University of British Columbia
Electrical and Computer Engineering
ELEC291/292

Lab 4: RS232, Temperature, C Programming, Python, Analog Input Protection

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

February 26, 2021

Objectives

- Use the serial port to connect the 8051 to a computer and interchange information.
- Measure temperature with the LM335.
- C and Python programming. Examples.
- Protecting the I/O pins with diodes and transistors.

For this lab you can work with a partner.

The Serial Port

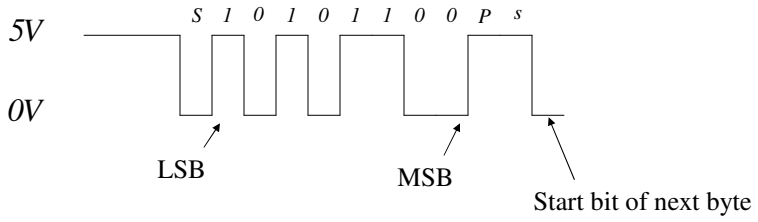
- The 8051 (both the EFM8LB12 and SoC-8052) as well as most popular microcontrollers have one or more serial ports.
- The serial port uses the RS-232 communication standard. It was introduced in 1962!
- Perhaps the easiest way to communicate between a microcontroller and a computer!
- Unlike SPI, RS-232 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 or 11.0592MHz) can generate all the standard baud rates up to 115200 baud! A crystal like that is called a "magic crystal". (Don't ask me why, if you Google "magic crystal" what you get is a bunch of new age BS)

"Magic Crystal" Frequencies

https://en.wikipedia.org/wiki/Crystal_oscillator_frequencies

Frequency (MHz)	comm	UART	A/V	RTC	Primary uses
0.032				✓	Real-time clocks, watches; allows binary division to 1 kHz signal (2^5×1 kHz).
0.032768				✓	Real-time clocks, quartz watches and clocks; allows binary division to 1 Hz signal (2^{15}×1 Hz); also low-speed low-power microcontrollers. Very common. Available as TCXO.[1]
					/180/150/110 kHz to both sides from the 10.7 MHz frequency)
11.0592		115200			UART clock (6×1.8432 MHz); allows integer division to common baud rates (96×115200 baud or 96×96×1,200 baud); common clock for Intel 8051 microprocessors[18]
					Used in CD-DA systems and CD-ROM drives; allows binary

24.5MHz, 33.3333MHz, 72MHz: not a "Magic" frequency, but error is acceptable.

Baud rate setup for the EFM8LB1

- Using timer 1 with the EFM8LB1:
  - Configure timer 1 in auto-reload mode
  - Load TH1 with the baud rate divider, for the EFM8 running at 24.5MHz. If the clock divider for timer 1 is set to 1:

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

$$TH1 = 256 - \frac{24.5MHz}{2 \times 115200} = 150$$

Serial port in the original 8051

- To use the serial port in the 8051:
  - Configure the baud rate using either timer 1, timer 2, or the baud rate generator. For example if the microcontroller is running at 24.5 MHz in the EFM8LB1, using timer 1:

```
// Configure Uart 0
SCON0 = 0x10;
CKCON0 |= 0b00001000 ; // Timer 1 uses the system clock.
TH1 = 0x100-((SYSCLK/BAUDRATE)/2L);
TL1 = TH1;      // Init Timer 1
TMOD &= ~0xf0;  // TMOD: Timer 1 in 8-bit auto-reload
TMOD |=  0x20;
TR1 = 1; // START Timer1
TI = 1;  // Indicate TX0 ready
```

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

Baud rate setup for the 8051, EFM8LB1, and SoC-8052 using Timer 1

- Using timer 1 with the EFM8LB1:
  - Configure timer 1 in auto-reload mode
  - Load TH1 with the baud rate divider, for the EFM8 running at 72.0MHz. If the clock divider for timer 1 is set to 12:

$$TH1 = 256 - \frac{f_{osc}}{2 \times 12 \times baud}$$

$$TH1 = 256 - \frac{72.0MHz}{2 \times 12 \times 115200} = 230$$

Serial port in the original 8051, EFM8 (clock divider=12) and SoC-8052 using timer 1

- For example if the microcontroller is running at 24.5 MHz in the EFM8LB1, using timer 1:

```
// Configure Uart 0
SCON0 = 0x10;
CKCON0 |= 0b00001000 ; // Timer 1 uses the system clock/12.
TH1 = 0x100-((SYSCLK/BAUDRATE)/(12L*2L));
TL1 = TH1;       // Init Timer 1
TMOD &= ~0xf0;   // TMOD: Timer 1 in 8-bit auto-reload
TMOD |=  0x20;
TR1 = 1; // START Timer1
TI = 1;  // Indicate TX0 ready
```

- Configure the serial port mode using SFR SCON. (SCON0 for EFM8LB1)
- Transmit and receive using SFR SBUF. (SBUF0 for EFM8LB1)

Baud rate setup for original 8051 and SoC-8052 using timer 2

- Using timer 2 (not possible in the EFM8LB1):
  - Configure timer 2 in auto-reload mode
  - Load RCAP2H and RCAP2L with the baud rate divider:

$$[RCAP2H,RCAP2L]=65536-\frac{f_{osc}}{32\times baud}$$

$$[RCAP2H,RCAP2L]=65536-\frac{33.3333MHz}{32\times 115200}=65527$$

Configure the serial port

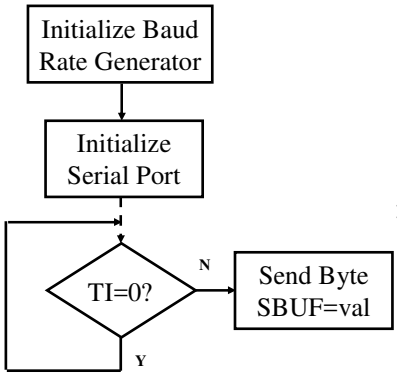
- The serial port in the 8051 has four operating modes. For RS-232 communications (same as personal computers) configure the serial port in mode 1.
- Use register SCON or SCON0. The 8051 microcontroller serial port in 8-bit mode can be configured only for 8 data bits, no parity, one stop bit:
  - mov SCON, #52H; ; Mode 1, REN=1, TI=1

SCON SFR (Address 98H)

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

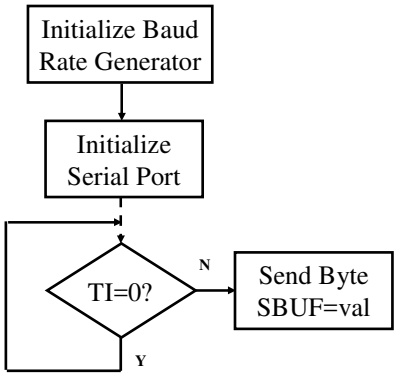
- RI: If this bit is set, there is a newly received byte in register SBUF.
- TI: If this bit is set, the transmit buffer is empty. Writing a byte to SBUF will initiate transmission.
- RB8, TB8: The 9th bit in 9-bit UART mode.
- REN: Setting this bit to one enables serial reception.
- SM0, SM1: Configures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

Serial Transmission




```
InitSerialPort:
    ; Configure serial port and baud rate
    orl PCON,#0x80
    mov SCON,#0x52
    mov BDRCON,#0x00
    mov BRL,#244
    mov BDRCON,#0x1E ; BRR|TBCK|RBCK|SPD
    ret

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


Serial Transmission (cleaner version)



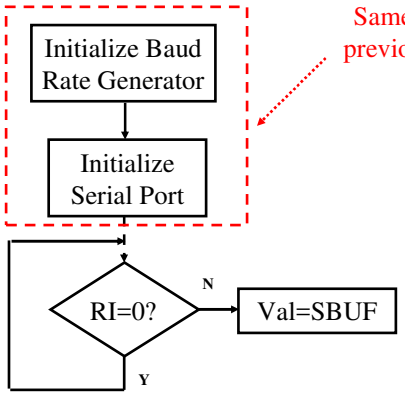

```
FREQ    EQU 22118400
BAUD    EQU 115200
BRG_VAL EQU 256-(FREQ/(16*BAUD))

InitSerialPort:
; Configure serial port and baud rate
    orl PCON,#0x80
    mov SCON,#0x52
    mov BDRCON,#0x00
    mov BRL,#BRG_VAL
    mov BDRCON,#0x1E ; BRR|TBCK|RBCK|SPD
    ret

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

Example: Sending a String

```
$MODLP51
org 0000H
   ljmp MainProgram

FREQ   EQU 22118400
BAUD   EQU 115200
BRG_VAL 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
    orl PCON,#0x80
    mov SCON,#0x52
    mov BDRCON,#0x00
    mov BRL,#BRG_VAL
    mov BDRCON,#0x1E; BRR|TBCK|RBCK|SPD
    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/BSD: 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.

Serial port in C for the 8051

- putchar() and getchar() already defined in the standard libraries.
- printf() uses putchar().
- scanf() uses getchar(). (There is a bug in scanf() that prevents reading to a 'unsigned char' or 'char' variable properly. I don't know how to fix it even though I wrote the function! Lines 369 and 384 of scan_format.c. The work around is to read to an integer variable.)
- Floating point arithmetic supported. Double not supported!
- Don't call scanf() or printf() from inside an interrupt service routine. EVER. Not even when you are using a fancy ARM or MIPS processor.

C Source code EFM8

```c
#include <EFM8LB1.h>
#include <stdio.h>

#define SYSCLK      24500000L  // SYSCLK frequency in Hz
#define BAUDRATE      115200L  // Baud rate of UART in bps

char _c51_external_startup (void)
{
    // Disable Watchdog with key sequence
    SFRPAGE = 0x00;
    WDTCN = 0xDE; //First key
    WDTCN = 0xAD; //Second key

    VDM0CN |= 0x80;
    RSTSRC = 0x02;

    // Use a 24.5MHz clock
    SFRPAGE = 0x00;
    CLKSEL  = 0x00;
    CLKSEL  = 0x00;
    while ((CLKSEL & 0x80) == 0);

    P0MDOUT |= 0x10; // Enable UART0 TX as push-pull output
    XBR0     = 0x01; // Enable UART0 on P0.4(TX) and P0.5(RX)
    XBR1     = 0x00;
    XBR2     = 0x40; // Enable crossbar and weak pull-ups

    // Configure Uart 0
    SCON0 = 0x10;
    CKCON0 |= 0b_0000_1000 ; // Timer 1 uses the system clock.
    TH1 = 0x100-((SYSCLK/BAUDRATE)/2L);
    TL1 = TH1;      // Init Timer1
    TMOD &= ~0xf0;  // TMOD: timer 1 in 8-bit auto-reload
    TMOD |=  0x20;
    TR1 = 1; // START Timer1
    TI = 1;  // Indicate TX0 ready

    return 0;
}
```

```c
// Uses Timer3 to delay <us> micro-seconds.
void Timer3us(unsigned char us)
{
    unsigned char i;               // usec counter

    // The input for Timer 3 is selected as SYSCLK by setting T3ML (bit 6) of CKCON0:
    CKCON0|=0b_0100_0000;

    TMR3RL = (-(SYSCLK)/1000000L); // Set Timer3 to overflow in 1us.
    TMR3 = TMR3RL;                 // Initialize Timer3 for first overflow

    TMR3CN0 = 0x04;                // Start Timer3 and clear overflow flag
    for (i = 0; i < us; i++)       // Count <us> overflows
    {
        while (!(TMR3CN0 & 0x80));  // Wait for overflow
        TMR3CN0 &= ~(0x80);         // Clear overflow indicator
    }
    TMR3CN0 = 0 ;                   // Stop Timer3 and clear overflow flag
}

void WaitXms (unsigned int ms)
{
    unsigned int j;
    for(j=ms; j!=0; j--)
    {
        Timer3us(249);
        Timer3us(249);
        Timer3us(249);
        Timer3us(250);
    }
}
```

```c
void main (void)
{
    WaitXms(500); // Give PuTTY a chance to start.
    printf( "Hello, world!\r\n" );
}
```

Posted on Canvas

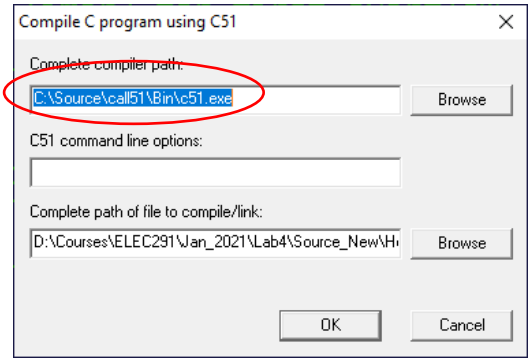
C Source code SoC-8052

```c
#include <stdio.h>
#include <CV_8052.h>

#define CLK 33333333L
#define BAUD 115200L
#define TIMER_2_RELOAD (0x10000L-(CLK/(32L*BAUD)))

void main(void)
{
    setbaud_timer2(TIMER_2_RELOAD);

    LEDRA=0x00;
    LEDRB=0x00;
    printf("Hello, World!\r\n");
}
```

Compiling with C

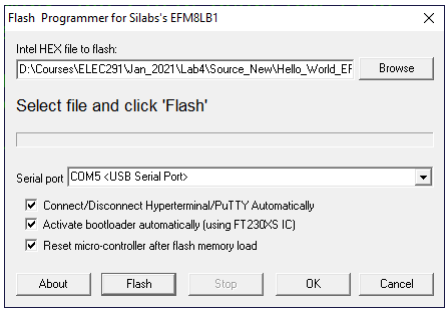
Click here

C compiler



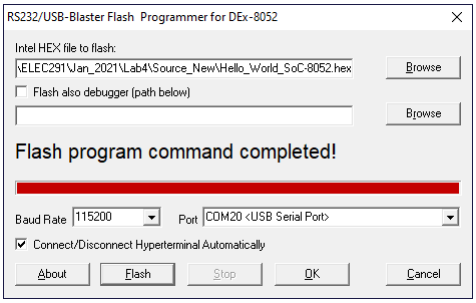
Load as Usual

Loading the HEX file on EFM8



Loading HEX file on SoC-8052

- Two options:
  - Quartus Signal Tap II (as before)
  - If USB to serial present 'DEx 8052':

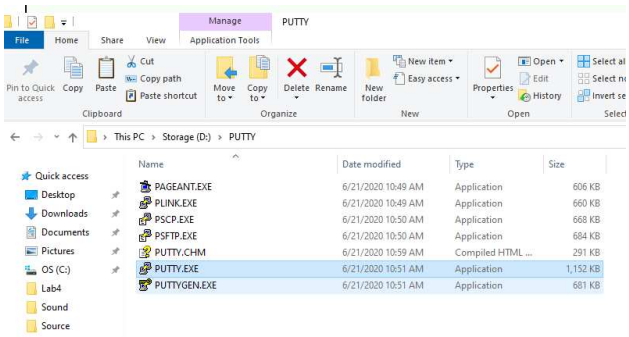


PuTTY

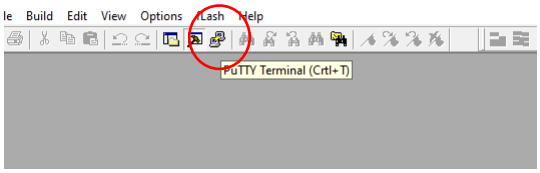
- Download from here:

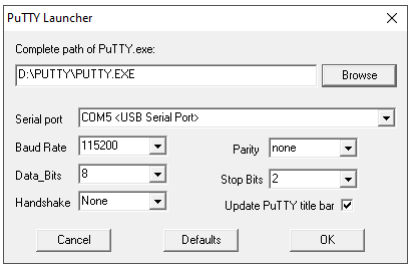
https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

- Install in a folder **without spaces** in the name.



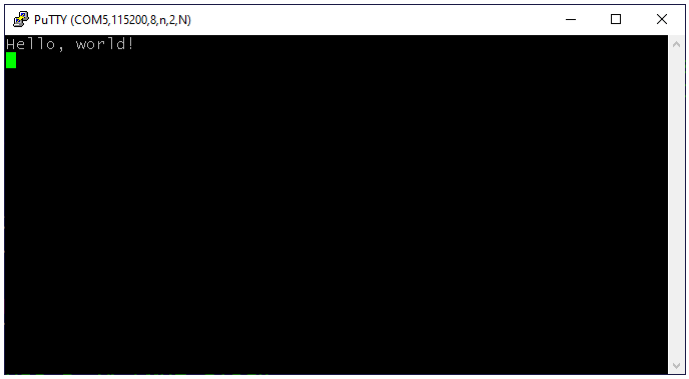
Calling PuTTY from CrossIDE





Running PuTTY

<u>Press the RESET</u> push button on the EFM8 board:



Using the Serial Port with Python

serial_in.py - D:\Courses\ELEC291\Jan_2021\Lab4\Python\se...

File Edit Format Run Options Window Help

```python
import time
import serial

# configure the serial port
ser = serial.Serial(
    port='COM5', # Change as needed
    baudrate=115200,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_TWO,
    bytesize=serial.EIGHTBITS
)
ser.isOpen()

while 1 :
    strin = ser.readline()
    print (strin)
```

Ln: 16 Col: 13

Python 3.8.6 Shell

File Edit Shell Debug Options Window Help

```
Python 3.8.6 (tags/v3.8.6:db45529, Sep 23 2020, 15:52:53) [MSC v.1927 64 bit (AMD
64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======== RESTART: D:\Courses\ELEC291\Jan_2021\Lab4\Python\serial_in.py ========
b'Hello, world!\r\n'
```

Must receive '\n' to accept input

Ln: 5 Col: 0

Using the Serial Port with Python

serial_in.py - D:\Courses\ELEC291\Jan_2021\Lab4\Python\s...

File Edit Format Run Options Window Help

```python
import time
import serial

# configure the serial port
ser = serial.Serial(
    port='COM5', # Change as needed
    baudrate=115200,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_TWO,
    bytesize=serial.EIGHTBITS
)
ser.isOpen()

while 1 :
    strin = ser.readline()
    print (strin.decode('utf-8'))
```

Ln: 17 Col: 0

Different

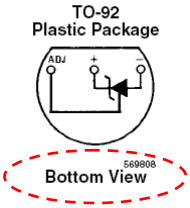
Python 3.8.6 Shell

File Edit Shell Debug Options Window Help

```
Python 3.8.6 (tags/v3.8.6:db45529, Sep 23 2020, 15:52:53) [MSC v.1927 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======== RESTART: D:\Courses\ELEC291\Jan_2021\Lab4\Python\serial_in.py ========
Hello, world!
```

Ln: 5 Col: 0

LM335 Temperature Sensor



For the LM335:

+10mV/°K, -40°C<t<100°C

For the EFM8 :

ADC: 14-bit, 0.0V<V_{in}<3.3V

Un-calibrated temperature error: 2 to 6°C

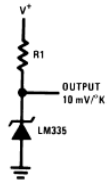
$$-40^0 C = (273-40)K = 233K \rightarrow 2.33V$$

$$+57^0 C = (273+57)K = 330K \rightarrow 3.3V$$

From the datasheet: "Included on the LM335 chip is an easy method of calibrating the device for higher accuracies. A pot connected across the LM335 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."

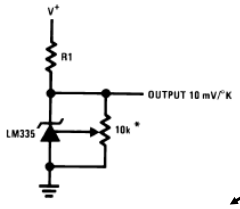
LM335 Temperature Sensor

Figure 15. Basic Temperature Sensor



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

Figure 16. Calibrated Sensor

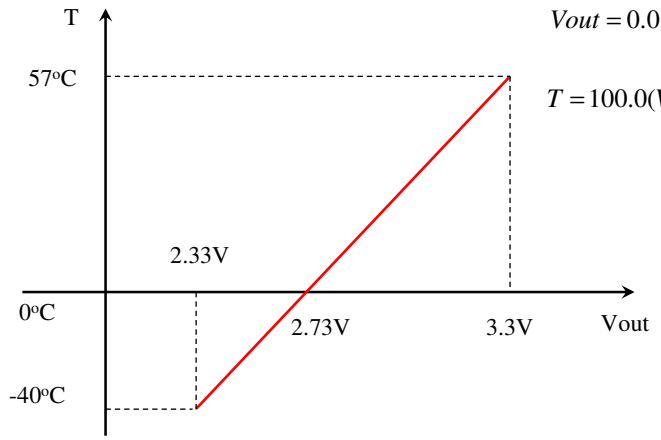


*Calibrate for 2.982V at 25°C

V^+=5V

How do you check if your LM335 is working?

LM335 Transfer Function



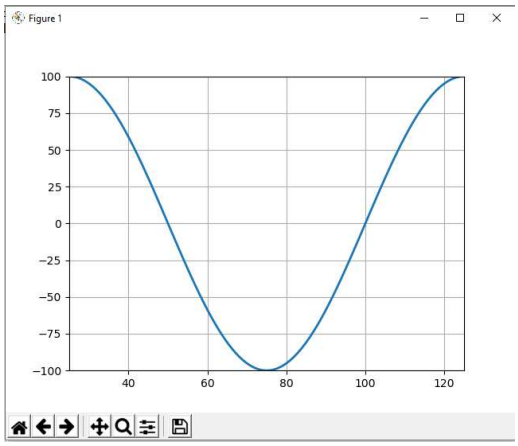
$$Vout = 0.01\frac{V}{^oC} \times T + 2.73V$$

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

Examples Provided

- For the EFM8 board:
  - Hello_World_EFM8.c
  - ADC_EFM8.c
  - EFM8_LCD_4bit.c
- For the SoC-8052:
  - Hello_World_SoC-8052.c
  - SoC-8052-LTC2308_test.c
  - SoC-8052-AD7928_test.c
- Python:
  - stripchart_sinewave.py
  - serial_in.py

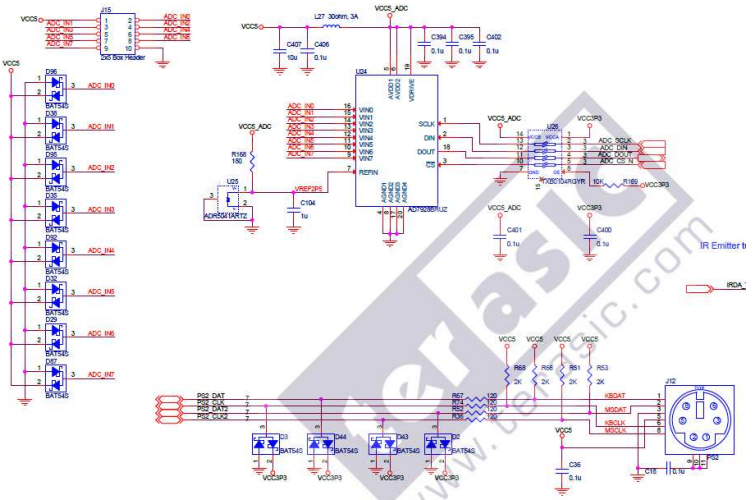
Output of stripchar_sinewave.py



Two Different Versions of DE1-SoC Board!

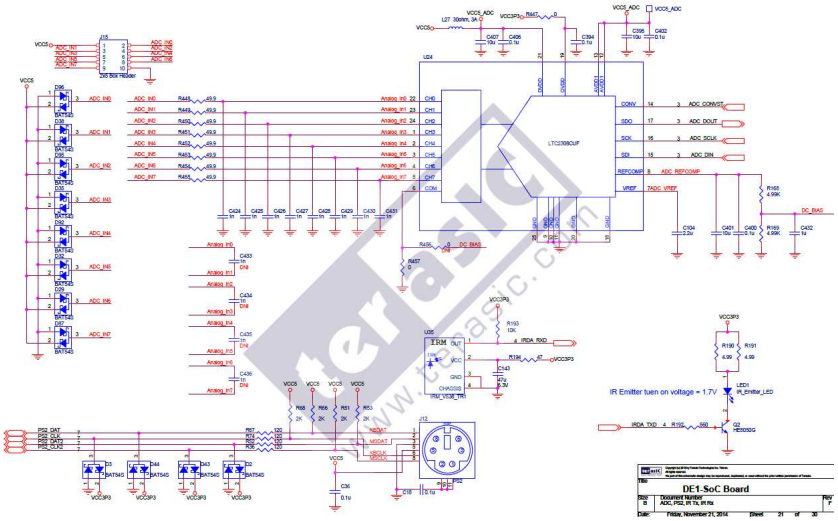
- Revision 4 uses the AD7928 SPI ADC
- Revision 5 uses the LTC2308 SPI ADC
- Examples using the SPI ADC provided for both versions.
- Why the change? Take a look at the schematics:

DE1-SoC Revision 4



SPI, RS232, Temperature, 32-bit Arithmetic, and Macros 37

DE1-SoC Revision 5

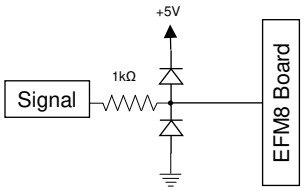


SPI, RS232, Temperature, 32-bit Arithmetic, and Macros 38

Differences

- Revision 4 has the input protection diodes but is missing the protection resistors!
- Revision 5 has the input protection diodes but the protection resistors are in the wrong place!!!
- The EFM8 board has neither protection diodes nor resistors. OUCH!
- Under the current situation (COVID-19) better safe than sorry:

Protecting the Analog Inputs (or any input for that matter!)



Max voltage the EFM8 Board sees: 5.2V
Min voltage the EFM8 Board sees: -0.2V

- Protection diodes should be Schottky diodes, but even a regular diode (1N4148) is way better than nothing!
- The Schottky diode in your kit has part number **MBR150**.
- The datasheet says that the EFM8 is 5V tolerant.
- For the DE1-SoC configured as SoC-8052 a 100Ω to 1kΩ resistor in series should be enough because protection Schottky diodes are included.

Final Remarks

- From now on, support only for Windows.
- For macOS:
  - Virtual Machine (parallels or Oracle VB)
  - Boot Camp.
- For Linux
  - Virtual Machine (Oracle VB)
  - WINE.
- In case you didn't know: "Currently enrolled UBC students with a valid CWL account now qualify for one license of Windows 10 Education."

https://it.ubc.ca/services/desktop-print-services/software-licensing/windows-10-education