Playing WAV Files with a PSoC Device

AN13945

Author: Chris Paiano
Associated Project: Yes
Associated Part Family: CY8C27143 (project), CY8C27xxx, CY8C29xxx
Software Version: PSoC Designer™ 4.3
Associated Application Notes: None

Application Note Abstract

The PSoC's internal ROM is used to store samples converted from computer sound files. A Psuedo Random Sequence (PRS) generator module is used for a low overhead method of output. A simple Visual Basic 6 application (executable and source code) is used for converting standard 8-bit mono .WAV files into a formatted, ready-to-use PSoC C header.

Introduction

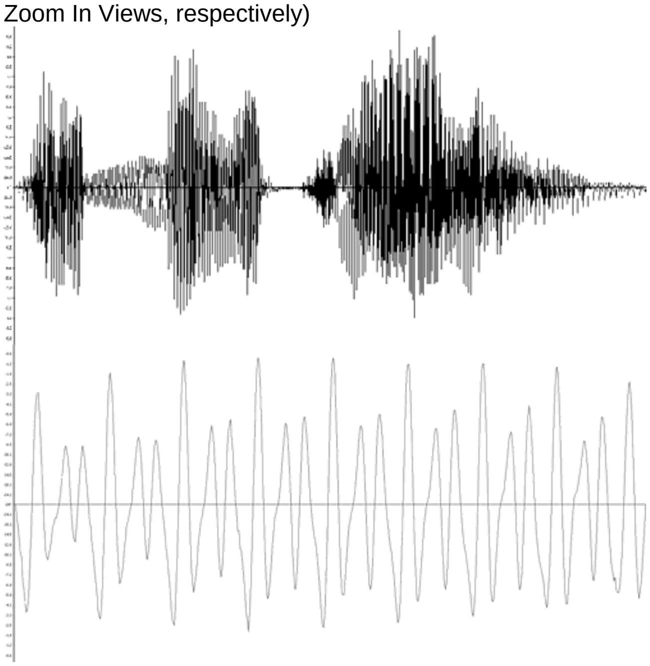
Sound at its lowest level is vibrating air molecules moving sensors in our ears (tympanic membrane or eardrums). Also, sound waves cannot propagate in a vacuum, demonstrated because there is no sound in outer space. Sound waves also travel through solids and liquids but with restrictions that end up altering the sound (for example, a conversation underwater).

Sounds are defined by the frequencies and amplitudes of the air vibrations over time; or how fast, how much, and when the molecules shake. The shapes of the individual vibrations also contribute to the sound. Our brains take care of identifying sounds after the eardrum receives them with incredibly fast and fuzzy memory comparisons.

By setting up a bionic eardrum (a microphone –basically just a membrane that picks up air vibrations and turns them into a voltage) and looking at various sounds with an oscilloscope (or by recording as digital audio with a computer), sound waves become visible.

Figure 1 is an example of a sound wave – particularly, of the "One more time!" voice recording used in this application note's project (original .WAV file is also included). The top wave is a zoomed out view of the entire phrase, while the bottom wave is a zoomed in view of the beginning of the word "more." In this view, the exact pattern and shape of sound waves that the speaker transfers to the air in front of it is seen.

Figure 1. "One More Time" Voice Recording (Zoom Out and Zoom In Views, respectively)



Besides amplification, reproduction of the sounds becomes possible by using these microphone output voltages to drive another membrane's position (a speaker). The speaker vibrates at the rate and intensity it wants to currently transfer to the air in front of it, and the resultant sound waves propagate outward for nearby biological and bionic eardrums to receive.

The difference between analog and digital audio is the method the microphone voltages are directed to drive the speaker to reproduce sound waves. Analog audio routes the voltage directly; digital audio involves a conversion of the voltages into representative numbers before being converted back into voltages to drive a speaker.

[+] Feedback

Analog audio has the advantage of never being forced into discrete steps of amplitude and frequency. This means that the reproduction of sound is (theoretically) exact.

Digital audio, at higher sample frequencies and bit rates (how many numbers are available to represent the range of voltages), reproduce any sound that a human ear responds to. Although the reproduction is not technically exact, it is close enough that human ears cannot tell the difference.

Analog audio is converted to digital audio for a computer (or microprocessor/microcontroller) to make use of it. Furthermore, there are many good audio effects that are achieved completely in the analog domain, but none compare to the variety and flexibility of Digital Signal Processing (DSP) effects – that requires digital audio.

Digital audio recordings have the distinct advantage of random access, at any time. If a computer has the sound in memory (or even on a hard drive), it is possible to play back any part of a digital sound file. Analog audio recordings require you to wind or rewind the tape and cue up the start position ahead of time, to have access to a certain portion of audio.

Software and PSoC Implementation

The PSoC has many options and is capable of reproducing digital audio from samples. The most common is the use of a DAC that directly translates samples to voltages. An alternative is to represent the samples with the pulse width of a PWM, and using a simple R/C circuit to convert these pulse widths to voltages.

However, a DAC module takes up valuable analog resource space. Its possible output frequencies are limited and it is power hungry (especially with the required analog output buffer).

A PWM is a better choice, as every PSoC has many more digital blocks than analog blocks. Driving a digital pin consumes considerably less current. The resulting digital output is converted to a voltage with an R/C circuit at high switching frequencies and with some additional filtering at lower switching frequencies.

With an 8-bit PWM module, the maximum switching frequency that is achieved, using all 8 bits of pulse width, is around 200 kHz (48 MHz clock divided by 256 counts). This is theoretically fine, as it switches above the human range of hearing (~20 kHz) and is not audible. However, if a 16-bit PWM module is desired for increased fidelity, the maximum switching frequency is around 700 Hz and very audible. This makes 16-bit sample playback unfeasible with a PWM.

The pulse width of a PWM module is updated via an interrupt right at the start of the count to avoid pulse glitches. This tends to overrun the PSoC, generating interrupts at a 200 kHz rate. So, either the PWM gets slowed down (potentially introducing more audible switching noise) or a slower synchronized interrupt generator is required and is tricky to configure. Either way, the PSoC's processing time is used up fast with the interrupts.

There is another lesser known alternative – the Psuedo-Random Sequence (PRS) generator. This handy little digital PSoC module has a Compare Out signal that is used directly with a simple R/C circuit to represent sound wave voltages. It is updated with a new sample at any time without noise, and its switching frequency is typically around 12 MHz for both 8-bit and 16-bit samples.

The way the PRS module works to represent digital audio is fascinating. The module itself is configured to generate a random stream of numbers evenly distributed around its range (for the purpose of this application note – 8 bits or 256 counts).

After the module is started with a seed value (simply a value given to kick start the random sequence generation), the seed register is used as a compare register. As the random numbers are generated at the clock rate (12 MHz), they are compared to the value in the compare register and the Compare Out signal is driven either high or low, depending on the result.

The Compare Out signal, when viewed on an oscilloscope, is a very fast and random square wave. However, because the number stream is evenly distributed around its range, an interesting phenomenon occurs. The Compare Out signal's overall duty cycle in any given window of time is equal to the value in the Seed/Compare registers. This is due to the probability that any supplied random number is above or below the value in the Seed/Compare registers.

When you write the new value (sample) to the Seed/Compare registers, the resultant duty cycle begins to "fade" to the new value. Because the PRS module is completely refreshed every clock cycle, this happens very quickly and imperceptibly – entirely too fast for any effect on audio signals. Therefore, the PRS module becomes an ideal method through which digital audio is represented with literally zero processing overhead and only a single digital block (for 8-bits).

Besides the PRS module, all that is needed for PSoC to play back digital audio samples is a sample rate clock to control exactly when each new sample is loaded into the PRS's Seed/Compare registers for output. This takes some math to get close to standard sample rates (11.025 kHz, 22.050 kHz, and 44.100 kHz). But with the VC1/VC2/VC3 dividers and an 8-bit counter, just about any rate is possible.

This particular application note makes use of 11.025 kHz sampling frequency to match the input sound file. Actually, the PSoC was able to hit 11.019 kHz which is close enough.

Note The play back sound samples at different frequencies, then they are sampled at results in either a higher/faster or a lower/slower effect that may or may not be desirable based on the particular application.

The PSoC code, illustrated in Appendix A. PSoC Source Code, demonstrates one way to implement the necessary logic. A basic flowchart is shown in Figure 4 and the configuration of the PSoC's resources are shown in Figure 5, Figure 6, Table 2, Table 3, Table 4, and Table 5.

Converting WAV to PSoC

Do you have a sound file on the computer that you like to implement into the PSoC project? Take the following into consideration:

1. What format is the sound file in? If it is not already in a standard 8-bit mono PCM .WAV format, you either need to convert it or modify the included Wav2H converter application to suit the need.
2. How large is the sound file (assuming it is already in an 8-bit mono PCM WAV format)? This is an indication of how much space is required of the PSoC's internal ROM, excluding about 45 bytes for the WAV's header information. The application note project uses a CY8C27143 that has approximately 16 kilobytes of ROM.

If the WAV file is too large, implement the following:

- Remove any unnecessary silence at the beginning or end of the sound file. This requires a basic WAV editing application. Silence takes up the same amount of storage space as sound in PCM WAV files.
- Reduce the sampling rate of the sound file. This is best done in a good WAV editing application that resamples the sound internally (such as Sound Forge).
- Scale up to a CY8C29xxx family PSoC that doubles the amount of available ROM to 32 kilobytes.

Theoretical maximum sound file length in seconds versus sampling frequency and PSoC ROM size are shown in Table 1. This data takes into account the 540 bytes of configuration or execution code that is in this application note's project, and leaves room for nothing else. The formula for this table is:

$$MaxLength = \frac{\left[\left(1024 * PsocRomSizeInKB\right) - 150\right]}{SampleFreqInHz}$$

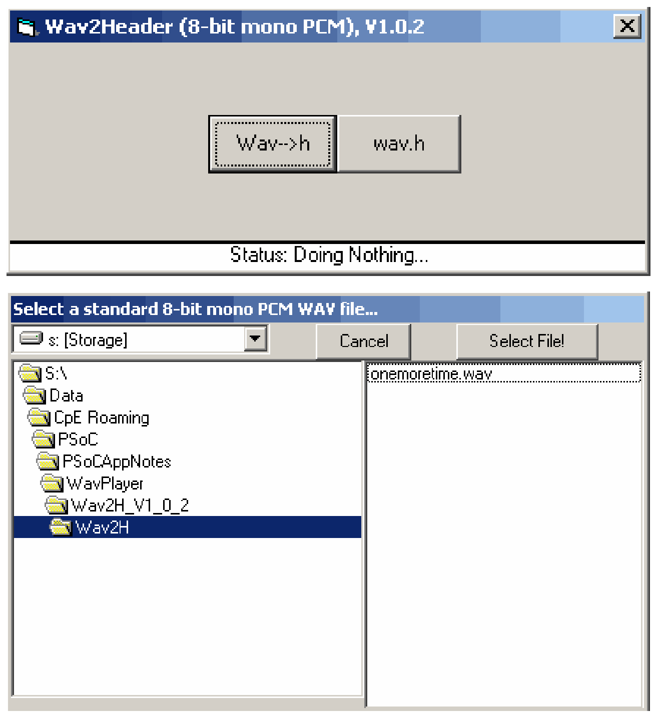
Equation 1

Table 1. Sampling Frequency vs. Maximum WAV Length

Sampling Frequency (kHz)	4 kB PSoC (Seconds)	16 kB PSoC (Seconds)	32 kB PSoC (Seconds)
8.000	0.4445	1.9805	4.0285
11.025	0.3225	1.4371	2.9232
22.050	0.1613	0.7185	1.4616
44.100	0.0806	0.3593	0.7308

When you have an 8-bit mono PCM WAV file that fits into the project, the next step is to run it through the Wav2H converter application and generate a PSoC header file. Figure 2 displays a screenshot of this application.

Figure 2. Screenshots of Wav2Header Conversion Application



Note Installation of the Microsoft Visual Basic 6 (Service Pack 5) Runtime Library is necessary to execute this application, and of course Visual Basic 6 to edit and recompile this application. At the time of writing this application note, the MSVB6sp5 Runtime Library is available at:

http://www.microsoft.com/downloads/details.aspx?FamilyID=bf9a24f9-b5c5-48f4-8edd-cdf2d29a79d5&DisplayLang=en

Using Wav2H

To use Wav2H:

1. Click **Wav→h**.
2. Acknowledge the information on what this program does (click **OK**) and select the WAV file in the browser window.
3. As Wav2H processes the WAV file's format information, it displays various progress messages. Click **OK** to continue.

Note If the WAV file does not conform to the format supported, Wav2H displays an error message with an explanation.

After all the messages are finished, the actual data extraction begins. When the status bar goes back to “Doing Nothing,” the PSoC header file is completed. It is saved as *wav.h* in the Wav2H application directory.

To view this file, click the *wav.h* button to launch the preferred text viewer automatically. Now, you either copy and paste the text file into the PSoC's *main.c* program or add the *wav.h* file to the project.

Assuming you are working with the project that comes with this application note (that is a great starting point for any WAV-playing PSoC project), copy the newly generated *wav.h* file over the old one. Now calculate how to achieve the sampling frequency with the included counter and change the array variable names to match the program (or vice versa). Now the PSoC is ready to play back digital audio.

The included project plays the WAV sound upon startup in its current state. A macro, mOutputWave, is included to cycle through the digital samples and make them available to the sample rate counter's interrupt. This updates the PRS Seed/Compare registers that is basically a command that plays the sound back once.

Hardware Implementation

Due to the extremely high frequency of the PRS module, a simple R/C filter easily removes the digital component in the audio. A 4.7k series resistor and a .1 uf capacitor to ground give a clean 1v signal.

Improvements and Possibilities

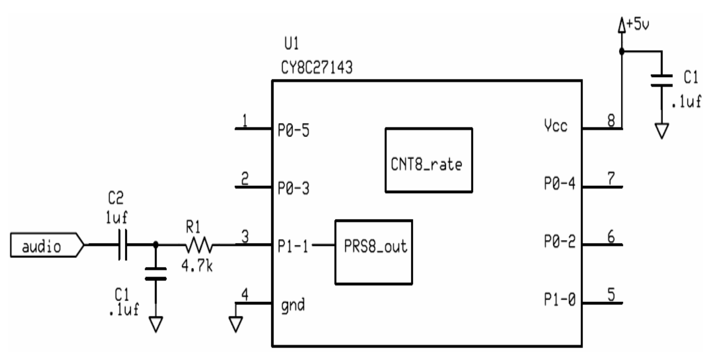
The PSoC has limited memory to store samples. Adding some external memory/EEPROM or a Flash drive interface enables playing back much longer sound files.

By adding some sort of compression algorithm, either algorithmically or through a codec chip, the duration of time of the stored samples in the PSoC's internal ROM is increased dramatically. Linear PCM WAV files are completely uncompressed and lend themselves to direct representation of the sound waves they attempt to reproduce. The PRS module still requires uncompressed samples to reproduce sounds. As long as decompression happens in real time inside the PSoC (or externally through another chip), compression is viable.

Three simple effects that are easily achieved with no additional overhead are:

1. **Speeding Up Playback** – Have the samples output at a faster rate than when they were recorded; they sound higher in pitch and faster.
2. **Slowing Down Playback** – Follow the same concept; only output the samples at a slower rate than when they were recorded; they sound lower in pitch and slower.
3. **Reverse Playback** – Alter the mOutputWave macro (duplicate or create a new macro) to cycle through the audio samples backwards from last to first.

Figure 3. Schematic



[+] Feedback

Appendix A. PSoC Source Code

main.c:

```
#include <m8c.h>          // part specific constants and macros
#include "PSoCAPI.h"      // PSoC API definitions for all User Modules
#include "wav.h"
//-----------------------------------------------------------------------------------------
#define      bPOLY_255    0xB8         // Modular Polynomial = [8,6,5,4]
#define      bSEED_255    0xFF         // Seed value
#define      InitPrs      PRS8_Out_WritePolynomial(bPOLY_255);
PRS8_Out_WriteSeed(bSEED_255); PRS8_Out_Start()
//-----------------------------------------------------------------------------------------
unsigned int i;
unsigned char NextSample;
BOOL SampleOutput=0;
#define mOutputWave i=0; while(i <= onemoretimeNumSamples) \
      { \
         NextSample=onemoretime[i]; \
         SampleOutput=0; \
         i++; \
         while(!SampleOutput); \
      }
//-----------------------------------------------------------------------------------------
void main()
{
   InitPrs;
   CNT_Rate_EnableInt(); CNT_Rate_Start();
   M8C_EnableGInt;
   mOutputWave     //enable this line to output the wave once at startup

   while(1)
   {
//    mOutputWave  //enable this line to output the wave continuously and repeatedly
   }
}
CNT_RateINT.asm:
include "PRS8_Out.inc"
push A
push X
   mov A, [_NextSample]
   mov reg[PRS8_Out_SEED_REG], A
   mov [_SampleOutput],1
pop X
pop A
```

Appendix B. Program Flowchart and Pin Configuration

Figure 4. Program Flowchart

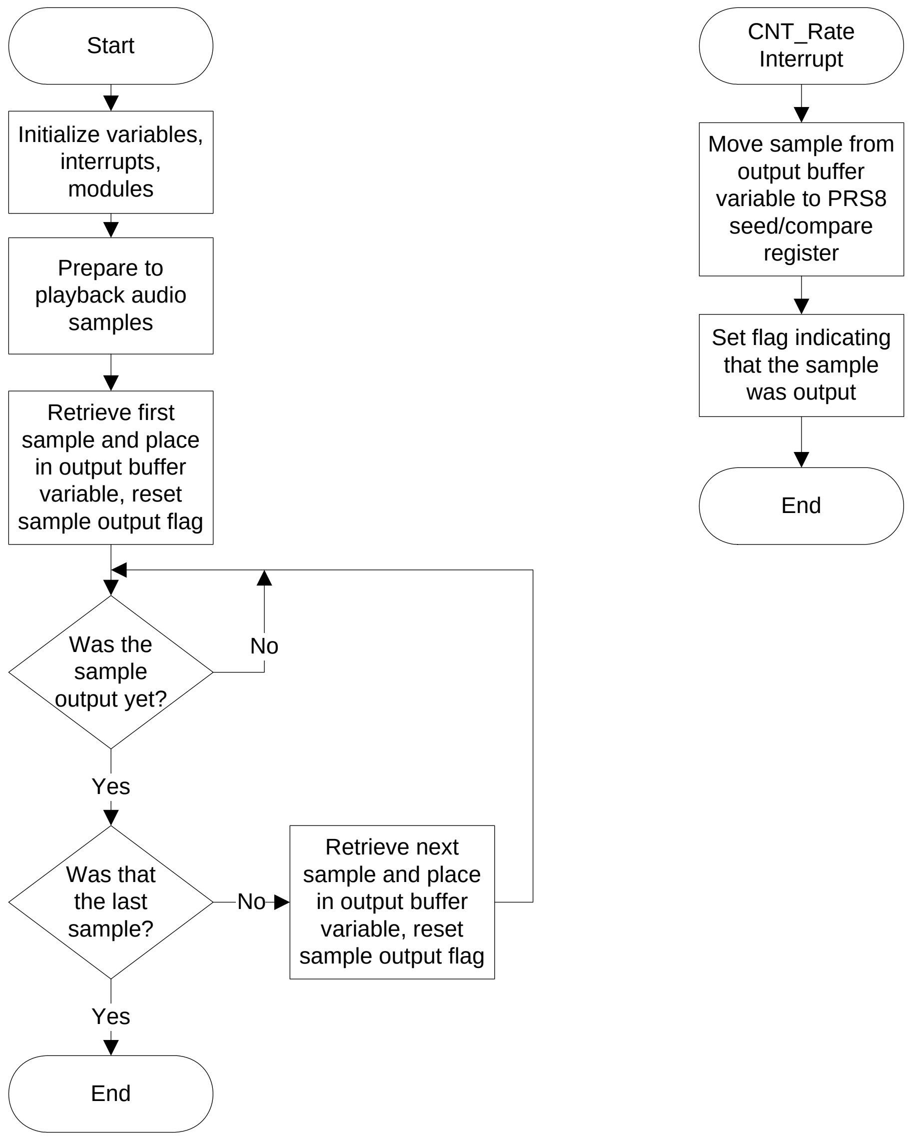


Figure 5. PSoC Pin Configuration

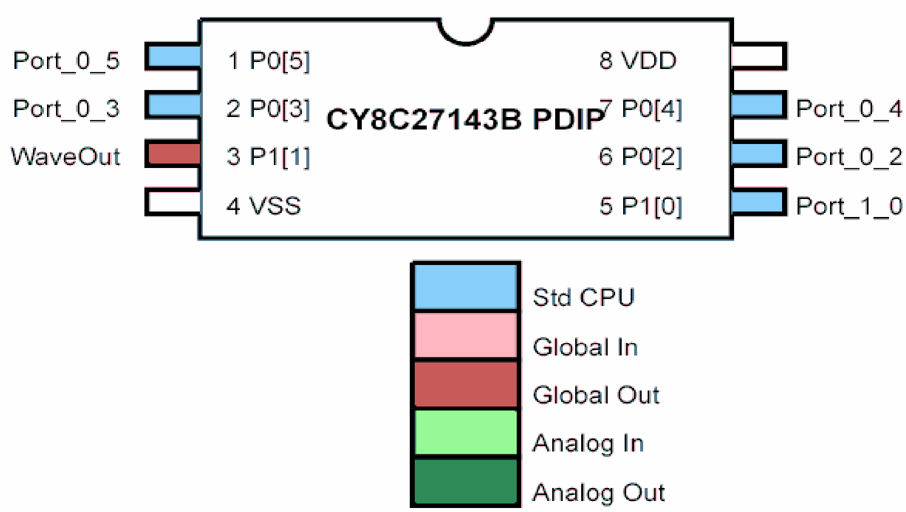
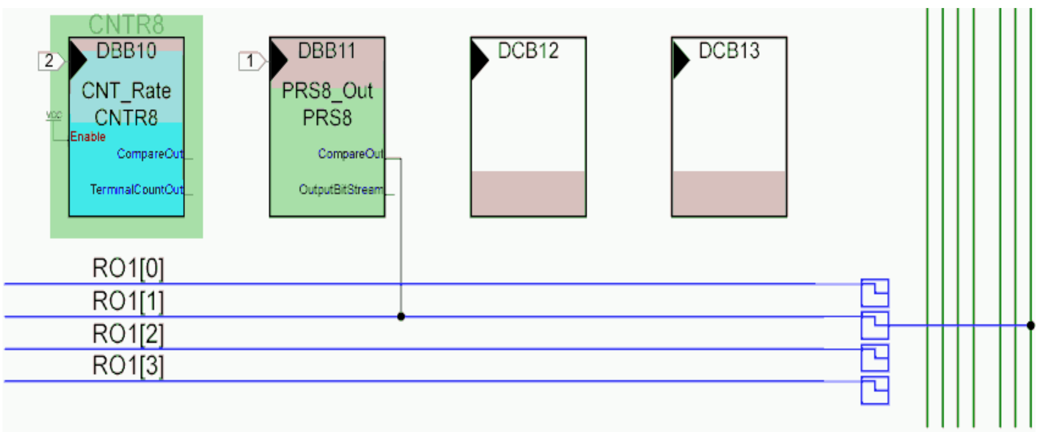


Figure 6. PSoC Module Configuration GUI



Appendix C. Project Settings

Table 2. CNT_Rate PSoC Module Configuration Settings

User Module Parameters	Value
Clock	VC2
ClockSync	Sync to SysClk
Enable	High
CompareOut	None
TerminalCountOut	None
Period	98
CompareValue	1
CompareType	Less Than Or Equal
InterruptType	Terminal Count
InvertEnable	Normal

Table 3. RPS8_Out PSoC Module Configuration Settings

User Module Parameters	Value
Clock	VC1
OutPutBitStream	None
CompareOut	Row_1_Output_1
CompareType	Less Than Or Equal
ClockSync	Sync to SysClk

Table 4. PSoC Global Resource Settings

Global Resources	Value
CPU_Clock	12_MHz (SysClk/2)
32K_Select	Internal
PLL_Mode	Disable
Sleep_Timer	512_Hz
VC1=SysClk/N	2
VC2=VC1/N	11
VC3 Source	SysClk/1
VC3 Divider	1
SysClk*2 Disable	No
Analog Power	All Off
Ref Mux	(Vdd/2)+/-BandGap
Op-Amp Bias	Low
A_Buff_Power	Low
Trip Voltage	4.18V
LVDThrottleBack	Disable
Supply Voltage	5.0V
Watchdog Enable	Disable

Table 5. PSoC Global Port Settings

Name	Port	Select	Drive	Interrupt
Port_0_2	P0[2]	stdCPU	High Z Analog	DisableInt
Port_0_3	P0[3]	stdCPU	High Z Analog	DisableInt
Port_0_4	P0[4]	stdCPU	High Z Analog	DisableInt
Port_0_5	P0[5]	stdCPU	High Z Analog	DisableInt
Port_2_0	P1[0]	stdCPU	High Z Analog	DisableInt
WaveOut	P1[1]	GlobalOutOdd_1	Strong	DisableInt

About the Author

Name:	Chris Paiano
Title:	Owner, CpE
Background:	24+ years programming experience 7+ years PSoC-specific experience
Contact:	http://www.chrispaiano.com chris.paiano@gmail.com

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone: 408-943-2600
Fax: 408-943-4730
http://www.cypress.com/