FlashFlex Microcontroller Using the Programmable Counter Array (PCA)

Application Note
September 2008

1.0 INTRODUCTION

The Programmable Counter Array (PCA) present on the SST89E/V58RDx and SST89E/V516RDx is a special 16-bit timer that has five 16-bit capture/compare modules. Each of the modules can be programmed to operate in one of four modes: rising and/or falling edge capture, software timer, high-speed output, or pulse width modulator. Module 4 can be programmed as a Watchdog Timer in addition to the other four modes. Each module has a pin associated with it in port 1. Module 0 is connected to P1.3 (CEX0), module 1 to P1.4 (CEX1), module 2 to P1.5 (CEX2), module 3 to P1.6 (CEX3), and module 4 to P1.7 (CEX4).

2.0 PCA OVERVIEW

PCA provides more timing capabilities with less CPU intervention than the standard timer/counter. Its advantages include reduced software overhead and improved accuracy. The PCA consists of a dedicated timer/counter which serves as the time base for an array of five compare/capture modules. Figure 3-1 shows a block diagram of the PCA. External events associated with modules are shared with corresponding Port 1 pins. Port pins not used by the PCA modules can still be used for standard I/O. Each of the five modules can be programmed in any of the following modes:

- Rising and/or falling edge capture
- Software timer
- High speed output
- Watchdog Timer (Module 4 only)
- Pulse Width Modulator (PWM)

3.0 PCA TIMER/COUNTER

The PCA timer is a free-running 16-bit timer consisting of registers CH and CL (the high and low bytes of the count values). The PCA timer is common time base for all five modules and can be programmed to run at 1/12 the oscillator frequency, 1/4 the oscillator frequency, Timer 0 overflow, or the input on the ECI pin (P1.2). The timer/counter source is determined from the CPS1 and CPS0 bits in the CMOD SFR as shown in Table 3-1. Table 3-2 summarizes Modes 0-3 clock inputs at two common frequencies.

TABLE 3-1: PCA Timer/Counter Source

CPS1	CPS0	12 Clock Mode	6 Clock Mode
0	0	f_{OSC} /12	f_{OSC} /6
0	1	f_{OSC} /4	f_{OSC} /2
1	0	Timer 0 overflow	Timer 0 overflow
1	1	External clock at ECI pin (maximum rate = f_{OSC} /8)	External clock at ECI pin (maximum rate = f_{OSC} /4)

T3-1.0 2052

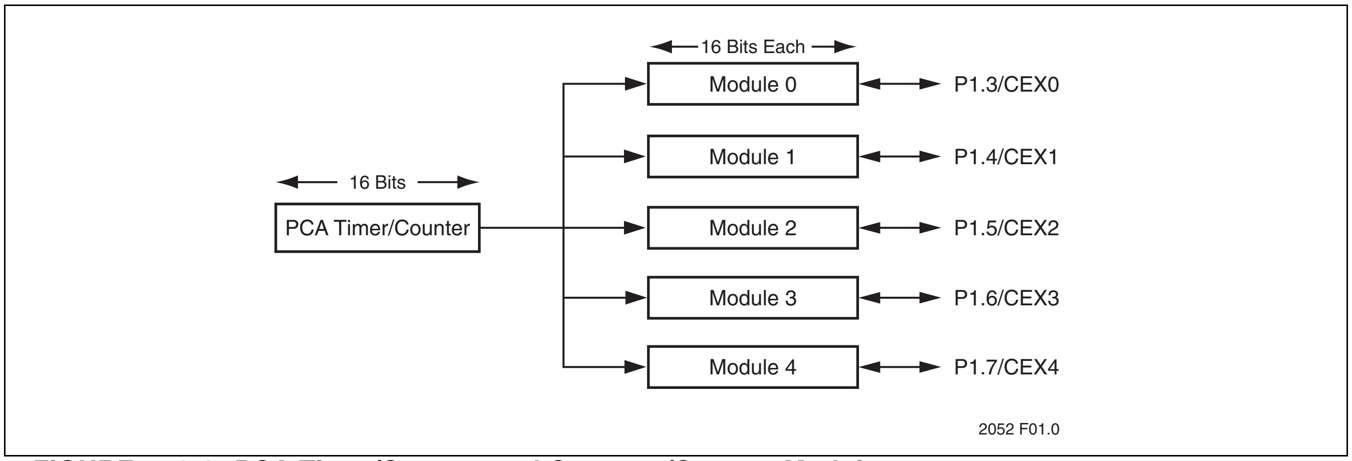

FIGURE 3-1: PCA Timer/Counter and Compare/Capture Modules

TABLE 3-2: PCA Timer/Counter Inputs

PCA Timer/Counter Mode	Clock Increments	
	12 MHz	16 MHz
Mode 0: $f_{OSC}/12$	1 µsec	0.75 µsec
Mode 1:	330 nsec	250 nsec
Mode 2: Timer 0 Overflows[1]		
Timer 0 programmed in:		
8-bit mode	256 µsec	192 µsec
16-bit mode	65 msec	49 µsec
8-bit auto-reload	1 to 255 µsec	0.75 to 191 µsec
Mode 3: External Input MAX	0.66 µsec	0.50 µsec

T3-2.0 2052

1. In Mode 2, the overflow interrupt for Timer 0 does not need to be enabled.

PCA Timer/Counter Mode Register[1] (CMOD)

Location	7	6	5	4	3	2	1	0	Reset Value
D9H	CIDL	WDTE	-	-	-	CPS1	CPS0	ECF	00xxx000b

1. Not bit addressable

Symbol	Function
CIDL	Counter Idle Control: 0: Programs the PCA Counter to continue functioning during idle mode 1: Programs the PCA Counter to be gated off during idle
WDTE	Watchdog Timer Enable: 0: Disables Watchdog Timer function on PCA module 4 1: Enables Watchdog Timer function on PCA module 4
-	Not implemented, reserved for future use. **Note:** User should not write ‘1’s to reserved bits. The value read from a reserved bit is indeterminate.
CPS1	PCA Count Pulse Select bit 1
CPS0	PCA Count Pulse Select bit 2

CPS1	CPS0	Selected PCA Input[1]	
0	0	0	Internal clock, $f_{OSC}/6$ in 6 clock mode ($f_{OSC}/12$ in 12 clock mode)
0	1	1	Internal clock, $f_{OSC}/2$ in 6 clock mode ($f_{OSC}/4$ in 12 clock mode)
1	0	2	Timer 0 overflow
1	1	3	External clock at ECI/P1.2 pin
			(max. rate = $f_{OSC}/4$ in 6 clock mode, $f_{OSC}/8$ in 12 clock mode)

1. f_{OSC} = oscillator frequency

Symbol	Function
ECF	PCA Enable Counter Overflow interrupt: 0: Disables the CF bit in CCON 1: Enables CF bit in CCON to generate an interrupt

Table 3-3 lists the CMOD initialization values associated with selecting different PCA count pulse sources. This list assumes that PCA will be left running during idle mode.

The CCON register shown below is associated with all PCA timer functions. It contains the run control bit (CR) and overflow flags for the PCA timer (CF) and all modules (CCFx). To run the PCA the CR bit (CCON.6) must be set by software. Clearing the bit will turn off PCA. When the PCA counter overflows, the CF (CCON.7) will be set, and an interrupt will be generated if the ECF bit in the CMOD register is set. The CF bit can only be cleared by software. Each module has its own timer overflow flag or capture flag (CCF0 for module 0, CCF4 for module 4, etc.). They are set when either a match or capture occurs. These flags can only be cleared by software. CF and CCFn bits can also be set by software.

TABLE 3-3: CMOD Values

PCA Count Pulse Selected	CMOD Value	
	Without Interrupt Enabled	**With Interrupt Enabled**
Internal clock, $f_{OSC}/12$	00H	01H
Internal clock, $f_{OSC}/4$	02H	03H
Timer 0 overflow	04H	05H
External clock at P1.2	06H	07H

T3-3.0 2052

PCA Timer/Counter Control Register[1] (CCON)

Location	7	6	5	4	3	2	1	0	Reset Value
D8H	CF	CR	-	CCF4	CCF3	CCF2	CCF1	CCF0	00x00000b

1. Bit addressable

Symbol	Function
CF	PCA Counter Overflow Flag Set by hardware when the counter rolls over. CF flags an interrupt if bit ECF in CMOD is set. CF may be set by either hardware or software, but can only cleared by software.
CR	PCA Counter Run control bit Set by software to turn the PCA counter on. Must be cleared by software to turn the PCA counter off.
-	Not implemented, reserved for future use. **Note:** User should not write '1's to reserved bits. The value read from a reserved bit is indeterminate.
CCF4	PCA Module 4 interrupt flag. Set by hardware when a match or capture occurs. Must be cleared by software.
CCF3	PCA Module 3 interrupt flag. Set by hardware when a match or capture occurs. Must be cleared by software.
CCF2	PCA Module 2 interrupt flag. Set by hardware when a match or capture occurs. Must be cleared by software.
CCF1	PCA Module 1 interrupt flag. Set by hardware when a match or capture occurs. Must be cleared by software.
CCF0	PCA Module 0 interrupt flag. Set by hardware when a match or capture occurs. Must be cleared by software.

4.0 COMPARE/CAPTURE MODULES

Each PCA module has a mode SFR with it. These registers are: CCAPM0 for module 0, CCAPM1 for module 1, etc. Each register contains 7 bits that are used to control the mode in which each module will operate. See Table 4-1.

The ECCF bit (CCAPMn0 where n = 0, 1, 2, 3, or 4 depending on module) will enable the CCF flag in the CCON SFR to generate an interrupt when a match or compare occurs. PWM (CCAPMn1) enables the pulse width modulation mode. The MATn (CCAPMn3) bit when set, will cause the CCFn bit in the CCON register to be set when there is a match between the PCA counter and the module's capture/compare registers. Additionally, the TOG bit (CCAPMn2) when set, causes the CEXn output pin associated with that module to toggle when there is a match between the PCA counter and the module's capture/compare registers.

Bits CAPN (CCAPMn4) and CAPP (CCAPMn5) determine whether the capture input will be active on a positive edge or negative edge. The CAPN bit enables a capture at the negative edge, and the CAPP bit enables a capture at the positive edge. When both bits are set, both edges will be enabled and a capture will occur for either transition. The last bit in the register ECOM (CCAPMn6) when set, enables the comparator function. Tables 4-2 and 4-3 show the CCAPMn settings for the various PCA functions.

There are two additional registers associated with each of the PCA modules: CCAPnH and CCAPnL. They are registers that hold the 16-bit count value when a capture occurs or a comparison occurs. When a module is used in PWM mode, these registers are used to control the duty cycle of the output. See Table 4-4.

TABLE 4-1: PCA Compare/Capture Module Mode Register[1] (CCAPMn)

Location	7	6	5	4	3	2	1	0	Reset Value
DAH	-	ECOM0	CAPP0	CAPN0	MAT0	TOG0	PWM0	ECCF0	00xxx000b
DBH	-	ECOM1	CAPP1	CAPN1	MAT1	TOG1	PWM1	ECCF1	00xxx000b
DCH	-	ECOM2	CAPP2	CAPN2	MAT2	TOG2	PWM2	ECCF2	00xxx000b
DDH	-	ECOM3	CAPP3	CAPN3	MAT3	TOG3	PWM3	ECCF3	00xxx000b
DEH	-	ECOM4	CAPP4	CAPN4	MAT4	TOG4	PWM4	ECCF4	00xxx000b

T4-1.0 2052

1. Not bit addressable

TABLE 4-2: PCA Module Modes Without Interrupt enabled

Reserved[1]	ECOMy[2]	CAPPy[2]	CAPNy[2]	MATy[2]	TOGy[2]	PWMy[2]	ECCFy[2]	Module Code
-	0	0	0	0	0	0	0	No Operation
-	0	1	0	0	0	0	0	16-bit capture on positive-edge trigger at CEX[4:0]
-	0	0	1	0	0	0	0	16-bit capture on negative-edge trigger at CEX[4:0]
-	0	1	1	0	0	0	0	16-bit capture on positive/negative-edge trigger at CEX[4:0]
-	1	0	0	1	0	0	0	Compare: software timer
-	1	0	0	1	1	0	0	Compare: high-speed output
-	1	0	0	0	0	1	0	Compare: 8-bit PWM
-	1	0	0	1	0 or 1[3]	0	0	Compare: PCA WDT (CCAPM4 only)[4]

T4-2.0 2052

1. User should not write '1's to reserved bits. The value read from a reserved bit is indeterminate.
2. y = 0, 1, 2, 3, 4
3. A 0 disables toggle function. A 1 enables toggle function on CEX[4:0] pin.
4. For PCA WDT mode, also set the WDTE bit in the CMOD register to enable the reset output signal.

TABLE 4-3: PCA Module Modes With Interrupt enabled

Reserved[1]	ECOMy[2]	CAPPy[2]	CAPNy[2]	MATy[2]	TOGy[2]	PWMy[2]	ECCFy[2]	Module Code
-	0	1	0	0	0	0	1	16-bit capture on positive-edge trigger at CEX[4:0]
-	0	0	1	0	0	0	1	16-bit capture on negative-edge trigger at CEX[4:0]
-	0	1	1	0	0	0	1	16-bit capture on positive/negative-edge trigger at CEX[4:0]
-	1	0	0	1	0	0	1	Compare: software timer
-	1	0	0	1	1	0	1	Compare: high-speed output
-	1	0	0	0	0	1	X[3]	Compare: 8-bit PWM
-	1	0	0	1	0 or 1[4]	0	X[5]	Compare: PCA WDT (CCAPM4 only)[6]

T4-3.0 2052

1. User should not write ‘1’s to reserved bits. The value read from a reserved bit is indeterminate.
2. y = 0, 1, 2, 3, 4
3. No PCA interrupt is needed to generate the PWM.
4. A 0 disables toggle function. A 1 enables toggle function on CEX[4:0] pin.
5. Enabling an interrupt for the Watchdog Timer would defeat the purpose of the Watchdog Timer.
6. For PCA WDT mode, also set the WDTE bit in the CMOD register to enable the reset output signal.

TABLE 4-4: PCA High and Low Register Compare/Capture Modules

Symbol	Description	Direct Address	Bit Address, Symbol, or Alternative Port Function MSB … LSB	RESET Value
CCAP0H	PCA Module 0 Compare/Capture Registers	FAH	CCAP0H[7:0]	00H
CCAP0L		EAH	CCAP0L[7:0]	00H
CCAP1H	PCA Module 1 Compare/Capture Registers	FBH	CCAP1H[7:0]	00H
CCAP1L		EBH	CCAP1L[7:0]	00H
CCAP2H	PCA Module 2 Compare/Capture Registers	FCH	CCAP2H[7:0]	00H
CCAP2L		ECH	CCAP2L[7:0]	00H
CCAP3H	PCA Module 3 Compare/Capture Registers	FDH	CCAP3H[7:0]	00H
CCAP3L		EDH	CCAP3L[7:0]	00H
CCAP4H	PCA Module 4 Compare/Capture Registers	FEH	CCAP4H[7:0]	00H
CCAP4L		EEH	CCAP4L[7:0]	00H

T4-4.0 2052

5.0 PCA OPERATIONAL MODES

5.1 Capture Mode

Capture mode is used to capture the PCA timer/counter value into a module's capture registers (CCAPnH and CCAPnL). The capture will occur on a positive edge, negative edge, or both on the corresponding module's pin. To use one of the PCA modules in the capture mode, either one or both the CCAPM bits CAPN and CAPP for that module must be set. When a valid transition occurs on the CEX pin corresponding to the module used, the PCA hardware loads the 16-bit value of the PCA counter register (CH and CL) into the module's capture registers (CCAPnL and CCAPnH). The CCFn bit for the module in the CCON SFR is set by hardware. If the ECCFn bit in the CCAPMn SFR are set, then an interrupt will be generated. In the interrupt service routine, the 16-bit capture value must be saved in RAM before the next event capture occurs. If a subsequent capture occurred, the original capture values would be lost. After the event flag (CCFn) has been set by hardware, the user must clear the flag in software. (See Figure 5-1)

A common use for the PCA capture mode is to measure the properties of a waveform. Properties such as the period, pulse width or the phase difference of two waveforms are measured by determining the difference in capture values between two edges of the waveform. The hardware support of the PCA capture mode allows accurate measurement of these properties with low software overhead. The following sample code shows how the PCA capture mode can be used to measure the pulse width of a waveform on the CEX0 pin.

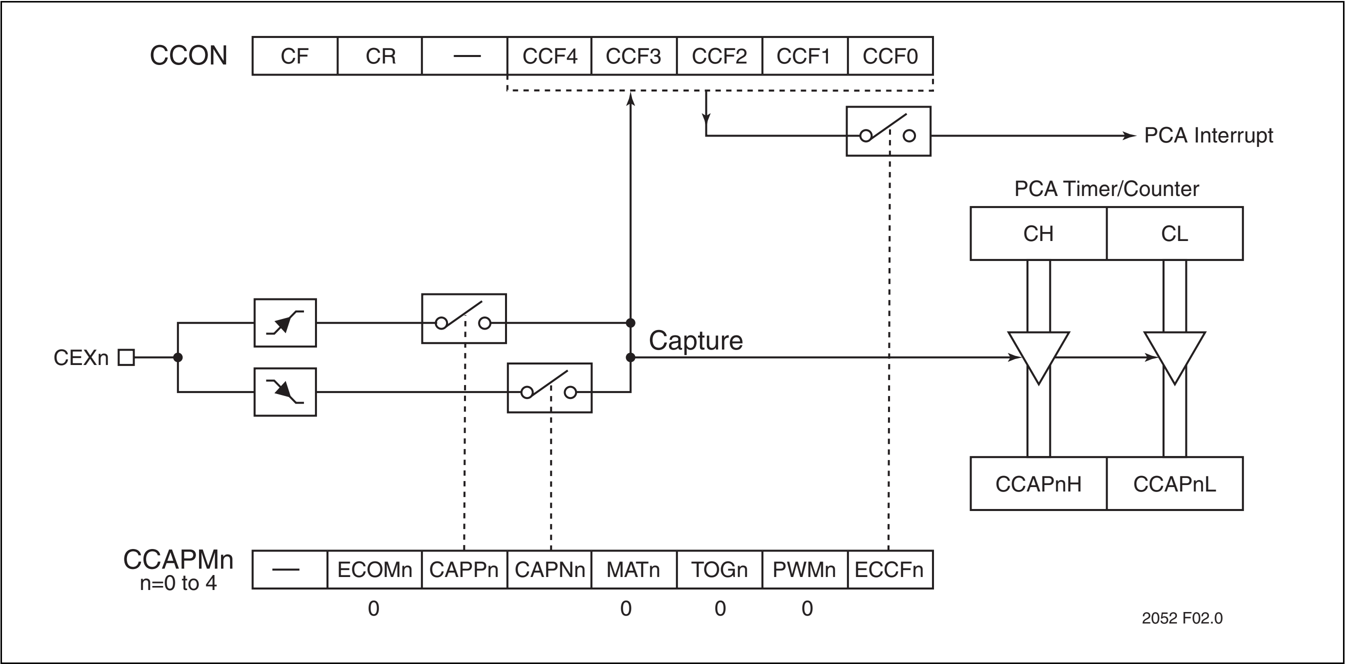

FIGURE 5-1: PCA Capture Mode

5.1.1 Sample Code For PCA Capture Mode

```
#include <stdio.h>
#include “SST89x5xxRDx.h”

// This program is designed to use the PCA module to calculate the width of a
// detected pulse. The pulse must begin with a rising edge and end with a falling
// edge on the CEX0 pin.

// The HEADER FILE “SST89X5xxRDx.h” is an SST proprietary header file that defines SST’s SFRs.
// This file can be found on the SST website, or the BSL Demo Kit.

bit flag = 0;                                   // Rising or falling edge flag.
unsigned int pulse_width = 0x00;                // Final pulse width calculation is stored here.
unsigned int capture1 = 0x00;                   // Rising edge captured here.
unsigned int capture2 = 0x00;                   // Falling edge captured here.

void PCA_ISR() interrupt 6 using 1// PCA Interrupt Service routine
{
        CCF0 = 0;                               // Clear the PCA flag
        if (flag == 0)
        {
                capture1 = CCAP0L | (CCAP0H << 8);  // If positive edge, store in
                CCAPM0 = 0x11;                  // capture1.  Setup module
                flag = 1;                       // 0 for capturing falling edge
        }
        else
        {
                capture2 = CCAP0L | (CCAP0H << 8);  // Capture falling edge
                pulse_width = capture2 - capture1;  // Final calculation.
                CCAPM0 = 0x21;                  // Setup module 0 for
                                                // capturing rising edge.
                flag = 0;                       // Reset flag
        }
}

void main()
{
        CMOD = 0x03;                            //Setting up the PCA counter
        CH = 0x00;
        CL = 0x00;
        CCAPM0 = 0x21;                          // Setup module zero in capture mode
        Flag = 0;

        IE = 0xC0;                              // Enable PCA interrupt
        CR = 1;                                 // Start PCA counter

        while (1)                               // Software trap
        {}
}
```

5.2 16-bit Software Timer Mode

The 16-bit software timer mode is used to trigger interrupt routines, which must occur at periodic intervals. It is setup by setting both the ECOM and MAT bits in the module's CCAPMn register. The PCA timer will be compared to the module's capture registers (CCAPnL and CCAPnH) and when a match occurs, an interrupt will occur, if the ECCFn (CCAPMn SFR) bit for the module is set.

If necessary, a new 16-bit compare value can be loaded into CCAPnH and CCAPnL during the interrupt routine. The user should be aware that the hardware temporarily disables the comparator function while these registers are being updated so that an invalid match will not occur. Thus, it is recommended that the user write to the low byte first (CCAPnL) to disable the comparator, then write to the high byte (CCAPnH) to re-enable it. If any updates to the registers are done, the user may want to hold off any interrupts from occurring by clearing the ECCFn bit or the EA bit. (See Figure 5-2.)

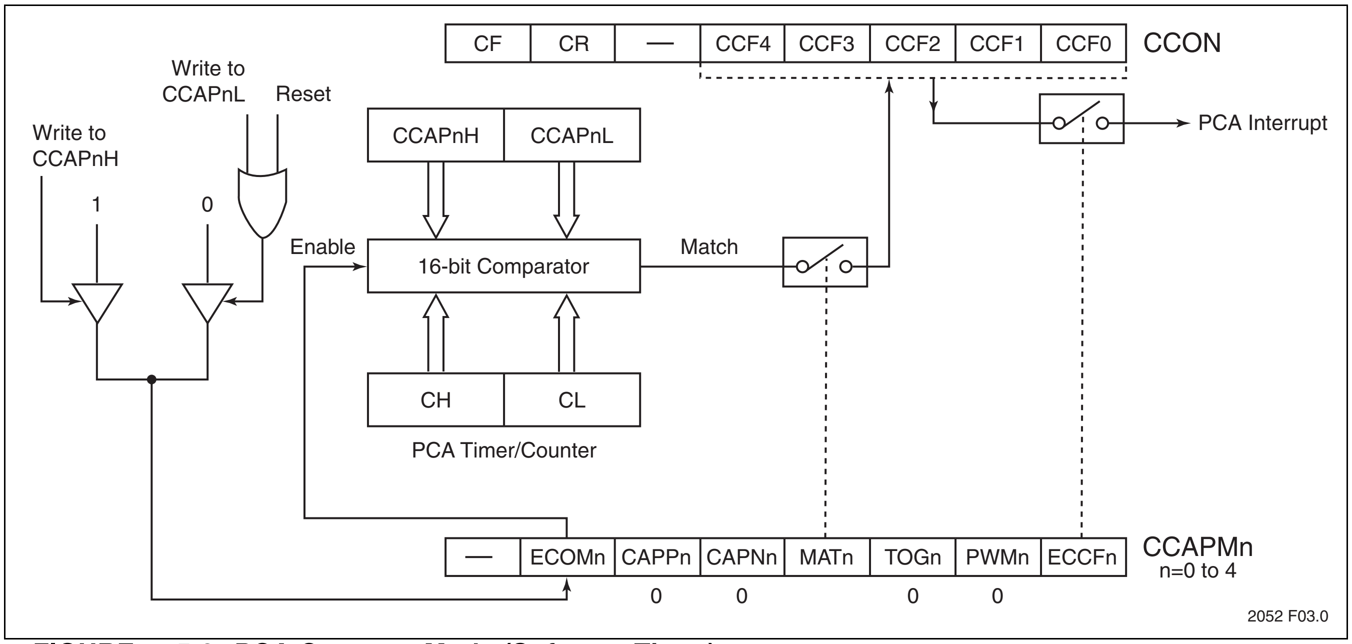

FIGURE 5-2: PCA Compare Mode (Software Timer)

5.2.1 Sample Code For 16-bit Software Timer

```
#include <stdio.h>
#include "SST89x5xxRDx.h"

// This program uses the PCA module 0 in 16-bit Timer mode.
// This program will trigger an interrupt event every 20000 counts.
// Assuming a 12MHz clock, an event should trigger every 20ms.

// The HEADER FILE "SST89X5xxRDx.h" is an SST proprietary header file that defines SST's SFRs.
// This file can be found on the SST website, or the BSL Demo Kit.

void PCA_ISR() interrupt 6 using 1
{
                unsigned int temp;

                IE = IE & 0xBF;                     // Stop Interrupts
                CCF0 = 0;                           // Clear Int flag
                temp = CCAP0L | (CCAP0H << 8);      // The following four lines
                temp += 0x4E20;                     // of code increase the
                CCAP0L = temp;                      // compare value in CCAP0
                CCAP0H = temp >> 8;                 // by 20000, effectively
                                                    // refreshing the timer.
                IE = IE | 0x40;                     // Start interrupts
}

void main()
{
                CMOD = 0x01;                        // Setup PCA timer mode.
                CH = 0x00;                          // Init values
                CL = 0x00;
                CCAP0L = 0x20;                      // Set compare limit
                CCAP0H = 0x4E;
                CCAPM0 = 0x49;                      // Set Modules zero for 16bit Timer mode.


                IE = 0xC0;                          // Enable Interrupts
                CR = 1;                             // Start PCA timer

                while(1)
                {}
}
```

5.3 High Speed Output Mode

In this mode, the CEX output pin (on port 1) associated with the PCA module will toggle every time there is a match between the PCA counter (CH and CL) and the capture registers (CCAPnH and CCAPnL). To activate this mode, the user must set TOG, MAT, and ECOM bits in the module's CCAPMn SFR. High Speed Output mode is much more accurate than software pin toggling since the toggle occurs before branching to an interrupt. In this case, interrupt latency will not affect the accuracy of the output. When using High Speed Output mode, using an interrupt is optional. Only if the user wishes to change the time for the next toggle is it necessary to update the compare registers. Otherwise, the next toggle will occur when the PCA timer rolls over and matches the last compare value. (See Figure 5-3.)

The following code shows how the PCA high speed output mode can be used to generate a continuous square wave on the module 0 output CEX0. After the PCA counter is started, a match eventually occurs between the PCA counter and the module capture registers. At this point, the hardware will toggle the port pin. Then in the PCA interrupt subroutine, the CPU adds a constant value to the capture register to indicate when the next match should occur. Note however that because the capture register must be reloaded by software, the effective maximum toggle rate of the signal is limited by the machine cycle count of the interrupt subroutine. The PCA pulse width modulation mode discussed later in this document shows a less software intensive method of producing such waveforms.

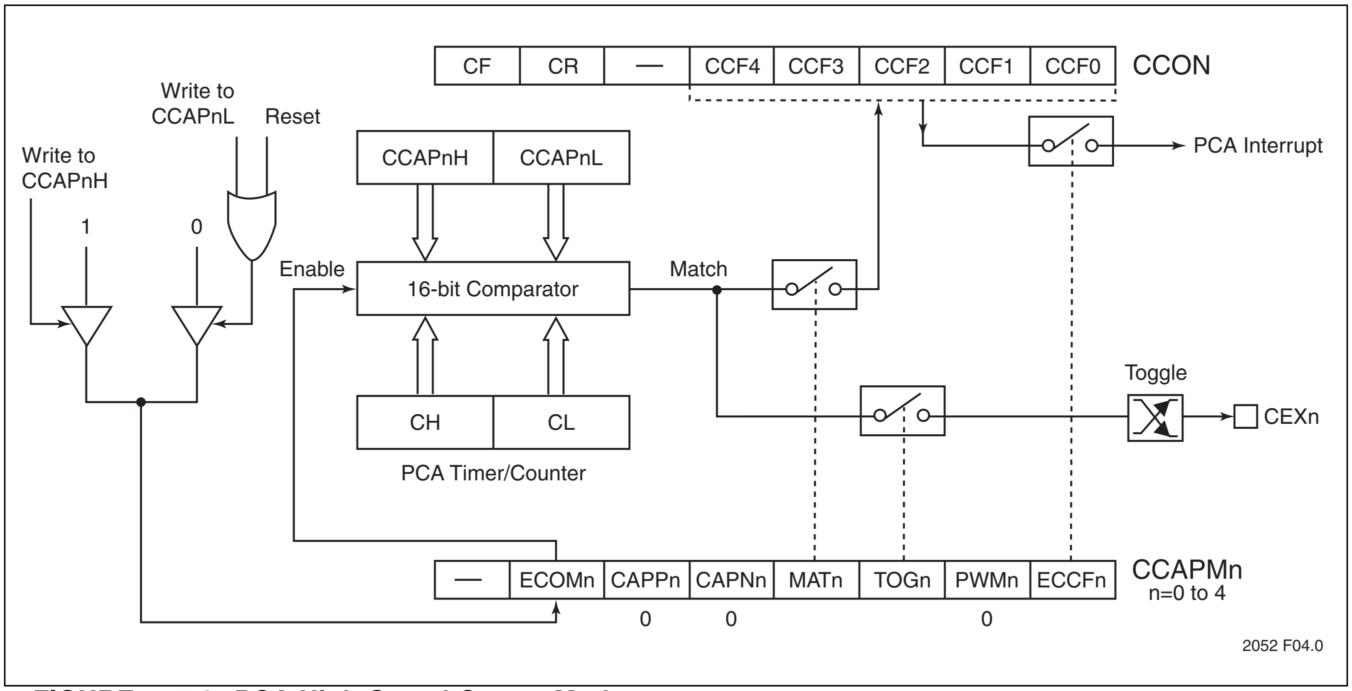

FIGURE 5-3: PCA High-Speed Output Mode

5.3.1 Sample Code For High Speed Output

```
#include <stdio.h>
#include “SST89x5xxRDx.h”

// The High Speed Output mode is used to toggle the CEX pin when a match occurs
// between the PCA timer and the pre-loaded value in the compare registers.

// The HEADER FILE “SST89X5xxRDx.h” is an SST proprietary header file that defines SST’s SFRs.
// This file can be found on the SST website, or the BSL Demo Kit.

// Maximum output with HSO mode without interrupts = 30.5 Hz signal.
// Frequency = 16 MHz
// PCA clock input = 1/4 x fosc -> 250 nsec
// Output with interrupts for increasing 3000h count value = 4 MHz/6000h = 162.7 Hz signal

void PCA_ISR() interrupt 6 using 1
{
                CCF0 = 0;
                CCAP0H += 0x30;                 // Increase compare values by 3000h counts
}
void main()
{
                CMOD = 0x02;                    // Setup PCA Timer
                CL = 0x00;
                CH = 0x00;

                CCAP0L = 0xFF;                  // Set Event trigger values
                CCAP0H = 0xFF;
                CCAPM0 = 0x4D;                  // Set PCA module 0 for HSO mode

                IE = 0xC0;
                CR = 1;                         // Start PCA timer.

                while(1)
                {}
}
```

5.4 Pulse Width Modulator

The Pulse Width Modulator (PWM) mode is used to generate a continuous square-wave with an arbitrary duty cycle. It generates 8-bit PWMs by comparing the low byte of the PCA timer (CL) with the low byte of the compare register (CCAPnL). When CL < CCAPnL the output is low. When CL ?>= CCAPnL the output is high. To activate this mode, the user must set the PWM and ECOM bits in the module's CCAPMn SFR. (See Figure 5-4 and Table 5-1) In PWM mode, the frequency of the output depends on the source for the PCA timer. Since there is only one set of CH and CL registers, all modules share the PCA timer and frequency. The frequency is fixed to 256 counts of the PCA timer. Duty cycle of the output is controlled by the value loaded into the high byte (CCAPnH). Since writes to the CCAPnH register are asynchronous, a new value written to the high byte will not be shifted into CCAPnL for comparison until the next period of the output (when CL rolls over from 255 to 00).

To calculate values for CCAPnH for any duty cycle, use the following equation:

$$\text{CCAPnH} = 256(1 - \text{Duty Cycle})$$

where CCAPnH is an 8-bit integer and duty cycle is a fraction.

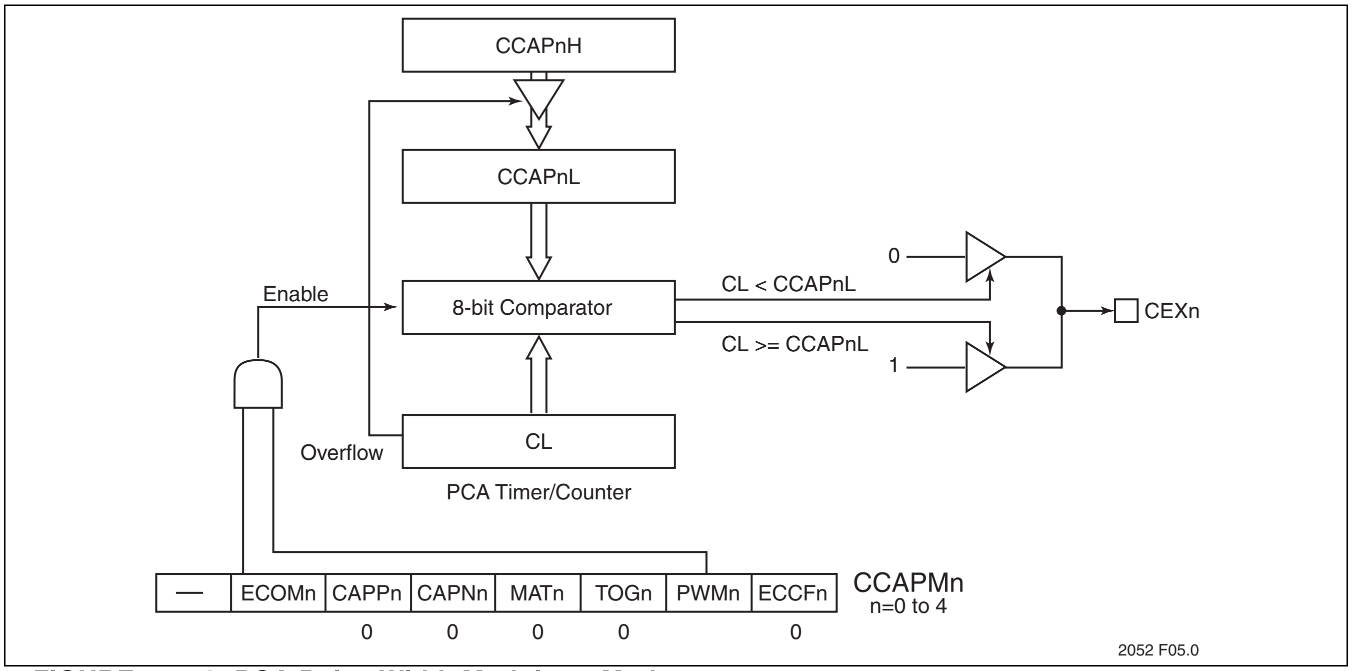

FIGURE 5-4: PCA Pulse Width Modulator Mode

TABLE 5-1: Pulse Width Modulator Frequencies

	PWM Frequency[1]	
PCA Timer Mode	**12 MHz**	**16 MHz**
1/12 Oscillator Frequency	3.9 KHz	5.2 KHz
1/4 Oscillator Frequency	11.8 KHz	15.6 KHz
Timer 0 Overflow:		
8-bit	15.5 Hz	20.3 Hz
16-bit	0.06 Hz	0.08 Hz
8-bit Auto-Reload	3.9 KHz to 15.3 Hz	5.2 KHz to 20.3 Hz
External Input (Max)	5.9 KHz	7.8 KHz

T5-1.0 2052

1. PWM Frequency = (F_{OSC} x F_{OSC} Divider) / 256

5.4.1 Sample Code For PWM

```
#include <stdio.h>
#include “SST89x5xxRDx.h”

// This program generates a pulse by comparing the CL register with the
// value stored in CCAP0L. If CL >= CCAP0L then the output is HIGH.
// If CL < CCAP0L then the output is LOW.
// The duty cycle is determined by the value stored in CCAP0H.

// The HEADER FILE “SST89X5xxRDx.h” is an SST proprietary header file that defines SST’s SFRs.
// This file can be found on the SST website, or the BSL Demo Kit.

void main()
{
                CMOD = 0x02;                    // Setup PCA timer
                CL = 0x00;
                CH = 0x00;

                CCAP0L = 0x40;                  // Set the initial value same as CCAP0H
                CCAP0H = 0x40;                  // 75% Duty Cycle
                CCAPM0 = 0x42;                  // Setup PCA module 0 in PWM mode.

                CR = 1;                         // Start PCA Timer.

                while (1)
                {}

}
```

5.5 Watchdog Timer

The Watchdog Timer mode is used to improve reliability in the system without increasing chip count (See Figure 5-5). Watchdog Timers are useful for systems that are susceptible to noise and/or power glitches. It can also be used to prevent a hardware or software deadlock. During the execution of the user's code, if there is a deadlock, the Watchdog Timer will time out and an internal reset or WDT interrupt will occur. Only module 4 can be programmed as a Watchdog Timer (but still can be programmed to other modes if the Watchdog Timer is not used).

To use the Watchdog Timer, the user pre-loads a 16-bit value in the compare register. Just like the other compare modes, this 16-bit value is compared to the PCA timer value. If a match is allowed to occur, an internal reset will be generated. In order to hold off the reset, the user has three options: 1. periodically change the compare value so it will never match the PCA timer, 2. periodically change the PCA timer value so it will never match the compare values, or 3. disable the watchdog timer by clearing the WDTE bit before a match occurs and then re-enable it. The first two options are more reliable because the Watchdog Timer is never disabled as in option #3. If the program counter ever goes astray, a match will eventually occur and cause an internal reset. The second option is also not recommended if other PCA modules are being used. Remember that the PCA timer is the common time base for all modules; changing the time base for other modules would not be a good idea. Thus, in most application the first solution is the best option.

The code below demonstrates how to initialize the Watchdog Timer. Module 4 can be configured in either compare mode, and the WDTE bit in CMOD must also be set. The user's software then must periodically change (CCAP4H, CCAP4L) to keep a match from occurring with the PCA timer (CH, CL).

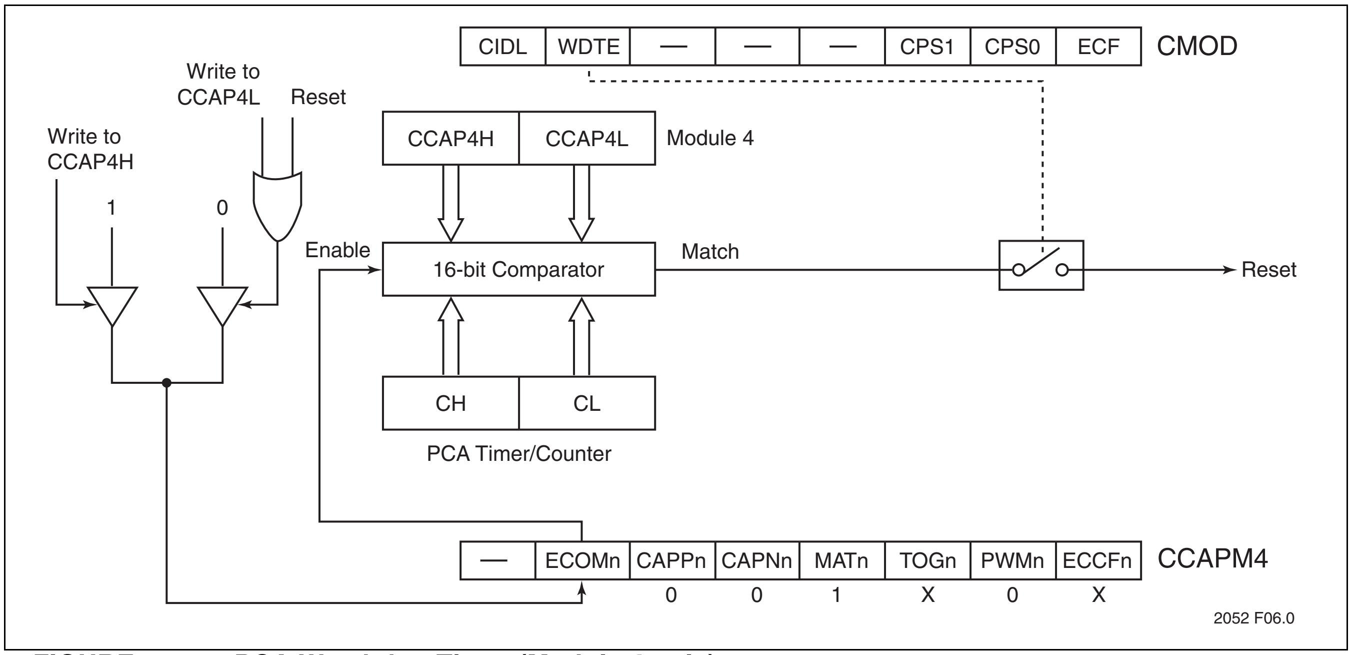

FIGURE 5-5: PCA Watchdog Timer (Module 4 only)

5.5.1 Sample Code For Watchdog Timer

```
#include <stdio.h>
#include “SST89x5xxRDx.h”

// This program uses PCA module 4 for the Watchdog timer. Note that only module 4
// can be used for the Watchdog timer. There are two ways of refreshing the timer to
// avoid a Watchdog reset. The first way is to reset the CH and CL registers. This is
// not recommended if you are using other PCA modules since disturbing the CH and
// CL registers would disturb the functionality of those other modules. The second choice is
// to change the values in CCAP4H/L. This is the best way to go since it accomplishes
// the same task of preventing Watchdog reset without the possibility of affecting the
// other PCA modules.

// The HEADER FILE “SST89X5xxRDx.h” is an SST proprietary header file that defines SST’s SFRs.
// This file can be found on the SST website, or the BSL Demo Kit.

void refresh_WDT();
void delay();

void main()
{
                CCAP4L = 0xFF;                  // Setup PCA module 4 for Watchdog Timer
                CCAP4H = 0xFF;
                CCAPM4 = 0x4C;

                CMOD = CMOD | 0x40;
                while (1)
                {
                        refresh_WDT();          // This function refreshes the WDT and should be
                                                // used periodically.
                        delay();
                }
}

void delay()
{
                unsigned int I;

                for(i = 0; i < 1024; i++);
}

/**********WATCH DOG REFRESH*********/
void refresh_WDT()
{
                EA = 0;

                CCAP4L = 0;
                CCAP4H = CH;

                EA = 1;
}
```

6.0 COMPREHENSIVE PCA PROGRAM

The following PCA code will run Module 0 in Capture mode, Module 1 in 16 bit timer mode, Module 2 in HSO mode, Module 3 in PWM mode, and Module 4 in WDT mode simultaneously.

```
#include <stdio.h>
#include <stdlib.h>
#include “SST89x5xxRDx.H”

bit flag = 0;                                   // Rising or falling edge flag.
unsigned int pulse_width = 0x00;                // Final pulse width calculation is stored here.
unsigned int capture1 = 0x00;                   // Rising edge captured here.
unsigned int capture2 = 0x00;                   // Falling edge captured here.
unsigned int HSO_Frequency = 0xFFFF;            // HSO Frequency counter

void PCA_ISR() interrupt 6 using 1// PCA Interrupt Service routine
{
        unsigned int temp = 0;

        if (CCF0 == 1)
        {
                IE = IE & 0xBF;                 // Stop PCA interrupt
                CCF0 = 0;                       // Clear the PCA flag
                if (flag == 0)
                {
                        capture1 = CCAP0L | (CCAP0H << 8);      // If positive edge, store in
                        CCAPM0 = 0x11;                          // capture1.  Setup module
                        flag = 1;                               // 0 for capturing falling edge
                }
                else
                {
                        capture2 = CCAP0L | (CCAP0H << 8);      // Capture falling edge
                        pulse_width = capture2 - capture1;      // Final calculation.
                        CCAPM0 = 0x21;                          // Setup module 0 for
                                                                // capturing rising edge.
                        flag = 0;
                }
                IE = IE | 0x40;                 // Start PCA interrupt
        }
        else if (CCF1 == 1)
        {
                IE = IE & 0xBF;                 // Stop PCA interrupt
                CCF1 = 0;                       // Clear Int flag
                temp = CCAP1L | (CCAP1H << 8);  // The following four lines
                temp += 0x4E20;                 // of code increase the
                CCAP1L = temp;                  // compare value in CCAP0
                CCAP1H = temp >> 8;             // by 20000, effectively
                                                // refreshing the timer.
                IE = IE | 0x40;                 // Start PCA interrupt
        }
        else if (CCF2 == 1)
        {
                IE = IE & 0xBF;                 // Stop PCA interrupt
                CCF2 = 0;
                CCAP2H += 0x30;                 // Increase compare values by 3000h counts
                IE = IE | 0x40;                 // Start PCA interrupt
        }
}
```

```
void delay()
{
        unsigned int i;

        for(i = 0; i < 1024; i++);
}

void refresh_WDT()
{
        IE = IE & 0xBF;                         // Stop PCA interrupt
        CCAP4L = 0;
        CCAP4H = CH;
        IE = IE | 0x40;                         // Start PCA interrupt
}

void main()
{
        CMOD = 0x43;                            //Setting up the PCA counter
        CH = 0x00;
        CL = 0x00;

        CCAPM0 = 0x21;                          // Setup Module 0 in capture mode

        CCAP1L = 0x20;                          // Set Module 1 compare limit
        CCAP1H = 0x4E;
        CCAPM1 = 0x49;                          // Set Module 1 for 16bit Timer mode.

        CCAP2L = 0xFF;                          // Set Event trigger values
        CCAP2H = 0xFF;
        CCAPM2 = 0x4D;                          // Set PCA module 2 for HSO mode

        CCAP3L = 0x40;                          // Set the initial value same as CCAP3H
        CCAP3H = 0x40;                          // 75% Duty Cycle
        CCAPM3 = 0x42;                          // Setup PCA module 3 in PWM mode.

        CCAP4L = 0xFF;
        CCAP4H = 0xFF;
        CCAPM4 = 0x4C;                          // Setup PCA module 4 for Watchdog Timer

        IE = 0xC0;                              // Enable PCA interrupt
        CR = 1;                                 // Start PCA counter

        while (1)                               // Software trap
        {
                refresh_WDT();                  // This function refreshes the WDT and should be
                                                // used periodically.
                delay();

        }
}
```