Chapter 21LC

Lab C3 SiLabs 3: Timers; PWM; Comparator

Contents

21LC.1 PORT 0 Pin Use in This Lab .................................................. 2
21LC.2 Timer: Blink an LED, Again .................................................. 2
   21LC.2.1 A Task for You: Timer Register Initializations ........................ 2
21LC.3 Using Configuration Wizard to Set Up the Timer ......................... 5
   21LC.3.1 The Wizard Produces Code ............................................... 6
   21LC.3.2 The Bit-flip Timer Program .............................................. 8
   21LC.3.3 Timers Do Various Tricks ............................................... 9
21LC.4 Pulse Width Modulation (PWM) (Dim an LED) ................................ 10
   21LC.4.1 PWM, Analog and Digital Versions ..................................... 10
   21LC.4.2 PWM Code: Slow Ramp of LED Brightness ............................. 11
   21LC.4.3 Improvements: 1: Change Color ....................................... 13
   21LC.5 A Task for You: Improvements: 2: Ramp Up, Ramp Down .............. 14
21LC.6 Comparator: an Oscillator as A Start on Something More Interesting .................................................. 14
   21LC.6.1 Oscillator Hardware .................................................. 14
   21LC.6.2 Comparator Configuration .............................................. 15
   21LC.6.3 Code: RC Oscillator .................................................. 16
21LC.7 Apply the Oscillator: “Suntan Alarm” ......................................... 17
   21LC.7.1 Code for Suntan Alarm .................................................. 18
   21LC.7.2 A Task for You: Improvements: 3: Improve the Monitor by Adding the DONE signal ............................. 20

REV 1; March 23, 2015.

1Revisions: init caps for headings, no space after §(3/15); add headerfile and index (7/14); change photoQ and buzzer, to those available (4/13); from fall 11 version: add point that Wizard2 is a separate utility, not accessible from within IDE; add images of photo-R and photo-transistor and buzzer (4/11); add code to integrate suntan exposure, and pose task of writing ‘Done’ subroutine (jan11).
21LC.1 PORT0 Pin Use in This Lab

Again we use one PORT0 pin to drive an LED, and we will add other peripherals, at four other pins:

Figure 1: PORT0 Pin Use in this Lab

21LC.2 Timer: Blink an LED, Again

We confessed, back in the first of these controller labs, that it was a bit doopey to tie up a controller in a loop of a few hundred thousand cycles, in order to slow the blinking of an LED to a rate that a human could notice. Now we invite you to try the controller’s better way to do such a task, using its built-in hardware timer.

The 8051, like almost any controller, includes a collection of such timers. We will try, first, one whose design is pretty standard (though it is an enhancement to the very first version of the 8051). This is called “Timer 2.” The ‘410 offers three or four other timers (depending on whether you consider a “Real time clock” to be a timer). As usual, we will have to initialize a set of registers. But once that is done, the program will occupy almost none of the processor’s attention.

The usual way to use a timer is to treat it like an oven timer: set a delay time, and ask the timer to let us know when time is up. Usually, “time’s up” will trigger an interrupt, since the point of using the timer is to free the processor for other operations more interesting than keeping time or measuring delay time. The program in § 21LC.2.1 behaves this way. It invokes an interrupt service routine to toggle an LED each time its timer overflows. It then automatically reloads the delay value and the cycle begins again.

It is also possible to let a timer drive an output pin directly: CA Counter/Timer can do that in “High-speed output mode.” We have not done that, here. Our ISR is so simple that it is virtually equivalent to the PCA direct-drive. Our ISE simply toggles the LED, but it could, of course, do something (even-) more exciting.

As usual, our program is extremely simple. So, most of our programming labor therefore goes into reading the datasheet to find what register initializations are needed. This time, we will ask you to determine these initializations.

21LC.2.1 A Task for You: Timer Register Initializations

21LC.2.1.1 Initialize TMR2CN...

For this particular program, apart from the USUAL SETUP, almost the entire task consists in initializing just one register, named TMR2CN (“Timer 2 Control”). As usual, the details of even this single register can look quite overwhelming (so much information!) We would like you to set up this register as follows:

- 16-bit timer
- enable the timer/counter
• clock source: system clock / 12
• timer initial value (used only on first pass, which you may want to watch in single-step): FF F0h
• timer reload value (used on all subsequent passes; his is the value re-loaded after overflow): 00 00h
• interrupt on overflow (called “Timer2 Interrupt”)

We invite you to do this task using either of two methods:

**Laborious Way: Proceed bit-by-bit** The heading makes this sound bad. But, in fact, this process probably will take you only two or three minutes. By scrutinizing each bit of the TMR2CN register described below, you can determine what value that register needs.

**Less Laborious Way: Invoke the Configuration Wizard** (Usually, this method will be the better choice. Today, it’s likely to take you longer than the “laborious” method.) Start up the Configuration Wizard on your laptop (the wizard, named “Wizard2” in its present incarnation, is a separate utility—not accessible from within the SiLabs IDE), and follow the process described in a supplementary note call “SiLabs Configuration Wizard.” The Wizard will deliver to you the register value that it prescribes for TMR2CN (along with a couple of others, already present in the listing of § 21LC.3.2 on page 8).

**21LC.2.1.2 Details, for People Setting up TMR2CN Bit-by-bit:**

![SFR Definition 24.8. TMR2CN: Timer 2 Control](image)

Figure 2: Bit-by-bit details of the Timer Control register that we need to initialize
We reprint this in full detail not because you need this much information, but to give you a sense of the initial scariness of a controller’s data “sheet” (a “sheet” that in this case is 270 pages long!). If you find this detail concerning a single register daunting, then you’re like us. We hope, though, that a few minutes’ inspection will reveal that—at least this time—we need know only a little of this overwhelming detail.

Proceeding bit by bit:

d7: TF2H  Yes, we need to pay attention to this bit, which will indicate when the timer has “timed out.” But this is not a bit we need to initialize, except in the sense that once it has been set by an overflow it is up to us to clear it (as the last sentence in the TF2H description says explicitly). This flag bit can serve to generate an interrupt. We use it so in this program.
Do we need to clear this on startup? No. The “Reset Value” shown at top right of fig. 2 on the preceding page shows that this bit, like all bits of this register, starts out low after a Reset.

d6: TF2L  We’ll ignore this bit, which indicates an 8-bit overflow—overflow of the low byte of the 16-bit timer2.

d5: TF2LEN  We’ll leave this bit, which could interrupt on an 8-bit overflow, disasserted. The Reset condition, zero, gets us the result that we want.

d4: TF2CEN  No, we aren’t using the “capture” mode (and won’t here go into what it might mean; if you’re interested, you can see it used in one of the two period-measuring programs shown in Lab micro 6, where you are asked to ‘do something with the standalone controller’). So we leave the bit Low, at its inactive Reset level.

d3: T2SPLIT  We need this bit low, to configure the timer as a single 16-bit timer, rather than as two 8-bit timers. We leave it in its Reset condition, Low.

d2: TR2  Yes, this is important: this bit we must set high, to enable Timer 2.

d1: T2RCLK  This bit does not matter to us, since it has no effect unless we have selected “capture” mode (by setting TF2CEN High), as we have not done.

d1: T2XCLK  Yes, this matters. It chooses a clock source for the timer—but since we are not using the capture mode, we will leave the bit in its Reset state, giving a clock rate that is System Clock / 12.

You may need a pencil and paper to record the bits that need your attention (we needed such a crutch!). When you have done that, you need to decide how to get the bits to the levels that you want.

Specifically, should you use a MOV into the register (such an operation determines the levels on every bit)? Or should you, instead, use bit operations? A preliminary question is whether bit operations are permitted, for this register? The answer to that question is Yes, as the line “Bit Addressable” indicates, near the top right corner of the register description (fig. 2 on the previous page).

Often bit addressing is useful, permitting the change of only bits that interest us, while leaving the others untouched. But here a MOV seems a better idea, for two reasons: first, we know the initial condition of all bits (the Reset value), so we need not fear overwriting some useful information with a MOV; second, some bits need to be cleared (or kept Low), others need to be set (High). A MOV can take care of both Setting and Clearing, at a stroke.

So, you should figure the byte value that you want for TMR2CN (probably expressed in hexadecimal, just to be consistent with the other values that are listed). Then MOV that value into the register.

One More Register Involved—though marginally: CKCON  It’s pretty much true that the single register TMR2CN is enough to initialize the timer; that’s a nice change from the more usual case. One other register could be said to be involved as fig. 2 on the preceding page suggests: CKCON. The T2MH bit (d5) in CKCON
could, if set, determine the Timer2 clock rate. We leave this bit in its Reset state, Low, and doing that allows TMR2CN to set the clock rate. A line in the present program reiterates the default setting, unnecessarily but harmlessly: \texttt{ANL CKCON, #DIV12 ; allow TMR2CN to set timer clock rate.}

We mention this not just to irritate you but to remind you that register initializations usually involve more than a single register, and that (more perplexing) the significance of a bit on one register may depend on a setting in a different register (as T2MH here depends on the level of the T2XCLK bit). All this register-initializing is, as we have said before, the price we pay for the controller’s versatility.

### 21LC.3 Using Configuration Wizard to Set Up the Timer

The Configuration Wizard makes things much easier than if you pored over the datasheet’s timer section.

We opened the Wizard, chose Options/Code Format/Asm, then Timers, then Timer2, and checked the boxes shown in fig. 3:

![Configuration Wizard](image)

**Figure 3:** The Configuration Wizard makes setting up the timer not hard

The timer initialization (which applies only on the first pass) and the values given the timer reload registers are explained in the subsection just below, § 21LC.3.1.1 on the following page.
21LC.3.1 The Wizard Produces Code

The Wizard’s configuration code for the timer bitflip program is shown in fig. 3 on the previous page and is reiterated in fig. 4:

```assembly
; Peripheral specific initialization functions, ; Called from the Init_Device label
Timer_Init:
    mov TRR2CN, #004h
    mov TRR2L, #0Fah
    mov TRR2H, #0FFh
    ret

Port_IO_Init:
    ; PO.0 - Unassigned, Open-Drain, Digital
    ; PO.1 - Unassigned, Open-Drain, Digital
    mov XBRI, #040h
    ret

Interrupts_Init:
    mov IE, #040h
    ret

; Initialization function for device, ; Call Init_Device from your main program
Init_Device:
    lcall Timer_Init
    lcall Port_IO_Init
    lcall Interrupts_Init
    ret
```

Figure 4: The Configuration Wizard produces a code listing—a set of initializing subroutines

In the timer_bitflip program of § 21LC.3.2 on page 8 the values shown for this configuration code were inserted by hand; we used the Wizard served only to check our work. But the Wizard’s code listing can, instead, be simply appended to a program and then invoked as a subroutine. In the PWM program of § 21LC.4.2.2 on page 12 we did that: there, the Wizard’s configuration code—a collection of initialization subroutines—is invoked by the line `acall Init_Device`.

21LC.3.1.1 …a Detail: Initializing the Reload Registers

When the timer overflows, it automatically reloads a 16-bit start value from two registers. Here is a block diagram to suggest how the timer behaves in auto-reload mode:

![Block diagram of Timer 2, showing auto-reload registers](image)

Figure 5: Block diagram of Timer 2, showing auto-reload registers

We determine the LED blink rate by initializing two timer settings: the clock rate (discussed in the preceding section, § 6 on the facing page) and the number of clocks the timer counts between overflows.

---

2Fig. 5 reproduces fig. 24.4 of the datasheet of C8051F410 (used with permission).
The timer reload registers (TMR2RLH and TMR2RLL) determine the number of counts per overflow cycle. Here are the initializations we used—showing the short first-loop, where we started at FF F8h, close to the final count of FF FFh—and the usual loop, where we start at zero for maximum delay:

```assembly
HITIME SET 0FFh ; set start count close to the overflow val
LOTIME SET 0FAh ; ...and this is the low byte
RELOAD SET 0h ; these are reload values, for maximum delay
```

Then these values are used to load both the timer registers (for first pass) and reload registers (for all subsequent iterations):

(A detail: when specifying a hexadecimal value that begins with an alphabetical character, such as the value FF hex, you must provide a leading zero. This zero tells the assembler that this value is, indeed, a number, not a character string. MOV TMR2RLH, #FFh, for example, would generate an assembly error.)

21LC.3.1.2 A Debugging Opportunity: Watch the Overflow Generate an Interrupt...

The first two values, HITIME and LOTIME are FF FAh, extremely close to the top count, FF FFh. We used this peculiar one-time start count so as to facilitate debugging. We wanted you to be able to look at the timer in action: you can, after a reset, single-step the program and watch the program as it approaches its overflow.

Adding Watch As you may have read in the IDE note, you can select a register to watch in the debugger by highlighting the register in the window, then right-clicking to get a menu that offers Add...to Watch. If you do this for TMR2RLH and TMR2RLL you can view the two timer registers, as in fig. 7.

Then a few more single-steps will cause the timer to overflow, from FF FFh to 00 00h, generating an interrupt request that causes the program hop to its ISR:
This first overflow—the quick one that we rigged by initializing the timer to nearly its full count—occurred after just a few increments of the timer. This process generates no useful delay, when run at full-speed. It was useful only to let us watch the timer at work.

**...The Usual Case: Timer Counts Many Cycles Between Overflows** But on all later passes—once the counter has overflowed and been reloaded with the reload values (TMR2RLH and TMR2RLL, which we initialize to zero)—the counter then provides maximum delay, with a PWM-bit start value of 00 00h.

Even this delay is under one second, but long enough to make the blink rate of the LED comfortably visible.

### 21LC.3.2 The Bit-flip Timer Program

; timer2_biflip.a51 use Timer2 to toggle LED, on interrupt by timer overflow

```
$NOSYMBOLS ; keeps listing short
$INCLUDE (C:\MICRO\8051\RAISON\INC\c8051f410.inc)
$INCLUDE (C:\MICRO\8051\RAISON\INC\VECTORS320.INC) ; Tom’s vectors definition file
STACKHOT EQU 80h ; put stack at start of scratch
DIV12 EQU 0DFh ; timer clock mask to div by 12, in CKCON
TIM2INTEN EQU IE.5
TIM2_ENABLE EQU TMR2CN.2
SOFTFLAG EQU 0 ; software flag that ISR uses to talk to Main
BLUE_LED EQU P0.0 ; LED to toggle
GLOBAL_INTEN EQU EA ; an easier mnemonic for the overall interrupt enable

HITIME SET 0FFh ; set start count close to the overflow val
LOTIME SET 0FAh ; ...and this is the low byte
RELOAD SET 0h ; these are reload values, for maximum delay

; port use:
; LED at P0.0

ORG 0
SJMP STARTUP
```
ORG 80h
STARTUP: MOV SP, STACKBOT-1 ; initialize stack pointer
        ACALL USUAL_SETUP
        ACALL TIMER_INITS
        SETB TIM2_ENABLE

STUCK: SJMP STUCK ; await interrupts. All the action is there

;-------------------
; ISR: JUST TOGGLE LED
ORG TIMER2VECTOR
ISR: CPL BLUE_LED
CLR TF2H ; Clear timer-2 overflow flag (does this clear interrupt flag?)
RETI

;---------- INITIALIZATIONS
;---- SUBROUTINES ----
USUAL_SETUP: ANL PCA0MD, #NOT(040h) ; Disable the WDT.
        ORL OSCICN, #04h ; sysclk = 24.5 Mhz / 8
        MOV XBR1, #40h ; Enable Crossbar
        RET

TIMER_INITS:
anl CKCON, #DIV12 ; set timer clock to system/12
        MOV TM2CN, #04h ; enable timer 2, and 16-bit auto-reload
        MOV TM2H, #HITIME ; these are the initial values (for short first pass,
        MOV TM2L, #LOTIME ; when we may be watching in single-step)
        MOV TM2RHL, #RELOAD ; reload values (chosen for maximum delay, after quick first pass)
        MOV TM2RLL, #RELOAD
        SETB TIM2INTEN ; timer-2 interrupt enable
        SETB GLOBAL_INTEN ; Global int enable
        RET

END

The delay available from this timer is only about 0.5 seconds, as configured. One could stretch this delay by letting the ISR include a software count. A single register, there, could extend the delay to a couple of minutes. For really long delays, the so-called “Real Time Clock” is made to order. We will not try that peripheral, in these labs, but you may want to, on your own.

21LC.3.3 Timers Do Various Tricks

The timer can be used in other modes, as well as for the ‘wake me when you’re done’ scheme of § 21LC.2 on page 2. Timer2 or Timer3 can, for example, measure the frequency of an external signal, relative to system clock, using “External...Capture Mode.”3 Timers 0, 1 and 3 can also count edges on an input pin rather than measure time.4 We will not take time, now, to explore these many options.

---Footnotes---
3See datasheet p. 241, 246.
4PCA takes as input a signal named ECL, p. 249, which can be assigned to a pin using the crossbar, as shown at p. 149.
21LC.4 Pulse Width Modulation (PWM) (*Dim an LED*)

21LC.4.1 PWM, Analog and Digital Versions

The technique of varying drive to a load by varying duty cycle on an output pin is quite easy to implement on a controller. The hardware required is almost nil, in contrast to what is required for continuous variation of voltage or current, which requires a DAC. PWM can be done entirely in software. But the '410 offers PWM in an easier form. The '410 offers a “Programmable Counter Array” (PCA), a timer that can be used to count an 8- or 16-bit value, and to set a bit when the count exceeds a reference value.

Thus in purely digital form it can mimic the analog PWM method, in which a waveform ramps (sawtooth or triangle) and triggers a comparator when ramp value exceeds an analog reference value. Here is a circuit sketch from Lab Op Amps 3, to refresh your recollection of the scheme.

![Figure 9: Analog PWM circuit of Lab Op Amps 3](image)

Here is the digital equivalent, which the '410 uses to implement an 8-bit PWM. When Count = Reference, the output bit goes High; when Count rolls over, output bit goes Low.\(^5\)

![Figure 10: '410 implements PWM in digital form](image)

---

\(^5\)Fig. 10 reproduces fig. 25.8 of the datasheet of C8051F410 (used with permission).
For constant duty cycle hold the reference register constant (this REFERENCE is PCA0CPHn in fig. 10 on the facing page and PCA0CPH0 in the program of § 21LC.4.2.1). The COUNT register increments at a rate set by the ‘PCA timebase.’ When COUNT exceeds reference, PWM output goes high; when COUNT rolls over to zero again, PWM goes low. REFERENCE thus determines duty cycle (from about 0.4% to 100%).

21LC.4.2 PWM Code: Slow Ramp of LED Brightness

The program below implements the gradual brightening of an LED. It does this by slowly increasing the Reference value (over a span of about three seconds), then rolling it over, so that the Reference value forms a slow sawtooth, sweeping brightness from minimum to maximum.

21LC.4.2.1 PWM configuration Using Wizard

Using the Wizard, first we turn on the PCA, and choose PCA0 (the only counter array available on the ’410), setting it up as 8-bit PWM:

![Figure 11: First, turn on the PCA (the counter array) and 8-bit PWM…](image)

In fig. 11 we have also (arbitrarily) set the duty cycle at 50%. Do you see how?

Then we place the PWM output at a particular pin.

---

6Code can take duty cycle to 0%, by writing to a separate control register.
7For this 8-bit PWM we have set the REFERENCE value at the midpoint: 80h.
We set up the PWM output as push-pull, and place this function on a free PORT0 pin, P0.2, skipping two pins that will be used in the next lab.8

21LC.4.2.2 The PWM Program

; pwm_by_wizard_nov10.a51 8-bit ramping pwm
$NOSYMBOLS ; keeps listing short
$INCLUDE ('C:\MICRO\8051\RAISON\INC\c8051f410.inc')
$INCLUDE ('C:\MICRO\8051\RAISON\INC\VECTORS320.INC') ; Tom’s vectors definition file
STACKBOT EQU 07Fh ; put stack at start of scratch indirectly-addressable block (80h and up)

; PWM output on P0.2
ORG 0h
LJMP STARTUP

ORG 080h
STARTUP: MOV SP, #STACKBOT
acall USUAL_SETUP
acall Init_Device
mov PCA0CPH0, #0 ; clear output, for orderly startup

UP: acall DELAY
inc PCA0CPH0
sjmp UP

;------- SUBROUTINES -------
; long delay:
DELAY: PUSH ACC ; save registers that’ll get messed up
PUSH B
MOV A, #000h ; set outer-loop delay value: 0, max is about 3 seconds
INITINNER: MOV B, #0 ; initialize inner loop counter

8P0.0, used last time for an LED, will be used in Lab C4 for a DAC; P0.1 will serve as input to the ADC.
INLOOP: DJNZ B, INLOOP ; count down inner loop, till inner hits zero
DJNZ ACC, INITINNER ; ...then dec outer, and start inner again.
POP B
POP ACC
RET ; Now back to main program.

;------- INITIALIZATIONS

USUAL_SETUP: anl PCA0MD, #NOT(040h) ; Disable the WDT
ret

;--------------------------------------
; Generated Initialization File --
;--------------------------------------

; Peripheral specific initialization functions,
; Called from the Init_Device label
PCA_Init:
    mov PCA0CN, #040h ; enable PCA
    mov PCA0CPM0, #042h ; 8-bit PWM, and enable Timer2
    ret

Port_IO_Init:
    ; P0.0 - Skipped, Open-Drain, Digital
    ; P0.1 - Skipped, Open-Drain, Digital
    ; P0.2 - CEX0 (PCA), Open-Drain, Digital
    mov P0SKIP, #003h ; put PWM on P0.2, skipping LED/DAC and ADC
    mov XBR1, #041h
    mov P0MDOUT, #04h ; make PWM output, P0.2, push-pull rather than open-drain
    ret

Oscillator_Init:
    mov OSCICN, #087h ; full-speed clock (just for variety)
    ret

; Initialization function for device--this fragment produced by Configuration Wizard--
; Call Init_Device from your main program
Init_Device:
    lcall PCA_Init
    lcall Port_IO_Init
    lcall Oscillator_Init
    ret
end

21LC.4.3 Improvements: 1: Change Color

Ramping the LED’s brightness is fun—but the thrill may wear off, in time. When that happens, you may want to change the hardware slightly so as to achieve a gradual change not of brightness but of color. To make that change, substitute a three-lead bicolor LED, putting an inverter between its two cathodes (use a 74HC14 or 74HC04). As one of the two LED’s becomes dimmer, the other becomes brighter; hence the mixing of colors and gradual transition from one color to the other.

Figure 13: Bicolor LED changes color as PWM varies brightness of Green vs Red LED’s within package
21LC.5  A Task for You: Improvements: 2: Ramp Up, Ramp Down

Instead of applying a sawtooth waveform to the brightness of the LED or LED’s it may be prettier to apply a triangle. That is, let the ramping down in brightness be as gradual as the ramping up. Try that, modifying the code of the program in §10 on page 10. A CJNE instruction could watch the ramping ACC value, reversing the ramp sense when ACC hits either extreme—zero or 0FFh.

21LC.6  Comparator: an Oscillator as A Start on Something More Interesting

Among the analog peripherals included in the ’410 are two comparators. They can be configured as simple analog parts: two analog inputs, one logic-level output at a port pin. Hysteresis is programmable.

More interesting, the comparator output can be polled, or can be used to generate an interrupt. In the program below, §21LC.6.3 on page 16, the comparator does no more than zero the timing capacitor. It could do something more interesting, as §21LC.7 on page 17 suggests.

We will start modestly, using just a resistor in place of the photodiode that we propose below in §21LC.7 on page 17, so the circuit will replicate the op amp RC oscillator of Lab Op Amps 3—except that it runs on a single supply.

21LC.6.1 Oscillator Hardware

Here is the hardware:

![Figure 14: Oscillator formed with ’410 comparator](image)
21LC.6.2 Comparator Configuration

21LC.6.2.1 Configuration Using Wizard

Here are comparator configuration choices:

Figure 15: Comparator configuration wizard: simple RC oscillator

In order to place the comparator inputs where we want them—at P0.6, P0.7—we must tell the crossbar to skip all the prior bits (the crossbar places the inputs as low as it is permitted to). The MOSFET drive bit, P0.3, is shown set to push-pull, so that we need not add a pullup to the MOSFET gate drive.

21LC.6.2.2 Configuration Choices, Noted Bit-by-bit

In case you like to see the choices bit-by-bit:

<table>
<thead>
<tr>
<th>Register</th>
<th>bit/byte-value</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPT0CN</td>
<td>d7 (= CP0EN)</td>
<td>1 ⇒ enable comparator 0</td>
</tr>
<tr>
<td></td>
<td>d6 (= CP0OUT)</td>
<td>1 ⇒ output high if {input +} &gt; {input -}</td>
</tr>
<tr>
<td></td>
<td>d3,d2 (= CP0HYPx)</td>
<td>01 ⇒ pos hysteresis 5 mV</td>
</tr>
<tr>
<td></td>
<td>d1,d0 (= CP0HYN)</td>
<td>01 ⇒ neg hysteresis 5 mV</td>
</tr>
<tr>
<td>CPT0MX</td>
<td>d7..D4 (CMX0Nx)</td>
<td>0011 ⇒ neg input assigned to P0.7</td>
</tr>
<tr>
<td></td>
<td>d3..D0 (CMX0Px)</td>
<td>0011 ⇒ pos input assigned to P0.6</td>
</tr>
<tr>
<td>CPT0MD</td>
<td>d1,D0 (=CPT0MDx)</td>
<td>11 ⇒ comparator to slowest and lowest power setting</td>
</tr>
<tr>
<td>P0MDOUT</td>
<td>d3 (byte 08h)</td>
<td>1 ⇒ set P0.3 to push-pull, to drive discharge MOSFET</td>
</tr>
<tr>
<td>P0MDIN</td>
<td>d7,d6 (byte 3Fh)</td>
<td>0 ⇒ set up pin (P0.7, P0.6) as analog input</td>
</tr>
<tr>
<td>P0</td>
<td>d7,d6 (byte 0C0h)</td>
<td>1 ⇒ set latch high, to permit pin use as input</td>
</tr>
<tr>
<td>P0SKIP</td>
<td>d0..d5</td>
<td>1 ⇒ skip these to place comparator inputs at d6, d7 in crossbar I/O assignments</td>
</tr>
</tbody>
</table>

Not likely, we admit. Reading these register descriptions is about as much fun as reading the U.S. tax code.
21LC.6.3 Code: RC Oscillator

And here is the code that implements an oscillator with the hardware of fig. 14 on page 14.

```assembly
; comparator_oscillator_jan11.a51 comparators watch charging cap, make oscillation;
; Jan. 11 re-assign MOSFET gate to P0.3, and let Wizard handle configuration

$NOSYMBOLS ; keeps listing short
$INCLUDE (C:\MICRO\8051\RAISON\INC\c8051f410.inc)

$INCLUDE (C:\MICRO\8051\RAISON\INC\VECTORS320.INC) ; Tom's vectors definition file
STACKBOT EQU 80h ; put stack at start of scratch indirectly-addressable block (80h and up)

; PORT USE:
; comparator inputs: - is P0.6, + is P0.7
; discharge MOSFET: P0.3

ORG 0h
LJMP STARTUP
ORG 080h

STARTUP: MOV SP, #STACKBOT-1
ACALL USUAL_SETUP
ACALL Init_Device
CLR P0.3 ; MOSFET off
SETB P0.6 ; write One's to pins to be used as inputs
SETB P0.7

OSC_LOOP: MOV A, CPT0CN ; get this where we can test rising-edge flag
JNB ACC.6, OSC_LOOP ; branch on comparator output (not flag)
SETB P0.3 ; discharge cap (turn on MOSFET)
ACALL delay ; ...allow enough time for full discharge of cap
CLR P0.3 ; ...turn off MOSFET
SJMP OSC_LOOP

;------- INITIALIZATIONS

USUAL_SETUP: ANL PCA0MD, #NOT(040h) ; Disable the WDT.
; Clear Watchdog Enable bit

; Configure the Oscillator
ORL OSCICN, #04h ; sysclk = 24.5 Mhz / 8
RET

DELAY: PUSH ACC
MOV A, #80h ; works pretty well: 10h caused step up on release of reset
DJNZ ACC, $
POP ACC
RET

; Called from the Init_Device label
Comparator_Init:

mov CPT0CN, #085h ; enable comparator 0; output high if + > - ; hysteresis +- 5 mV
; clr A ; Wait 10us for initialization
; djnz ACC, $
; anl CPT0CN, #0CFh ; clear two Comp0 flags (both rising and falling edge)--not necessary
mov CPT0MX, #033h ; establish input pins for comparators (P0.6, P0.7)
mov CPT0MD, #003h ; lowest power, lowest speed setting for comparator
ret

Port_IO_Init:

; P0.0 - Skipped, Open-Drain, Digital
; P0.1 - Skipped, Open-Drain, Digital
; P0.2 - Skipped, Open-Drain, Digital
; P0.3 - Skipped, Push-Pull, Digital
; P0.4 - Skipped, Open-Drain, Digital
; P0.5 - Skipped, Open-Drain, Digital
; P0.6 - CP0 (Cmp0), Open-Drain, Analog
; P0.7 - CP0A (Cmp0), Open-Drain, Analog

mov P0M0IN, #03Fh ; make the two comparator inputs analog (P0.6, P0.7)
mov P0M0DOUT, #008h ; make P0.3 push-pull, to drive MOSFET gate
mov P0M0SIP, #03Fh ; skip portpins so as to assign comparator inputs to P0.6, P0.7
```
mov XBR1, #040h ; enable crossbar
ret

; Initialization function for device,
; Call Init_Device from your main program
Init_Device:
    lcall Comparator_Init
    lcall Port_IO_Init
ret
END

21LC.7 Apply the Oscillator: “Suntan Alarm”

This circuit and code does nothing exciting, as it stands. It does nothing you couldn’t achieve with a single-supply comparator or op amp—or with a ’555 oscillator. Except that it uses one supply, the circuit is essentially the same as that of the first RC oscillator, in Lab Op Amps 3. But this program could be the foundation for something more interesting—like AoE’s suntan monitor (AoE §15.2).¹⁰

To apply this oscillator to that task we would need just two changes:

- replace the resistor to V+, shown in fig. 14 on page 14, with a photodiode or photoresistor, so that the frequency of oscillation is proportional to light-intensity;

![Photoresistor](image1)

**Figure 16: Photoresistor, to replace resistor in oscillator circuit**

- let the comparator output do more than simply zero the capacitor: each time it discharges the cap, the program could increment a many-bit counter. At 16 bits and a 1kHz oscillation, such a counter would allow measuring about a minute’s duration; at 24-bits, the counter would allow much longer exposures.

When the count reached a target value (set perhaps by keypad, perhaps by a pot feeding an ADC, as in the AoE design), the program could do whatever you think appropriate: sound an alarm, or perhaps deliver a mild electric shock to the user.¹¹ Better than a shock, probably, is an audible alarm, which we propose that you might install in the next section.

Here is one of the small 5-volt buzzers that your ‘Done’ signal might turn on:

![Small 5-volt buzzer](image2)

**Figure 17: Small 5-volt buzzer: could be used to signal that tanning exposure is complete**

¹⁰That AoE suntan monitor was born as an exam question that we wrote as a joke. You may imagine our surprised satisfaction, then, when we discovered that someone has patented just such a suntan monitor. No kidding. It is U.S. patent # 4, 428,050, “Tanning Aid” (Pellegrino, 1984); the invention’s description includes these remarks: “...Finally, the device includes an alarm for giving an appropriate warning when the preset dose for each session and the total dosage are achieved, and a preset “turnover” feature which can be used to divide the session...for the purposes of tanning the front of the body and the back of the body to the same extent.” We are sad to report that the invention does not seem to have made Mr. Pellegrino rich.

¹¹Just kidding. We don’t recommend that scheme.
21LC.7.1 Code for Suntan Alarm

The program below adds the features suggested in § 21LC.7 on the previous page:

- it relies on a photoresistor (whose conductance is approximately proportional to illumination)\(^\text{12}\) or a phototransistor (current proportional to illumination; this is the part you met back in the first op amp lab).
- It increments a 16-bit counter on each cycle of the oscillator;
- we’ll ask you to finish the code by writing a subroutine that compares the high byte of the 16-bit count against the keypad value. When these are equal, the program turns on a DONE signal. This signal, as we have suggested, could be discreet—turning an LED—or brash, turning on a buzzer.
- ...The program then halts, awaiting a reset from its user.

```asm
ORG 0h
LJMP STARTUP
ORG 100h
STARTUP: MOV SP, #STACKBOT-1
ACALL USUAL_SETUP
ACALL Init_Device
ACALL MISC_INITS

OSC_LOOP: MOV A, CPT0CN ; get this where we can test bit that senses level of comparator output
JNB ACC.6, OSC_LOOP ; branch on comparator output (not flag): await Vcap > Vref
SETB P0.3 ; discharge cap (turn on MOSFET)
ACALL delay ; ...allow enough time for full discharge of cap
CLR P0.3 ; ...turn off MOSFET
MOV P1, R7 ; display hi byte of cycle counter
SMFlyp OSC_LOOP ; carry on (we ask you to replace this with code that signals DONE when COUNT_high equals KEYPAD input

;----SUBROUTINE THAT WE’D LIKE YOU TO WRITE:
CHECK_END: ; get current count for comparison
DONE: ; if not yet done, keep counting
HALT: SUMP $ ; once COUNT_high equals KEYPAD input, hang here till Reset

;-----------------------------
ACALL Check_End

; ISR for comparator interrupt
ACALL COMP0_INT
ANL CPT0CN, #0DFh ; clear comparator interrupt flag
```

\(^{12}\)Typical sensitivity specification for the N5AC501085 photoresistor is 0.85 Ω/lux. The spec omits the minus sign that must be assumed.
LJMP COMPARATOR_INT_RESPONSE

ORG 150h
COMPARATOR_INT_RESPONSE: PUSH ACC ; save scratch register
16_BIT_INC: MOV A, #1 ; set up an increment that will affect CY flag
ADD A, R6 ; increment low byte of count
MOV R6, A
MOV A, R7 ; get hi byte
ADDX A, #0 ; increment hi byte if there was a carry from low byte
MOV R7, A
POP ACC
RETI

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

;------- INITIALIZATIONS
MISC_INITS: CLR P0.3 ; MOSFET off
SETB LED ; overflow (tan done) LED off
SETB P0.6 ; write One’s to pins to be used as inputs
SETB P0.7
MOV R7, #0 ; clear osc count
MOV R6, #0
RET

USUAL_SETUP: ANL PCA0MD, #NOT(040h) ; Disable the WDT.
; Clear Watchdog Enable bit
; Configure the Oscillator
ORL OSCICN, #04h ; sysclk = 24.5 Mhz / 8
RET

DELAY: PUSH ACC ; this is duration of MOSFET turn-on, used to discharge cap fully
MOV A, #80h ; works pretty well: 10h caused step up on release of reset
DJNZ ACC, $ ;
POP ACC
RET

; Called from the Init_Device label
Comparator_Init:

mov CPT0CN, #085h ; enable comparator 0; output high if + > - ; hysteresis +- 5 mV
clr A ; Wait 10us for initialization
djnz ACC, $ ; THIS IS A BIT PERFECTIONIST, FOR OUR PURPOSES!
anl CPT0CN, #0CFh ; clear two Comp0 flags (both rising and falling edge)--bars interrupt
mov CPT0MX, #033h ; establish input pins for comparators (P0.6, P0.7)
mov CPT0MD, #023h ; lowest power, lowest speed setting for comparator; int on rising edge
mov CPT0MD, #023h ;
ret

Port_IO_Init:

; P0.0 - Skipped, Open-Drain, Digital
; P0.1 - Skipped, Open-Drain, Digital
; P0.2 - Skipped, Open-Drain, Digital
; P0.3 - Skipped, Push-Pull, Digital
; P0.4 - Skipped, Open-Drain, Digital
; P0.5 - Skipped, Open-Drain, Digital
; P0.6 - CP0 (Cmp0), Open-Drain, Analog
; P0.7 - CP0A (Cmp0), Open-Drain, Analog

mov P0MDIN, #03Fh ; make the two comparator inputs analog (P0.6, P0.7)
mov P0MDOUT, #008h ; make P0.3 push-pull, to drive MOSFET gate
mov P0SKIP, #03Fh ; skip portpins so as to assign comparator inputs to P0.6, P0.7
mov XBRI, #040h ; enable crossbar
ret

Interrupts_Init:

mov EIR1, #020h ; enable interrupt on rising edge of Comp0 (flag)
mov IE, #080h ; global interrupt enable
ret

; Initialization function for device.
; Call Init_Device from your main program
Init_Device:
lcall Comparator_Init
lcall Port_IO_Init
lcall Interrupts_Init
ret

END
21LC.7.1 Try the Suntan Integrator

You should see the LCD data display (driven by PORT1) slowly counting up—at a rate proportional to light intensity sensed by the photoresistor/transistor. Try shading the sensor with your hand, to simulate a cloud passing over the sunbathing geek.

21LC.7.2 A Task for You: Improvements: 3: Improve the Monitor by Adding the DONE signal

Now try writing the subroutine labelled \texttt{CHECK\_END} in § 21LC.7.1 on page 18. This is the routine that compares the high byte of the 16-bit COUNTER (let’s call it “\texttt{COUNT\_HI}”) against the value on the KEYPAD. When the two match, the program should indicate that sunbathing is complete (by turning on LED or a buzzer) and then should get trapped in an endless loop (from which only a \texttt{RESET*} can break out). So long as the two values, \texttt{COUNT\_HI} and KEYPAD, do not match, the program should let the oscillations continue. You will find the command \texttt{CJNE} useful.\footnote{\texttt{CJNE} is “Compare Jump if Not Equal.”}
Index

BPV11 (lab), 17

CEM-1205C lab), 17
comparator
    in SiLabs controller (lab), 14
configuration wizard, SiLabs (lab), 5

interrupt
    on timer overflow, SiLabs (lab), 7

LED
    bi-color (lab), 13

oscillator
    applied as suntan alarm, SiLabs (lab), 17
    implemented with SiLabs comparator (lab), 14

PCA, SiLabs (lab), 11
PWM
    controller hardware, SiLabs (lab), 10–14
PWM
    configuration, SiLabs (lab), 11

suntan alarm (lab), 17

timer
    code, SiLabs (lab), 8
    configuration, SiLabs (lab), 5