University of British Columbia
Electrical and Computer Engineering
ELEC291/292

Lab 3: SPI, RS232, Temperature, 32-bit Arithmetic, and Macros.

Dr. Jesús Calviño-Fraga P.Eng.
Department of Electrical and Computer Engineering, UBC
Office: KAIS 3024
E-mail: jesusc@ece.ubc.ca
Phone: (604)-827-5387

January 20, 2017

About Lab #3

- There are two versions of the lab (pick one):
  - Using Matlab. Unfortunately Matlab is not free but you may have it from another course. Installed in the lab computers.
  - Using Python. You can download Python for free. The version I use is WinPython (http://winpython.github.io/)
  - Alternatively you can download Python from this link and uncompress it to some folder:

  http://courses.ece.ubc.ca/281/2017/WinPython-64bit-3.5.1.2.zip
- For this lab you need to use the serial port, the MCP3008 SPI ADC, Python/Matlab, and the temperature sensor LM335. Several examples are provided in Connect.
- Since some math is required to convert the ADC value to temperature, a library of 32-bit unsigned integer arithmetic and conversion routines is provided.

Objectives

- Use SPI devices with the AT89LP52 microcontroller.
- Use the serial port to connect the AT89LP52 to a computer and interchange information.
- Measure temperature with the LM335.
- Perform 32-bit unsigned arithmetic using a library in assembly language.

The Serial Peripheral Interface (SPI)

- Used to add multiple external I/O devices or functions to the microcontroller.
- SPI devices are **extremely** simple to connect to most modern microcontrollers/microcomputers.
- SPI devices are usually smaller and cheaper than their parallel bus equivalents.
- Many SPI devices available. Go to Digikey/Mouser and search for 'SPI'.
- SPI article in Wikipedia is excellent!

Copyright © 2009-2016, Jesus Calvino-Fraga. Not to be copied, used, or revised without explicit written permission from the copyright owner.

SPI Synchronous Data Communication Data Format

- Data length can be any size but optimized for 8 bits on most 8-bit microcontrollers.
- Clock is transmitted in a separate wire. Both the polarity of the clock and its phase are selectable.
- Data and clock are transmitted at the same time. Data is also transmitted and received at the same time!
- Most microcontrollers, including the 8051, can be setup as master or slaves to implement master/slave networks.

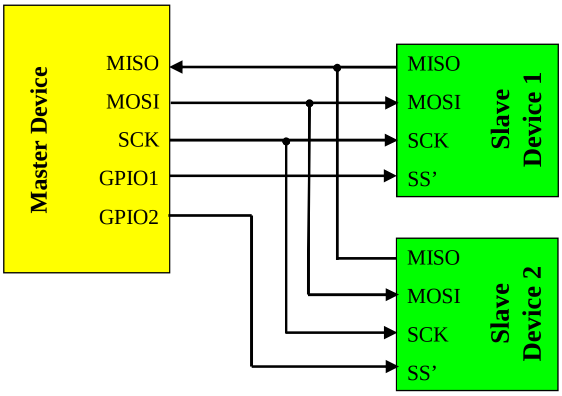
Standard SPI Signals

- Master in / Slave out (MISO)
- Master out / Slave in (MOSI)
- Serial Clock (SCK)
- Slave select (SS')

SPI Wiring modes

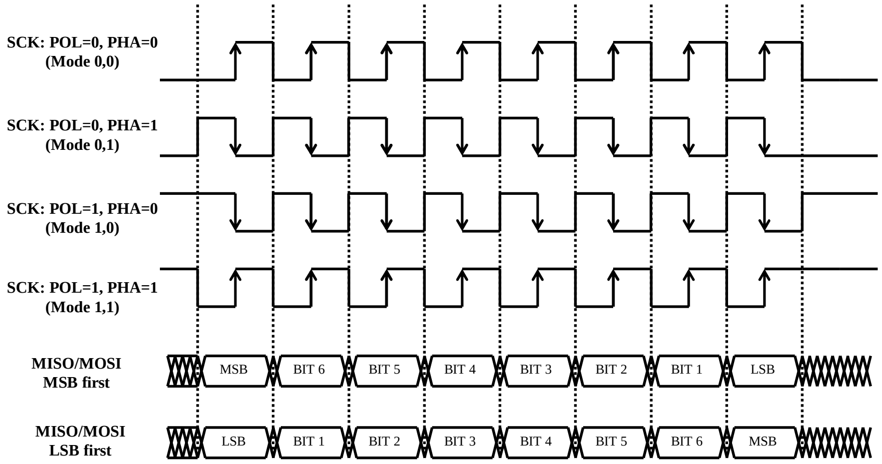
- Master: The device initiates and controls the flow of data. Can be configured as:
  - Multi-master.
  - Three wire single master.
  - Four (or more) wire master.
  - Four wire master with daisy-chained slaves.
- Slave: The device needs to be selected, and the clock is provided to it.

SPI Wiring

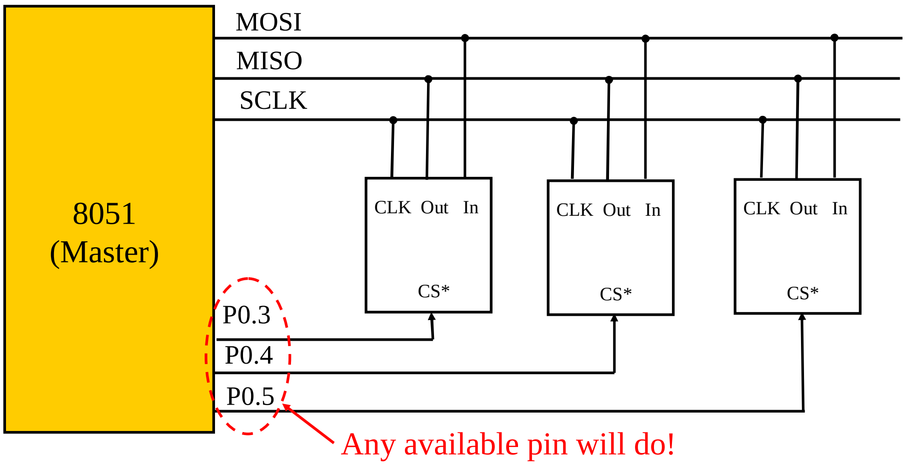


4-Wire Single Master and 4-Wire Multiple Slave Mode Connection Diagram. This is the most frequent configuration.

SPI Timing



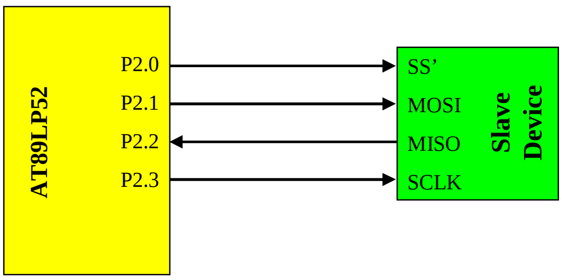
Connecting Devices to the SPI port



Bit Bang SPI

- Many modern microcontrollers have one or more built-in SPI controllers. This includes almost all Freescale (formerly Motorola), PICs from Microchip, 8051s derivatives from Silabs, Atmel, Dallas/Maxim, and NXP (formerly Philips), and ARMs from Atmel, NXP, and many others.
- Occasionally you may need to connect a SPI device to a microcontroller without an SPI controller. In this case we can **Bit Bang** SPI signals using GPIO pins!
- The AT89LP52 has a built-in SPI controller but it is shared with the serial port SFRs. Also, I used the SPI pins to In-System-Program (ISP) the µC, so… Bit Bang SPI it is!

Bit Bang SPI



GPIO pins "arbitrarily" chosen… (they are actually the closest pins to the MCP3008 as you'll see in the next slides)

Bit Bang SPI in Mode (0,0)

```
CE_ADC    EQU  P2.0
MY_MOSI   EQU  P2.1
MY_MISO   EQU  P2.2
MY_SCLK   EQU  P2.3

DO_SPI:
    clr MY_SCLK             ; Mode 0,0 default
    clr CE_ADC              ; Enable device (active low)
    mov R1, #0              ; Received byte stored in R1
    mov R2, #8              ; Loop counter (8-bits)
DO_SPI_LOOP:
    mov a, R0               ; Byte to write is in R0
    rlc a                   ; Carry flag has bit to write
    mov R0, a
    mov MY_MOSI, c
    setb MY_SCLK            ; Transmit
    mov c, MY_MISO          ; Read received bit
    mov a, R1               ; Save received bit in R1
    rlc a
    mov R1, a
    clr MY_SCLK
    djnz R2, DO_SPI_LOOP
    setb CE_ADC             ; Disable device
    ret
```


Bit Bang SPI: more general version

The caller must enable the slave device before calling this subroutine, and disable it after.

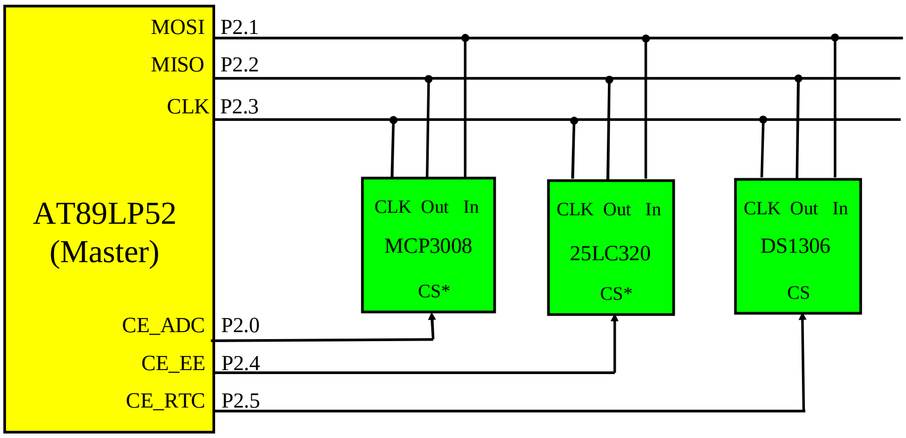
```
MY_MOSI   EQU  P2.1
MY_MISO   EQU  P2.2
MY_SCLK   EQU  P2.3

INI_SPI:
    setb MY_MISO            ; Make MISO an input pin
    clr MY_SCLK             ; Mode 0,0 default
    ret

DO_SPI_G:
    mov R1, #0              ; Received byte stored in R1
    mov R2, #8              ; Loop counter (8-bits)
DO_SPI_G_LOOP:
    mov a, R0               ; Byte to write is in R0
    rlc a                   ; Carry flag has bit to write
    mov R0, a
    mov MY_MOSI, c
    setb MY_SCLK            ; Transmit
    mov c, MY_MISO          ; Read received bit
    mov a, R1               ; Save received bit in R1
    rlc a
    mov R1, a
    clr MY_SCLK
    djnz R2, DO_SPI_G_LOOP
    ret
```

Mode (0,0)

Example: Connecting the MCP3008, 25LC320, and DS1306 to the AT89LP52



Examples: Connecting the MCP3008, 25LC320, and DS1306 to the AT89LP52

At the initialization of main code we must put all enables into the inactive state because the devices are all connected to the same SPI bus:

```
setb CE_ADC
setb CE_EE
clr  CE_RTC ; RTC CE is active high
```

Example: Using the MCP3008 10-bit, 8-channel ADC

Functional Block Diagram

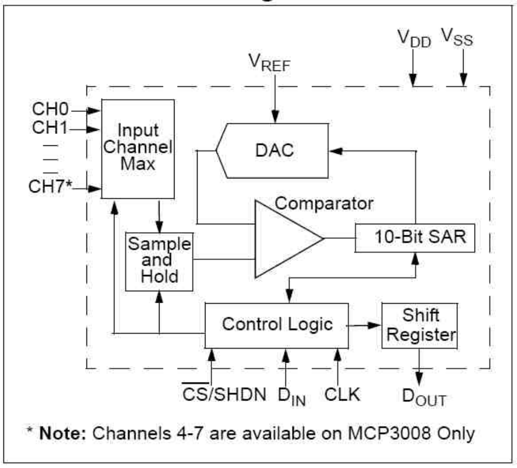
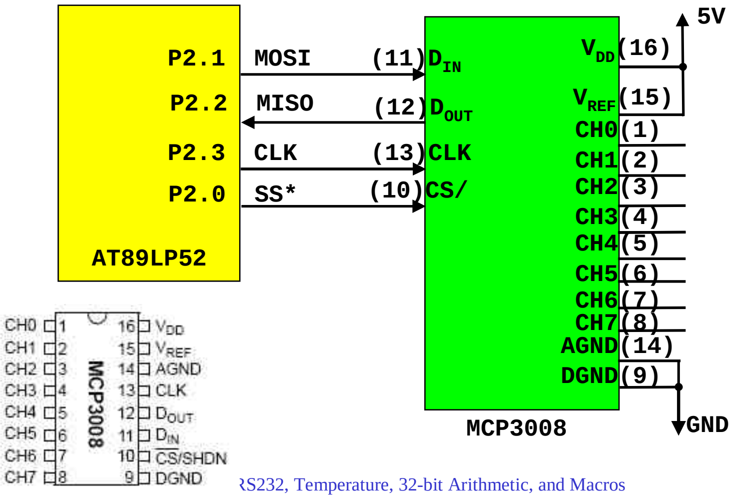


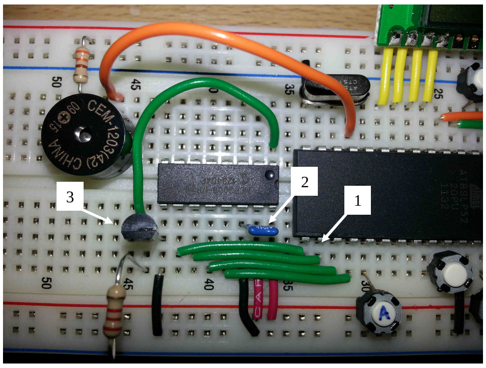
Figure taken From Microchip's MCP3008 datasheet

Example: Using the MCP3008 10-bit, 8-channel ADC



VREF=5V ideally we should use a good voltage reference!

Wiring the Circuit (the easy way)



1. P2.0 (pin 20) to P2.3 (pin 23) connections to ADC.
2. Decoupling cap (0.1µF near V_{DD} and V_{REF})
3. LM335 (un-adjusted) connected to channel 0.

Example: Using the MCP3008 10-bit, 8-channel ADC

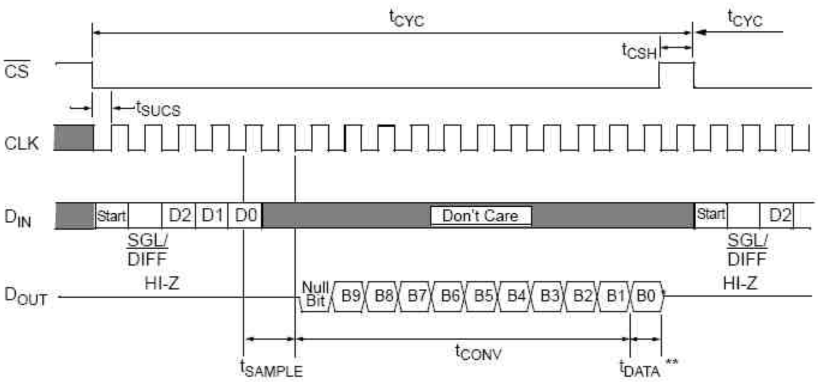


Figure taken from Microchip's MCP3008 datasheet

Example: Using the MCP3008 10-bit, 8-channel ADC

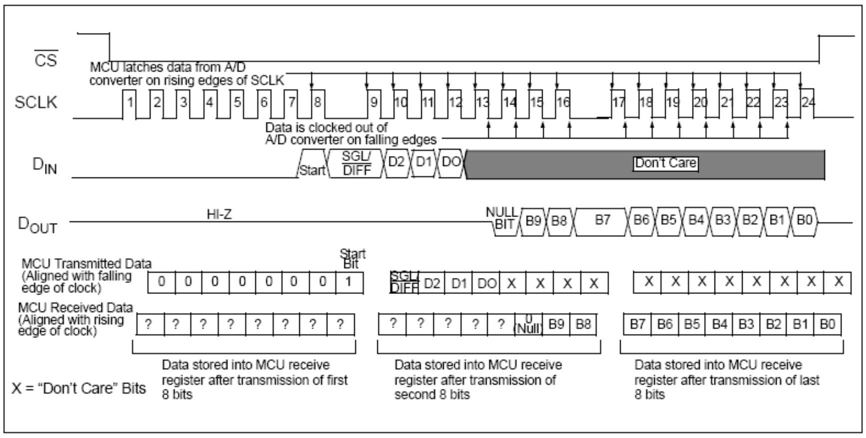


Figure taken from Microchip's MCP3008 datasheet

FIGURE 6-1: *SPI Communication with the MCP3004/3008 using 8-bit segments (Mode 0,0: SCLK idles low).*

Example: Using the MCP3008 10-bit, 8-channel ADC

```
    lcall INIT_SPI

Forever:
    clr CE_ADC
    mov R0, #00000001B ; Start bit:1
    lcall DO_SPI_G

    mov R0, #10000000B ; Single ended, read channel 0
    lcall DO_SPI_G
    mov a, R1          ; R1 contains bits 8 and 9
    anl a, #00000011B  ; We need only the two least significant bits
    mov Result+1, a    ; Save result high.

    mov R0, #55H       ; It doesn't matter what we transmit...
    lcall DO_SPI_G
    mov Result, R1     ; R1 contains bits 0 to 7.  Save result low.
    setb CE_ADC
    lcall Delay

    lcall Do_Something_With_Result
    sjmp Forever
```

```
DSEG at 30H
.
Result: ds 2
.
```

Warning: not a complete program!

The Serial Port

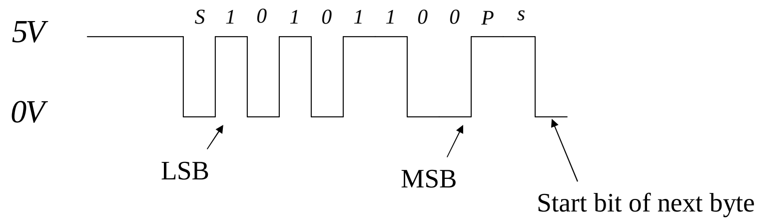
- The AT89LP52 as well as most popular microcontrollers have a serial port.
- The serial port uses the RS-232 communication standard. It was introduced in 1962!
- Perhaps the easiest way to communicate between a microcontroller and a computer!
- Unlike SPI, RS-232 is asynchronous: the clock is not shared between the processors.

Asynchronous Data Communication Data Format

- A start bit used to synchronize the data. ‘0’ or space.
- 5 to 8 data bits. For the standard 8051 the number of data bits is usually 8.
- Optional parity bit. Set or reset so that the number of ones transmitted is either odd or even. For the standard 8051 the parity is set to none by default.
- One, one and half, or two stop bits. Always ‘1’ or mark. For the standard 8051 is set to one stop bit.

Asynchronous Data Communication Data Format

- For example, transmit “00110101” using 8 bits, odd parity, one stop bit:



Baud Rate

$$BR = \frac{1}{t_{bit}}$$ Unit is ‘baud’

- Standard baud rates are: 110, 300, 600, 1200, 4800, 9600, 14400, 19200, 38400, and so on…
- The 8051/8052 with the correct crystal (For example 22.1184 MHz) can generate all the standard baud rates up to 115200 baud!

Serial port in the AT89LP52

- To use the serial port in the AT89LP52:
  - Configure the baud rate using either timer 1 or timer 2. For example if the microcontroller is running at 22.1184 MHz:

```
; Configure serial port and baud rate
clr TR1 ; Disable timer 1
anl TMOD, #0x0f ; Mask the bits for timer 1
orl TMOD, #0x20 ; Set timer 1 in 8-bit auto reload mode
orl PCON, #80H ; Set SMOD to 1
mov TH1, #244 ; for 115200 baud
mov TL1, #244
setb TR1 ; Enable timer 1
mov SCON, #52H
```

  - Configure the serial port mode using SFR SCON.
  - Transmit and receive using SFR SBUF.

Baud rate setup

- Using timer 1 with the AT89LP52 in fast mode:
  - Configure timer 1 in auto-reload mode
  - Load TH1 with the baud rate divider:

$$TH1 = 256 - \frac{2^{SMOD} \times f_{osc}}{32 \times baud}$$

$$TH1 = 256 - \frac{2^{1} \times 22.1184MHz}{32 \times 115200} = 244$$

SMOD is bit 7 of SFR PCON (@ 87H)

Baud rate setup

- Using timer 2 in fast mode:
  - Configure timer 2 in auto-reload mode
  - Load RCAP2H and RCAP2L with the baud rate divider:

$$[RCAP2H,RCAP2L]=65536-\frac{f_{osc}}{16\times baud}$$

$$[RCAP2H,RCAP2L]=65536-\frac{22.1184MHz}{16\times 115200}=65524$$

Configure the serial port

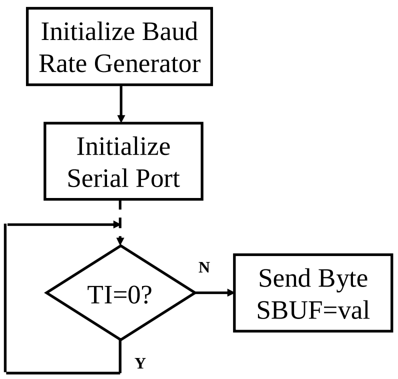
- The serial port in the 8051 has four operating modes. For RS-232 communications (same as personal computers) configure the serial port in mode 1.
- Use register SCON. The 8051 microcontroller serial port in 8-bit mode can be configured only for 8 data bits, no parity, one stop bit:
  - mov SCON, #52H; ; Mode 1, REN=1, TI=1

SCON SFR (Address 98H)

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

- RI: If this bit is set, there is a newly received byte in register SBUF.
- TI: If this bit is set, the transmit buffer is empty. Writing a byte to SBUF will initiate transmission.
- RB8, TB8: The 9th bit in 9-bit UART mode.
- REN: Setting this bit to one enables serial reception.
- SM0, SM1: Configures serial port mode. For now set it to [0, 1], 8-bit UART
- SM2: Enables multiprocessor communication.

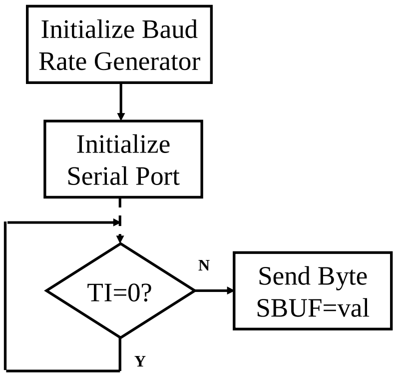
Serial Transmission Using Timer 1




```
InitSerialPort:
; Configure serial port and baud rate
    clr TR1 ; Disable timer 1
    anl TMOD, #0x0f ; Clear bits of timer 1
    orl TMOD, #020H ; timer 1 as 8-bit auto r
    mov TH1, #244 ; for 115200 baud @22.11MHz
    mov TL1, #244
    orl PCON, #80H ; Set SMOD to 1
    setb TR1 ; Enable timer 1
    mov SCON, #52H
    ret

putchar:
    jnb TI, putchar
    clr TI
    mov SBUF, a
    ret
```

Serial Transmission Using Timer 1 (cleaner version)




```
FREQ   EQU 22118400
BAUD   EQU 115200
T1LOAD EQU 256-(FREQ/(16*BAUD))

InitSerialPort:
; Configure serial port and baud rate
    clr TR1 ; Disable timer 1
    anl TMOD, #0x0f ; Clear bits of timer 1
    orl TMOD, #020H ; timer 1 as 8-bit auto r
    mov TH1, #T1LOAD
    mov TL1, #T1LOAD
    orl PCON, #80H ; Set SMOD to 1
    setb TR1 ; Enable timer 1
    mov SCON, #52H
    ret

putchar:
    jnb TI, putchar
    clr TI
    mov SBUF, a
    ret
```


Example: Sending a String

```
$MODLP52
org 0000H
   ljmp MainProgram

FREQ   EQU 22118400
BAUD   EQU 115200
T1LOAD EQU 0x100-(FREQ/(16*BAUD))

InitSerialPort:
; Debounce the reset button!
    mov R1, #222
    mov R0, #166
    djnz R0, $   ; 3 cycles=22.51us
    djnz R1, $-4 ; 22.51us*222=4.998ms
; Configure serial port and baud rate
    clr TR1 ; Disable timer 1
    anl TMOD, #0x0f
    orl TMOD, #020H
    mov TH1, #T1LOAD
    mov TL1, #T1LOAD
    orl PCON, #80H ; Set SMOD to 1
    setb TR1 ; Enable timer 1
    mov SCON, #52H
    ret
```

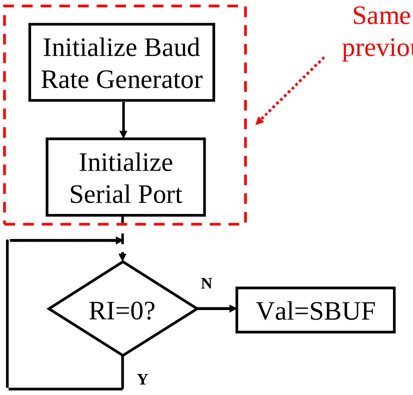
```
putchar:
    JNB TI, putchar
    CLR TI
    MOV SBUF, a
    RET

SendString:
    CLR A
    MOVC A, @A+DPTR
    JZ SSDone
    LCALL putchar
    INC DPTR
    SJMP SendString
SSDone:
    ret

Hello:  DB  'Hello, World!',0AH, 0DH, 0
MainProgram:
    MOV SP, #7FH
    MOV PMOD, #0
    LCALL InitSerialPort
    MOV DPTR, #Hello
    LCALL SendString

    SJMP $
END
```

Serial Reception



Same code as previous slides.

```
getchar:
    jnb RI, getchar
    clr RI
    mov a, SBUF
    ret
```


Reading Strings

```
DSEG at 30H
buffer: ds 30

CSEG
GeString:
    mov R0, #buffer
GSLoop:
    lcall getchar
    push acc
    clr c
    subb a, #10H
    pop acc
    jc GSDone
    MOV @R0, A
    inc R0
    SJMP GSLoop
GSDone:
    clr a
    mov @R0, a
    ret
```

The trick with receiving strings is to know when to stop!

On Windows: line ends with CR+LF, or 0DH, 0AH.

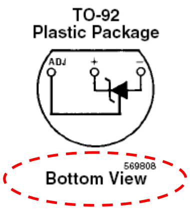
On Linux/Unix: line ends with LF, or 0AH.

On Macs: Nowadays, same as Linux/Unix.

Any of these is less than 10H!

Warning: buffer overrun is possible.

LM335 Temperature Sensor



For the LM335:

+10mV/°K, -40°C<t<100°C

For the MCP3008 :

ADC: 10-bit, 0.0V<V_{in}<5V

Un-calibrated temperature error: 2 to 6℃

$$-40^0 C = (273-40)^0 K = 233^0 K \rightarrow 2.33V$$
$$+100^0 C = (273+100)^0 K = 373^0 K \rightarrow 3.73V$$

From the datasheet: “Included on the LM335 chip is an easy method of calibrating the device for higher accuracies. A pot connected across the LM135 with the arm tied to the adjustment terminal allows a 1-point calibration of the sensor that corrects for inaccuracy over the full temperature range.”

LM335 Temperature Sensor

Figure 15. Basic Temperature Sensor

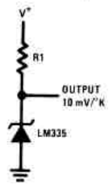
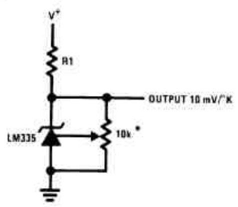


Figure 16. Calibrated Sensor

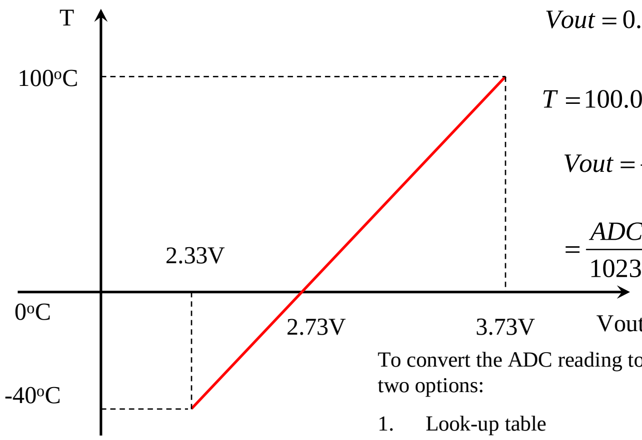


*Calibrate for 2.982V at 25°C

R1=2kΩ or 2.2kΩ, if you check the datasheet, most of the specs are @ 1 ma.

V^+=5V

LM335 Transfer Function



$$Vout = 0.01\frac{V}{^{o}C}\times T + 2.73V$$

$$T = 100.0(Vout - 2.73V)\frac{^{o}C}{V}$$

$$Vout = \frac{ADC}{\left(2^{10}-1\right)}\times V_{REF}$$

$$= \frac{ADC}{1023}\times 5.00V$$

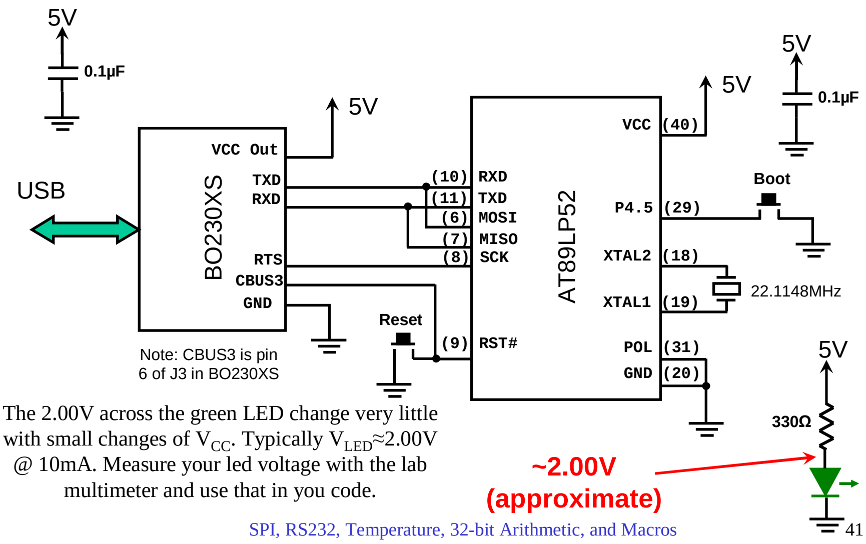
To convert the ADC reading to temperature you have two options:

1. Look-up table
2. Arithmetic

Voltage Reference

- Ideally we should use a good voltage reference for the "V_{REF}" pin of the ADC. For example the LM4040AIZ-5.0: 2.51$, 0.1%, 100ppm/°C. Unfortunately, to get 5.000V we need an input voltage of about 7V.
- We keep V_{CC} connected to V_{REF}, but connect a lower reference voltage (LM4040AIZ-2.5 for example) to one of the inputs of the ADC. With that we can easily determine the actual value of V_{CC}.
- Good references are expensive. So, things we can do instead for this course:
  - Measure V_{CC} and store the value in your code. Works fine, but if V_{CC} changes, all measurements will be converted to voltage incorrectly. Different computers and even different USB ports in the same computer may give different values of V_{CC}: nominal voltage 5 V ± 5% (4.75V<V_{CC}<5.25V).
  - Use an constant voltage to measure V_{CC}. For example the forward voltage drop across a LED:

Constant Voltage



The 2.00V across the green LED change very little with small changes of V_{CC}. Typically $V_{LED} \approx 2.00V$ @ 10mA. Measure your led voltage with the lab multimeter and use that in you code.

I_{LED} vs. V_{LED}

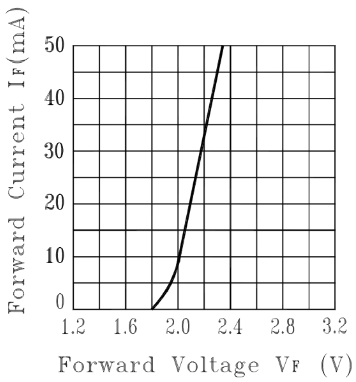
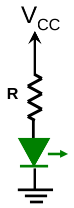


Fig.2 Forward Current vs. Forward Voltage

http://media.digikey.com/pdf/Data%20Sheets/Lite-On%20PDFs/LTL-4233.pdf

Constant Voltage from LED



LED #	R in Ω	V_{LED} in V @ V_{CC}=4.75V	V_{LED} in V @ V_{CC}=5.00V	V_{LED} in V @ V_{CC}=5.25V
1	326.4	2.065	2.078	2.092
2	328.1	2.055	2.067	2.079
3	329.3	2.058	2.071	2.084
4	328.7	2.060	2.073	2.085
5	329.0	2.065	2.078	2.091
6	328.9	2.053	2.065	2.077
7	328.8	2.056	2.068	2.081
8	328.1	2.063	2.076	2.089
9	327.9	2.058	2.070	2.083
10	329.7	2.059	2.071	2.084
Ave.	328.5	2.0592	2.0717	2.0845

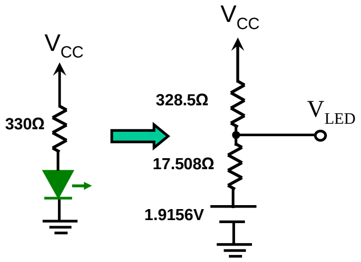
Constant Voltage from LED

From the tests above:

V_{LED}=2.0717V @ 8.91448mA, r_{LED}=17.508Ω

Model LED using voltage drop + resistance:

V_0 = 2.0717V - 8.91448mA * 17.508Ω = 1.9156V



V_{CC}	V_{LED}
4.75V	2.059V
5.00V	2.072V
5.25V	2.084V

V_{LED}≈2.072±0.6%

Finding V_{CC} using V_{LED}: code

```
VLED EQU 207 ; Measured (with multimeter) LED voltage x 100
DSEG ; Tell assembler we are about to define variables
Vcc: ds 2 ; 16-bits are enough to store VCC x 100 (max is 525)
CSEG ; Tell assembler we are about to input code
; Measure the LED voltage.  Used as reference to find VCC.
mov b, #7 ; VLED connected to input ‘7’ of MCP3008 ADC
lcall Read_ADC_Channel ; Read voltage, returns 10-bits in [R6-R7]
mov y+3, #0 ; Load 32-bit “y” with value from ADC
mov y+2, #0
mov y+1, R7
mov y+0, R6
load_x(VLED*1023) ; Macro to load “x” with constant
lcall div32 ; Divide “x” by “y”, the result is VCC in “x”
mov Vcc+1, x+1 ; Save calculated VCC high byte
mov Vcc+0, x+0 ; Save calculated VCC low byte
```


Math32 Library

- ***Math32.asm*** has the following functions:
  - **Hex2bcd**: Converts the 32-bit binary number in 'x' to a 10-digit packed BCD in 'bcd' using the double-dabble algorithm.
  - **Bcd2hex**: Converts the 10-digit packed BCD in 'bcd' to a 32-bit binary number in 'x'.
  - **add32**: x = x + y
  - **sub32**: x = x − y
  - **mul32**: x = x * y
  - **div32**: x = x / y
  - **x_lt_y**: mf=1 if x < y (mf is a bit)
  - **x_gt_y**: mf=1 if x > y
  - **x_eq_y**: mf=1 if x = y
  - **x_gteq_y**: mf=1 if x >= y
  - **x_lteq_y**: mf=1 if x <= y
- ***Math32.asm*** has the following macros:
  - **Load_X**: load x with a 32-bit constant
  - **Load_Y**: load y with a 32-bit constant

Math32 Test Program

- ***Mathtest.asm*** shows how to:
  - Define the x, y, bcd, and mf variables.
  - Include the math32 library in your program.
  - Use the Load_X and Load_X macros.
  - Convert a binary to BCD using bin2bcd and display it using the LCD.
  - Use the add32, sub32, mul32, and div32 functions.
  - Evaluate a formula using only integers.

Warning: Using Integer Arithmetic

$$Vout = \frac{ADC}{\left(2^{10}-1\right)} \times V_{CC} = \frac{ADC}{1023} \times 500$$

Suppose ADC=612, compute Vout

$$Vout = \frac{612}{1023} \times 500$$

Wrong!

Vout=(612/1023) x 500 = (0) x 500 =0

$$\boxed{\text{Vout=(612 x 500) / 1023 = (306000) /1023 = 299}}$$

Assembly Macros

- "A macro is a name assigned to one or more assembly statements or directives. Macros are used to include the same sequence of instructions in several places. This sequence of instructions may operate with different parameters, as indicated by the programmer."
- The MAC directive is used to define the start of a macro. A macro is a segment of instructions that is enclosed between the directives MAC and ENDMAC. The format of a macro is as follows:

```
name MAC ; comment
.
.
.
ENDMAC
```

- You can use macros to add some "flavour" of high level language to your assembly program.

Macro Example: Duplicated code

```
Loop:
    mov P3, #0 ; all bits zero!
    lcall mydelay

    setb P3.7
    lcall mydelay
    jnb P2.4, L1a
    clr P3.7
L1a:
    setb P3.6
    lcall mydelay
    jnb P2.4, L2a
    clr P3.6
L2a:
    setb P3.5
    lcall mydelay
    jnb P2.4, L3a
    clr P3.5
L3a:
.
.
[more code here]
```

Similar code for each bit, good candidate for a macro:

```
ADC_bit MAC
    ;ADC_bit(%0, %1, %2)
    setb %0
    lcall mydelay
    jnb %1, %2
    clr %0
%2:
ENDMAC
```

Macro Example: first try

```
ADC_bit MAC
    ;ADC_bit(%0, %1, %2)
    setb %0
    lcall mydelay
    jnb %1, %2
    clr %0
%2:
ENDMAC
.
.
myprogram:
.
.
.
```

```
Loop:
    mov P3, #0 ; all bits zero!
    lcall mydelay

    ADC_bit(P3.7, P2.4, L1a)
    ADC_bit(P3.6, P2.4, L2a)
    ADC_bit(P3.5, P2.4, L3a)
    ADC_bit(P3.4, P2.4, L4a)
    ADC_bit(P3.3, P2.4, L5a)
    ADC_bit(P3.2, P2.4, L6a)
    ADC_bit(P3.1, P2.4, L7a)
    ADC_bit(P3.0, P2.4, L8a)

    mov val, P3 ; Save the result

    ljmp Loop
```


Macro Example: better macro

```
ADC_bit MAC
    ;ADC_bit(%0, %1)
    setb %0
    lcall mydelay
    jnb %1, skip%M
    clr %0
skip%M:
ENDMAC
.
.
myprogram:
.
.
.
```

%M: Macro counter

```
Loop:
    mov P3, #0 ; all bits zero!
    lcall mydelay

    ADC_bit(P3.7, P2.4)
    ADC_bit(P3.6, P2.4)
    ADC_bit(P3.5, P2.4)
    ADC_bit(P3.4, P2.4)
    ADC_bit(P3.3, P2.4)
    ADC_bit(P3.2, P2.4)
    ADC_bit(P3.1, P2.4)
    ADC_bit(P3.0, P2.4)

    mov val, P3 ; Save the result

    ljmp Loop
```

Check the .lst file to see how the macro expanded:

Macros after expansion:

```
001E            36   Loop:
001E 75B000     37       mov P3, #0 ; all bits zero!
0021 120003     38       lcall mydelay
0024            39
0024            40       ;ADC_bit(P3.7, P2.4)
0024 D2B7       40       setb P3.7
0026 120003     40       lcall mydelay
0029 30A402     40       jnb P2.4, skip1
002C C2B7       40       clr P3.7
002E            40   skip1:
002E            41       ;ADC_bit(P3.6, P2.4)
002E D2B6       41       setb P3.6
0030 120003     41       lcall mydelay
0033 30A402     41       jnb P2.4, skip2
0036 C2B6       41       clr P3.6
0038            41   skip2:
0038            42       ;ADC_bit(P3.5, P2.4)
0038 D2B5       42       setb P3.5
003A 120003     42       lcall mydelay
003D 30A402     42       jnb P2.4, skip3
0040 C2B5       42       clr P3.5
0042            42   skip3:
.
.
.
```


Calling Subroutines Using Macros

Different names

```
Read_ADC_Channel MAC
    mov b, #%0
    lcall _Read_ADC_Channel
ENDMAC

_Read_ADC_Channel:
    clr CE_ADC
    mov R0, #00000001B ; Start bit:1
    lcall DO_SPI_G
    mov a, b
    swap a
    anl a, #0F0H
    setb acc.7 ; Single mode (bit 7).
    mov R0, a
    lcall DO_SPI_G
    mov a, R1 ; R1 contains bits 8 and 9
    anl a, #00000011B  ; We need only the two least significant bits
    mov R7, a ; Save result high.
    mov R0, #55H ; It doesn't matter what we transmit...
    lcall DO_SPI_G
    mov R6, R1 ; R1 contains bits 0 to 7.  Save result low.
    setb CE_ADC
    ret
```

We use this macro in our code to read an ADC channel like this:

`Read_ADC_Channel(6)`

Finding V_{CC} using V_{LED}: code

```
VLED EQU 207 ; Measured (with multimeter) LED voltage x 100
DSEG ; Tell assembler we are about to define variables
Vcc: ds 2 ; 16-bits are enough to store VCC x 100 (max is 525)
CSEG ; Tell assembler we are about to input code
; Measure the LED voltage.  Used as reference to find VCC.
Read_ADC_Channel(6) ; Read voltage, returns 10-bits in [R6-R7]
mov y+3, #0 ; Load 32-bit “y” with value from ADC
mov y+2, #0
mov y+1, R7
mov y+0, R6
load_x(VLED*1023) ; Macro to load “x” with constant
lcall div32 ; Divide “x” by “y”, the result is VCC in “x”
mov Vcc+1, x+1 ; Save calculated VCC high byte
mov Vcc+0, x+0 ; Save calculated VCC low byte
```


Final Remarks

- Lab 3 is an excellent starting point for project 1!
- Macros take a lot of pain away from programming in assembly. Use them!