Chapter 19LC

Lab C1 SiLabs 1: Startup

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19LC.1</td>
<td>Microcontroller (the Second Branch)</td>
<td>1</td>
</tr>
<tr>
<td>19LC.2</td>
<td>Our controller: the SiLabs C8051F410</td>
<td>2</td>
</tr>
<tr>
<td>19LC.2.1</td>
<td>C2 Programming Pins</td>
<td>3</td>
</tr>
<tr>
<td>19LC.3</td>
<td>Pinout of ‘410, As Assigned in These Lab Exercises</td>
<td>7</td>
</tr>
<tr>
<td>19LC.4</td>
<td>A Very Simple 8051 Program: Code to Blink an LED</td>
<td>8</td>
</tr>
<tr>
<td>19LC.4.1</td>
<td>Hardware: ’410 sinks current from an LED</td>
<td>8</td>
</tr>
<tr>
<td>19LC.4.2</td>
<td>Code: blink an LED</td>
<td>8</td>
</tr>
<tr>
<td>19LC.5</td>
<td>Try Code on a ’410</td>
<td>10</td>
</tr>
<tr>
<td>19LC.6</td>
<td>add Delay, to permit Full-Speed Run</td>
<td>10</td>
</tr>
<tr>
<td>19LC.6.1</td>
<td>A Little Task For You</td>
<td>11</td>
</tr>
<tr>
<td>19LC.6.2</td>
<td>Some Peculiarities of the Delay Loop</td>
<td>11</td>
</tr>
<tr>
<td>19LC.6.3</td>
<td>Try it at full-speed; try a breakpoint</td>
<td>12</td>
</tr>
<tr>
<td>19LC.6.4</td>
<td>A Better Way to Do Delay?</td>
<td>12</td>
</tr>
</tbody>
</table>

1Revisions: add header file (7/14); add note saying slide switch on LCD board must be in C2 position (7/13); add photo and references to LCD card rather than programming pod (4/13); add index (7/12); correct pin-use figure, update it to Nov. 11 version (4/12); correct bitflip inline 151 listing (11/11); add clarifying figures re cable and header (4/11); insert redrawn Jamie figs (4/11); add pin use image (dec10); correct ref below fig. 2 to plural “two 1k resistors” (nov10); plant small task: student to enter delay multiplier in delay bitflip (8/10).
converge, and at that point we hope you will dream up an application of your own for the microcontroller—whichever route you took to reach this junction.

19LC.2 Our controller: the SiLabs C8051F410

We chose an 8051-type controller, first (perhaps obviously) so as to make the two “branches” of this course consistent: one discussion of internal architecture and assembly language covers this 8051 and the Dallas part used in the other branch. A great many 8051 variations are available, of course, and many would have been satisfactory. Here is a summary of considerations that led us to the ’F410:

- modest package size (32 pins), quite easy to solder to a DIP carrier (though not so convenient as a part that is issued in DIP, like the AVR parts).2
- good set of included peripherals:
  - both ADC and DAC3
  - analog comparators
  - PWM output (handy for varying brightness of an LED, or DC-motor drive—as you know from Lab 8, where you used PWM to drive a small motor)
  - hardware serial protocols:
    - UART: RS232 standard serial port, useful to let the controller communicate with a full-size computer (just about every controller offers this)
    - SPI (the simplest of serial protocols for communication with peripherals); this scheme is simple enough so that on the other branch of labs we were able to implement SPI in code, for the Dallas part that lacks the built-in SPI hardware (see Big Board Lab µ5). But SPI is even easier when one finds it implemented in hardware, as it is for the SiLabs ’410
    - I2C (a fancier serial protocol)
  - on-chip oscillator (accurate enough to permit UART communication and other work, without addition of a crystal oscillator)
  - versatile signal and power interfacing:
    - the part includes an on-chip voltage regulator, so it can be powered by +5V while generating the 2.5V that it uses for its “core” logic
    - its Vio pin allows one to set the swing of input and output logic levels to one’s convenience: ordinarily, we will use +5V; in one exercise—when interfacing to a 3.3V serial RAM—we will apply 3.3V to this Vio pin4
    - reasonable speed: 24.5MHz clock rate, and many instructions execute (as on the Dallas part) in one clock cycle (in contrast to the 12 cycles used by the original 8051)

These are the main features that show the part’s competence. But more important than any one of these is the debugging facility that SiLabs offers. Like any controller, the ’410 can be loaded with code from a full-scale computer (the Dallas part does this, too). But the ’410 also allows crucial debugging options—without which a controller can become a maddeningly-mysterious black box. More specifically, here’s what the debugging interface permits, in addition to the obvious program load:

- single-step (this we achieve for the Dallas part with an external piece of hardware: the single-step PAL logic);

---

2DIP parts are becoming so scarce that there may be some value in introducing you to a way to work with these surface-mount parts. As SMT replaces DIP, a few resourceful manufacturers continue to produce adapters or carriers that permit breadboarding a prototype with a DIP. Some soldering skill—or at least patience—is required. But the relatively-generous 0.8mm lead spacing of the ’410—tight, but not so tight as the 0.5mm spacing of some other parts—makes the task not difficult.

3ADC’s are common; DAC’s less so.

4The low-voltage of the “core” is characteristic of recent IC’s. SiLabs, which offers several parts aimed at low-power designs, includes some controllers that can be powered at 0.9V. This sounds a little more magical than it is: the supply voltage is stepped up, on-chip, by a flying-capacitor charge pump[CHECK THIS]. But 0.9V is pretty spectacular, anyway.
display of the contents of most registers and ports of the 8051, and of internal RAM. This facility is not quite so refined as for the RIDE simulator that you can use to test code before loading it into the controller. But the display capabilities are good. They provide strong clues to what the controller is doing as it executes your code.

19LC.2.1 C2 Programming Pins

The LCD board that you have been using as display in recent labs includes C2 signals—C2D (data) and C2CK—to drive the '410’s respective pins 32 and 2. The C2 signals are provided on a $5 \times 2$ header at the top left corner of the LCD card.

19LC.2.1.1 Route USB signals to C2 path

*NOTE:* in order to use the LCD board for C2 programming, you must make sure that the slide switch labelled “C2” versus “Serial” at lower left of the board is in the C2 position:

![Slide switch](image)

Figure 1: Switch selects C2 link to controller

19LC.2.1.2 Details of C2 wiring: Our Breadboarded C2 Link

It is possible simply to connect the two C2 pins (clock and data) from the programming pod directly to the '410, as shown in fig. 2.

![Simplest wiring](image)

Figure 2: Simplest possible C2 wiring—a scheme that disables two pins. We don’t use this arrangement

But we’d like to avoid committing two pins to the debugging function, given the small number of pins available on this controller. For contrast, note that on some of its larger controllers SiLabs uses the pin-greedy but standard JTAG interface. JTAG occupies four pins, full-time. But the waste of even two pins is not tolerable for a little 32-pin part like the '410. And we need a RESET* function, in any case.

So, instead of using the simplest wiring that is shown in fig. 2 we adopt the slightly more complicated arrangement of fig. 4 on the following page. This wiring preserves the utility of both debug lines: pin 2 can serve as a manual-reset* input when not in use for C2; pin 32 can serve as a line for general-purpose I/O (GPIO).

The wiring that permits such pin-sharing is quite fussy, and we use a home-brew scheme to make the C2
connection. We do this because we want the ‘410 controller to feel to you like just another piece of hardware, like the many others that you have breadboarded in this course. We are trying to minimize magic.

Perhaps we ought to admit, though, that there is an easier way to try out a controller, though it is a way we will not use today. The easier way is to buy from SiLabs a “daughter card,” a small printed circuit with a ’410 and a few other parts including points for soldering to its I/O pins. This daughter card can be pressed onto a SiLabs “base adapter” card, which ties to a personal computer through a USB cable. There is nothing at all to build.

After this course, we expect you will use such a scheme the next time you use a controller. We have enjoyed the versatility of SiLabs’ daughter cards. When we needed many I/O pins, for example, as we did for the LCD card, we just chose a SiLabs part with 60-odd lines and shoved it onto the same base adapter that we had used with the ’410, and had used also with a still smaller controller. The development system, like the base adapter, works happily with a wide range of parts from this manufacturer. But today we ask you to work a little harder.

**Connector** To link the SiLabs programming connector to a breadboard, using its 10-line cable, we bend leads on a header, as shown in fig. 3.5

![Figure 3: 10-pin header accepts SiLabs cable: leads must be bent to straddle center “trench” of a breadboard](image)

Another view of this header appears in fig. 8 on page 6.

Fig. 4 shows the cable and connector pinout. Two of the header’s ten pins are not used. Do not connect the 3.3V output today.

![Figure 4: Cable and header pinout: two “C2” signal lines, plus one “readback” line, along with power and ground](image)

The cable will deliver power to the ’410 (+5V), drawn from the USB connection to the PC. So, your circuits will run without any additional power supply. The 3.3V output you will not use until Lab C6, when we meet a serial RAM that needs the lower voltage.

Fig. 5 shows the pin-sharing that permits use of the link to a PC without sacrificing two ’410 pins.

---

5 The header is a TE Connectivity part 5103310: low-profile, right-angle. Digikey part number is A33179.
The wiring is odd and fussy, as promised—and some details of this wiring are not at all obvious.

- The two 1k resistors placed between the C2D (data) connection and a peripheral labelled “P2.7 GPIO” are not hard to understand. These resistors provide a simple sort of isolation: C2D can drive the ’410 without fighting the peripheral.
- The 100Ω resistors between C2 and ’410: these are just for protection. They are included in order to limit current in the ’410 pin-protection diodes, if C2 signals happen to be applied when the ’410 is not powered.
- The connection at the junction between the two 1k resistors, in the case of the C2D line: this is a terminal that the programming pod uses to hold the peripheral voltage constant despite activity on the C2 line. This works even when the peripheral is a ’410 output (a case illustrated in fig. 6). This feature is not always necessary, but becomes useful if one single-steps a program that uses P2.7. In that case, the C2 lines are intermittently active even as the ’410 is driving this P2.7 line used as an output.

Below, one can see the effect of this extra connection, which we have labelled “readback.” The scope image shows a program that blinks an LED (you’ll see this very soon, in today’s lab). The program is running in single-step, controlled by the C2 link.

As the program runs, C2 is communicating with the ’410, and these signals appear as the quick pulses that mix with the slow square wave on both top traces. The C2 signals appear at the peripheral in line “b)” of the left-hand scope image of fig. 6, but disappear when corrected by the additional “readback” connection. The clean LED drive appears in the lower right-hand image of fig.6.

This signal cleanup is not of major importance, and in full-speed operation it matters not at all. But it is easily achieved, so let’s take advantage of this option.

---

6GPIO is “General Purpose Input Output.”
C2, Power and Ground to the ’410

Power and ground connections for the ’410 are few:

- $V_{reg}$: +5V: this is the input to an on-chip regulator that generates the core 2.1V or 2.5V used by the ’410
- $V_{io}$: +5V: this determines the top of I/O output swing. Setting this to +5V provides input and output properties like those of 74HC: $V_{I_{min}} = 3.5V$; $V_{O_{min}} = 4.5V$ when sourcing 3mA in “push-pull” mode (an enhanced mode, much stronger in sourcing current than the generic 8051 output, which, in the style of TTL, is highly asymmetric)
- gnd
- leave $V_{dd}$ open—but decouple it with a tantalum cap to ground (at least 1µF). This is the output of the on-chip regulator. At the moment, we are not using it.

Fig. 8 shows the way we wired the header-connector and pushbutton, on a breadboard. The left-hand image shows the board ready to accept the ’410 controller. Power and ground connections, along with two decoupling capacitors, also are shown. The LCD board provides the +5 power supply (borrowed from the USB source). The right-hand image shows controller and pod cable in place.

Make sure to keep access to at least one row of connection points at each of the 3 ports—P0, P1 and P2. Even better, give yourself access to two rows on top (at P2, P0), because we use some P0 pins for double duty, and it’s convenient to be able to leave wires in place, rather than remove and later re-install. We hope you will include a label on your ’410 carrier: it helps a lot.
The network of resistors is a little convoluted, arranged that way in order to keep it compact. You may find a neater way to do this wiring.

19LC.3 Pinout of ’410, As Assigned in These Lab Exercises

SiLabs gives a user some choice in the assignment of signals to pins, so a ’410 pinout does not look like that of an ordinary IC. (In this respect, the ’410 resembles the PALs that you have met; there, the freedom to assign signals was even greater than for this microcontroller.) This freedom is provided by a port “crossbar.” This crossbar—which is not a native 8051 feature—must be enabled in any program where it is used.

Here is the way we use the pins of the ’410 in these labs, C1 through C6. We understand that fig. 9 shows much more than you need to know today, and refers to some signal names that may be puzzling. We include all these signal so that you will have a reference to go to, if later you want it. The numbers shown in parentheses indicate the particular labs for which the pins are so used.

Some of the pins show a second use: “…/ADC0,” for instance, at P0.0, the pin that today drives an LED. That second use comes in a later lab, and at that time the earlier use will have to be disconnected (in this example, the LED). For some functions, we had complete freedom to assign a pin (LED, for example). For others we had no freedom (RX0, TX0, for example).

Because the ’410 pinout is not fixed, we will show pin use near the beginning of each ’410 lab. Today’s use is the simplest: just an LED at P0.0:
A Very Simple 8051 Program: Code to Blink an LED

As we have said before, everyone gets a kick out of seeing an LED blink, when that blinking means that something you have built is working.7

19LC.4.1 Hardware: ’410 sinks current from an LED

Wire an LED to P0.0 (top right pin), so as to turn the LED on by sinking current from the +5V supply.8

The SiLabs ’410 behaves like a traditional asymmetric 8051 by default—but can be told to provide symmetric output drive, instead (SiLabs calls this option “push-pull,” to describe the symmetric CMOS output structure). The sink/source capabilities of the push-pull are nearly symmetric: it can source current into an LED (3mA @ 4.5V), as well as sink current (8.5mA @ 0.4V).9 But we have left the pin in its usual open-drain condition, and therefore we are obliged to sink the LED current.10

We did this because we think it’s useful for you to get a habit of sinking current from a load, since you will often meet devices that show the asymmetry of TTL-like outputs: good at sinking current, poor at sourcing it. The CMOS PALs are such devices; so is the 8051 in its original output mode. P0.0 pulls current through the LED, whose anode is fed, through a 1k resistor, from the +5V supply.

![Figure 11: LED wiring: sink current through LED, rather than source it](image)

19LC.4.2 Code: blink an LED

19LC.4.2.1 The Main Loop...

The core of the program is as simple as this:

```
FLIPIT: CPL P0.0 ; flip LED, ON, then OFF...
SJMP FLIPIT
```

CPL is assembly-language shorthand for complement; SJMP means short jump, where “short” means “short enough so that a single byte will specify how far to jump.” Since the “how far” byte is a two’s-complement value, the available range is -128 to +127 (1000 0000b to 0111 1111b).

19LC.4.2.2 ...The Assembly Language Source

Here is that little loop, along with some preliminaries that the SiLabs controller requires; these initializations are explained in Classnotes μ 1.

---

7In fact, LED-love goes deeper than this. An LED may not even need to blink to keep us happy. A glowing LED reassures anyone who plugs in a well-designed battery charger. The chargers that save 50 cents by omitting the LED seem likely to fade from the market. Users like reassurance.

8This is the standard way to drive a load from an output that shows asymmetric “TTL-like” drive. Old “TTL” logic can sink much more current than it can source. If this applied only to TTL, we could dismiss this as a historical quirk. But we cannot be so dismissive, because many CMOS parts choose to mimic TTL. The Xilinx PAL that you met recently does this. So does the Dallas 8051 that appears in the other branch of these micro labs. Such devices can sink substantial currents (1.6mA, for the Dallas 8051, for example), but can source only the feeblest trickle (50µA, for the Dallas 8051). Ordinary CMOS parts, in contrast, provide symmetric drive, as you know.

9’410 datasheet Table 18.1, at p. 163.

10PORT0 of the 8051 is peculiar in that its default configuration is open-drain, in contrast to the other ports, which provide a weak pullup resistor, and a brief stronger High drive that speeds charging of a load’s stray capacitance.
Lab C1 SiLabs 1: Startup

; bitflip.a51  this blinks LED—but at a rate too fast to see unless run in single-step

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

ORG 0 ; tells assembler the address at which to place this code

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

; Enable the Port I/O Crossbar
MOV XBR1, #40h ; Enable Crossbar

SETB P0.0 ; start with LED off (it's active low), just to make it predictable

FLIPIT: CPL P0.0 ; flip LED
SJMP FLIPIT

END

19LC.4.2.3  …Machine Code Produced from the Assembly-Language Source

Above, in § 19LC.4.2.2 on the preceding page, we have listed the assembly-language version of this program. The SiLabs (or RIDE) assembler program on your PC converts this to code the 8051 can execute. This executable code appears below, alongside the original assembly language:

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>CODE</th>
<th>line #</th>
<th>ASSEMBLY CODE</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>266</td>
<td>53</td>
<td>ORG 0 ; tells assembler the address at which to place this code</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>53D9BF</td>
<td>267</td>
<td>ANL PCA0MD, #NOT(040H) ; Clear Watchdog Enable bit</td>
<td></td>
</tr>
<tr>
<td>0003</td>
<td>75E240</td>
<td>268</td>
<td>MOV XBR1, #40H ; Enable Crossbar</td>
<td></td>
</tr>
<tr>
<td>0006</td>
<td>D280</td>
<td>269</td>
<td>SETB P0.0 ; start with LED off (it’s active low)</td>
<td></td>
</tr>
<tr>
<td>0008</td>
<td>B280</td>
<td>270</td>
<td>FLIPIT: CPL P0.0 ; toggle LED, On, Off, …</td>
<td></td>
</tr>
<tr>
<td>000A</td>
<td>80FC</td>
<td>271</td>
<td>SJMP FLIPIT</td>
<td></td>
</tr>
</tbody>
</table>

We have explained, in the note on the SiLabs IDE, that all ’410 programs require initializations. That requirement applies even to this tiny two-line LED-blink program.

Here are the register initializations—simple enough, this time, so that there may be no need to list them as we do here. But we want to make this our standard practice. We think it underlines a fact about controllers: the initializations can be more complex than the program itself. Often that will be true in the simple programs that we offer in these labs. When you do something more substantial, it will cease to be true. But we hope that it helps to see the register initializations separated out.

<table>
<thead>
<tr>
<th>Register</th>
<th>bit/byte-value</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCA0MD</td>
<td>d6 (= WDTE)</td>
<td>0 (\Rightarrow) watchdog timer disabled</td>
</tr>
<tr>
<td>XBARE</td>
<td>d6 (= XBARE)</td>
<td>1 (\Rightarrow) crossbar enabled (permits pins to serve as outputs)</td>
</tr>
<tr>
<td>XBR1</td>
<td>40h</td>
<td>this is the byte value of the register that includes the XBARE bit</td>
</tr>
</tbody>
</table>

Don’t spend any intellectual energy trying to digest these initialization details. You will see many of them, over and over—including the two shown just above—and getting used to them will serve you well enough. Save your brain for more interesting challenges. Next time, we will begin to use the SiLabs “Configuration Wizard,” which makes setting up the ’410 quite manageable. You will not need to go into the dismaying bit-by-bit detail shown in the table just above. Instead, you will be able to check boxes in order to select particular behaviors: a much more manageable task.
19LC.5 Try Code on a ’410

Let’s now get on with the fun part: loading and running this code.

Follow the procedure described in “…SiLabs IDE” to “assemble,” “make” and download bitflip.a51 to the ’410. You may want to use View/Debug Windows/SFRs/Ports to let you watch P0 as the program single-steps.

The window below shows the nested menus that let you choose to watch that port on-screen.

You should see P0’s value toggle between FEh and FFh (7 lines float high; the LSB toggles as the program runs).

But the main way to judge success for your test program surely will be to see whether it does, indeed, toggle the blue LED at a few Hz when run in multiple-step mode. When it does, you will know that in principle you now can control the world with your little ’410.

19LC.6 …add Delay, to permit Full-Speed Run

If you run the program of § 19LC.4.2 on page 8 at full speed (as the head of the program warns), you will not see any interesting result. The LED will glow at half brightness (since it is on for half the time); the debugger will show you nothing, because it cannot monitor signal levels at full speed.

But if we patch in some code that wastes time between toggles, we can run the program at full speed and see the result. Here is the bitflip program, with such a delay included—except that we have not quite finished the program...
19LC.6.1 A Little Task For You

We have left undecided, in the code below, the number of times the two-byte delay loop should be run. This is a value that will be determined by the value that you load into register R4 in the line that begins `MOV R4, #________`. The “#” indicates that the value that will fill the blank is to be treated as a constant, not as the address where some other value is stored. This addressing mode is called “immediate.”

Fill in that value, aiming for a delay of about one second, each pass through the loop. Assume that the two-byte delay (the time that R4 will multiply) is about 10ms. Please enter your R4 value in hexadecimal form, indicated by an $h$ following the value, as in the line `MOV XBR1, #40h ; Enable Crossbar`.

With these additions, this small program is starting to look quite ugly: the important, central loop is hard to make out because the flow is interrupted by the patch of code that implements DELAY. We will improve this situation next time, when we meet the subroutine form. That will allow breaking this ungainly program into smaller modules, and will let us give prominence to the part that interests us—here, the bitflip loop.

19LC.6.2 Some Peculiarities of the Delay Loop

Some details of the DELAY loop are odd, and should be explained. The DELAY loop uses three 8-bit registers to generate a long delay (about one second, with the values shown, if you initialize R4 appropriately; a maximum delay would be under three seconds, running the clock at 24.5MHz, as we do here).

Initialization here are strange:

- First, it is odd, at first glance, to find us initializing two registers to zero in a count-down loop. That looks like a value that would produce minimal delay. But the zero value turns out to deliver maximum delay because the count-down operation, DJNZ (“Decrement Jump if Not Zero”), does a decrement before testing for zero.
- Second, it may seem strange that we do not re-initialize these two registers within the loop. We do re-initialize R4 (DELAY: `MOV R4, #10h`), but not A or B. We get away with this simplification because at the end of each DJNZ loop the register that has been decremented is left with value zero.
- Finally, a detail so fussy that we’re embarrassed to have to mention it: register A sometimes demands to be called by the name “ACC.” It likes this name in the DJNZ operation, and we will see, next time,
that it also insists on this in the stack PUSH operation. We don’t know why this ever made sense; the explanation is lost in the mists of history at Intel, where the 8051 was designed long ago. Assembler conventions have perpetuated this odd, and annoying, tradition.

19LC.6.3 …Try it at full-speed; try a breakpoint

After you download this program, run it at full speed: click on the large green icon (“Go”), at center-left top of the screen in the IDE.

This program also gives us a chance to demonstrate the value of a breakpoint. Place a breakpoint somewhere in the bitflip loop: highlight the line where you want to place the break, then right-click, and select “Insert/Remove Breakpoint.”

Once the breakpoint is in place, clicking Go will run the program at full-speed, but only up to the breakpoint, where execution pauses. If you click “Go” repeatedly, you should find the LED toggling each time you do this—with about a 1-second delay as the program runs the Delay code and then again hits the breakpoint.

19LC.6.4 A Better Way to Do Delay?

But does it not seem perverse to take a fast processor (in this case, clocked at its maximum normal rate, 24.5MHz\textsuperscript{11}), and then to slow its operation to a crawl? Yes, it is rather perverse—and there is, indeed, a better way to slow execution than by trapping the processor in a loop for hundreds of thousands of cycles. This better way we will explore in a later lab (C3), using the controller’s hardware timers. These can be loaded once with a delay value, and then told to notify the main program when the delay time has expired. This scheme leaves the processor free to do something useful during the timekeeping process: a much better plan than the one we demonstrated today in § 19LC.6 on page 10.

\textsuperscript{11}One more factor of two in speed would be available, if we chose to double the effective clock rate, using the CLKMUL register.
Index

breakpoint
   SiLabs (lab), 12

c2 interface, SiLabs (lab), 3–7
C8051F410
   pin use, all labs (lab), 7
   C8051F410 (lab), 2
   crosbar (lab), 8

delay loop, Silabs (lab), 11

watchdog timer (lab), 8