AN14278

PSoC® 1: Measurement and Compensation of the Internal Low Speed Oscillator

Author: Kristopher Young

Associated Project: Yes
Associated Part Family: CY8C24x23A, CY8C27x43, CY8C29xx66, CY8C21xxx
Software Version: PSoC® Designer™ 5.4
Related Application Notes: None

AN14278 outlines the use of the PSoC® 1 internal main oscillator (IMO) to measure and compensate for wider tolerance on the internal low speed oscillator (ILO). This application note describes how to use the main oscillator to measure the frequency of the low speed oscillator to compensate for the ILO inaccuracy. This provides greater accuracy for sleep duration and other low power clocking needs.

Introduction

The PSoC Internal Low Speed Oscillator (ILO) is a 32 kHz internal oscillator that is physically separate from the Internal Main Oscillator (IMO). The ILO is used for the sleep timer, watchdog timer, and also as a clock input for digital blocks. Unlike the IMO, the ILO continues operating even when the PSoC is in its low power sleep mode. The main drawback of the ILO is its accuracy — the ILO has an accuracy specification that corresponds to a tolerance of between -50% and +100%, compared to the IMO specification of ± 2.5%. The accuracy of the ILO is improved by using an external crystal, but there is a cheaper way to get more accurate results from the ILO. By implementing firmware that uses the more accurate IMO to measure the ILO, the PSoC determines its ILO frequency, allowing the application to compensate for the actual frequency of the ILO. A few firmware tricks provide this measurement and produce a compensation value, giving an extra boost in accuracy at the expense of a few lines of code!

Why Compensate?

The most common need for ILO compensation is to more tightly define sleep intervals. The PSoC offers four sleep intervals defined by their interruption rate (see Table 1). However, these rates are based on the ILO being 32.768 kHz. Most battery powered applications need to sleep most of the time and do something constructive at regular intervals. To get the design to work every 60 seconds and by basing sleep intervals directly on the ILO, the work is done as quickly as 30 seconds, or as slowly as 120 seconds. This is good enough for some designs, but others do not tolerate this kind of inaccuracy.

Table 1. PSoC Sleep Timer Intervals

Sleep Interval OSC_CR[4:3]	PSoC Designer Setting	Sleep Period (nominal)	ILO Clocks
00b	512 Hz	1.95 ms	64
01b	64 Hz	15.6 ms	512
10b	8 Hz	125 ms	4096
11b	1 Hz	1 sec	32,768

The ILO performance using the compensation techniques described in this application note is not quite as accurate as the IMO, but approaches the IMO accuracy. According to experiments, using these techniques to compensate for the ILO frequency yields average timing within ± 10% over temperature and voltage. Using the earlier example with this compensated tolerance, the 60 second event occurs somewhere between 54 seconds and 66 seconds, which is a big improvement over the raw ILO tolerance. If this is still not accurate enough, there are a few additional options. Any reference frequency input is used to measure the ILO period instead of using the IMO. The measurement accuracy increases if an external reference frequency is used that is more accurate than the IMO. If all else fails, a 32.768 kHz crystal gives you parts-per-million accuracy for a dime.

Figure 1. What We Are Trying to Avoid

Compensation Tricks

The ILO has a trim register (ILO_TR) that controls a few aspects of its behavior. There is a frequency trim setting that consists of four bits calibrated during manufacturing as part of the testing process. There are also a couple of bias trim bits. It is to be noted that this is a write-only register.

One problem is that there are actually two power modes for the ILO. The high power mode is typically used when the part is active, while the low power mode is ALWAYS used when the device is asleep. The high power mode keeps the ILO frequency more stable with respect to voltage and temperature. Since the ILO behaves differently between the two modes, if the compensation is being done to more accurately perform in sleep mode, then we need a way of forcing the ILO into this low power mode before doing the measurement. This is done by setting the most significant bit in the ILO_TR register.

Since the ILO_TR register is write-only, how do we put the ILO into low power mode without overwriting the ILO trim value set during the factory testing of the device? The answer is we cannot. But we can put our own trim value into the frequency and bias trim bits at the same time that we are setting the low power mode bit. Note that by doing this, we may actually put ourselves further away from the ideal 32 kHz frequency that this register was trimmed toward in manufacturing. We are making the decision to trade off the absolute accuracy of the ILO to more accurately know its frequency. For example, a particular device may have an ILO frequency of 35 kHz. It is possible that when we overwrite the trim setting, the ILO frequency may be changed to 40 kHz. However, since we have set the low power bit, we are able to obtain a more accurate estimate of the ILO frequency in sleep mode. We then use this knowledge to allow our application to compensate for the known frequency by taking action after a specific calculated number of ILO cycles have occurred.

Also note that there are warnings in the *Technical Reference Manual* against changing the ILO trim settings. Our code uses a nominal trim setting that does not cause any problems. However, instability problems may result if extreme trim settings are applied. For this reason, it is recommended to only use the trim setting from the example code in this application note.

Code 1 shows the line of C code to force the ILO into low power mode and rewrite the trim value. Note that you only need to execute this once in your application; there is no need to re-trim or to change the power mode back to high. Since we are continually calibrating with respect to the IMO, we just keep the ILO in this same power mode and trim setting.

Code 1. C Statement for the ILO_TR Setting

```
//Set the ILO_TR to Low Power, Maximum
Bias, and
//set Frequency to Nominal Trim Value of
0xB
ILO_TR = 0x9B;
```

The Measurement

So now we have an ILO that is in the same power mode whether the PSoC is asleep or awake. What is next is the ILO measurement itself. There are several ways to do this measurement. We need to measure the number of IMO clocks in some period related to the ILO clock. The easiest way to do this is to run a timer clocked with the IMO (or a derivative of the IMO) and capture this timer on ILO-based events. Note that there is a bit of jitter on the ILO clock (especially in low power mode), so it is best to make the measurement on many ILO clock cycles to get a good average. The example project shows one way to do this by capturing sleep timer intervals by polling for capture with a 16-bit timer clocked with VC1 (set to 12 MHz). This measurement is also accomplished using code in the sleep interrupt service routine (ISR) to record values of a free running IMO-based timer. Another technique that can be used is to look at the value of an ILO-based timer before and after running through a fixed number of instruction cycles, which are, of course, based on the IMO.

Once the number of IMO clocks in an ILO clock period is determined, find the actual ILO period by multiplying the obtained number by the ideal IMO period. Knowing the actual ILO period also yields the actual sleep timer interval (refer again to Table 1). The next step is to translate this knowledge into a useful parameter. For the 60-second sleep example, this translation involves finding the number of actual sleep timer intervals that will occur in 60 seconds. Then we count sleep timer intervals and do our task when the correct number is reached. For our example project, we use a slightly different implementation by determining the period value for an 8-bit counter which, when clocked with our ILO, results in 2 millisecond periods. We only use the sleep timer for the ILO measurement itself.

To analyze the accuracy of the ILO measurement, an 8-bit counter that is clocked with the ILO is set to a period value of 255. Measuring the period of the output of this clock on a scope and dividing it by 256 yields the average ILO period. Upon first running the measurement routine on a CY8C29466-24PXI, the code produced an ILO period measurement of 304 VC1 counts, where VC1 was 12 MHz. This yielded an ILO of 39.5 kHz. Measuring the counter output on the scope produced a period of 6.56 ms / 256, which is a frequency of 39.0 kHz. So the calculated period is just over a percent higher than the measured frequency. With the measurement uncertainty on the scope and the tolerance on the IMO, the measurement appears to be an acceptable estimate of the ILO frequency.

Another Bump

There is one other bump in our path. The confirmatory measurement was made with the part awake and the ILO in low power mode. Do things change when the part goes into sleep mode? The answer is yes, so we are not quite done yet. Because the ILO and digital blocks are still active while sleeping, it was possible to observe that the 39.0 kHz ILO drops to around 35.6 kHz while sleeping, even with the ILO low power bit set in both modes. However, this difference between the awake ILO frequency and the asleep ILO frequency in the PSoC is generally consistent among all parts in the same product family and package. So in addition to setting the low power bit, a constant also needs to be added to or subtracted from the results of the firmware measurement to obtain the optimal estimate of the asleep ILO frequency. It is recommended that you test several units to find the appropriate constant for your specific device. Once the constant is put into the calculation, you get a reliable measurement of your sleeping ILO frequency. It is worth noting that the wake / sleep difference does vary some versus the ILO frequency of the device, with greater differences at greater frequencies. While this application note achieves acceptable performance using a simple constant to compensate for the difference, even more accurate performance could likely be obtained by using a more sophisticated offset calculation.

This measurement routine need to be called periodically, as changing temperature and voltage conditions changes the frequency of the ILO. The environmental conditions and accuracy requirements of the application determine how frequently the routine needs to be run.

A New Way to Wake

The sleep timer built in to the PSoC provides an interrupt which allows waking upon regular intervals, and then going back to sleep. For the PSoC to do something every 60 seconds, find the actual ILO period, convert to sleep periods, multiply 60 by this number, and get to a final count value. The PSoC increments a counter upon every sleep interrupt, and when it gets to this final count value, it performs the required task. However, remember that the sleep intervals are fixed to specific multiples of the ILO (see Table 1). What if you want to wake up more frequently than the fastest setting or more seldom than the slowest setting? This is done by substituting a Timer User Module for the sleep timer. The PSoC wakes upon any interrupt, so the sleep interrupt can be disabled and a digital block interrupt used instead. This provides a tremendous increase in flexibility with respect to the number of ILO cycles between wake events. And because of another trick in some devices, an 8-bit counter may be all you need, even for much longer sleep intervals. In the CY8C24x23A, CY8C29x66, CY8C21x34, CY8C24x94, and CY8C21x23 devices, there is actually a way to use the sleep timer output as an input clock for digital blocks. To do this, set the most significant bit of the OSC_GO_EN register, which drives the sleep timer output onto GOE[7]. Code 2 shows the line of code to do this. The sleep timer output signal being driven onto GOE[7] is the ILO already pre-divided by the number corresponding to the sleep timer setting from Table 1. Using this method with a sleep timer setting of 1 Hz, an 8-bit digital block provides a sleep timer interval of over 4 minutes. The processor never has to wake during this period, providing optimal low current performance.

Code 2: C Statement to enable the sleep timer output onto GOE[7] in compatible devices

```
OSC_GO_EN |= 0x80;
```

For devices that do not have the capability to drive the sleep timer output signal onto GOE[7] (for instance, the CY8C27x43), an alternative is to use another counter instance clocked with CPU_32_kHz to divide down the ILO output and generate the signal that would be used by the capture input for the ILO measurement. This would be equivalent to the OSC_GO_EN technique described above, but it uses extra digital resources to implement the counter.

Example Project

The example project targets the CY8C29466-24PXI. Figure 2 shows the pinout for the example project, while Figure 3 shows the digital block interconnect view for this project. A 16-bit timer (MeasureTimer) is clocked with VC1 (12 MHz). MeasureTimer's capture input is connected to RI0[3], which in turn is connected to GIE[7]. GOE[7] is connected to GIE[7] via the interconnect, which is shown by Figure 4. This connection was made by clicking on the GOE[7] bus and selecting "OutputToInput" as the "Interconnect" setting. As described previously, the measurement routine sets the most significant bit of OSC_GO_EN to drive out the sleep interval clock onto GOE[7], which is routed to the MeasureTimer capture input (code shown in code 2). Note that this connection enabled by the firmware write to OSC_GO_EN is not shown in the PSoC Designer interconnect view, since this connection is only enabled in firmware and not through the GUI. The result of the setup just described is that the capture input for MeasureTimer is now connected to the sleep timer output. Therefore MeasureTimer is configured to take a measurement of the time period between sleep intervals.

Figure 2. Example Project Pinout

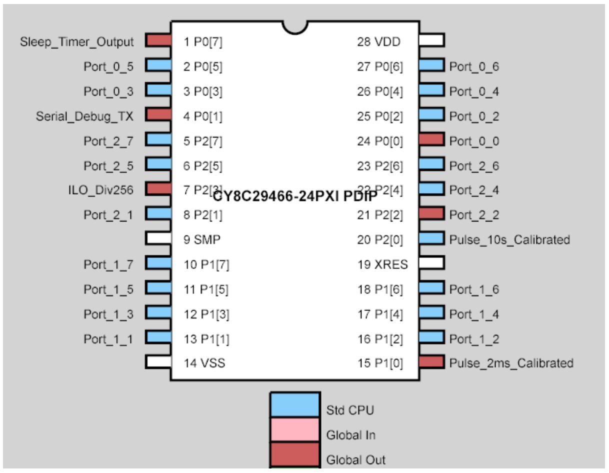


Figure 3. Example Project Interconnect View

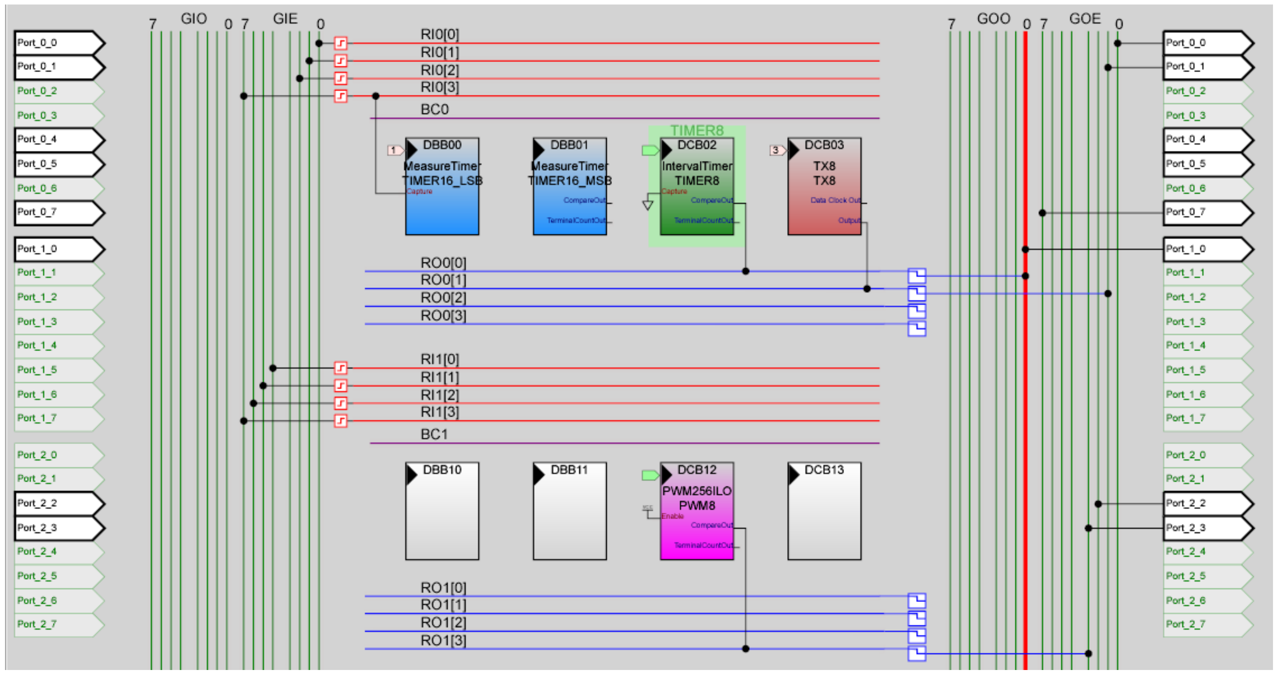
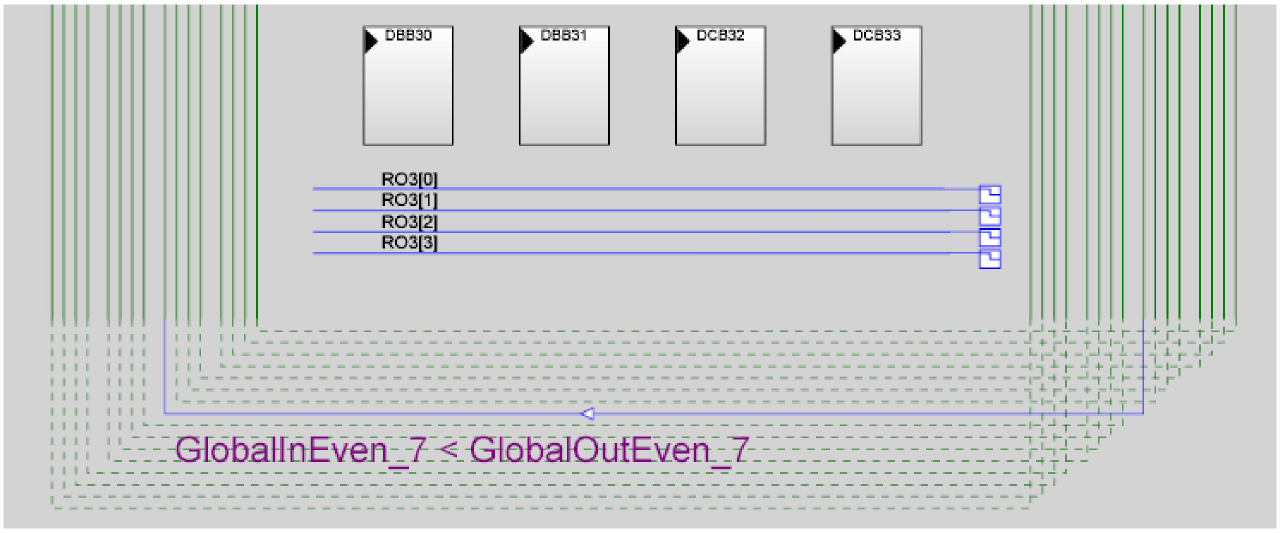


Figure 4. Example Project Interconnect View (bottom portion showing output to input connection)



The main routine calls the measurement routine, which uses MeasureTimer to determine the number of VC1 clocks between sleep intervals. The measurement routine sets up the measurement, takes the measurement by polling for capture, and then does a little processing on the result. The actual number of VC1 clocks between sleep intervals is placed in the *iRawMeasVal* global variable. The measurement routine then processes this value further by dividing it by 256. This division by 256 includes dividing by 64 to convert the measurement from sleep interval to the ILO period (note the assumption of the 512 Hz sleep timer setting), and an additional divide by four to reduce the number of possible measurement values. The result is also checked against the maximum and minimum possible values of the ILO, and offset by the shortest period limit so that the result is indexed to zero. The short period limit offset and the divide by four are both done to facilitate using a lookup table calculation, which is discussed next. The processing also adds in an offset to account for the wake / sleep difference in ILO frequency.

To go from the number of VC1 clocks in an ILO clock period (called N) to the clock period value to be loaded into the IntervalTimer, use the equation ((2 ms * 12 MHz / N) – 1). Subtract one because the value used by the timer needs to be one less than the actual period count. Instead of doing this math in the PSoC, the example project implements a lookup table routine with pre-calculated period values for all the possible values of N. Why waste CPU cycles (and power consumption) doing math when we can do the math in advance and give the processor a little more time to sleep. The input for this lookup table routine is the N value that has been divided by four and offset by the lower period limit. These were both done to create an 8-bit zero indexed value, which is easily implemented in a PSoC lookup table. Since the IntervalTimer is only 8 bits, the period adjustment is fairly coarse, so we do not lose a lot by doing this. For finer period control, the resolution of the IntervalTimer should be increased, and the measurement and lookup routines be modified accordingly. A spreadsheet is included that shows how the lookup table values were calculated using the equation above and compensating for the divide by four and the limit offset. Since the table is being calculated in a spreadsheet, it is straightforward to add more advanced calculations, such as incorporating a sleep / wake offset that change with frequency.

This project has an 8-bit timer (IntervalTimer) used to create a 2 ms interrupt that is accurate when the PSoC is sleeping. The IntervalTimer is clocked by the ILO and has a period value that is determined using the ILO measurement. The 2ms output pulse from IntervalTimer can be found on P1[0]. There is a software counter in the 2 ms interrupt service routine that re-measures the ILO and provides a pulse on P2[0] every 10 seconds (perfect for blinking an LED).

There are a few additional debugging features in this example project that make it easier to figure out how well it is working. P2[3] (output of PWM256ILO) provides a pulse with a period 256 times the actual ILO period. P0[7] is the sleep timer output signal that is being driven on to GOE[7]. The signals on P1[0], P2[0], P0[7], and P2[3] are all active when the PSoC is awake and asleep. P2[3] is a good way to measure the actual ILO frequency with an oscilloscope, while P1[0] is useful for testing the effectiveness of the ILO measurement and IntervalTimer period value calculation. If you do not have an oscilloscope handy, connecting an LED to P2[0] and using a stopwatch to time the blinking (10-second period) is another way to verify the accuracy of the compensation. There is also a serial transmitter that can be enabled on P0[1] that outputs the ILO measurements and some other useful values at 19200 bps, with one stop bit, 8 data bits, and no parity. The serial transmitter functions only when the device is always awake. Whether the PSoC sleeps or not is a *#define* setting in *main.c*. Enabling the serial output is also a *#define* setting.

Waking Use

It is worth noting that the techniques described here are not limited to low power applications. Perhaps the PSoC does not sleep during an application, but knowledge of the ILO frequency is desired to save digital blocks or VCx clocking resources. In situations such as these, simply run the measurement routine without putting the ILO into low power mode and use the result without adding in the wake or sleep offset value.

Summary

With the tips, tricks, and tools presented here, it is possible to measure the PSoC ILO against the more accurate IMO and use the measurement to make much more accurate use of the ILO than its specifications suggest is possible. This enables accurate clocking during the low power sleep mode without the use of any external crystals or frequency references.

About the Author

Name:	Kristopher Young.
Title:	Field Apps Engr Principal

Document History

Document Title: PSoC® 1: Measurement and Compensation of the Internal Low Speed Oscillator - AN14278

Document Number: 001-14278

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	1625546	KLY	10/15/2007	New application note
*A	3205763	KLY	03/25/2011	Updated Software Version as “PSoC® Designer™ 5. 1” in page 1. Updated A New Way to Wake: Updated description. Updated Example Project: Updated description. Added Figure 2 (for example project pinout). Added Figure 3 and Figure 4 (for example project interconnect view). Updated attached example project to PSoC Designer 5.1 SP1 and tested.
*B	4307559	RJVB	03/13/2014	Updated to new template. Completing Sunset Review.
*C	4579437	ASRI	11/25/2014	Updated Software Version as “PSoC® Designer™ 5.4” in page 1. Updated Abstract: Updated description. Updated Why Compensate?: Updated description. Updated Compensation Tricks: Updated description. Updated Example Project: Updated Figure 3 and Figure 4. Updated Summary: Updated description. Updated attached example project to PSoC Designer 5.4 and tested.

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

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc cypress.com/go/plc
Memory	cypress.com/go/memory
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

psoc.cypress.com/solutions

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

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

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

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com