

Glitch-Free PWM

By: Uroš Platiše
Associated Project: Yes
Associated Part Family: CY8C25xxx, CY8C26xxx
PSoC Designer Version: 4.2

Summary

PSoC™ digital blocks can operate as 8-bit pulse width modulators (PWM). Due to the special design of the PSOC, these modules do not incorporate glitch-free hardware logic but instead logic must be considered in software. This document presents a solution with a first-order predictor adequate to precisely generate any function.

Introduction

Pulse width modulation represents one of the most versatile ways of expressing signals in a digital control. Such a signal can be amplified easily and, consequently, is the key element of all power supplies and power converters. Discrete values that are referred to as the duty cycle of some period are represented as a ratio of the time when output is active, t_{active} , to the whole period, T . This is presented in Figure 1 and Equation (1).

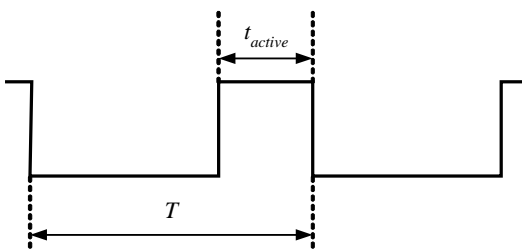


Figure 1. Duty Cycle

$$duty = \frac{t_{active}}{T} \quad (1)$$

The values are transmitted in counts with a period of T . Normally, T and t_{active} are of forms:

$$T = (N+1) dt \quad (2)$$

and

$$t_{active} = n dt. \quad (3)$$

dt is the smallest time step equal to the PWM clock frequency, f_{clk} . The number of active steps, usually called pulse width, is n and $N+1$ is the total number of steps per period.

Both n and N are referenced from zero because of the hardware implementation.

The PSOC PWM is implemented as a down counter with a period register that holds the N value and a compare register (also called pulse width register) that holds the n value. The PSOC block runs down to zero from the maximum value defined in the period register. At the zero point, the down counter reloads the content from the period register and repeats this cycle. Output is set high whenever the down counter is less than (LESS) the value stored in the compare register or less-than-or-equal-to (LESS-EQUAL) the value stored in the compare register. Otherwise, it is cleared.

The length of the period determines the resolution of the PWM that is identical to the effective number of bits:

$$B = \log_2(N+1) \quad (4)$$

N , n , dt , and type of comparison (LESS or LESS-EQUAL) parameters completely define the PWM output.

PWM output frequency is defined as:

$$f_{pwm} = \frac{f_{clk}}{N+1} \quad (5)$$

The type of comparison defines the output according to n .

When the comparison is set to LESS-EQUAL (i.e., $down_counter \leq compare_register$, the higher the value in the compare register brings out a higher value of duty cycle), n is at least 1 and ranges from 1 to maximum value. Using the LESS comparison ($down_counter < compare_register$), n ranges from 0 to maximum value-1. The maximum value represents the highest possible number to be stored in the compare register. In the case of an 8-bit PWM, this value is 255. When the period and compare values are both set to 255, the period will take 256 cycles and the output asserts for 255 cycles.

Finally, it is important to notice how the PWM block generates interrupts. Interrupts can be generated by terminal count (TC). TC is when the counter reaches zero and reloads with the contents of the period register, or when output rising edge (RE) occurs. Note that the output may remain in high (active) state or low (inactive) state for several periods, skipping the generation of an RE interrupt.

Changing the Pulse Width (n)

PWM steady conditions do not require special attention. Users can simply modify the pulse width once in a while. Possible glitches would be insignificant over several tens or hundreds of periods.

We shall focus on cases where the pulse width is changed constantly, from period to period. Because the PSoC PWM does not provide extra buffering on the pulse width register, writing a new value to it may cause an immediate change in the result of the down counter for compare/pulse width register comparison.

Proper modifications to the compare register can be achieved through interrupt handlers or by observing interrupt status flags. Indeed, many power converters require maximum pulse widths equal to 50% of the period (duty cycle = 50% $\geq n = N/2$). The TC interrupt should be used in these cases, as shown in Figure 2.

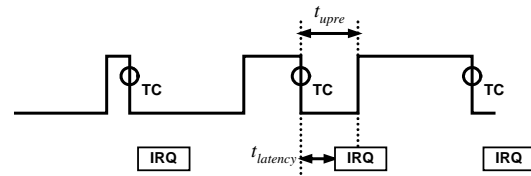


Figure 2. Terminal Count Interrupt

Note that changes written to the compare register in the interrupt handler IRQ caused by the TC interrupt of the current (i^{th}) period (past period) are valid for the following ($i+1$) period. Maximum glitch-free pulse width is limited by interrupt latency time plus the time needed to update the compare register. In Figure 2, t_{upre} represents a maximum update-to-rising edge window in which the compare register is safely updated.

Generating Glitches

The story becomes more interesting when the duty cycle is in the upper range close to 100% or when it varies over all ranges from 0 to 100%. The following figure shows an example of how easily a glitch may be generated.

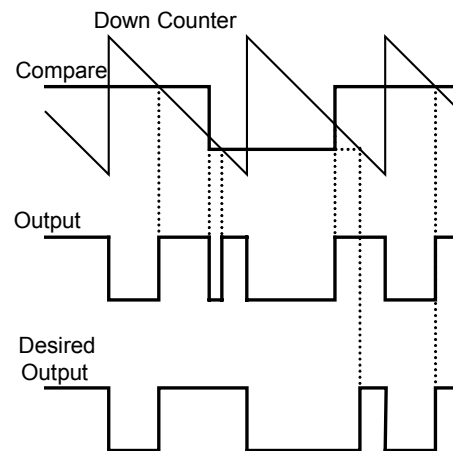


Figure 3. Glitch Example

In general, we may conclude that the PWM compare register of lower duty cycle signals (up to approximately 70% depending on f_{pwm} and CPU speed) may safely be updated on TC interrupts and on RE interrupts with higher duty cycles. For perfect operation, updates on RE interrupts need an additional delay estimator, as shown later.

Implementation of a PWM with a full range (0 to 100%) of duty cycles is required to use a combination of RE and TC interrupts.

Here, we shall propose a solution with a first-order predictor. A predictor is used to estimate the optimal interrupt source in subsequent periods. A first-order predictor fulfills the requirements of most applications.

Solution with First-Order Predictor

Before exploring an example pattern of a PWM signal, rules, notes and basic features of the first-order predictor are described.

Let us assume a PWM signal that is over several periods. Each period is assigned a pulse width, $W(i)$, where $W(0)$ is the width of the current period ($i=0$), $W(1)$ is width of the next period ($i=1$), $W(2)$ is the width of the second period, etc. Our purpose is to find an algorithm that implements:

$$W(i) = \text{func}(i). \quad (6)$$

$\text{func}(i)$ is an arbitrary function, such as sin-wave, triangle, saw-tooth, etc.

The pulse width must be updated at least once per period to obtain the desired stream. This update is, of course, always done inside the interrupt service routine (ISR), also referred to as the interrupt handler. The purpose of the predictor is to best select the interrupt source so the pulse width is updated in time.

Predictor Rules

The first-order predictor uses a current pulse width $W(0)$ and one pulse width in advance $W(1)$ to estimate the optimal interrupt source. As said, there are two different interrupt sources but three different cases:

- TC: terminal count interrupt source
- RE: rising edge interrupt source
- TCRE: terminal count instead of rising edge when RE cannot be generated

TCRE, the last interrupt source, uses a TC interrupt whenever an RE interrupt cannot be generated (when output is always high). It is identified as a different interrupt source for easier understanding.

As we have three different cases, the predictor has nine transition possibilities. Predictor rules are summarized in the following table.

States A, B, and C represent the current state in period i , and states 1, 2, and 3 represent a new optimal interrupt source for the next period, $i+1$. Hereinafter, we assume $i=0$, unless otherwise specified. Later, we shall use these state signatures in describing transitions.

Table 1. Predictor Transition Possibilities

A: TC	B: RE	C: TCRE	
$W \leq T$ [Update]	$W \leq T$ [SkipUp] [ParUp]	$W \leq T$ [SkipUp] [ParUp]	1: TC
$M > W > T$ [Update]	$M > W > T$ [Wait] [Update]	$M > W > T$ [Wait] [Update] [SkipNextRE]	2: RE
INVALID	$W \geq M$ [Update]	$W \geq M$ [Update*]	3: TCRE

- W stands for pulse width $W(1)$.
- T is equal to PWM_THRESHOLD and defines the transition point of TCRE interrupt sources. It is in the range $[0, N]$.
- M is equal to PWM_MAXREWIDTH and defines the maximum width at which output is always high. For example, if $N=255$, $n=254$, and less comparison is selected, then RE is never generated since the output is always set. In this case $M=255$.
- [Update] command instructs the pulse width to be updated and the next $W(1)$ to be fetched.
- [Update*] command may be skipped because the output state is always high (this is just a note).
- [SkipUp] command skips the pulse width update of value $W(1)$ in this interrupt and performs it in the next interrupt.
- [ParUp] command performs a partial update. It changes the current width to the width of value PWM_THRESHOLD. This gives extra time to the TC interrupt executed at the end of the period to properly modify the pulse width. Note that this command waits for a specific amount of time before the width is updated. If this amount of time is too little, several glitches may occur.
- [Wait] command adds delay of form $T_{delay} = [W(0)-W(1)]k - n$ only if $T_{delay} > 0$. The Wait function is described later.

- [SkipNxtRE] command skips the next RE interrupt.

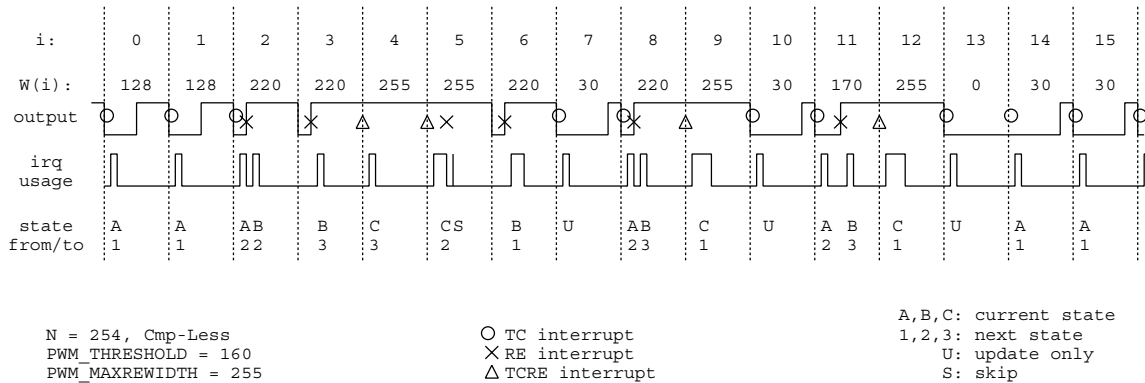


Figure 4. PWM Pattern Example

The first-order predictor shall fail only in two cases:

- Accurate transition from pulse width under PWM_THRESHOLD to pulse widths close to the maximum pulse width. This is due to the TC interrupt latency and time it takes to update the compare register.
- Transition from pulse width under PWM_THRESHOLD to maximum pulse width. This is an INVALID transition.

Description of Transitions/Example Pattern

Figure 4 illustrates all nine transition possibilities, places of pulse width updates, and all three types of interrupts. The represented sequence has 16 periods. Each transition shall be described separately.

Transition A1

State A (TC) to 1 (TC) is the default state after power up. Thus, the initial value must be under PWM_THRESHOLD for proper operation. Transition A1 means no change in interrupt source as seen in cycles 0, 1, 14 and 15.

The pulse width is updated at the very beginning of the ISR to support the highest possible frequencies. The update is marked with the [Update] command in Table 2.

Transition A2

State A (TC) to 2 (RE) occurs whenever the pulse width $W(i)$ is over PWM_THRESHOLD and below PWM_MAXREWIDTH. This condition occurs in cycles 2, 8 and 11.

Two interrupts are ALWAYS generated per period; one TC from the previous period (State A) to set up the width of this period, and one RE to set up the next period. Note that one RE interrupt is already represented by next State B.

The pulse width is updated at the very beginning of the ISR to support the highest possible frequencies. The update is marked with the [Update] command in Table 3.

Transition A2 has one weakness as it can be seen in cycles 2 and 8. Whenever $W(i)$ is close to the PWM_MAXREWIDTH and, of course, over PWM_THRESHOLD, the ISR has little time to write a new pulse width to the compare register on TC interrupt at the beginning of cycles 2 and 8. This problem is represented in this case. It is likely that the pulse width in periods 2 and 8 will not be 220 but a little less. A higher-order predictor can help solve this case.

From this we can also find the maximum possible value of the PWM_THRESHOLD. This was partly described previously under "Changing the Pulse Width (n)" and shall be described in greater detail later.

Transition A3

This is an INVALID transition and is not shown in the figure. This transition could be supported by the first-order predictor but:

- 100% duty cycle cannot be achieved even if a predictor supports that feature.
- The transition can be supported by higher-order predictors only – which look at least two cycles forward.

- There are not many applications that require accurate transition from below PWM_THRESHOLD to 100% PWM duty cycle.

Transition B1

State B (RE) to 1 (TC) occurs whenever the pulse width of the next period is under PWM_THRESHOLD. The steps to shorten the width are represented by the following:

1. Shorten the pulse width to PWM_THRESHOLD only, referenced as the [ParUp] command.
2. Update pulse width to the desired value, referenced as the [SkipUp] command.

The first step is done in the RE interrupt generated by State B. The second step is done in the first TC interrupt marked as U in Figure 4. High frequency designs may combine the two steps into a single step to reduce interrupt count.

Transition B2

State B (RE) to 2 (RE) occurs whenever the pulse width remains in the range above PWM_THRESHOLD and below PWM_MAXREWIDTH.

It should be stressed that before shortening the pulse width register, an extra delay is required before the update, otherwise a glitch may occur. This is marked as the [Wait] command followed by the [Update] command. The delay must be estimated in advance and is described in greater detail ahead.

Transition B3

State B (RE) to 3 (TCRE) occurs whenever the pulse width W(1) is to be set to its maximum. Transition is smooth and is represented in cycles 3, 8 and 11.

Transition C1

State C (TCRE) to 1 (TC) occurs whenever the pulse width changes from its maximum value to a value below PWM_THRESHOLD. The sequence to achieve accurate results is the same as described under Transition B1.

Transition C2

State C (TCRE) to 2 (RE) occurs whenever the pulse width changes from its maximum value to a value above PWM_THRESHOLD. The sequence is generally the same as Transition B2 except that an additional RE interrupt is generated.

The additional RE interrupt is skipped as referenced by the [SkipNxtRE] command.

Transition C3

State C (TCRE) to 3 (TCRE) occurs whenever the pulse width remains at its maximum value. The pulse width register does not need to be updated.

Changing the Interrupt Source

Changing the interrupt source is done by setting the proper bit in the PWM register. Whenever the interrupt source is changed from TC to RE and output is high or becomes high, the RE interrupt is **always** generated! This includes the transition from maximum pulse width, where the output is high and the RE interrupt source is selected.

RE-TC Threshold Parameter

Let us focus on the threshold that defines the border above which the RE interrupt source is selected and below or equal to the selected TC interrupt source. The RE interrupt sources are less welcome. They require additional CPU time, using delay loops when the pulse width is shortened. Let us define:

$$IRQ_CYCLES = (T_{irqup} + T_{irq\ latency}) / dt \quad (7)$$

T_{irqup} is the maximum time required by the interrupt handler without delay loops when LESS comparison is chosen and:

$$PWM_MAXREWIDTH = N \quad (8)$$

...when LESS-EQUAL comparison is chosen.

Wait Function

The WAIT function is represented by the [Wait] command in Table 1q. It removes possible glitches that could occur when the pulse width is decreased. The WAIT function is called before the update to the compare register is done in RE and TCRE interrupts.

The WAIT(dn) function declares the time to loop. It may be written as follows:

$$T_{wait\ loop} = (W(0) - W(1) - IRQ_CYCLES) dt \quad (9)$$

Usually smooth functions $func(i)$ have low first derivatives, $W(0) - W(1) < dW$. In such cases, computation of $T_{wait\ loop}$ is too time consuming.

It is easier to declare constant $T_{wait\ loop}$ calculated or estimated for worst case using only dW to update the pulse width register. $T_{irq\ latency}$ is the maximum possible interrupt invocation latency and IRQ_CYCLES referenced to the PWM clock frequency and not to the CPU clock frequency. The value $T_{irqup} + T_{irq\ latency}$ is also equal to the t_{upre} , as described earlier.

Therefore, the `PWM_THRESHOLD` variable should be set as:

$$PWM_THRESHOLD = N - IRQ_CYCLES \quad (10)$$

Under these considerations, the interrupt handler CPU consumption is the lowest.

Maximum Pulse Width Parameter

This important parameter defines the most upper border when the RE interrupt cannot be used, as it is not generated if the output does not change.

$$PWM_MAXREWIDTH = N+1 \quad (11)$$

Example of Implementation

Appendix A provides source code of the first-order implementation. 'C' language was used to describe the algorithm. In an effort to be clear and easy to understand, the code is not completely optimized.

There are two API functions; one to start the operation and a second to modify the pulse width from anywhere in the program. The core of the algorithm resides in the ISR `gfpwm_update()`. This ISR should have the highest interrupt priority. Lower priority increases maximum latency time, which influences `PWM_THRESHOLD`. Consequently, the ISR uses more CPU power due to the [Wait] command.

Applications that update pulse widths in every period should modify the `gfpwm_w2` variable directly in the ISR `gfpwm_update()` as shown in the source code by reading the `foo_table[0]`.

Example source generates the same stream as shown under the example pattern. The time consumed by the ISR is set as HIGH state on output pin P1[3]. Additional synchronization is provided on output pin P1[1]. It is set HIGH only when the first sample is fetched from the `foo_table[0]`.

Conclusion

Applications rarely require an accurate, fast transition from 0% duty cycle to 99% or 100%; the first-order predictor cannot accurately handle such situations. Nevertheless, such functions as:

- Sine wave
- Triangular
- Saw-tooth (with positive gradient only)
- Other smooth functions

...are perfectly generated using the first-order predictor without a single glitch.

Implementation may differ from application to application. Sometimes a single interrupt is used for timer and PWM updates and some applications require very fast PWM updates. The last case requires good examination of the algorithm. Some parts of the code may be rearranged or changed, as for example; double update in states B1 or C1 may be replaced by a single update to reduce interrupts.

The algorithm described here is general purpose and handles all possible cases and states of the first-order predictor.

Appendix A: Source Code

```

// main.c, Glitch Free PWM
// Uros Platise (c) 27. July 2003, <uros.platise@ijs.si>
//
//      This source represents Glitch Free Implementation with first order predictor.
//
// Features:
//      - full range of duty cycle support [0, 100]%
//      - simple user API
//      - operates up to 100 kHz
//      - If W(i)=foo(i), then foo(i) may be any smooth function with
//      limited first derivate dfoo(i)/dt
//
// Do not forget:
//      - to estimate delay loops
//      - invalid transition from pulse of width PWM_MAXREWIDTH to pulse
//      of width under PWM_THRESHOLD!
//      - hardly handles transitions from pulse with below PWM_THRESHOLD to
//      pulse width close to maximum duty cycle!

#include <gfpwmap.h>

#define PWM_INITVAL          128          // initial PWM pulse width
#define PWM_THRESHOLD        160          // TC/RE threshold
#define PWM_MAXREWIDTH       255          // Max Rising Edge Irq Pulse Width

// Uncomment below line to analyze IRQ usage on output pin P1[3] and sync on P1[1]
#define PWM_IRQUSAGE_DEBUG

// Variables

unsigned char gfpwm_w0;
unsigned char gfpwm_w1 = PWM_INITVAL;
unsigned char gfpwm_w2 = PWM_INITVAL;

// Example of a PWM pattern to demonstrate all possible transition cases
unsigned char phase = 0;                // table index

const unsigned char foo_table[16] = {   // sample table
    128, 128, 220, 220, 255, 255, 220, 30, 220, 255, 30, 170, 255, 0, 30, 30
};

//-----
// Glitch Free PWM Interrupt
// Note: avoid calling functions from this ISR. If you do call any function
// from this ISR, C compiler will add plenty of pushes and pops, consuming a
// lot of CPU power.
//-----

#define PWMIRQ_RE_SKIP 3
#define PWMIRQ_TCRE 2
#define PWMIRQ_RE 1
#define PWMIRQ_TC 0

#pragma interrupt_handler gfpwm_update

void gfpwm_update()
{
    static unsigned char int_type = PWMIRQ_TC;
    unsigned char lock = 1;

    if (int_type == PWMIRQ_RE_SKIP)
    {
        int_type = PWMIRQ_RE;
        return;
    }

#ifdef PWM_IRQUSAGE_DEBUG
    if (phase>0)
    {
        // turn on P1[3] - CPU/PWMIRQ usage analyzer
        PRT1DR = 8;
    }
    else
        // turn on P1[3] and P1[1] - to provide ext. sync. for scope
        PRT1DR = 10;
#endif

    // SERVE Terminal-Counter Interrupts
    if (int_type == PWMIRQ_TC)

```

```

    {
        // be direct, call to sub-routine will confuse C compiler and eat extra cycles!
        PWM8_1_PWIDITH_REG = gfpwm_w1;
    }

    // Obtain new value
    // Note: gfpwm_w2 may be also updated outside the ISR. In such case
    // comment the foo_table line (and phase++ of course) but you should
    // keep the lock statement!
    if (int_type == PWMIRQ_TC || gfpwm_w1 > PWM_THRESHOLD)
    {
        gfpwm_w2 = foo_table[(unsigned char)phase & 0x0F];
        phase++;
        lock = 0;
    }

    // SERVE Rising Edge and TC-RE Interrupts
    if (int_type > PWMIRQ_TC)
    {
        if (gfpwm_w1 > PWM_THRESHOLD)
        {
            // WAIT(W(0)-W(1))
            // Note: delay should be estimated in form of: delay = (W0-W1)*loop_dly - irq_dly
            // where loop_dly is unit delay and irq_dly is preceding delay of a ISR, such as
            // computation of new value etc... To verify proper settings an output IRQ usage
            // on pin P1[3] may be turned on.
            if (gfpwm_w0 > gfpwm_w1)
            {
                gfpwm_w0 -= gfpwm_w1;
                gfpwm_w0 >>= 1;
                for (; gfpwm_w0 > 0 /* irq_dly */; gfpwm_w0--) // k = 0.5 // irq_dly = 0
                {
                    // unit delay - see asm to predict the number of cycles
                }
            }
            PWM8_1_PWIDITH_REG = gfpwm_w1;
        }
        else
        {
            // fixed delay down to threshold - give time TC to provide accurate value
            // in the next cycle
            if (gfpwm_w0 > PWM_THRESHOLD)
            {
                gfpwm_w0 -= PWM_THRESHOLD;
                gfpwm_w0 >>= 1;
                gfpwm_w0 += 10;
                for (; gfpwm_w0 > 0 /* irq_dly */; gfpwm_w0--) // irq_dly = 0
                {
                    // unit delay - see asm to predict the number of cycles
                }
            }
            PWM8_1_PWIDITH_REG = PWM_THRESHOLD;
        }
    }

    // 1st order predictor
    if (gfpwm_w1 >= PWM_MAXREWIDTH)
    {
        int_type = PWMIRQ_TCRE;
        PWM8_1_FUNC_REG = 0x31; // Terminal Count
    }
    else if (gfpwm_w1 > PWM_THRESHOLD)
    {
        if (int_type == PWMIRQ_TCRE)
            int_type = PWMIRQ_RE_SKIP; // very special case, simply skip the next RE interrupt
        else
            int_type = PWMIRQ_RE;
        PWM8_1_FUNC_REG = 0x39; // Rising Edge
    }
    else
    {
        int_type = PWMIRQ_TC;
        PWM8_1_FUNC_REG = 0x31; // Terminal Count
    }

    // W(i) FIFO
    if (lock == 0)
    {
        gfpwm_w0 = gfpwm_w1;
        gfpwm_w1 = gfpwm_w2;
    }

#ifdef PWM_IRQUSAGE_DEBUG
    // turn off P1[3] - CPU/PWMIRQ usage analyzer
    PRT1DR = 0;
#endif
}

```

```

//-----
// Glitch Free PWM API / next value is determined in the ISR itself!
//-----

void gfpwm_start()
{
    PWM8_1_WritePulseWidth(PWM_INITVAL);
    PWM8_1_Start();
    PWM8_1_EnableInt();
}

// To update PWM width from any point in the source, ISR, main, ...
void gfpwm_width(unsigned char new_width)
{
    gfpwm_w2 = new_width;
}

// Test Main
void main()
{
    gfpwm_start();
    M8C_EnableGInt;
    while(1);
}

```

About the Author

Name: Uroš Platiše
Title: Research and Development
Background: Mixed signal designs,
 computer and CPLD/FPGA
 programming, control,
 sensors, measurement, ...
Contact: Uros Platise
 Seljakovo naselje 45
 SI-4000 Kranj
 Slovenia
 Email: uros.platise@ijs.si
 Web: <http://www.andeuos.org>

Cypress MicroSystems, Inc.
 2700 162nd Street SW, Building D
 Lynnwood, WA 98037
 Phone: 800.669.0557
 Fax: 425.787.4641

<http://www.cypress.com/> / <http://www.cypress.com/support/mysupport.cfm>

Copyright © 2004 Cypress MicroSystems, Inc. All rights reserved.

PSoC™, Programmable System-on-Chip™, and PSoC Designer™ are trademarks of Cypress MicroSystems, Inc.

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

The information contained herein is subject to change without notice. Made in the U.S.A.