Chapter 22LC

Lab C4 SiLabs 4: Interrupt; DAC & ADC

Contents

22LC Port Pin Use in this Lab ................................................................. 2
22LC Interrupts .................................................................................. 2
22LC.2 First Interrupt Demonstration: Increment or Decrement of Display .................................................. 2
22LC.3 DAC ........................................................... 7
  22LC.3.1 12-bit Resolution? Yes and No ............................................... 8
  22LC.3.2 Setting up the DAC using the Configuration Wizard .................. 8
  22LC.3.3 DAC1 Initializations, from Configuration Wizard ...................... 9
  22LC.3.4 DAC test Program: a beginning, for you to complete ................ 9
  22LC.3.5 8-bit Resolution .......................................................... 12
  22LC.3.6 Code: the full program—minus the RAMP, which we leave to you ............................................. 13
22LC.4 ADC .......................................................... 14
  22LC.4.1 Hardware: A Signal Source .............................................. 14
  22LC.4.2 Code ........................................................................ 14
  22LC.4.3 Set up ADC with Configuration Wizard ............................... 15
  22LC.4.4 Preliminary ADC Test: potentiometer Voltage on Display ....... 17
  22LC.4.5 ADC to DAC upon Interrupt: full program ......................... 17
22LC.5 Apply ADC and DAC ....................................................... 19
  22LC.5.1 Watch ADC to DAC, for Sinusoid ....................................... 19
  22LC.5.2 Filter the DAC Output .................................................. 20
  22LC.5.3 and Listen to the DAC Output .......................................... 20
  22LC.5.4 A task, if you’re in the mood: Modify Waveforms ................. 20

REV 01; October 21, 2014.

1Revisions: add headerfile and index (7/14); amend wizard screenshot to show ADC at P0.0 (4/13); correct DAC1 wizard use, adding crucial enable (11/12); reduce DAC output voltage to fit its limited compliance (4/12); insert CALL to “Init_Device” in 12-bit DAC program, in place of Port_Init (11/11); correct ref to INT1* on p. 19; many errors in port use for ADC and DAC corrected (ADC0
22LC.1 Port Pin Use in this Lab

Two pins serve as analog lines, in and out of the converters: an input to an ADC, an output from a DAC. One—the ADC at P0.0—replaces the LED that has occupied this pin in Labs C1 through C3. Disconnect that LED.

Of the two interrupts shown in fig. 1, one will reappear (INT0*, used again in lab C6); one will not (INT1*).

22LC.2 Interrupts

As the class notes suggest, it takes some effort to devise a program that surely benefits by use of interrupts. Many of our lab programs that use interrupts will do the strange trick of letting the interrupt set a software flag that the main program polls. This technique seems to vitiate the value of interrupt, but does not entirely do so: apart from the value of being able to break into a running program, interrupt also can offer the utility of an on-chip flip-flop that can record the event of an interrupt request. The presence of this flop then permits two useful results:

- the flop amounts to a flag that will stay pending until the interrupt request is honored. This will seem unnecessary, if you assume that an interrupt always gets a quick response; but some circumstance can prevent such an immediate response. Two such circumstances are quite common:
  - interrupts may have been temporarily disabled, within the main program. This is the case in the storage scope exercise of Lab SiLabs C5.
  - Or—a more common case—the processor may be busy responding to another interrupt, and may be set up to finish that routine before responding to another request. Such is always the case when interrupts are running with their so-called “natural” priority. It is also the case when a lower-priority interrupt request occurs during response to a higher-priority request.

- the edge-triggering of the internal flop makes it easy to implement a do-it-once response to the transition of a request line, whereas ordinary polling would require extra code to achieve something like edge-sensing. We will take advantage of interrupt’s edge response in all of the interrupt programs of this lab.

22LC.2.1 First Interrupt Demonstration: Increment or Decrement of Display

If you want to justify this little program as more than just the simplest demonstration we could devise, then imagine that our goal is to keep track of the number of passengers on a platform with limited capacity.

to P0.0, DAC1 to P0.1 (4/11); add line assigning EX0 and EX1 to particular port pins added to hand-done interrupt-init subroutine (apr11); configuration wizard prescribed, rather than hand them the completed program (nov10); DAC output moved to P0.1, per Jason’s suggestion (8/10).
(Maybe it’s the glass platform that is cantilevered over the Grand Canyon.) One turnstile produces a high-low transition when a person enters the platform; a second turnstile does the same when a person leaves the platform. The display shows the number on the platform (and, for simplicity, we will permit our display to show a negative number on the platform!)

Interrupt 0 increments the value; Interrupt 1 decrements the value, and the main program simply loops, displaying the current number. We’ll note the simple hardware, then the program.

**22L.C.2.1.1 Hardware: Two Pushbuttons, more-or-less Debounced**

Bounce on an edge-triggered input, we have learned, can cause mischief: it will look like multiple requests. You might be inclined to dismiss bounce in the present application for a mistaken reason. As our hypothetical confused designer said, in the note called “Ready Flag: An Orgy of Error,” ‘the bounce doesn’t last long—just a few milliseconds; so don’t worry about it.’ This was thoroughly wrong, when applied to a controller that can respond within microseconds.

But it turns out that we can get away with a pseudo-debouncer that would not suffice if applied to a true edge-triggered input, like the clock of a flip-flop or counter. The simple $RC$ shown in fig. 2 is a good-enough debouncer for the present case.

![Figure 2: Pseudo debouncer to feed interrupt request pins: just an $RC$ slowdown circuit](image)

The 10k, 0.1$\mu F$ $RC$ provides the slowdown; the 4.7k is included to protect the ’410 by limiting current from the charged capacitor on power-down.

Why does a simple $RC$ slowdown suffice? We can see why, if we recall why a slowdown was not sufficient in the case of a true edge-triggered input. A slow edge caused trouble there because the edge-response was fast enough to cause indecision as the slow edge passed through the threshold region. A first response to the rising edge caused the flop output to switch; that event caused a power-supply disturbance that, in turn, caused the input stage to change its mind. And so on—reproducing some of the neurotic indecision that we saw in spectacular form when we teased a ’311 comparator with a slow edge, back in Lab Op Amps 3.

The controller’s interrupt inputs, though described as edge-sensitive, are not truly so. Instead, the controller samples those inputs on successive cycles of the internal clock. The response to the interrupt request is not so quick or direct as to cause a power-supply disturbance and consequent indecision about whether the input has crossed the input threshold. So, a simple $RC$ slowdown circuit, which eliminates large signal swings during bounce, is sufficient here. The edge-triggered interrupt does not get confused by a slow edge, as a true edge-triggered input would.

Wire such a pseudo-debounced pushbutton to each of the two Interrupt request pins: INT0* and INT1*.

**22L.C.2.1.2 Code: INC or DEC within the ISR’s**

The program is odd: the MAIN loop simply displays the value of the A register, endlessly:

```
STUCK: MOV DISPLAY, A ; show display--constant, till interrupt inc's it
SJMP STUCK
```

2This won’t happen often in life: only, perhaps, when someone parachutes in and then walks off.

3We borrowed this detail from the circuit of the SiLabs ’410 development kit.
All the action happens in the two tiny ISR’s: one increments the value of the A register, the other decrements that value.

AoE warns against trying to do too much within an ISR (AoE 15.4, subsection titled “some comments”). We think these ISR’s pass that test handsomely: each ISR consists of one line of code, and then the RETI that returns to the main program from the ISR.

Here, as usual, are the register initializations. This time, they are quite fussy. We won’t reiterate the register loadings that appear in the “USUAL_SETUP” routine. As usual, feel free to ignore these bit-by-bit details. Rely on the configuration, instead, as we do.

<table>
<thead>
<tr>
<th>Register</th>
<th>bit/byte-value</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>IT01CF</td>
<td>d7 (= INT1PL)</td>
<td>0 ⇒ INT1 active low</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>11b ⇒ assign INT1* to P0.7</td>
</tr>
<tr>
<td></td>
<td>d3 (= IN0PL)</td>
<td>0 ⇒ INT0 active low</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>11b ⇒ assign INT0* to P0.6</td>
</tr>
<tr>
<td>TCON</td>
<td>d2, d0 (= IT1, IT0)</td>
<td>11b ⇒ both interrupts edge-triggered</td>
</tr>
<tr>
<td>IE</td>
<td>d7, d2, d0 (= EA, EX1, EX0)</td>
<td>1 ⇒ enable interrupts: global, external 1, external 0</td>
</tr>
<tr>
<td>IP</td>
<td>d2 (= PX1)</td>
<td>1 ⇒ INT1 to high priority level</td>
</tr>
</tbody>
</table>

The Configuration Wizard Can Set Up Interrupts  
The Wizard eases this fussy work a great deal. The screens help to remind a user of the choices that must be made. Fig. 3 shows the principal selections, including assignment of interrupts to particular pins. (Another menu screen, not shown, permits altering the “natural priority” among interrupt sources). We placed INT0* at P0.5, INT1* at P0.7. We did not alter natural priority.

Strangely, the Wizard seems to have omitted the issue of edge versus level sensing on the interrupts. Edge-triggered interrupts are useful. To get edge-triggered behavior, we write (as you will see in the program listing, §22LC.2.1.3 on the next page):

```
SETB IT0 ; make INTO Edge-sensitive (IT0 = TCON.0)
```
...but, This Time, We'll Configure by Hand    
We’re a little chagrined to say that after showing the wonderful labor-saving Wizard once again, in this case we prefer to set up interrupts by hand. We do this partly because the Wizard forgot one option (edge-triggering), and more generally because we think it’s easier to follow the code if it is done and explained (in a comment within the listing), one action at a time. But this is only a pedagogical point: we expect you will use the Wizard whenever you can.

22LC.2.1.3 Interrupt Demo: Full Program

; int_inc_dec.a51 try interrupt
; should increment display each time interrupted by INT0, decrement for INT1

$NOSYMBOLS ; keeps listing short, lest...
$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 ($60h and up)
DISPLAY EQU P1 ; ...as in earlier byte in,out programs
; INTO* is P0.5
; INT1* is P0.7
ORG 0h
LJMP STARTUP

ORG 5D0h

STARTUP: MOV SP, #STACKBOT
ACALL USUAL_SETUP
CLR A
MOV DISPLAY, A
ACALL INT_INITS
CLR A ; (for clean startup, as usual)
STUCK: MOV DISPLAY, A ; show display--constant, till interrupt inc's it
; so you must clock several times while high, then low)

;-----------------------------
; ISR0: This is response to INT ZERO: INCREMENT A
ORG INT0VECTOR ; this is defined in VECTORS3210.INC, included above.
; It is address 03h, the address to which micro hops
; in response to interrupt ZERO
ISR0: PUSH PSW ; just to demonstrate swapping in a fresh set of scratch registers
MOV PSW, #08h ; select set #1 of scratch registers (vs #zero used for Main)
INC A
POP PSW ; restore old set of registers
RETI

; ISR1: This is response to INT ONE: DECREMENT A
ORG INT1VECTOR ; this is defined in VECTORS3210.INC, included above.
; It is address 13h, the address to which micro hops
; in response to interrupt ONE
ISR1: PUSH PSW ; just to demonstrate swapping in a fresh set of scratch registers
MOV PSW, #10h ; select set #2 of scratch registers (vs #zero used for Main, and #1 for Int0)
DEC A
POP PSW
RETI

;--- SUBROUTINES ----
USUAL_SETUP: ANL PCA0MD, #NOT(040h) ; Disable the WDT.
; Clear Watchdog Enable bit
; Configure the Oscillator
ORL OSCICN, #04h ; sysclk = 24.5 Mhz / 8
; Enable the Port I/O Crossbar
MOV XBR1, #40h ; Enable Crossbar (this is done by the Wizard, too)
RET
Lab C4 SiLabs 4

; --- NOW ENABLE INTERRUPTS (these hand-done inits seem clearer than those done by Wizard, shown below; and the Wizard forgot
; to permit the edge-trigger option [DO Wizards forget?]) ----
INT_INITS: MOV IT01CF, #075h ; assign interrupts Zero and One to portpins P0.5 and P0.7
; SETB IT0 ; make INT0 Edge-sensitive (IT0 = TCON.0)
; SETB IT1 ; ditto for INT1
; SETB PX1 ; give higher priority to INT1 (PX1 = IP.2) (try this if you feel like it)
; SETB EX0 ; ...and enable INT0 (EX0 = IE.0)
; SETB EX1 ; ...and INT1
; SETB EA ; Global int enable (EA = IE.7)
RET

-----------------
; THESE REMAINING LINES WERE PRODUCED BY THE WIZARD--and are not used in this program, since we set up interrupts by hand
; Peripheral specific initialization functions,
; Called from the Init_Device label
Port_IO_Init:
   mov XBR1, #040h
   ret

Interrupts_Init:
   mov IT01CF, #075h
   mov IE, #085h
   ret

; Initialization function for device,
; Call Init_Device from your main program
Init_Device:
   lcall Port_IO_Init
   lcall Interrupts_Init ; commented out because the hand-done version seems clearer
   ret

END

22LC.2.1.4 Run the Program

Slow Motion Checkout: Single-Step Try it in single-step or multiple-step, first. You should see the program loop in the main “STUCK” loop, until you press one of the interrupt pushbuttons. When you do press that button, hold it down for at least a couple of cycles of the multiple-stepping: a brief pulse on the interrupt will have no effect, because these are not true edge-triggers, as we noted in § 22LC.2.1.1 on page 3.

You should see execution move to one or the other of the ISR’s, in response to a press of a pushbutton. If you keep the button down, you should not see any further response to this interrupt request: in the important sense, these interrupts are edge-triggered, even though they do not respond to a brief pulse.

If you press the INT0* pushbutton while single-stepping through ISR1, does INT0* break into the ISR? It should not, even though INT0* does have priority over INT1*. This sort of priority—known by the strange name “natural”—determines only which interrupt request wins when both requests come at the same time.

Full Speed Run it at full-speed, and try the two pushbuttons. As in single-step, one button-press should produce a single increment or decrement of the display value, thanks to the edge-triggered behavior of these interrupts.

Test the Pseudo-Debouncer If you wonder whether the pseudo-debouncing works, try removing the slow-down capacitor from one of the switches. You should find, when the program is running at full speed, that instead of incrementing or decrementing, a button-press sometimes changes the display by several counts.

22LC.2.1.5 Try Altering the “Natural Priority” of Interrupts

Within the interrupt-initialization section appears one line that we have commented-out: SETB PX1. This line would have two effects:
• it would reverse the normal priority rank between interrupts zero and one, and...

• would allow INT1 to break into an INT0 service routine, whereas the normal, “natural” priority scheme lets even a lower-priority routine finish before the controller will respond to the higher-priority interrupt, as we mentioned in § 22LC.2.1.4 on the preceding page.

If you feel like watching these effects in action, remove the semicolon that comments out SETB PX1, and watch as you single-step (or multiple-step) the program. Does INT1* now break into the ISR for INT0* (let’s call that “ISR0”)? If an INT0* request comes during ISR1, what happens after the return to Main from ISR1?4

22LC.3 DAC

It would be nice to set ADC and DAC ranges equal. We cannot. ADC input range is 0 to 2.2V (the value of \( V_{\text{reference}} \)); DAC “output compliance” range is smaller: 0 to 1.3V. We will set the DAC output range slightly below that limit: 0 to 1.2V.5

The DAC output is a current, not a voltage. The DAC sources this current, and the easiest scheme for converting this current to a voltage is just to pass it through a resistor to ground. For a larger output swing we would need to add an op amp.

![Figure 4: DAC output is a current; \( R \) needed to convert it to a voltage](image)

We set the DAC full-scale current at 1mA (our old favorite) and let it flow through a 1.2k resistor to ground.

22LC.3.0.6 Justification: A Choice, Left- or Right-

Like many DAC’s and ADC’s, this DAC allows us to choose whether to use left- or right- justification of its 12-bit result:

![Figure 5: DAC input, like ADC result, can be justified either left- or right-](image)

We have chosen to left justify because this arrangement permits us to use the eight MSB’s by themselves, treating DAC (and, later, ADC) as if it were an 8-bit part. Ignoring the low 4 bits does not corrupt the high 8; it simply reduces the converters’ resolution. Left-justifying the DAC input value makes it easy, in the exercise that follows (§ 22LC.3.5 on page 12), to change from 12-bit to eight-bit output.

---

4Because we have made the interrupts edge-sensitive, the INT0* request will have been saved on a flip-flop, and will be honored when the response to INT1* concludes.

5DAC “output voltage compliance” worst-case limit is \( V_{\text{DD}} - 1.2V = 1.3V \) (datasheet Table 6.1, p. 75).
22LC.3.1 12-bit Resolution? Yes and No

As we have said in today’s class notes ($\mu$4), DAC and ADC offer more resolution than we can handle in our breadboarded setup: noise normally buries the effect of the lowest two or three bits of the available twelve. But just to show off the 12-bit resolution, let’s run the DAC in slow motion, watching its output with a DVM. The DVM’s averaging effect will make even the LSB steps visible.

As usual, one is obliged to make many initialization choices, in order to use this peripheral. We had to choose even what event should be used to update the DAC. We chose a write to the DAC register for this updating event; it seemed the most straightforward scheme. We rejected an update on any of four timer overflows and an update on an edge of an input line named CNVSTR.

22LC.3.2 Setting up the DAC using the Configuration Wizard

Choosing the DAC as peripheral, and then DAC1 which we chose because we like its (rigid) assignment to P0.1 (leaving P0.0 free for other use), we get a menu screen, as usual:

Choose...

- enable DAC1
- left-justify
- 1mA output current\(^6\)
- update on write to high byte
- using the sub-menu, set the crossbar to “skip” P0.1, leaving it free for the DAC
- ...and set it as an analog pin

Install a 1.2k resistor to ground. This will convert the 1mA (max) DAC0 current output to a voltage.

\(^6\)...because of our longstanding love affair with the value ONE, of course. It makes arithmetic so easy!
22LC.3.3 DAC1 Initializations, from Configuration Wizard

The result will be a set of initialization commands, which we ask you to append to the incomplete program listed at § 22LC.3.4:

DAC_Init:
  mov IDA1CN, #0F2h
  ret

Port_IO_Init:
; P0.0 - Unassigned, Open-Drain, Digital
; P0.1 - Skipped, Open-Drain, Analog
; P0.2 - Unassigned, Open-Drain, Digital
; P0.3 - Unassigned, Open-Drain, Digital
; P0.4 - Unassigned, Open-Drain, Digital
; P0.5 - Unassigned, Open-Drain, Digital
; P0.6 - Unassigned, Open-Drain, Digital
; P0.7 - Unassigned, Open-Drain, Digital
  mov P0MDIN, #0FDh
  mov P0SKIP, #002h
  mov XBR1, #040h
  ret

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

end

You should append these initialization commands to the incomplete program that we called dac_test_12_apr11.a51, below in § 22LC.3.4.

22LC.3.4 DAC test Program: a beginning, for you to complete

Here we’ve written a loop that adds one to the 12-bit DAC value, each time around. (The details of this 12-bit increment are spelled out after the program listing.) You can use the Wizard to add the initializations.

; dac_test_12_apr11.a51 12-bit resolution: apparent only in slow-motion, when watching with DVM
; changed to DAC1
$NOSYMBOLS ; keeps listing short
$INCLUDE (C:\MICRO\8051\RAISON\INC\c8051f410.inc)
STACKBOT EQU 080h ; put stack at start of scratch indirectly-addressable block (80h and up)

ORG 0h
LJMP STARTUP

ORG 080h
STARTUP: MOV SP, #STACKBOT-1
   ACALL USUAL_SETUP
   ACALL ANALOG_SETUP
   ACALL Init_Device ; this calls the initializations generated by Configuration Wizard--and assumes you’ve

; here’s 12-bit ramp
   CLR A
   CLR C ; clear carry bit
   MOV IDA1L, A ; first-pass, to see voltage at zero, on DAC1
   MOV IDA1H, A
RAMP_12: MOV A, IDA1L ; recall low nybble
ADD A, #10h ; increment low nybble, left-justified (updating carry)
MOV IDA1L, A ; update low byte of DAC
MOV A, IDA1H ; recall high byte
ADDC A, #0 ; use carry out of low byte, if any, to update high byte
MOV IDA1H, A
SJMP RAMP_12

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

ANALOG_SETUP: SETB P0.0 ; make sure latch is high
RET

; NEXT SECTION IS INCOMPLETE: (here append the commands prescribed by the Configuration Wizard)-------
; You don’t need to use the next four lines; the Wizard should generate these lines, and others as needed
;PORT_SETUP:
;
; set up Vref to 2.2V (necessary for ADC, probably not for DAC)
MOV P0MDIN, ; set up DAC1 pin for analog
MOV P0SKIP, ; tell crossbar to skip DAC1 bit (P0.1)
MOV IDA1CN, ; enable DAC1; update on write to high byte; left-justified; 1mA full-scale
RET

END

22LC.3.4.1 Some details of the “12-bit increment”

The 12-bit ramp loop extends its increment beyond one byte using the trick we saw in class notes \( \mu_2 \):
ADDC A, #0, incrementing if the CARRY bit is set. The increment of the low nybble may look odd, at first.

A preliminary point: we must not use INC A, for any greater-than-byte operation, because INC affects no flags. Thus INC could not be used for this 16-bit operation, which uses the Carry flag in its second step.\(^7\)

A second point: given that we must use ADD rather than INC, why add 10h rather than 01h for an increment? Because the lowest nybble (that is, the lowest four bits) of the 12-bit value lives in the high nybble of the register, IDA1L. Adding 10h may look funny, but it does, indeed, simply increment the lowest nybble of the 12-bit value.

22LC.3.4.2 Trying the 12-bit Ramp

Note that this program includes no delay. So, don’t expect to see anything coherent if you run it at full-speed. There are two ways to look for the tiny 12-bit LSB increments:

- single-step, so as to allow time for the DVM to settle and average out noise;
- plant a breakpoint somewhere in the loop (see details, below).

Either method should show the increments of a single LSB.

Voltage value of an LSB

What LSB step size do you observe, for this 12-bit DAC? Given the full-scale value of 1.2V, what step size would you expect?\(^8\)

\(^7\)This kind of detail is buried in the description of the INC operation, in the Philips 8051 Instruction Set Reference, www.nxp.com/acrobat/80C51FAM_PROG_GUIDE_1.pdf.

\(^8\)That’s right: 1.2V/2^{12}.}
Set up a WATCH window...

You can watch the IDA1 value in either of two ways. The more laborious method, giving you better control, is to set up a WATCH window for particular registers (this lets you see only what you have chosen, whereas the IDAC window in fig. 9 shows more than may interest you: IDAC1 and a couple of initialization registers.

Watch Window for particular registers

Here, we’ve chosen to display as separate items the two-byte input register of the IDAC1:

...Add a Breakpoint

A breakpoint—which halts execution when program flow reaches it—allows us to see the alteration of the IDAC1 input after each pass around the loop. Each time you click the green “Go” button, you can see the IDAOL value increment (in its peculiar way: it will grow by 10h, but that is a 1-bit increment for the lowest nybble of the DAC input register). In fig. 8 we placed both input registers in the Watch window.

Instead of using a “watch” window, we could display the DAC input registers by choosing View/DebugWindows/SFR’s/IDAC

In fig. 9, using that IDAC display, we see the same information as in fig. 8. Again we see the value of the low-order nybble, IDA1L, after it has been incremented twice:
At the same time, watch the DVM, and you can note the size of the LSB step. Both figures—fig. 8 on the previous page and fig. 9 on the preceding page—show a count of two going to the DAC (whose LSB lives at d4 of the IDA1L register). At this count, we saw \( V_{out} \approx 0.9\,mV \).

Incidentally, you need to ground the DVM as close as you can to the '410’s ground: a few inches of breadboard ground line can drop several LSB’s in voltage (we saw a drop of about 3mV, along the six-inch length of the ground line on the breadboard strip).

### 22LC.3.5 8-bit Resolution

Eight-bit resolution is more useful, in our breadboarded circuit. A scope can display the individual steps of the DAC’s 8-bit ramp, and we will from this point use both DAC and ADC as 8-bit devices. This we can do by ignoring the low four bits, using only the IDA1H register of the DAC, and the equivalent high-byte register of the ADC.

At 8-bits, when we included some averaging, a scope showed the LSB steps of about 5mV apiece:

![Figure 10: DAC ramp at 8-bit, averaged: LSB steps visible](image)

### 22LC.3.5.1 Code: the RAMP Loop, A Task for You

This program is very simple, lacking the contortions required by the 12-bit increment. The ramp loop is as brief as the comments below suggest. We leave the code to you. Note that for 8-bit control of the DAC, we need to write only to the register that holds the top 8 of its 12 bits. This register is IDA1H.

Let’s use the accumulator (A) as scratch register, where we can do the incrementing.
Lab C4 SiLabs 4

EIGHT_BIT: ; transfer current "ramp" value to DAC1
; increment ramp value
; waste a little time
; keep doing this, forever

22LC.3.6 Code: the full program—minus the RAMP, which we leave to you

The full program includes a delay, so that we can run it at full-speed and watch the ramp on a scope. As usual, the initialization chores dwarf the code loop, in fussiness.

; dacl_Sbit_wizard_MT.a51 8-bit resolution
$NOSYMBOLS ; keeps listing short
$INCLUDE (C:\MICRO\8051\RAISON\INC\c8051f410.inc)
STACKBOT EQU 07Fh ; put stack at start of scratch indirectly-addressable block (80h and up)
ORG 0h
LJMP STARTUP
ORG 080h
STARTUP: MOV SP, #STACKBOT
acall USUAL_SETUP
acall Init_Device ; let the wizard do the work for us
setb p0.1 ; make sure DAC pin is high
; here's 8-bit ramp
CLR A
EIGHT_BIT: ______________ ; transfer current "ramp" value to DAC1 (high byte)
______________ ; increment ramp value
______________ ; waste a little time (use DELAYSHORT routine)
______________ ; keep doing this, forever

DELAYSHORT: PUSH ACC
MOV A, #10H
DJNZ ACC, $
POP ACC
RET
USUAL_SETUP: ANL PCA0MD, #not(040h) ; disable WDT
ret

;------------------------------------
; Generated Initialization File --
;------------------------------------
; Initialization function for device,
; Call Init_Device from your main program
Init_Device:
 lcall DAC_Init
 lcall Port_IO_Init
ret

; Peripheral specific initialization functions,
; Called from the Init_Device label
DAC_Init:
 mov IDA1CN, #0F2h
ret

Port_IO_Init:
 ; P0.0 - Unassigned, Open-Drain, Digital
 ; P0.1 - Skipped, Open-Drain, Analog
 ; P0.2 - Unassigned, Open-Drain, Digital
 ; P0.3 - Unassigned, Open-Drain, Digital
 mov POMDIN, #0F7h
 mov POSKIP, #002h
 mov XBR1, #040h
ret

end
A DVM will show you the LSB value—the increment visible if you use multiple-step and the small delay. We saw about 8mV. A scope image is more exciting. You may have to limit bandwidth (hiding fuzz) in order to make the individual steps clear.

### 22LC.4 ADC

#### 22LC.4.1 Hardware: A Signal Source

In the initial ADC test, it’s convenient to feed the ADC with a DC level. We can use a potentiometer, with its maximum value just a little beyond full-scale, and a moderate $R_{\text{Thevenin}}$. (What is the maximum $R_{\text{Thevenin}}$ for this circuit? This question will warm your heart, no doubt, carrying you back to day one of this course!)\(^9\)

![Figure 11: Potentiometer can provide a DC source, to feed the ADC](image)

\[^9\text{In case you aren’t instantly carried back to those happy days when you were younger, we will remind you that the maximum }R_{\text{Thevenin}}\text{ comes when the pot slider is set to its topmost position. Then }R_{\text{Thevenin}} = (1k \parallel 1k) = 0.5k. \text{ At every other setting }R_{\text{Thevenin}}\text{ is lower.}\]

#### 22LC.4.2 Code

The task of setting up the ADC is similar to setting up the DAC; the many options call for many choices. Our first ADC test program displays the 8-bit ADC result on the LCD, and the program loop is as simple as most of our programs. The routine that picks up a sample from the ADC is this:

```assembly
GET_SAMPLE: CLR CNVRT_START ; low on AD0BUSY (to permit rising edge)
            SETB CNVRT_START ; rising edge on AD0BUSY starts conversion
            JNB CNVRSN_DONE, $; hang here till conversion-done flag
            MOV SAMPLE_HI, ADC0H ; high byte
            RET
```

GET_SAMPLE includes a pair of commands that provide a rising edge on a bit that starts the ADC, plus a test loop that waits till a sample is ready (the “$” as jump destination means “this line,” so the JNB takes execution to the JNB line itself). The bits, “CNVRT\_START” and “CNVRSN\_DONE” are defined at the head of the program, as usual.

…Then the main program displays the ADC sample, and sends it to the DAC so we can watch on a scope:

```assembly
MOV DISPLAY_HI, SAMPLE_HI ; this to let us watch as it proceeds
MOV IDA0H, SAMPLE_HI       ; ...this to DAC, for scope display
```

#### 22LC.4.2.1 Many initializations required for ADC

And—as usual—the full program includes lots of initialization choices. Here are the register-initialization details:
<table>
<thead>
<tr>
<th>Register</th>
<th>bit/byte-value</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC0CN</td>
<td>d7</td>
<td>ADC0 enable: 1 ⇒ enable</td>
</tr>
<tr>
<td>...</td>
<td>d5</td>
<td>AD0INT: 1 ⇒ sample ready (this is a flag we test)</td>
</tr>
<tr>
<td>...</td>
<td>d4</td>
<td>AD0BUSY: rising edge starts conversion</td>
</tr>
<tr>
<td>...</td>
<td>d2</td>
<td>AD0LJST: 1 ⇒ Left Justify</td>
</tr>
<tr>
<td>...</td>
<td>d1, d0</td>
<td>selects event that starts conversion</td>
</tr>
<tr>
<td>...</td>
<td>00b</td>
<td>selects rising edge on AD0BUSY as event that starts conversion</td>
</tr>
<tr>
<td>ADC0MX</td>
<td>00h</td>
<td>ADC0 mux assignment ⇒ to P0.0</td>
</tr>
<tr>
<td>ADC0TK</td>
<td>d3...d0</td>
<td>tracking mode</td>
</tr>
<tr>
<td>...</td>
<td>FBh</td>
<td>dual-mode (this is the default)</td>
</tr>
<tr>
<td>ADC0CF</td>
<td>d2, d1</td>
<td>number of conversions per start-convert</td>
</tr>
<tr>
<td>...</td>
<td>00b</td>
<td>⇒ one sample/start-convert</td>
</tr>
<tr>
<td>P0</td>
<td>d1, d0</td>
<td>both bits high, to permit use as analog pins</td>
</tr>
<tr>
<td>P0MDIN</td>
<td>0FCh</td>
<td>ADC &amp; DAC pins set to analog (P0.0, P0.1)</td>
</tr>
<tr>
<td>P0SKIP</td>
<td>23h</td>
<td>skip INT0*, ADC and DAC, d1, d0</td>
</tr>
<tr>
<td>P1SKIP</td>
<td>04h</td>
<td>skip Vref pin</td>
</tr>
<tr>
<td>REF0CN</td>
<td>13h</td>
<td>enable Vref, set full-scale to 2.2V</td>
</tr>
</tbody>
</table>

The more complex the peripheral, the more painful the initializations!

22LC.4.3 Set up ADC with Configuration Wizard

Once again, the Wizard provides a minimally-painful way to select among the many options.

Most are straightforward; again, we want to left-justify, so that can use the ADC as an 8-bit device, if we like.  A few of the initializations are not at all obvious:

- we will start a conversion by giving a pulse to a bit called AD0BUSY

- we accept the default “tracking” mode, “dual.” Tracking describes the condition of the converter’s sample-and-hold while it is in sample or track mode (source connected to storage capacitor). After a Start-conversion signal, tracking runs long enough to acquire a sample, which then is converted. At the end of conversion, tracking resumes.

- we route the ADC input to a free pin, P0.0.

There are still other choices to make, as the table in § 22LC.4.2.1 on the preceding page told us. In addition to left-justifying and setting range, as for the DAC, we need to set up a voltage reference:
The port configuration also is more complicated for the ADC than for earlier programs. We'll leave the DAC setup as before, at P0.1, and will add an interrupt input, and will set up the voltage reference, as well.
The crossbar makes both DAC and ADC pins analog (we’ll soon be using both ADC and DAC in a program), and permits use of the voltage reference, brought out at another pin, P1.2. This pin is made analog, and is brought out, even though the ADC uses it only on-chip.

22LC.4.4 Preliminary ADC Test: potentiometer Voltage on Display

This ADC test amounts to demonstrating a digital voltmeter. Here is the head of the program. The ADC initializations you can add with the Wizard’s help.

```assembly
; adc_wizard_apr11.a51 ; ADC test, done with configuration wizard Jan 11: pins reassigned
; ADC as voltmeter (8 bits displayed on LCD), input on P0.0

$NOSYMBOLS ; keeps listing short, lest...
$INCLUDE (C:\MICRO\8051\RAISON\INC\c8051f410.inc) ; ...this line might produce huge list
; of symbol definitions [all '51 registers]
$INCLUDE (C:\MICRO\8051\RAISON\INC\VECTORS320.INC) ; Tom’s vectors definition file
STACKBOT EQU 080h ; put stack at start of scratch indirectly-addressable block (80h and up)
DISPLAY_LO EQU P2
DISPLAY_HI EQU P1
; port use: ADC0 on P0.0

ORG 0h
LJMP STARTUP

ORG 080h

STARTUP: MOV SP, #STACKBOT-1
acall USUAL_SETUP
acall SETUP_ANALOG
acall Init_Device

START_ADC: mov ADC0CN, #84h ; low on ADDBUSY
mov ADC0CN, #94h ; high on ADDBUSY starts conversion
jnb ADC0CN.5, $ ; hang here till conversion-done flag
mov DISPLAY_HI, ADC0H
mov DISPLAY_LO, ADC0L ; show ADC value on display
sjmp START_ADC

;------- INITIALIZATIONS

USUAL_SETUP: anl PCA0MD, #NOT(040h) ; Disable the WD'T. 
ret ; Clear Watchdog Enable bit

SETUP_ANALOG: setb P0.0 ; set pin high, to permit use as analog pin (this for ADC)
ret

;------- Wizard’s INITIALIZATIONS (These you can fill in with the help of the Configuration Wizard)

Once you are satisfied that both DAC and ADC work, you can let ADC talk to DAC, in the next section.

22LC.4.5 ADC to DAC upon Interrupt: full program

The program below takes a sample from the ADC each time the ’410 is interrupted. The program then displays the 8-bit sample on the LCD, and also sends the sample to the DAC. (This program surely is too long to type in. We hope you’ll download it from our website.)

; adc_dac_int_jan11.a51 : splices ADC_TEST with DAC_TEST, to do in-out on interrupt
; redone using Wizard, pins reassigned Jan 11

$NSYMBOLS ; keeps listing short, lest...
$INCLUDE (C:\MICRO\8051\RAISON\INC\c8051f410.inc) ; ...this line might produce huge list
; of symbol definitions (all '51 registers)
$INCLUDE (C:\MICRO\8051\RAISON\INC\VECTORS320.INC) ; Tom's vectors definition file
STACKBOT EQU 080h ; put stack at start of scratch indirectly-addressable block (80h and up)
SOFTFLAG EQU 0 ; software flag at bit 0: used by ISR to say 'time to sample'
CNVRSN_DONE EQU ADC0CN.5 ; flag bit indicates ADC sample is ready
CNVRT_START EQU ADC0CN.4 ; ADC start bit: needs low-going pulse
DISPLAY_HI EQU P1 ; display high 8 bits of 12
SAMPLE_HI EQU R7
; INT0 at P0.5, to signal time to take a sample
; DAC1 at P0.1
; ADC0 at P0.0

ORG 0h
LJMP STARTUP

ORG 080h

STARTUP: mov SP, #STACKBOT-1
acall USUAL_SETUP
acall Init_Device
acall INT_EDGE ; make INT1 edge-sensitive
acall PORT_SETUP ; set bits high, to free them for ADC and DAC

CHECK_FOR: jnb SOFTFLAG, $ ; hang here till interrupt says 'time to transfer'
clr SOFTFLAG
clr EA ; disable interrupts, to make sure it get played out to DAC
acall GET_SAMPLE
mov DISPLAY_HI, SAMPLE_HI ; this to let us watch as it proceeds
mov IDA0H, SAMPLE_HI
setb EA ; re-enable interrupts
sjmp CHECK_FOR

;--- SUBROUTINES ----
ISR0: This is response to INT ONE: set softflag
OR INT0VECTOR ; this is defined in VECTORS3210.INC, included above.
; It is address 13h, the address to which micro hops
; in response to interrupt ONE
ISR0: setb SOFTFLAG ; pseudo-polling: a flag that MAIN will check
RETI

GET_SAMPLE: CLR CNVRT_START ; low on AD0BUSY (to permit rising edge)
SETB CNVRT_START ; rising edge on AD0BUSY starts conversion
JNB CNVRSN_DONE, $; hang here till conversion-done flag
CLR CNVRSN_DONE ; not required, but tidy to clear this flag
MOV SAMPLE_HI, ADOCH ; high byte
RET

; ------INITS ---------
USUAL_SETUP: anl PCA0MD, #NOT(040h) ; Disable the WDT.
; Clear Watchdog Enable bit
; Configure the Oscillator
mov OSCICN, #087h ; max speed sysclk: 24.5 Mhz
RETI

INT_EDGE: setb IT0 ; make INT0 Edge-sensitive
RETI

PORT_SETUP: setb P0.0 ; make sure latch is high (this for DAC)
setb P0.1 ; ...and this for ADC
ret

;-------------------------------------------------------------
; Generated Initialization File --
;-------------------------------------------------------------
; Peripheral specific initialization functions,
; Called from the Init_Device label

ADC_Init:
    mov ADCUMX, #000h ; ADC0 at P0.0
    mov ADCUCN, #084h
    ret

DAC_Init:
    mov IDAICN, #0F2h
    ret

Voltage_Reference_Init:
    mov REF0CN, #013h ; enable internal reference, set value to 2.2V
    ret

Port_IO_Init:
    ; P0.0 - Skipped, Open-Drain, Analog : ADC0
    ; P0.1 - Skipped, Open-Drain, Analog DAC1
    ; P0.2 - Unassigned, Open-Drain, Digital
    ; P0.3 - Unassigned, Open-Drain, Digital
    ; P0.4 - Unassigned, Open-Drain, Digital
    ; P0.5 - Skipped, Open-Drain, Digital INT0*
    ; P0.6 - Unassigned, Open-Drain, Digital
    ; P0.7 - Unassigned, Open-Drain, Digital
    ; P1.0 - Unassigned, Open-Drain, Digital
    ; P1.1 - Unassigned, Open-Drain, Digital
    ; P1.2 - Skipped, Open-Drain, Analog ; this for voltage reference
    ; P1.3 - Unassigned, Open-Drain, Digital

    mov P0MDIN, #0FCh ; two analog pins, P0.1, P0.0
    mov P1MDIN, #0FBh ; one analog pin, for ADC voltage reference
    mov P0SKIP, #0A3h ; skip pins for interrupt, ADC and DAC
    mov P1SKIP, #004h ; skip pin for voltage reference
    mov XBR1, #040h ; enable crossbar
    ret

Interrupts_Init:
    mov ITOUCF, #075h ; place INT0* at P0.5
    mov IE, #081h ; enable INT0* and enable all interrupts (global: EA)
    ret

; Initialization function for device,
; Call Init_Device from your main program

Init_Device:
    lcall ADC_Init
    lcall DAC_Init
    lcall Voltage_Reference_Init
    lcall Port_IO_Init
    lcall Interrupts_Init
    ret

END

22LC.5 Apply ADC and DAC

22LC.5.1 Watch ADC to DAC, for Sinusoid

This program provides a nice setup for trying out the sampling rules that we discussed—and perhaps demonstrated in class—back on the day when you built an ADC in lab (Lab 18/D5).

Use a function generator to provide a sinusoid for the ADC to convert. Make sure the sinusoid does not exceed the input range of the ADC: 0 to 2.2V.

Use a TTL square wave, from breadboard or function generator, to drive INT0*, setting $f_{\text{sample}}$. As a first
test, watch analog in and analog out on a scope. If you set the sampling rate at 3 or 4 times $f_{in}$, you should see something like what’s shown in fig. 14.

![Figure 14: Sinusoid to ADC, DAC output reconstructed from ADC samples](image)

As you play with the relation between $f_{in}$ and $f_{sample}$, you can confirm, at least roughly, the teachings of Nyquist. For $f_{sample}$ below $2 \times f_{in}$ you should be able to recognize aliasing. But the reconstructed waveform may be so weird that it is hard to recognize as a sinusoid at a strangely-low frequency. The filter that we suggest in § 22LC.5.2 will help to make aliasing more obvious.

### 22LC.5.2 Filter the DAC Output

A good low-pass “reconstruction filter” allows one to sample efficiently, with a sampling rate just a little above twice the maximum $f_{in}$. You may recall the rule that the lowest sampling artifact appears at $f_{sample} - f_{in} - max$. The good filter available to us—the Max294—attenuates to below -60dB (one part in a thousand) just 20% above $f_{cutoff} \approx f_{3dB}$. So, we can safely sample at $2.2 \times f_{in} - max$ or above. You’re not likely to want to calculate this $f_{sample}$ when, instead, you can just play with $f_{sample}$ by adjusting a function-generator knob, while watching the results on a scope.

Today’s classnotes ($\mu 4$) set out the details of the filter circuit, and of the audio amplifier that permits listening to the reconstructed output. We will not repeat that detail, here.

You will need a third oscillator to drive the clock of the MAX294. If you don’t have a function generator available, wire up a ’555 oscillator to give yourself a square wave at a frequency adjustable from perhaps 5kHz to 1MHz. (See Lab Op Amps 3.)

### 22LC.5.3 …and Listen to the DAC Output

We hope you’ll wire up the audio amp described in today’s classnotes, and listen to the output waveforms. The information delivered by your ears will supplement what scope and eyes can reveal.

### 22LC.5.4 A task, if you’re in the mood: Modify Waveforms

Today’s classnotes propose some changes that the 8051 might apply to a sampled waveform, rather than just spit it out at once to the DAC. A few lines of code could implement a rectifier, for example—full-wave or half-wave. Perhaps more fun is a digital-lowpass filter, done with weighted-averaging between a current sample and a running average.

As we admit in those class notes, these exercises are essentially entertainments, since the 8051 is feeble, indeed, as a signal processor. But the exercise will give you a little practice in coding, while delivering the reward of vivid results, visible on a scope screen, and perhaps audible.
Index

ADC
  SiLabs (lab), 14–17, 19

DAC
  SiLabs (lab), 7–14
debounce
  pseudo (lab), 3
interrupt
  SiLabs (lab), 2–7
interrupt priority, SiLabs (lab), 6
justification
  left/right, DAC, SiLabs (lab), 7
priority
  interrupt, natural (lab), 6