Implementing 16-Bit PWM Using the PCA

Relevant Devices

This application note applies to the following devices:

C8051F000, C8051F001, C8051F002, C8051F005, C8051F006, C8051F010, C8051F011, and C8051F012.

Introduction

Pulse-width modulated (PWM) waveforms are often used in closed-loop feedback and control applications, such as controlling the on/off state of a heating element used to regulate the temperature of a laser in a DWDM (Dense Wavelength Division Multiplexing) application. In some applications, the built-in 8-bit PWM mode of the Programmable Counter Array (PCA) provides insufficient resolution for the task. This application note describes how to implement a PWM waveform achieving 16-bit resolution using the PCA in ‘High-Speed Output’ mode with minimal software overhead. Software examples in ‘C’ and assembly are provided at the end of this note.

Background

Figure 1 shows an example PWM waveform. The frequency of the PWM signal of the class used in feedback control applications is largely unimportant, as long as the waveform is ‘fast enough’, such that the step response of the control system is much slower than one period of the PWM signal. Signal information is encoded instead in the duty cycle of the waveform, the ratio of the time the waveform is high over one period of the PWM signal. The input to the PWM implementation is a number, usually an integer, that is proportional to the duty cycle desired at the output.

Implementation

There are many methods for implementing a PWM waveform in an 8051-based design: software loops, polled or interrupt-driven timers, etc. The examples in this note use the Programmable Counter Array (PCA). Using the PCA for this application results in a substantial reduction in CPU bandwidth requirements over any polled scheme (software or timer-based), and eliminates timing jitter caused by

---

\[ \text{duty cycle} = \frac{t_H}{t_P} \]

Figure 1. Example PWM Waveform
variable interrupt latency in interrupt-driven timer-based designs.

**An Introduction to the PCA**

The PCA consists of a single 16-bit counter/timer and five capture/compare modules, as shown in Figure 2. The counter/timer has a 16-bit timer/counter register (PCA0H:PCA0L), an associated mode register (PCA0MD), which selects the time base, and a control register (PCA0CN), which contains the timer/counter run control and the modules’ capture/compare flags. Each capture/compare module has a configuration register (PCA0CPMx) which selects the module’s mode (Edge-triggered Capture, Software Timer, High-Speed Output, or PWM) and a 16-bit capture/compare register (PCA0CPHn:PCA0CPLn).

Because the capture/compare modules share a common time base, they can operate in concert, to provide phase-locked excitation waveforms for motor control, for example. Or, because each module has its own control and capture/compare registers, it can operate independently of the other modules, as long as the routines for any module do not affect the shared time base (by stopping or resetting the counter/timer or by changing the counter/timer clock source).

The examples in this note configure the PCA modules to act independently; the routines for the allocated module affect only the configuration register and the capture/compare register for that module. The PCA Mode Register (PCA0MD) is configured once, then left alone, and the timer/counter register (PCA0H:PCA0L) is left free-running.

**Selecting the PCA Time Base**

The PCA time base can be derived from one of four sources: SYSCLK/12, SYSCLK/4, Timer0 overflows, or high-to-low transitions on an external pin, ECI. A block diagram of the PCA counter/timer is shown in Figure 2.
shown in Figure 3.

As will be shown in the following sections, the selection of the PCA time base determines the resulting frequency of the PWM waveform. As mentioned earlier, the frequency of the PWM waveform is generally not important, so long as it is ‘fast enough’.

One timing option that is not immediately obvious is that the PCA can be clocked at the SYSCLK rate by selecting Timer0 overflows as the PCA clock source, and setting Timer0 in 8-bit auto-reload mode with a reload value of ‘0xFF’.

The examples in this note all configure the PCA to use SYSCLK / 4 as the PCA clock source.

![Figure 3. PCA Counter/Timer Block Diagram](image-url)
8-Bit PWM Using the PCA

We first present a method for generating a PWM waveform with 8-bit precision, for completeness, and to introduce the PWM mode of the PCA.

In this mode, the capture/compare module is configured in PWM mode as shown in Figure 4. The period of the waveform at $CEX_n$ is equal to 256 PCA clocks. The low-time of the signal at $CEX_n$ is equal to the 8-bit value stored in the low-byte of the module’s capture/compare register ($PCA_{0 CPL_n}$). This relationship is shown in Figure 5.

At every overflow of the low-byte of the main PCA counter ($PCA_{0 L}$), the high-byte of the module’s compare register is copied into the low-byte of the module’s compare register ($PCA_{0 CPL_n} = PCA_{0 CPH_n}$). The duty cycle is changed by updating $PCA_{0 CPH_n}$. The copying process ensures glitch-free transitions at the output.

The duty cycle of the output waveform (in %) is given by:

$$\text{dutycycle} = \frac{256 - PCA_{0 CPH_n}}{256} \times 100$$

Because $PCA_{0 CPH_n}$ can contain a value between 0 and 255, the minimum and maximum programmable duty cycles are 0.38 % ($PCA_{0 CP0H} = 0xFF$) to 100 % ($PCA_{0 CP0H} = 0x00$). The resolution of the duty cycle selection is:

$$\text{resolution} = \frac{1}{256} \times 100 = 0.38$$

The key advantage of 8-bit PWM mode is that no CPU intervention is required to maintain an output waveform of a fixed duty cycle. In fact, if the CIDL bit ($PCA_{0 MD.7}$) is set to ‘0’ (RESET state), the output waveform will be maintained even if the CPU is in IDLE mode.
Changing the duty cycle is implemented by a single 8-bit write to PCA0CPHn.

An example of 8-bit PWM mode is included in the file ‘PWM8_1.c’ at the end of this note.

Additional notes on 8-bit PWM mode:

1. The output CEXn can be held low by clearing the ECOMn bit (PCA0CPMn.6) in the module configuration register. This allows a 0% duty cycle waveform to be generated. Normal PWM output can be resumed by writing a ‘1’ to this bit OR by writing any value to PCA0CPHn.

2. Setting the MATn and ECCFn bits (PCA0CPMn.3 and PCA0CPMn.0 respectively) to ‘1’ will cause an interrupt to be generated on the falling edge of CEXn.

16-Bit PWM Using the PCA

To implement a PWM waveform with 16-bit precision, we configure a PCA module in High-Speed Output mode, as shown in Figure 6. In this mode, the CEXn pin is toggled, and an optional interrupt is generated, each time a match occurs between the main timer/counter register (PCA0H:PCA0L) and the module’s capture/compare register (PCA0CPHn:PCA0CPLn).

In the example code, the interrupt handler for the PCA module is implemented in two states: a rising-edge state and a falling-edge state, depending on which edge on CEXn initiated the interrupt. Note that the actual CEXn pin is decoded as the state variable.

During the rising-edge state, the module’s capture/compare register is updated with the compare value for the next falling-edge (this value is called PWM in the attached example code). During the falling-edge state, the module’s capture/compare register is

![Figure 6. PCA Configuration in High-Speed Output Mode](image-url)
loaded with the compare value for the next rising-edge, which is zero (0x0000). This is shown in Figure 7. Note that the period of the PWM waveform is 65536 PCA clocks.

The duty cycle (in %) is given by:

\[
dutycycle = \frac{PWM}{65536} \times 100
\]

The minimum and maximum allowed duty cycles are determined by the maximum time it takes to update the compare value after \(CEXn\) changes (which triggers the process interrupt). In both the ‘C’ example code and the assembly example code (‘pwm16_1.c’ and ‘pwm16_1.asm’ respectively), the minimum value for PWM is 7 PCA clocks (28 SYSCLK cycles in this case). This results in minimum and maximum duty cycle values of 0.01 % and 99.99 % respectively. The resolution of the duty cycle (in %) is:

\[
resolution = \frac{1}{65536} \times 100 = 0.0015
\]

or about 15 ppm (parts per million).

The CPU overhead required to process these interrupts is minimal. In the assembly example, processing both edges takes a total of 41 SYSCLK cycles, not counting the interrupt call and vector itself. Both edges must be processed every 65,536 PCA clocks, or 65,536 * 4 = 262,144 SYSCLKs, if the PCA clock is equal to SYSCLK / 4. CPU bandwidth consumed (in %) is equal to (41 / 262,144 * 100) = 0.015 %.

Also note that the CPU can be left in IDLE mode, as is done in the examples, since the PCA module interrupt will ‘wake up’ the core when required for processing.

The duty cycle can be changed by a single 16-bit write to the variable \(PWM\) in the examples.

\section*{n-Bit PWM Using the PCA}

A generalized case of 16-bit PWM, we present n-Bit PWM for applications requiring more than 8-bits of precision but less than 16-bits of precision. One motivation for adopting the n-Bit approach is to achieve a higher output frequency from the PWM than can be obtained in the 16-bit implementation.

In this example (‘PWMn_1.c’), two 16-bit variables are used: \(PWM\_HIGH\), which holds the number of PCA clocks for the output waveform to remain high, and \(PWM\_LOW\), which correspondingly holds the number of PCA clocks for the output waveform to remain low. The period of the

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure7.png}
\caption{Capture/Compare Register Loading for 16-Bit PWM}
\end{figure}
output waveform is given by the sum of these two variables:

\[ \text{period} = \text{PWMHIGH} + \text{PWMLOW} \]

The duty cycle (in \%) is given by:

\[ \text{dutycycle} = \frac{\text{PWMHIGH}}{\text{PWMHIGH} + \text{PWMLOW}} \times 100 \]

The resolution of the duty cycle (in \%) is:

\[ \text{resolution} = \frac{1}{\text{PWMHIGH} + \text{PWMLOW}} \times 100 \]

Similar to the 16-bit PWM case, the interrupt handler is implemented in two states, one for the rising-edge and one for the falling-edge. The primary difference is that in the 16-bit case, a constant (PWM or zero) was loaded into the PCA module’s compare registers. In the n-Bit case, a constant (PWM_HIGH or PWM_LOW) is added to the current value in the module’s compare register. The addition operation takes a few more cycles than loading a constant, which restricts the minimum high or low time of the output waveform a little more than the corresponding 16-bit solution.

Note: the n-Bit PWM solution can be used to generate a waveform of an arbitrary frequency by programming the appropriate high and low values into PWM_HIGH and PWM_LOW.
Software Examples

PWM8_1.c

Example source code for implementing 8-bit PWM.

The PCA is configured in 8-bit PWM mode using SYSCLK/4 as its time base. <PWM> holds the number of PCA cycles for the output waveform to remain low per 256-count period. The waveform is high for (256 - PWM) cycles. The duty cycle of the output is equal to (256 - PWM) / 256.

Because the 8-bit PWM is handled completely in hardware, no CPU cycles are expended in maintaining a fixed duty cycle. Altering the duty cycle requires a single 8-bit write to the high byte of the module’s compare register, PCA0CP0H, in this example.

Achievable duty cycle ranges are 0.38% (PCA0CP0H = 0xff) to 100% (PCA0CP0H = 0x00).

#include <c8051f000.h> // SFR declarations

#define PWM 0x80 // Number of PCA clocks for waveform to be low // duty cycle = (256 - PWM) / 256 // Note: this is an 8-bit value

void main (void) {
WDTCN = 0xde; // Disable watchdog timer
WDTCN = 0xad;

OSCICN = 0x07; // set SYSCLK to 16MHz, // internal osc.

XBR0 = 0x08; // enable CEX0 at P0.0
XBR2 = 0x40; // enable crossbar and weak // pull-ups

PRT0CF = 0x01; // set P0.0 output state to // push-pull
PRT1CF = 0x20; // set P1.6 output to // push-pull (LED)

// configure the PCA
PCA0MD = 0x02; // disable CF interrupt
PCA0CPL0 = PWM; // PCA time base = SYSCLK / 4
PCA0CPH0 = PWM;
PCA0CPM0 = 0x42; // CCM0 in 8-bit PWM mode
PCA0CN = 0x40; // enable PCA counter

while (1) {
    PCON |= 0x01; // set IDLE mode
}

/** END OF FILE **/

PWM16_1.c

//-----------------------------------------------------------------------------
// PWM16_1.c
//-----------------------------------------------------------------------------
//
// AUTH: BW
//
// Target: C8051F000, F001, F002, F005, F006, F010, F011, or F012
// Tool chain: KEIL C51
//
// Description:
// Example source code for implementing 16-bit PWM.
// The PCA is configured in high speed output mode using
// SYSCLK/4 as its time base. <PWM> holds the number of
// PCA cycles for the output waveform to remain high. The
// waveform is low for (65536 - PWM) cycles. The duty
// cycle of the output is equal to PWM / 65536.
//
// Due to interrupt service times, there are minimum and
// maximum values for PWM, and therefore the duty cycle,
// depending on interrupt service times. On the Keil C51
// compiler (Eval version), the minimum PWM value is 7
// PCA clocks; the maximum value is 65530. This equates
// to a minimum duty cycle of 0.01% and a maximum duty
// cycle of 99.99%. This assumes a PCA time base of SYSCLK/4
// and no other interrupts being serviced.
//-----------------------------------------------------------------------------
`#include <c8051f000.h>` // SFR declarations

// Global CONSTANTS

`#define PWM_START0x4000` // starting value for the PWM
// high time
`sbit PWM_OUT = P0^0;` // define PWM output port pin

// Function PROTOTYPES

void main (void);` void PCA_ISR (void);` // PCA Interrupt Service Routine

// Global VARIABLES

`unsigned PWM =PWM_START;` // Number of PCA clocks for
// waveform to be high
// duty cycle = PWM / 65536
// Note: this is a 16-bit value

void main (void) {
    WDTCN = 0xde; // Disable watchdog timer
    WDTCN = 0xad;
    OSCICN = 0x07; // set SYSCLK to 16MHz,
    // internal osc.
    XBR0 = 0x08; // enable CEX0 at P0.0
    XBR2 = 0x40; // enable crossbar and weak
    // pull-ups
    PRT0CF = 0x01; // set P0.0 output state to
    // push-pull
    PRT1CF = 0x20; // set P1.6 output to
    // push-pull (LED)

    // configure the PCA
    PCA0MD = 0x02; // disable CF interrupt
    PCA0CPL0 = (0xff & PWM); // PCA time base = SYSCLK / 4
    PCA0CPH0 = (0xff & (PWM >> 8)); // initialize PCA compare value
    PCA0CPM0 = 0x4d; // CCM0 in High Speed output mode
EIE1 |= 0x08; // enable PCA interrupt
EA = 1; // Enable global interrupts
PCA0CN = 0x40; // enable PCA counter
while (1) {
    PCON |= 0x01; // set IDLE mode
}

//-----------------------------------------------------------------------------
// PCA_ISR
//-----------------------------------------------------------------------------
// This ISR is called when the PCA CCM0 obtains a match
// PWM_OUT is the CEX0 port pin that holds the state of the current edge:
// 1 = rising edge; 0 = falling edge
// On the rising edge, the compare registers are loaded with PWM_HIGH.
// On the falling edge, the compare registers are loaded with zero.
// void PCA_ISR (void) interrupt 9
{
    if (CCF0) {
        CCF0 = 0; // clear compare indicator
        if (PWM_OUT) { // process rising edge
            PCA0CPL0 = (0xff & PWM); // set next match to PWM
            PCA0CPH0 = (0xff & (PWM >> 8));
        } else { // process falling edge
            PCA0CPL0 = 0; // set next match to zero
            PCA0CPH0 = 0;
        }
    } else if (CCF1) { // handle other PCA interrupt
        CCF1 = 0; // sources
    } else if (CCF2) {
        CCF2 = 0;
    } else if (CCF3) {
        CCF3 = 0;
    } else if (CCF4) {
        CCF4 = 0;
    } else if (CF) {
        CF = 0;
    }
}

// *** END OF FILE ***
DESCRIPTION: Example source code for implementing 16-bit PWM.

The PCA is configured in high speed output mode using
SYSCLK/4 as its time base. PWM holds the number of
PCA cycles for the output waveform to remain high. The
waveform is low for (65536 - PWM) cycles. The duty
cycle of the output is equal to PWM / 65536.

Due to interrupt service times, the minimum value for
PWM is 7 PCA cycles, and the maximum value is 65529.
This equates to a minimum duty cycle of 0.01068% and a
maximum duty cycle of 99.9893%.

If the PCA time base is changed to SYSCLK / 12, the min and
max values for PWM change to 3 and 65533 respectively,
for min and max duty cycles of 0.0046% and 99.9954%
respectively.

Processing the rising edge interrupt handler takes 18 cycles.
Processing the falling edge interrupt handler takes 19 cycles.

One interrupt handler is called for each edge, and there are
2 edges for every 65536 PCA clocks. Using SYSCLK / 4 as the
PCA time base, that means that 37 cycles are consumed for
dge maintenance for every (65536 * 4) = 262,144 SYSCLK
cycles, not counting vectoring the interrupt.
CPU utilization is (37 / 262,144)*100% = 0.0141%

Using SYSCLK / 12 as the PCA timebase, 37 cycles are
consumed for edge maintenance for every (65536 * 12) =
786,432 SYSCLK cycles. CPU utilization is (37 / 786,432)
= 0.0047%.

The period of the waveform is 65536 PCA clocks. Using
SYSCLK / 4 as the PCA time base, the period is 262,144 SYSCLK
cycles. Using the default internal oscillator at 2MHz, the
period is 131ms (7.6Hz). Using the 16MHz internal
oscillator (as in this example), the period is 16.4us
(61 Hz).

Using SYSCLK / 12 as the PCA time base, the period is
65536 * 12 = 786,432 SYSCLK cycles. Using the default
internal oscillator at 2MHz, the period is 393ms (2.5Hz).
Using the 16MHz internal oscillator, the period is 49.2ms
(20Hz).

In this example, the output is routed to P0.0, which is
also labeled 'PWM_OUT'.

;---------------------------------------------------------------------------

; EQUATES

$MOD8F000

PWM EQU 32768 ; Number of PCA clocks for waveform
to be high
duty cycle = PWM / 65536
; max = 65529 (99.9893% duty cycle)
; min = 7 (0.01068% duty cycle)
; Note: this is a 16-bit constant

PWM_OUT EQU P0.0 ; define PWM output port pin

;-----------------------------------------------
; RESET AND INTERRUPT VECTOR TABLE
;-----------------------------------------------

CSEG
org 00h
ljmp Main

org 04bh
ljmp PCA_ISR ; PCA Interrupt Service Routine

;-----------------------------------------------
; MAIN PROGRAM CODE
;-----------------------------------------------

org 0b3h ; start at end of interrupt handler ; space

Main:

; Disable watchdog timer
mov WDTCN, #0deh
mov WDTCN, #0adh

; Enable the Internal Oscillator at 16 MHz
mov OSCICN, #07h

; Enable the Crossbar, weak pull-ups enabled
mov XBR0, #08h ; route CEX0 to P0.0
mov XBR2, #40h

orl PRT0CF, #01h ; Configure Port 0.0 as Push-Pull

; Configure the PCA
mov PCA0MD, #02h ; disable cf interrupt,
mov PCA0CPL0, #LOW(PWM) ; PCA time base = SYSCLK/4
mov PCA0CPFH0, #HIGH(PWM)
mov PCA0CPM0, #4dh ; CCM0 in High Speed output mode

; Enable interrupts
orl EIE1,#08h ; Enable PCA interrupt
setb EA ; Enable global interrupts

mov PCA0CN, #40h ; enable PCA counter

jmp $ ;

;-----------------------------------------------
; CCF0 Interrupt Vector
;
; This ISR is called when the PCA CCM0 obtains a match
; PWL_OUT is the CEX0 port pin that holds the state of the current edge:
; 1 = rising edge; 0 = falling edge
; On the rising edge, the compare registers are loaded with PWL_HIGH.
; On the falling edge, the compare registers are loaded with zero.

PCA_ISR:
    jbc CCF0, CCF0_HNDL ; handle CCF0 comparison
    jbc CCF1, PCA_STUB ; stub routines
    jbc CCF2, PCA_STUB
    jbc CCF3, PCA_STUB
    jbc CCF4, PCA_STUB
    jbc CF,    PCA_STUB

PCA_STUB:
PCA_ISR_END:
    reti

CCF0_HNDL:
    jnb PWM_OUT, CCF0_FALL ; handle rising edge

CCF0_RISE:
    mov PCA0CPL0, #LOW(PWM)
    mov PCA0CPH0, #HIGH(PWM)
    reti

CCF0_FALL: ; handle falling edge
    mov PCA0CPL0, #00
    mov PCA0CPH0, #00
    reti

; rising edge takes 4+3+11 = 18 cycles
; falling edge takes 4+4+11 = 19 cycles

;-----------------------------------------------------------------------------
; END
;-----------------------------------------------------------------------------
END
; *** END OF FILE ***

PWMn_1.c

@Configuration

// PWMn_1.c

// AUTH: BW

// Target: C8051F000, F001, F002, F005, F006, F010, F011, or F012
// Tool chain: KEIL C51

// Description:
// Example source code for implementing an n-bit PWM.
// The PCA is configured in high speed output mode using
// SYSCLK/4 as its time base. <PWM_HIGH> holds the number of
// PCA cycles for the output waveform to remain high.
// <PWM_LOW> holds the number of PCA cycles for the output
// waveform to remain low. The duty cycle of the output
// is equal to PWM_HIGH / (PWM_HIGH + PWM_LOW).
Due to interrupt service times, there are minimum and maximum values for PWM_HIGH and PWM_LOW, and therefore the duty cycle, depending on interrupt service times. Regardless of the efficiency of the compiler, duty cycles between 1% and 99% should be very easy to achieve.

With the eval version of the Keil compiler, the minimum high and low counts are 20 PCA cycles each (max frequency is about 100kHz w/ 16MHz internal SYSCLK). This assumes no other interrupts being serviced, and PCA time base is SYSCLK / 4.

---

```c
#include <c8051f000.h> // SFR declarations

#define PWM_START 0x8000 // starting value for the PWM_HIGH time and PWM_LOW time
sbit PWM_OUT = P0^0; // define PWM output port pin

void main (void); // PCA Interrupt Service Routine

void main (void) {
    WDTCN = 0xde; // Disable watchdog timer
    WDTCN = 0xad;
    OSCICN = 0x07; // set SYSCLK to 16MHz, internal osc.
    XBR0 = 0x08; // enable CEX0 at P0.0
    XBR2 = 0x40; // enable crossbar and weak
```
// pull-ups

PRT0CF = 0x01; // set P0.0 output mode to push-pull
PRT1CF = 0x20; // set P1.6 output to push-pull (LED)

// configure the PCA
PCA0MD = 0x02; // disable CF interrupt
PCA0CPL0 = (0xff & PWM_HIGH); // PCA time base = SYSCLK / 4
PCA0CPH0 = (0xff & (PWM_HIGH >> 8)); // initialize PCA compare value
PCA0CPM0 = 0x4d; // CCM0 in High Speed output mode
EIE1 |= 0x08; // enable PCA interrupt
EA = 1; // Enable global interrupts
PCA0CN = 0x40; // enable PCA counter

while (1) {
    PCON |= 0x01; // set IDLE mode
}

//------------------------------------------------------------------------------
// PCA_ISR
//------------------------------------------------------------------------------
// This ISR is called when the PCA CCM0 obtains a match
// PWM_OUT is the CEX0 port pin that holds the state of the current edge:
// 1 = rising edge; 0 = falling edge
// On the rising edge, the compare registers are updated to trigger for the
// next falling edge.
// On the falling edge, the compare registers are updated to trigger for the
// next rising edge.
// void PCA_ISR (void) interrupt 9
{
    unsigned temp; // holding value for 16-bit math

    if (CCF0) {
        CCF0 = 0; // clear compare indicator
        if (PWM_OUT) { // process rising edge
            // update compare match for next falling edge
            temp = (PCA0CPH0 << 8) | PCA0CPL0; // get current compare value
            temp += PWM_HIGH; // add appropriate offset

            PCA0CPL0 = (0xff & temp); // replace compare value
            PCA0CPH0 = (0xff & (temp >> 8));
        } else { // process falling edge
            // update compare match for next rising edge
            temp = (PCA0CPH0 << 8) | PCA0CPL0; // get current compare value
            temp += PWM_LOW; // add appropriate offset

            PCA0CPL0 = (0xff & temp); // replace compare value
            PCA0CPH0 = (0xff & (temp >> 8));
        }
    }
}
PCA0CPL0 = (0xff & temp);  // replace compare value
PCA0CPH0 = (0xff & (temp >> 8));

}  

} else if (CCF1) {  // handle other PCA interrupt
    CCF1 = 0;  // sources
} else if (CCF2) {
    CCF2 = 0;
} else if (CCF3) {
    CCF3 = 0;
} else if (CCF4) {
    CCF4 = 0;
} else if (CF) {
    CF = 0;
}

}  // *** END OF FILE ***