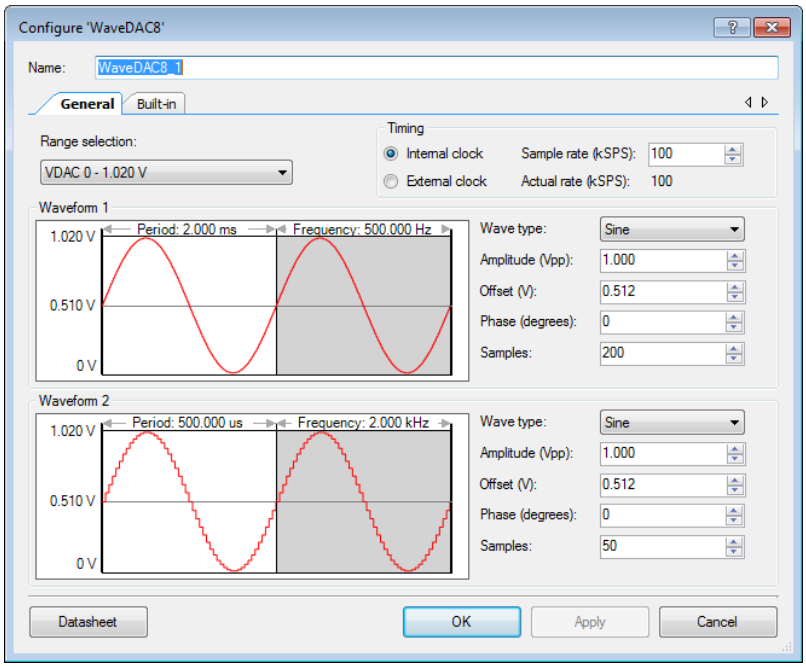
This very basic example generates a 1-kHz sine wave. Figure 5 shows the PSoC Creator schematic for this example.

Figure 5. WaveDAC8_SimpleSine Project Schematic



Notice that the schematic contains only three components, WaveDAC8, analog output pin (P00_Vout), and a logic “0” to select Waveform 1. The configuration of the WaveDAC8 is shown in Figure 6. Only the settings for “Range Selection”, “Clocking – Sample Rate” and parameters in the “Waveform 1” area matter because Waveform 2 is not selected in this project.

Figure 6. WaveDAC8 Configuration



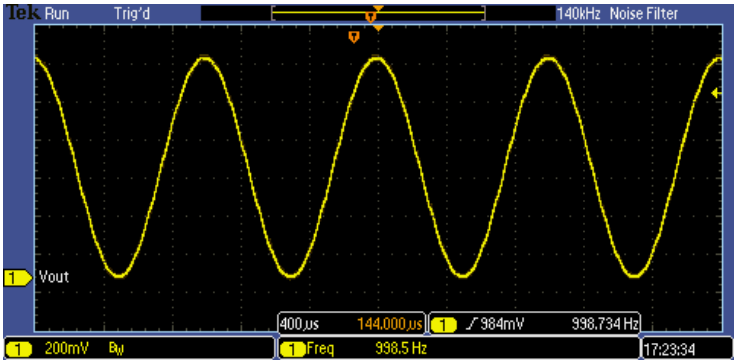
The code for this example is as simple as the schematic:

```
#include <device.h>
```

```
void main()
{
    /* Start WaveDAC8  */
    WaveDAC8_1_Start();

    for(;;);    /* Loop forever */
}
```

Notice that the only code required is the “WaveDAC8_1_Start();” function. Setting up the DAC, Internal Clock, and DMA is handled by the Start() function. After the project is built and the device is programmed, an oscilloscope can be connected to port pin P0[0] to see the waveform shown in Figure 7.

Figure 7. Output of Project 1_WaveDAC8_SimpleSine

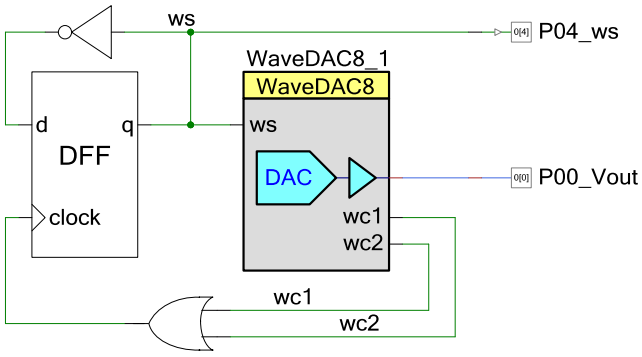


This is all that is required. This sine wave will be generated continually with zero CPU overhead until the device is powered down or the WaveDAC8_1_Stop() function is called. This example can be easily modified to switch between two waveforms. The easiest method is to connect the “ws” terminal to an I/O pin and control the waveform with an external switch.

4.2 2_WaveDAC8_TwoWaves Project

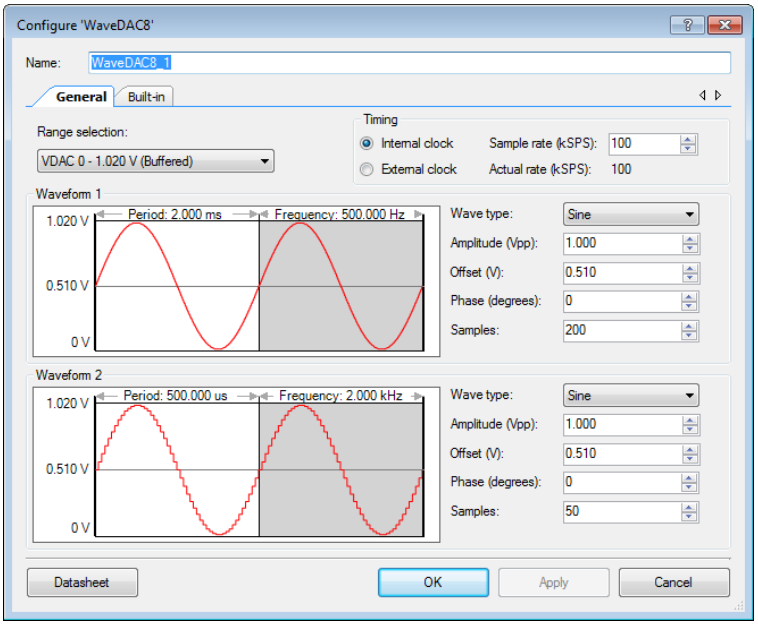
This next example adds a little more hardware to alternate between Waveform 1 and Waveform 2. It uses the two waveform-complete signals, “wc1” and “wc2”, so that it does not switch from one waveform to the other until the entire waveform is completed. The wc1 and wc2 signals go high after the last value in the waveform table for each waveform is written to the DAC, therefore, signaling the end of waveform period. Figure 8 is the schematic for this example project.

Figure 8. 2_WaveDAC8_TwoWaves Project Schematic



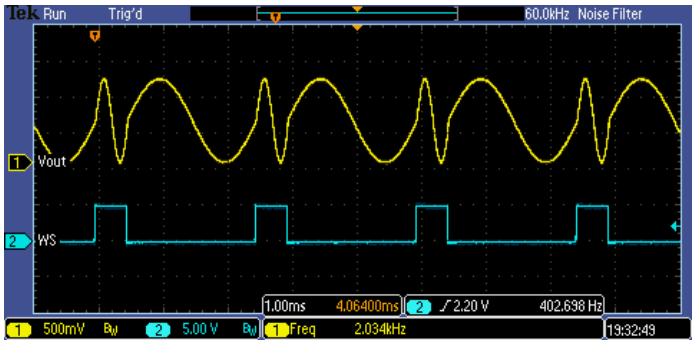
The configuration of the WaveDAC8 is shown in Figure 9. Again, the component is configured to use a buffered voltage DAC output, internal clock, sine wave, and a sample rate of 100 kHz. This time the **Samples** parameter for Waveform 1 is set to 200, which will generate an output frequency of 500 Hz. Waveform 2 will also generate a sine wave, but the **Samples** parameter is set to 50, which will output a frequency of 2000 Hz.

Figure 9. WaveDAC8 Configuration



When the wave select signal “ws” is low, Waveform 1 will be output, which is the slowest of the two waveforms at 500 Hz. When “ws” is high, Waveform 2 will be output with a frequency of 2000 Hz. If you place a scope probe on pins P0[0] and P0[4], you can see both the waveform output and the waveform selection signal shown in Figure 10.

Figure 10. Output of Project 2_WaveDAC8_TwoWaves



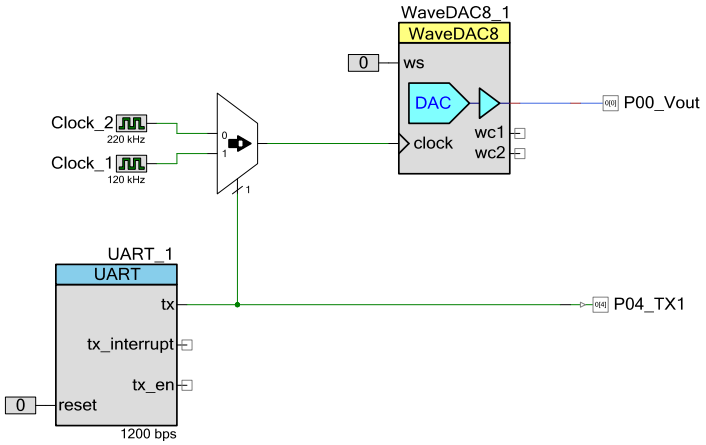
This output is a very simple form of FSK. As before, the code required to generate this modulated waveform is minimal:

```
#include <device.h>

void main()
{
    /* Initialize WaveDAC8 */
    WaveDAC8_1_Start();

    for(;;);  /* Loop forever */
}
```

4.3 3_WaveDAC8_UART_FSK Project

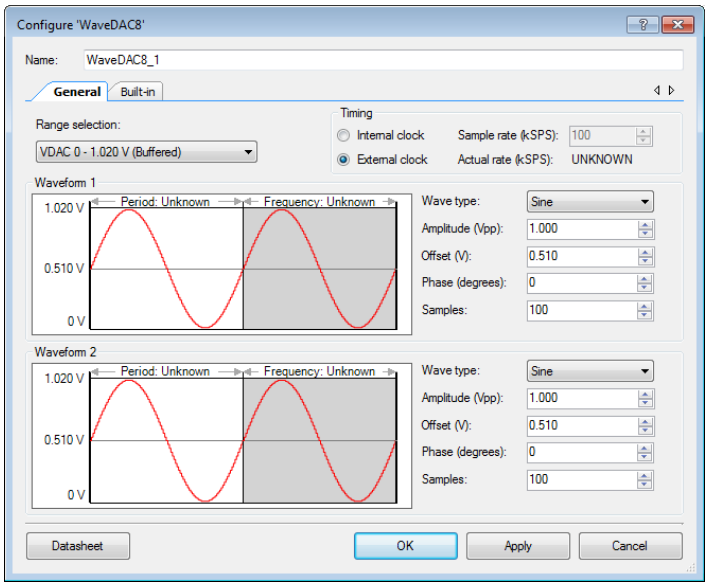
In this third example, another form of FSK is demonstrated. This time, a single waveform is used, but two external clocks are used to modulate the output signal. By adding the transmit (TX) half of a UART, a fully functional FSK transmitter will be demonstrated. The schematic for this example is shown in Figure 11.

Figure 11. 3_WaveDAC8 project Schematic



This example emulates a 1200-baud modem in which the output signal switches between 1200 and 2200 Hz. In the WaveDAC8 configuration shown in Figure 12, only the settings affecting Waveform 1 are important. Notice that the **Samples** parameter is set to 100; if you divide the two clock outputs by 100, you will get the 1200 and 2200 Hz sine waves required.

Figure 12. WaveDAC8 Configuration



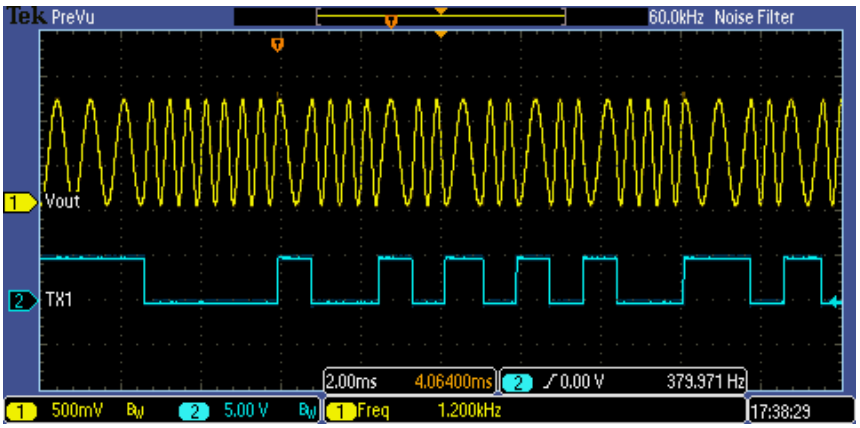
The code for this example contains a few more lines, but it does more too. It continuously transmits the message “Hello World” in an FSK-modulated format. The amount of code that is required for the designer is again rather small:

```
#include <device.h>
void main()
{
    /* Initialize WaveDAC8  */
    WaveDAC8_1_Start();
    UART_1_Start();  /* Initialize UART */
    Clock_1_Start(); /* Start both clocks */
```

```
    Clock_2_Start();
    for(;;)
    {
       /* Send "Hello World"  */
       UART_1_PutString((uint8 *)"Hello World");
       CyDelay(250);    /* Wait 500 mSec */
       CyDelay(250);
    }
}
```

The FSK-modulated output can be seen on pin P0[0] and the digital output of the UART transmitter may be observed on pin P0[4], as shown in Figure 13. Notice that the 2200-Hz tone is present when the TX line is low and the 1200-Hz tone is present when the TX line is high.

Figure 13. Output Waveforms



4.4 4_WaveDAC8_DTMF_EN Project

This fourth and final example implements a simple dual tone multiple frequency (DTMF) phone dialer. It requires two WaveDAC8s so that it can generate two tones at the same time. See Figure 14 for the schematic.

A single clock in conjunction with two counters generates the proper sampling clock for each WaveDAC8. The period of the two PWMs, used as frequency dividers, “Col_Divider” and “Row_Divider” is changed for each tone to provide the proper column and row tones respectively. An opamp “DTMF_Buffer” is added to buffer the output of the WaveDAC8s so a lower impedance load can be driven.

At first, connecting two voltage DACs in parallel might not sound like good design practice, but it is safe with the PSoC 3 or PSoC 5LP DACs. The PSoC voltage DACs are actually current DACs with an internal resistor. When you place two of these DACs in parallel, you are simply combining two current DACs and two parallel resistors. The output of these two combined DACs is the sum of the two signals divided by two. This is a useful feature when a design requires combining two signals.

Figure 14. DTMF Encoder Schematic

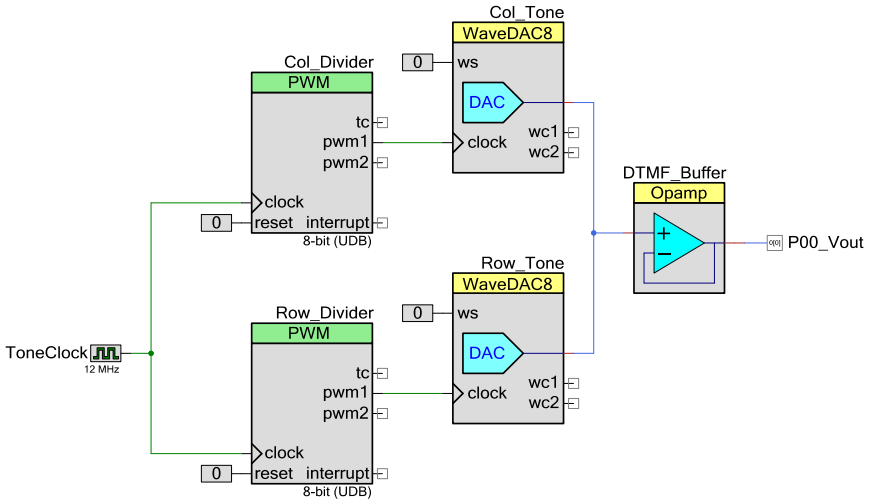


Table 1 shows the standard DTMF tones for each key. Although the digits 0 through 9, "*", and "#" are the most common, the characters A, B, C, and D are also supported. When a single key is pressed, two tones are mixed together. For example, if the "5" key is pressed, the column tone 1336 Hz and the row tone of 770 Hz are mixed together. These eight tones were selected to ensure that any harmonics and inter-modulation products do not cause a signal that can be mistaken for one of the other tones. Notice that no frequency is a sum, difference, or multiple of one or more of the other frequencies.

Table 1. DTMF Keypad Frequencies

		Column Tones			
		1209 Hz	**1336 Hz**	**1477 Hz**	**1633 Hz**
Row Tones	**697 Hz**	1	2	3	A
	770 Hz	4	5	6	B
	852 Hz	7	8	9	C
	941 Hz	*	0	#	D

To create each of the tone combinations in the DTMF table, the tones are divided into the column tones (1209 Hz, 1336 Hz, 1477 Hz, and 1633 Hz) and the row tones (697 Hz, 770 Hz, 852 Hz, 941 Hz). The row tones are generated by the "Row_Tone" instance of the WaveDAC8 and the column tones are generated by the "Col_Tone" instance of the WaveDAC8. Each tone frequency is a function of the clock "ToneClock", the dividers "Col_Divider" and "Row_Divider", and the number of samples per cycle set in the WaveDAC8s configuration.

$$ToneFreq = \frac{ToneClockFreq}{Divider \times Samples}$$ Equation 4

For this example, the samples per cycle **Samples** parameter is set to 50 for the column frequencies and 80 for the row frequencies. These values are set in the WaveDAC8 **Configure** dialog box for each waveform with the **Samples** parameter. The only thing left to calculate is the divider value, which is the period of each of the counters, Row_Divider and Col_Divider. If you solve for the divider value, you get:

$$Divider = \frac{ToneClockFreq}{ToneFreq \times Samples}$$ Equation 5

When using a counter as a divider, the divide by value is the counter period plus 1. Taking the previous equation and solving for the counter period:

$$CounterPeriod = \frac{ToneClockFreq}{ToneFreq \times Samples} - 1$$ Equation 6

For example, suppose you want to calculate the counter period to generate the row tone if the “5” key is pressed. Using the previous equation and substituting 770 Hz for the row tone, 80 for the sample size, and 12 MHz for the clock, you get a period of 194.

$$CounterPeriod = \frac{12{,}000{,}000}{770 \times 80} - 1 = 194$$ Equation 7

Taking the DTMF tone in Table 1 and using the above equation to calculate the counter period for the DTMF frequencies, the rest of the values can be filled in Table 2.

Table 2. Counter Period Values

		Col_Divider Period			
		198 (1209 Hz)	179 (1336 Hz)	161 (1477 Hz)	146 (1633 Hz)
Row_Divider Period	214 (697 Hz)	1	2	3	A
	194 (770 Hz)	4	5	6	B
	175 (852 Hz)	7	8	9	C
	158 (941 Hz)	*	0	#	D

The source code for the example is a few more lines but still very easy to follow. You can view the entire source code listing in Code Listing for 4_WaveDAC8_DTMF_EN Project.

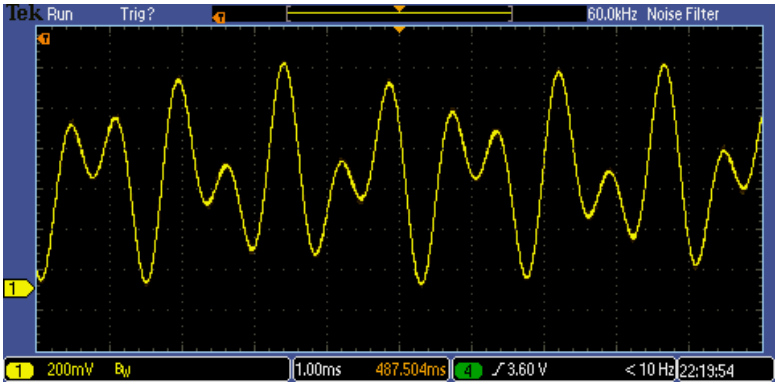
This implementation is not the only possible approach to making a DTMF dialer, but is a good compromise between hardware resources, code size, and ease-of-use. Here are some alternatives with which you may want to experiment:

- Use two clocks and remove the counters. Change the clock dividers to vary the output tones.
- Use only one WaveDAC8 but create a larger lookup table for each tone combination. This may require large lookup tables. To save RAM, just one lookup table can be used but modified each time a key is pressed.
- Modify the internal clock dividers to change frequencies.

4.4.1 Examining the output

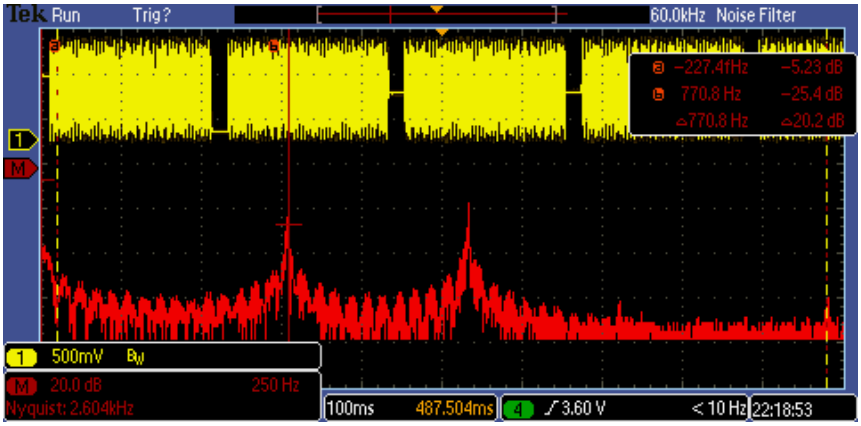
Figure 15 is an example of the output signal. The two mixed signals are 1336 Hz and 770 Hz simulating the “5” key pressed on the keypad.

Figure 15. Tones 770 Hz and 1336 Hz Combined



If you enjoy experimenting with some of the more advanced features of your scope such as the FFT module, try using the same dual tone output as in Figure 15 and enable FFT. Two waveforms appear as shown in Figure 16.

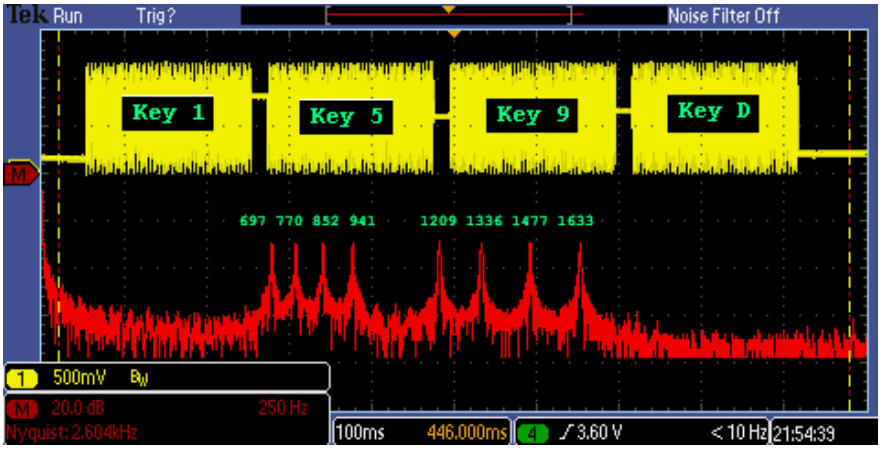
Figure 16. Analyzing DTMF Output of a "5" Key



The input (upper waveform) simulates a ‘5’ on a DTMF keypad being pushed repeatedly. The lower waveform is the FFT of the input signal. Just as expected, the 770- and 1336-Hz tones show up clearly in the FFT.

In this next test, a simulated key sequence of 1, 5, 9, and D are output. This tests each of the eight different tones used for the DTMF keypad. Figure 17 clearly shows each of the row and column tones in the FFT.

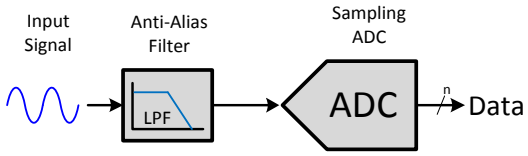
Figure 17. Simulated "1 5 9 D" Key Sequence



5 Over Sampling and Filters

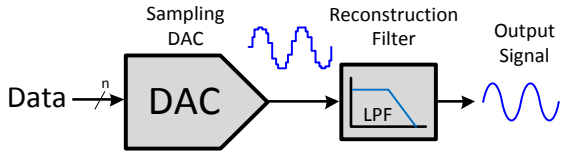
The WaveDAC8 is an easy component to use, but there are a couple of aspects that need to be considered when using any type of sampled system. A sampled system is one that uses an ADC to sample an AC waveform or a DAC to generate an AC waveform. The basic system and filter requirements for these systems are similar. Most engineers are familiar with the Nyquist sampling theorem, which states that you must sample a signal at more than twice its highest frequency component to reconstruct it. This exact concept holds true for a DAC. In the ADC system shown in Figure 18, the anti-alias filter limits the input signal bandwidth to less than half of the ADC's sample frequency.

Figure 18. ADC Sampling System



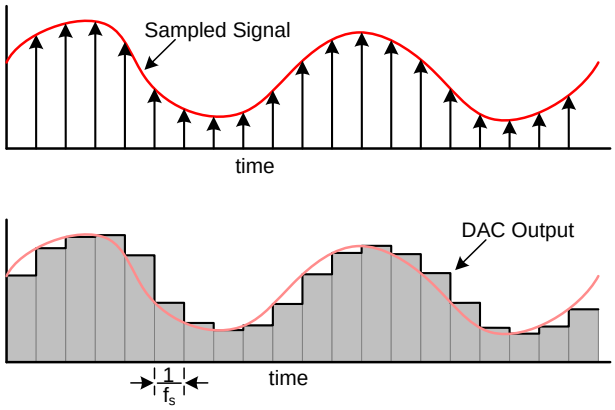
When using a DAC, the system looks identical but in reverse, except for replacing the ADC with a DAC.

Figure 19. DAC Sampling System



When sampling a continuous signal, the output is turned into a series of discrete points. The output of a DAC before being filtered may look like a series of stair steps that follow the original signal. The size of these steps, depend on the DAC resolution and the sample rate of the DAC. The width of these steps is the inverse of the sampling frequency.

Figure 20. Result of Sampling

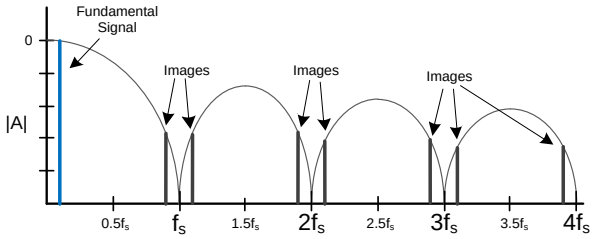


A waveform with large "stair step" edges has a broad frequency spectrum, much higher than the desired output bandwidth. The actual output spectrum of a DAC looks like the typical sync function.

$$Amplitude = \frac{\sin\left(\frac{\pi \cdot f}{f_s}\right)}{\left(\frac{\pi \cdot f}{f_s}\right)}$$ Equation 8

Figure 21 shows the output frequency spectrum of a DAC. Harmonic images appear on each side of the sampling frequency harmonics that mimic the original bandwidth of the sampled signal. To simplify the diagram, assume the desired output signal is a sine wave.

Figure 21. DAC Output Frequency Spectrum



f_s is the DAC sample rate

A reconstruction filter is used to attenuate the images to the point where the total harmonic distortion (THD) is low enough for the application. Ideally, you want to keep this filter small and simple to reduce the cost.

Another way to reduce THD is to increase the over sample rate. When a signal is sampled at more than twice the maximum bandwidth of that signal, it is said to be over sampled. The over sample rate is defined by:

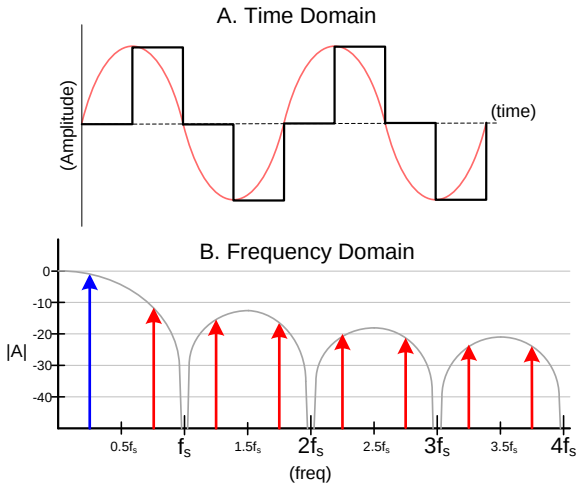
$$OverSampleRate = \frac{f_s}{2B}$$ Equation 9

Where fs is the sample frequency and B is the desired bandwidth of the signal.

Increasing the over sample rate increases the distance between the fundamental frequency and the first image. This, in turn, also reduces the signal distortion by reducing the amplitude of the harmonic images.

To explain this a little better, take the example where a 1-kHz sine wave must be generated and the sample rate is 4 ksps. Using Equation 9, this gives us an over sample rate of 2. Figure 22 shows the representation of a 2x over sample rate in both the time and frequency domain.

Figure 22. Example of 2x Over Sampling Rate



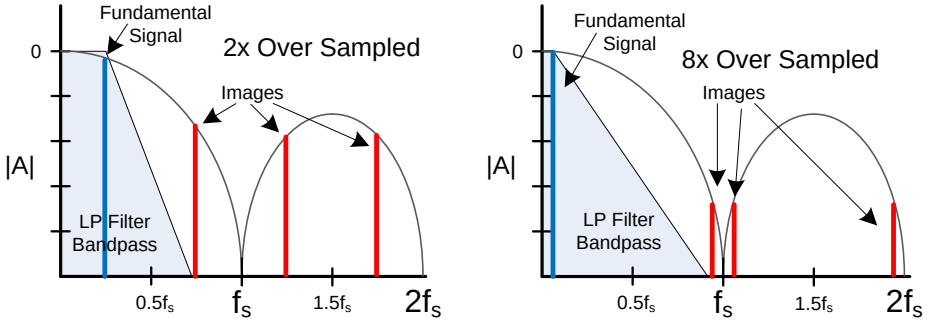
When looking at the DAC output in the time domain in Figure 22(A), it is easy to see that the signal looks more like a square wave than a sine wave. Looking at the frequency domain plot in Figure 22(B), it is not surprising that the image harmonics are significant.

Using the following three options, we can reduce or attenuate the effect of the harmonics.

- Add a reconstruction filter
- Increase the over sample rate
- Increase the over sample rate and add a filter

Depending on the application requirements, you may be forced into one of the options. If you need a high frequency output that is close to the maximum sample rate, your only option may be to use an external reconstruction filter. A filter will reduce the THD, but with a low over sample rate, it can be difficult to get the results needed with a simple first-order filter since the first image is close to the fundamental. Figure 23 shows the problem in the frequency domain between a 2x and 8x over sampled signal. Notice that the filter must be much sharper in the 2x system. Also, note that the harmonic images in the 8x system are much lower than those in the 2x system and will require a less complex and cheaper filter to get the same performance.

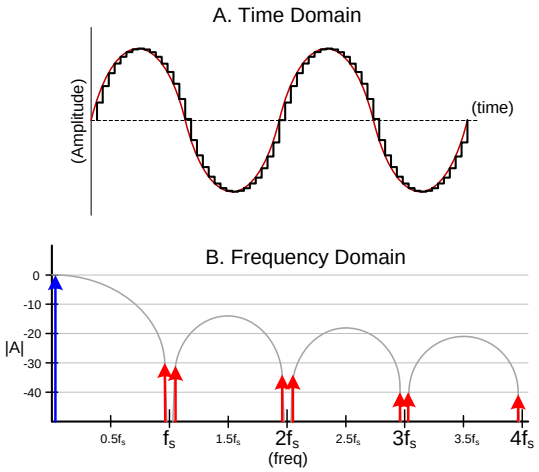
Figure 23. Reconstruction Filter



A first-order filter will reduce the distortion from about 46 percent down to about 12 percent. A second-order filter will further reduce the THD to less than 4 percent.

If we take the other approach and increase the over sample rate to 16x you can visually see that the generated signal looks much more like a sine wave in Figure 24 (A). Notice that the first harmonic image is down about 30 dB compared to only 10 db in the 2x over sampled system in Figure 22.

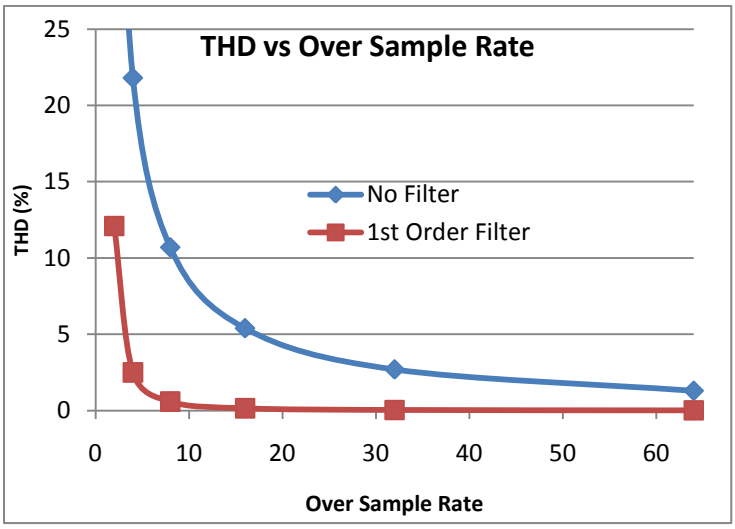
Figure 24. Example of a 16x Over Sample Rate



The THD now is about 5.4 percent, down from 46 percent, without adding a filter. Adding a simple first order filter, the THD will be less 0.5 percent.

The plot in Figure 25 shows how increasing the over sample rate or adding a simple filter can greatly decrease the output waveform distortion. The graph uses the harmonic images out to the 5th sampling harmonic to calculate the distortion with Equation 10.

Figure 25. THD vs Over Sample Rate Guide

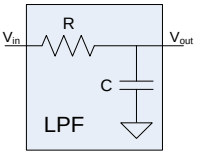


$$THD = \frac{\sqrt{V_2^2 + V_3^2 + ... + V_n^2}}{V_{fund}} \% \qquad \text{Equation 10}$$

Where V_n is each of the harmonics and V_{fund} is the fundamental frequency.

A good rule of thumb, is that each time the over sample rate is increased by a factor of two; the distortion goes down by a factor of two. A first-order filter combined with an over sample rate between 4 and 8 can bring the distortion down very quickly. This may be the best and most economical solution for many applications. An example of first-order passive RC filter is shown in Figure 26 and Equation 11.

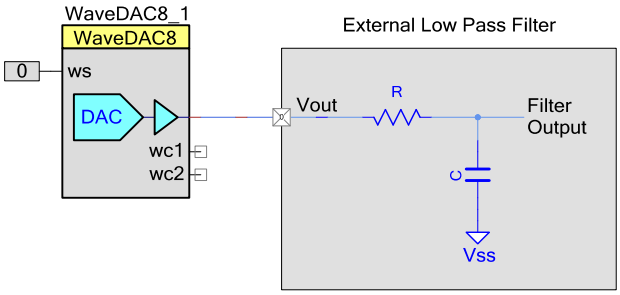
Figure 26. First Order RC Filter



$$F_c = \frac{1}{2\pi RC} \qquad \text{Equation 11}$$

The output resistance of the WaveDAC8 when in the voltage mode is 4 K and 16 K for the 1 V and 4 V ranges respectively. When connecting a filter directly to the output of the WaveDAC8, consider this resistance. In some applications, it may be a good idea to buffer the output of the WaveDAC8 with the buffer option as shown in the following figure.

Figure 27. WaveDAC8 with Buffered Output



This application note has touched on the basic requirements of a reconstruction filter. This information should be sufficient for most applications, but if your design requires something more complex, there are many filter software applications and web pages on the internet to assist in your design.

6 Summary

The WaveDAC8 component is a good example of how a highly integrated component can greatly simplify user code. These examples provide a good starting point for many applications. It is recommended that you refer to the WaveDAC8 datasheet for detailed descriptions of the API functions. These examples can be run on any of the PSoC 3 or PSoC 5LP development kits.

About the Author

Name: Mark Hastings

Title: Application Engineer MTS

Background: Mark Hastings graduated from Washington State University in 1984 with a BSEE degree. For most of the last almost 30 years he has been involved in embedded and mixed signal designs. His free time is spent hiking in the Washington Cascades taking pictures.

A Code Listing for 4_WaveDAC8_DTMF_EN Project

```
/*******************************************************************************
* File Name: main.c
*
* Version 1.2
*
* Description:
*  This example project demonstrates one way to make a DTMF dialer with
*  two WaveDAC8s, two counters, and a clock.
*
* Hardware Dependency:
*  Pin Connections:
*  P0[0]] --> Dialer output
*
* Code Tested With:
*  1. PSoC Creator  2.2 Service Pack 1
*  2. DP8051 Keil 9.03
*  3. ARM GCC 4.4.1
*
* Theory of operation:
*
* To dial each digit in a phone number, two independent tones must be generated
* at the same time.  Below is a table that shows each of the valid 16 keypad
* tones and the row and column tone for each digit.
*
*                      DTMF tone matrix
*                      Column tone (Hz)
*                  1209   1336   1477   1633
* ROW tone (Hz) *------*------*------*------*
*          697  |  1   |   2  |  3   |  A   |
*               *------*------*------*------*
*          770  |  4   |   5  |  6   |  B   |
*               *------*------*------*------*
*          852  |  7   |   8  |  9   |  C   |
*               *------*------*------*------*
*          941  |  *   |   0  |  #   |  D   |
*               *------*------*------*------*
*
*                     12.00 MHz
* Tone_Freq = -------------------------
*             ( Sample_Size * Divider )
*
* Solving for the divider
*
*                     12.00 MHz
* Divider =   --------------------------
*             ( Sample_Size * Tone_Freq )
*
* The Sample size for the column tones is 50 and the sample size for the row
* tones is 80. These values were selected to keep the counters at 8 bits, but
* at the same time keep the tone frequency error less than 1%.
*
* Example:
* To generate the row tone for a "5", which is 770 Hz and the sample size for
```

```
* row tones is 80;
*
*              12.00 MHz
* Divider =   ------------ = 195
*             ( 80 * 770 )
*
* Since the period length of a counter is the period register + 1, the value
* loaded into the counter period regiser would be 195 - 1 = 194.  ( PWMs
* are used for counters in this design for ease of use. )
*
* The divider values for both the row and column tones are pre-calculated in
* arrays. The "rowDiv" array has the values for each of the four rows and
* "colDiv" contains the divider values for the columns.
*
********************************************************************************/
#include <device.h>

#define TONE_DURATION_MS   80   /* Duration of tone in mSec */
#define TONE_SPACE_MS      50   /* Duration of space between tones in mSec */
#define TONE_PAUSE_MS     100   /* Pause caused by invalid code  */

/* Function prototypes */
void DialNumber( char * number );
void PlayTones( char key );
int8 KeyIndex(char * keyString, char key);

char keyCodes[] = "123A456B789C*0#D";  /* Valid character array */

/* The tables below store the dividers loaded into */
/* the counter period register to generate the row */
/* and column tones.                               */

/* Row Tones (Hz) 697  770  852  941      */
uint8 rowDiv[] = {214, 194, 175, 158};

/* Col Tones (Hz) 1209  1336  1477  1633  */
uint8 colDiv[] = { 198,  179,  161,  146};

void main()
{
    /* Initialize all components */
    ToneClock_Stop();
    Row_Divider_Start();
    Col_Divider_Start();
    Row_Tone_Start();
    Col_Tone_Start();
    DTMF_Buffer_Start();

      /* Dial a phone number */
      DialNumber( "555-123-4567" );

    for(;;)
    {
        /* Place user code here */
    }
```

```
}

/*******************************************************************************
* Function Name: DialNumber( char * number )
********************************************************************************
*
* Summary:
*  This function dials the number passed to it in a character string.  After
*  the number has been dialed the funtions returns.
*
* Parameters:
*  char * number:  Pointer to phone number string.
*
* Return:
*  void
*
*******************************************************************************/
void DialNumber( char * number )
{
   uint8 c = 0;
   while( number[c] != 0 )     /* Step through the dial string */
   {
      PlayTones( number[c] );  /* Play tone for each digit     */
      CyDelay(TONE_SPACE_MS);  /* Wait minimum space between tones */
      c++;
   }
}

/*******************************************************************************
* Function Name: PlayTones( )
********************************************************************************
*
* Summary:
*  Generates the two tones required to dial the given key code.  The valid
*  dial characters are "1 2 3 4 5 6 7 8 9 0 A B C D * #".  Any invalid
*  character will generate a pause.
*
* Parameters:
*  char key:  Keypad character to be dialed.
*
* Return:
*  void
*
*******************************************************************************/
void PlayTones( char key )
{
   int8 idx;
   uint8 row_div, col_div;

   idx = KeyIndex(keyCodes, key);  /* Convert key to "keyCodes" array index */

   if (idx >= 0 )    /* Is key valid */
   {  /* Valid Key */
      col_div = colDiv[(uint8)(idx & 0x03)];      /* Get divider for column tone */
      row_div = rowDiv[(uint8)((idx >> 2)& 0x03)];/* Get divider for row tone    */
```

```
        Row_Divider_WritePeriod(row_div);            /* Set both dividers  */
        Col_Divider_WritePeriod(col_div);

        ToneClock_Start();                           /* Turn on clock */
        CyDelay(TONE_DURATION_MS);                   /* Wait for the tone duration */
        ToneClock_Stop();                            /* Turn off clock */
    }
    else  /* Invalid key, just pause for set period of time. */
    {
        CyDelay(TONE_PAUSE_MS);
    }
}

/*******************************************************************************
* Function Name: KeyIndex( )
********************************************************************************
*
* Summary:
*  This function finds the location of a character in a string and returns
*  the index.  If the character is not found a -1 is returned.
*
* Parameters:
*  char * keyString:  String to search through.  This strings should be null
*                     terminated and less than 255 characters in lenth.
*  char   key:        Character to find in "String".
*
* Return:
*  int8: Index of "key" in "keyString".  If not found, return -1.
*
*******************************************************************************/
int8 KeyIndex(char * keyString, char key)
{
    int8 i;              /* String index */
    int8 charLoc = -1;   /* Location of character in string */

    /* Search through string for character match */
    for(i=0; (keyString[i] != 0); i++)
    {
        /* If character is found return index in string */
        if (key == keyString[i])
        {
              charLoc = i;
              break;
        }
    }
    return charLoc;
}
/* [] END OF FILE */
```

Document History

Document Title: AN69133 - PSoC® 3 / PSoC 5LP Easy Waveform Generation with the WaveDAC8 Component

Document Number: 001-69133

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3232074	MEH	09/27/2011	New Spec.
*A	3450410	MEH	11/29/2011	Update project to PSoC Creator 2.0. Updated template.
*B	3812634	MEH	11/23/2012	Updated Associated Part Family as “All PSoC 3 and PSoC 5LP parts”. Updated Software Version as “PSoC Creator™ 2.1 SP1 or later”. Updated Using these DACs in your Project (Updated figures 28 thru 32). Replaced PSoC 5 with PSoC 5LP in all instances across the document.
*C	4082747	MEH	07/31/2013	Replaced the counters in the “4_WaveDAC8_DTMF_EN” project with PWMs.
*D	4815750	MEH	06/29/2015	Updated projects and application note to reflect version 2.0 of the WaveDAC8, which is now part of the PSoC Creator installation. Updated template
*E	5459483	MEH	10/03/2016	Changed “and” to “an” in section 1. Updated template Added links to code examples
*F	5639500	MEH	05/11/2017	Updated projects to PSoC Creator 4.0 Updated template

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

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

© Cypress Semiconductor Corporation, 2011-2017. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.