University of British Columbia
Electrical and Computer Engineering
ELEC291/ELEC292

Lab 5 - Measuring Phasors

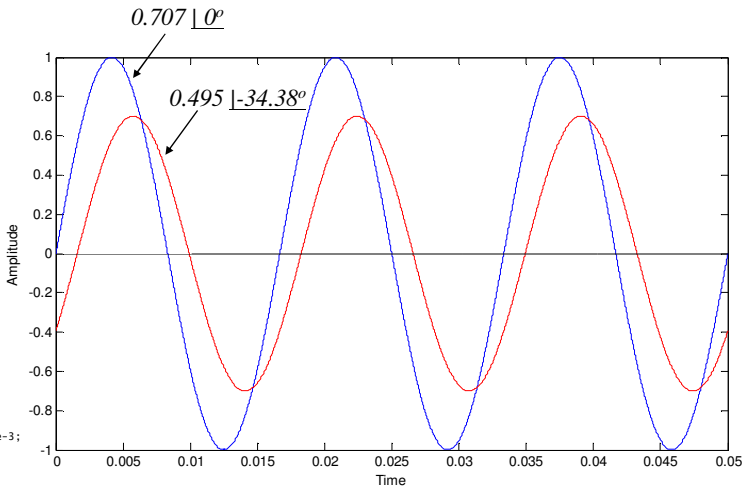
Dr. Jesús Calviño-Fraga P.Eng.
Department of Electrical and Computer Engineering, UBC
Office: KAIS 3024
E-mail: jesusc@ece.ubc.ca
Phone: (604)-827-5387

March 3, 2023

Objectives

- Understand what is a phasor.
- Measure sine wave amplitude.
- Measure sine wave phase.
- Work with Makefiles.

What is a Phasor?




```
t=0:0.01e-3:50e-3;
f=60;
w=2*pi*f;
y=sin(w*t);
z=0.7*sin(w*t-0.6);
plot(t,y,'b',t,z,'r',t,0*y,'k-');
xlabel('Time')
ylabel('Amplitude')
```

Matlab script

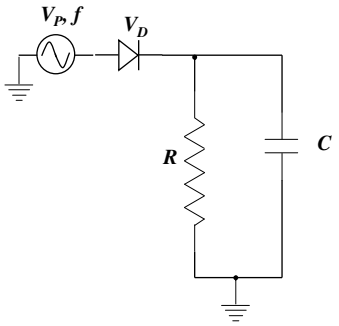
What do we need?

- Two sinusoidal signals. Without any DC, preferably! The new function generators in the lab can do that!!!
- A circuit or method to measure the peak value of the signals. Notice that phasors use RMS voltages/currents. We will need to convert V_{peak} to V_{RMS}.
- A circuit or method to measure the phase difference. We will be using degrees.

Measuring Peak AC or RMS

- We can use different methods:
  - Peak detector. For good accuracy, a precision peak detector may be used.
  - One shot measurement of the peak value by precisely timing the input waveform using a zero cross detector.
  - Fast ADC sampling:
    - look for the max value of the wave.
    - computation of the RMS using floating point arithmetic.
  - Direct RMS measurement using a RMS to DC converter such as the **AD536A** IC. Usually very expensive: 21$ to 112$!

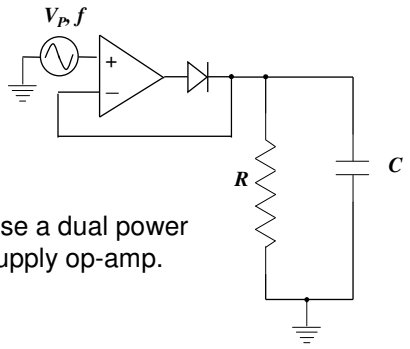
Simple Peak Detector



$$V_r = \frac{V_P - V_D}{f \times R \times C}$$

USE THIS CIRCUIT FOR PROJECT 2!

Precision Peak Detector



Use a dual power supply op-amp.

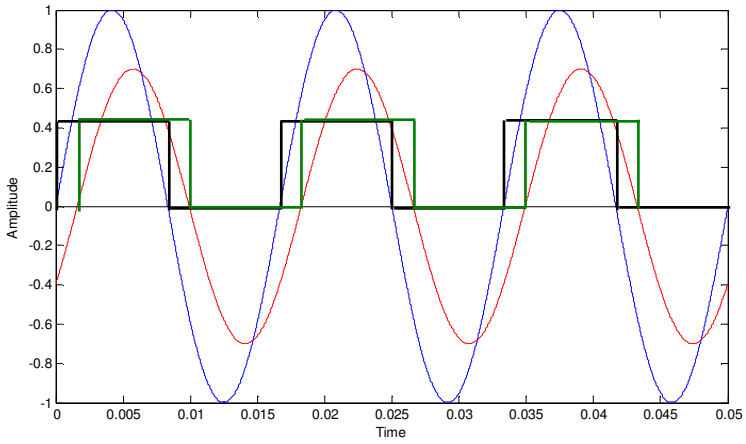
$$V_r = \frac{V_P}{f \times R \times C}$$

Good for measuring the magnitude of a phasor!

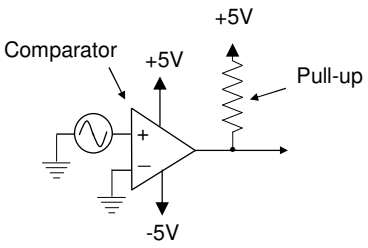
Zero Cross Detection

- If we know were the two waves cross zero we can:
  - Find the frequency of the waves.
  - Determine where the peak value is 'located'.
  - Measure the phase difference between the two waves.
- Since we need to know when the waves cross zero in order to determine their phase difference, we may as well use it to determine where the peak voltage is and measure it!

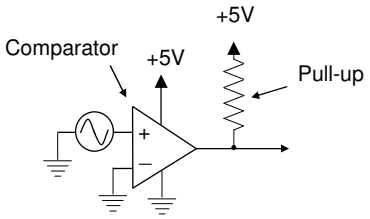
Zero Cross Detector Signals and Sine Waves



Zero Cross Detection

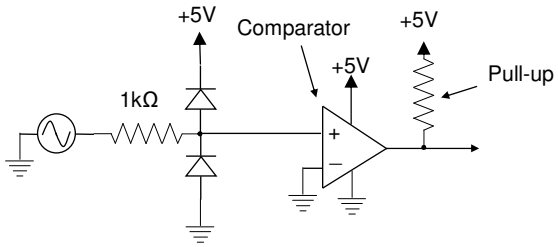


Works for any input voltage from -5V to +5V, but the output voltage of the comparator is either -5V or +5V



Works only for positive voltages from 0V to 5V, but the output voltage of the comparator is now either 0V or +5V.

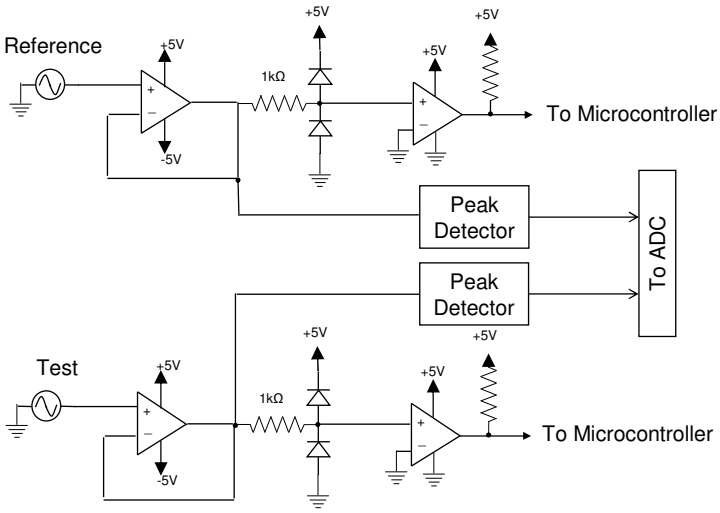
Zero Cross Detection



Works for “any” AC input voltage.
Comparator output is either 0V or +5V.

The 1kΩ resistor is used to limit the current throughout the diodes in case of over/under voltage.

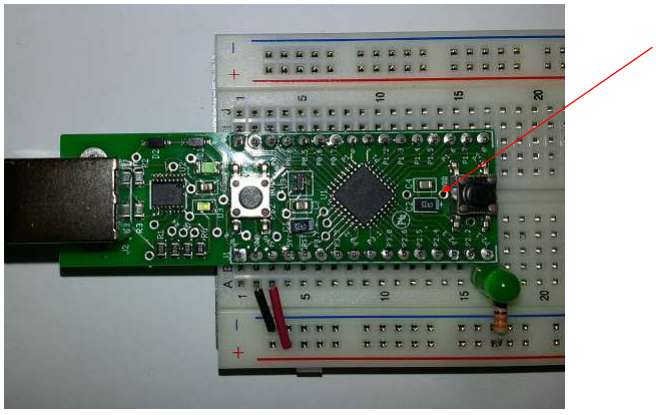
Zero Cross Detection & Peak Detector to ADC



Using the Built-in ADC in the EFM8LB1

- EFM8_ADC.c: ADC example available on Canvas:
  - This program measures the voltages at pins P1.4, P1.5, P1.6, and P1.7.
  - Uses the serial port to transmit the results to PuTTy.
  - The reference I used is V_{DD} (should be very close to 3.3V). If you also use V_{DD}, you should measure it.

EFM8LB1 VDD Voltage (Used in ADC example EFM8_ADC.c)



EFM8LB1 ADC

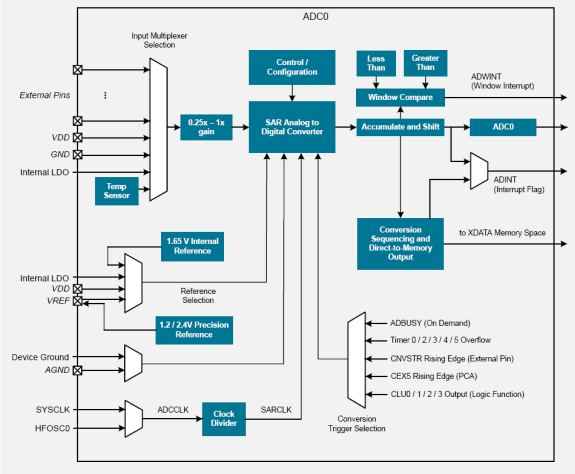


Figure 12.1. ADC Block Diagram

EFM8LB1 ADC Initialization

```
void InitADC (void)
{
    SFRPAGE = 0x00;
    ADEN=0; // Disable ADC

    ADC0CN1=
        (0x2 << 6) | // 0x0: 10-bit, 0x1: 12-bit, 0x2: 14-bit
        (0x0 << 3) | // 0x0: No shift. 0x1: Shift right 1 bit. 0x2: Shift right 2 bits. 0x3: Shift right 3 bits.
        (0x0 << 0) ; // Accumulate n conversions: 0x0: 1, 0x1:4, 0x2:8, 0x3:16, 0x4:32

    ADC0CF0=
        ((SYSCLK/SARCLK) << 3) | // SAR Clock Divider. Max is 18MHz. Fsarclk = (Fadcclk) / (ADSC + 1)
        (0x0 << 2); // 0:SYSCLK ADCCLK = SYSCLK. 1:HFOSC0 ADCCLK = HFOSC0.

    ADC0CF1=
        (0 << 7)   | // 0: Disable low power mode. 1: Enable low power mode.
        (0x1E << 0); // Conversion Tracking Time. Tadtk = ADTK / (Fsarclk)

    ADC0CN0 =
        (0x0 << 7) | // ADEN. 0: Disable ADC0. 1: Enable ADC0.
        (0x0 << 6) | // IPOEN. 0: Keep ADC powered on when ADEN is 1. 1: Power down when ADC is idle.
        (0x0 << 5) | // ADINT. Set by hardware upon completion of a data conversion. Must be cleared by firmware.
        (0x0 << 4) | // ADBUSY. Writing 1 to this bit initiates an ADC conversion when ADCM = 000. This bit should not be polled
        (0x0 << 3) | // ADWINT. Set by hardware when the contents of ADC0H:ADC0L fall within the window specified by ADC0GTH:ADC0
        (0x0 << 2) | // ADGN (Gain Control). 0x0: PGA gain=1. 0x1: PGA gain=0.75. 0x2: PGA gain=0.5. 0x3: PGA gain=0.25.
        (0x0 << 0) ; // TEMPE. 0: Disable the Temperature Sensor. 1: Enable the Temperature Sensor.

    ADC0CF2=
        (0x0 << 7) | // GNDSL. 0: reference is the GND pin. 1: reference is the AGND pin.
        (0x1 << 5) | // REFSL. 0x0: VREF pin (external or on-chip). 0x1: VDD pin. 0x2: 1.8V. 0x3: internal voltage reference.
        (0x1F << 0); // ADPWR. Power Up Delay Time. Tpwrtime = ((4 * (ADPWR + 1)) + 2) / (Fadcclk)

    ADC0CN2 =
        (0x0 << 7) | // PACEN. 0x0: The ADC accumulator is over-written.  0x1: The ADC accumulator adds to results.
        (0x0 << 0) ; // ADCM. 0x0: ADBUSY, 0x1: TIMER0, 0x2: TIMER2, 0x3: TIMER3, 0x4: CNVSTR, 0x5: CEX5, 0x6: TIMER4, 0x7: TIMER

    ADEN=1; // Enable ADC
}
```

SFR Example

12.4.2 ADC0CN1: ADC0 Control 1

Bit	7	6	5	4	3	2	1	0
Name	ADBITS		ADSJST			ADRPT		
Access	RW		RW			RW		
Reset	0x1		0x0			0x0		

SFR Page = 0x0, 0x30; SFR Address: 0xB2

Bit	Name	Reset	Access	Description
7:6	ADBITS	0x1	RW	**Resolution Control.**
	Value	Name		Description
	0x0	10_BIT		ADC0 operates in 10-bit mode.
	0x1	12_BIT		ADC0 operates in 12-bit mode.
	0x2	14_BIT		ADC0 operates in 14-bit mode.
5:3	ADSJST	0x0	RW	**Accumulator Shift and Justify.**
	Specifies the format of data read from ADC0H:ADC0L. All remaining bit combinations are reserved.			
	Value	Name		Description
	0x0	RIGHT_NO_SHIFT		Right justified. No shifting applied.

Configuring a Pin for Analog Input

Configuring Port Pins For Analog Modes

Any pins to be used for analog functions should be configured for analog mode. When a pin is configured for analog I/O, its weak pull-up, digital driver, and digital receiver are disabled. This saves power by eliminating crowbar current, and reduces noise on the analog input. Pins configured as digital inputs may still be used by analog peripherals; however this practice is not recommended. Port pins configured for analog functions will always read back a value of 0 in the corresponding Pn Port Latch register. To configure a pin as analog, the following steps should be taken:

1. Clear the bit associated with the pin in the PnMDIN register to 0. This selects analog mode for the pin.
2. Set the bit associated with the pin in the Pn register to 1.
3. Skip the bit associated with the pin in the PnSKIP register to ensure the crossbar does not attempt to assign a function to the pin.

From the EFM8LB1 reference manual, page 99.

EFM8LB1 Analog Pin Initialization

```
void InitPinADC (unsigned char portno, unsigned char pinno)
{
    unsigned char mask;

    mask=1<<pinno;
    SFRPAGE = 0x20;
    switch (portno)
    {
        case 0: // P0.0 and P0.3 can not be used for analog input
            P0MDIN &= (~mask); // Set pin as analog input
            P0|=mask; // Set the bit associated with the pin in the Pn register to 1
            P0SKIP |= mask; // Skip Crossbar decoding for this pin
        break;
        case 1:
            P1MDIN &= (~mask); // Set pin as analog input
            P1|=mask; // Set the bit associated with the pin in the Pn register to 1
            P1SKIP |= mask; // Skip Crossbar decoding for this pin
        break;
        case 2: // P2.0 and P2.7 can not be used for analog input
            P2MDIN &= (~mask); // Set pin as analog input
            P2|=mask; // Set the bit associated with the pin in the Pn register to 1
            P2SKIP |= mask; // Skip Crossbar decoding for this pin
        break;
        default: // P3 can not be used for analog input
        break;
    }
    SFRPAGE = 0x00;
}
```


Convert and read from ADC

```
#define VDD 3.299 // The measured value of VDD in volts

unsigned int ADC_at_Pin ( unsigned char pin)
{
    ADC0MX = pin;   // Select input from pin
    ADINT=0;       // Dummy conversion first to select new pin
    ADBUSY = 1;     // Convert voltage at the pin
    while (!ADINT); // Wait for conversion to complete
    return (ADC0);
}

float Volts_at_Pin( unsigned char pin)
{
    return ((ADC_at_Pin(pin)*VDD)/16383.0);
}
```

$$2^{14}-1=((1<<14)-1)=16383$$

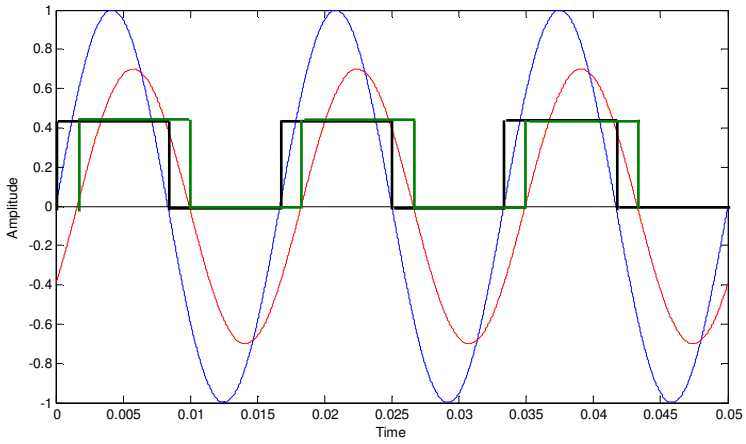
Convert and read from ADC

```
.
.
.
InitPinADC(1, 4); // Configure P1.4 as analog input
InitPinADC(1, 5); // Configure P1.5 as analog input
InitPinADC(1, 6); // Configure P1.6 as analog input
InitPinADC(1, 7); // Configure P1.7 as analog input
InitADC();

while(1)
{
    // Read 14-bit value from the pins configured as analog inputs
    v[0] = Volts_at_Pin(QFP32_MUX_P1_4);
    v[1] = Volts_at_Pin(QFP32_MUX_P1_5);
    v[2] = Volts_at_Pin(QFP32_MUX_P1_6);
    v[3] = Volts_at_Pin(QFP32_MUX_P1_7);
    printf ("V@P1_4=%7.5fV, V@P1_5=%7.5fV, V@P1_6=%7.5fV, V@P1_7=%7.5fV\r",
             v[0], v[1], v[2], v[3]);
    waitms(500);
}
```

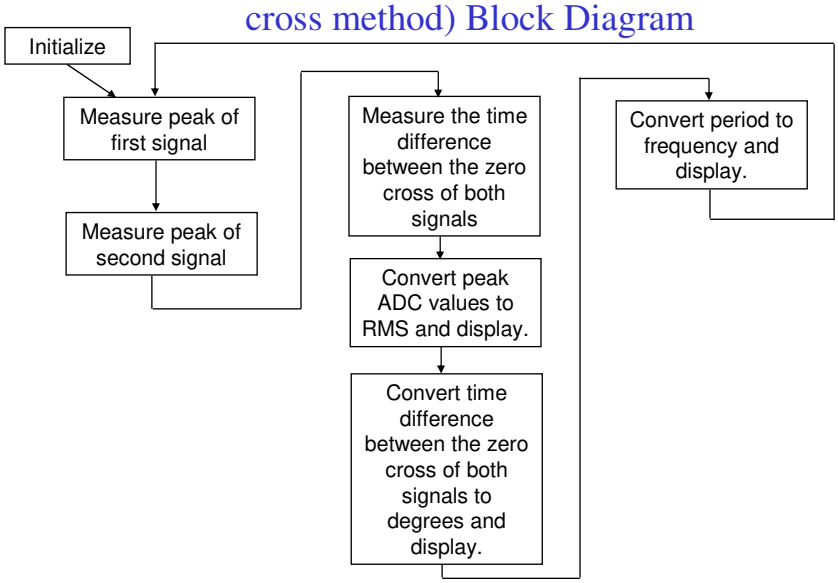

My experience with the EFM8LB1 ADC

- Speed, resolution, and accuracy are fantastic!
- The input is not protected for over/under voltages. Better safe than sorry: add a limiter circuit (two diodes and a resistor).
- Some pins can not be used as analog inputs: P0.0, P0.3, P2.0, P2.7, and all pins of P3.
- The main selling point of this microcontroller is the ADC!

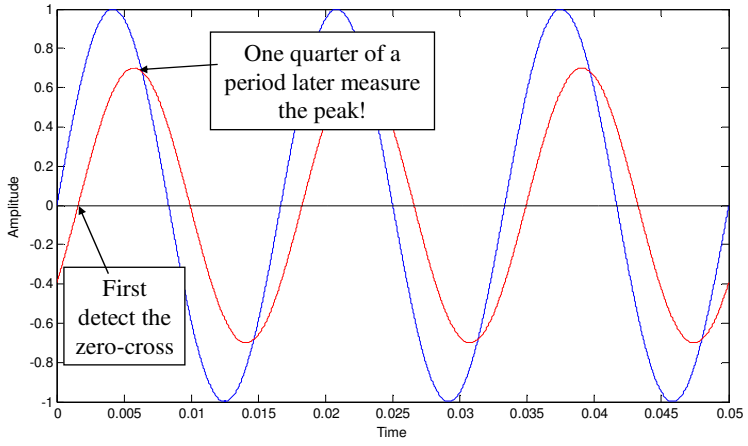
Zero Cross Detector Signals and Sine Waves



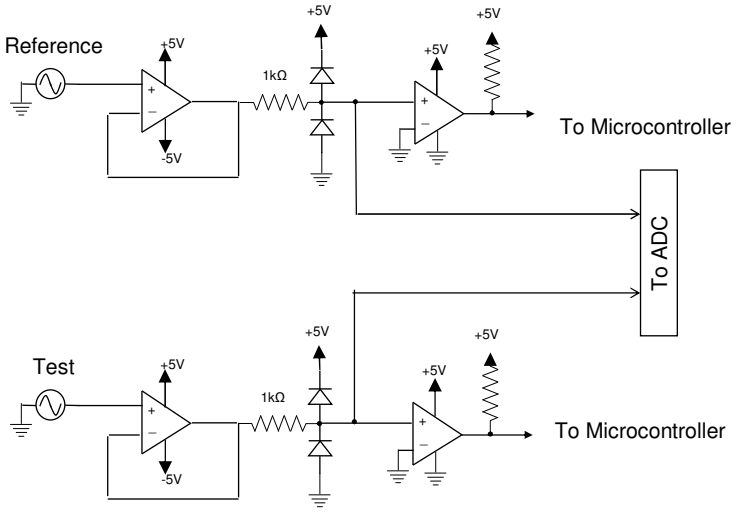
Phasor Measurement (using peak detector and zero cross method) Block Diagram



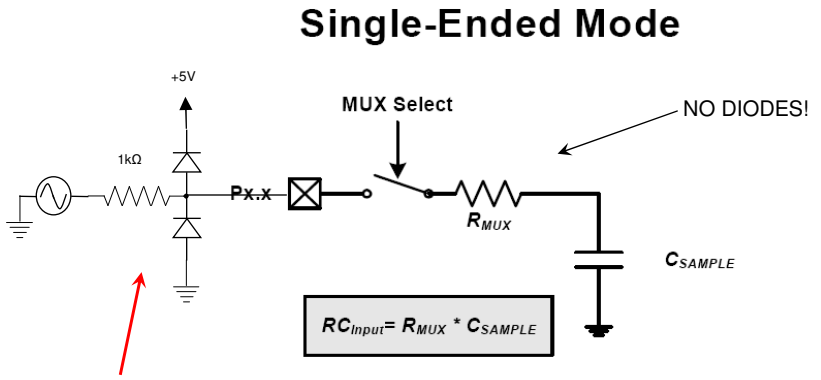
One Shot Measurement of the Peak Value with the ADC



One Shot Measurement of the Peak Value with the ADC



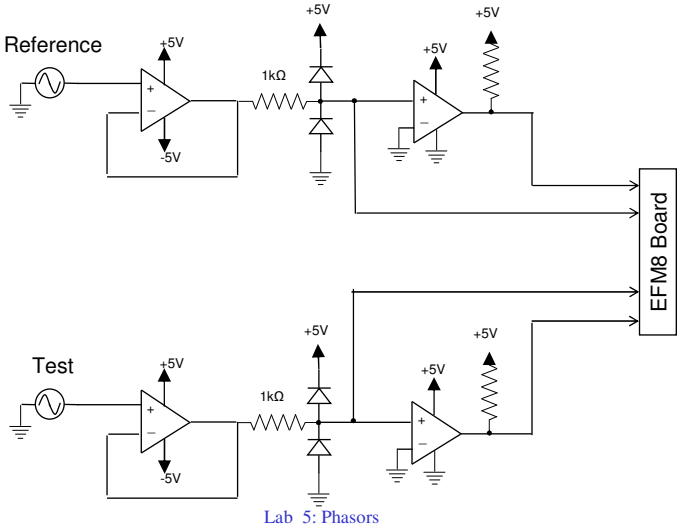
EFM8LB1 Analog Input Model



Protect the Analog inputs from under/over voltages!!!

Figure 12.4 in EFM8LB1 reference manual.

One Shot Measurement of the Peak Value with the ADC and Zero Cross



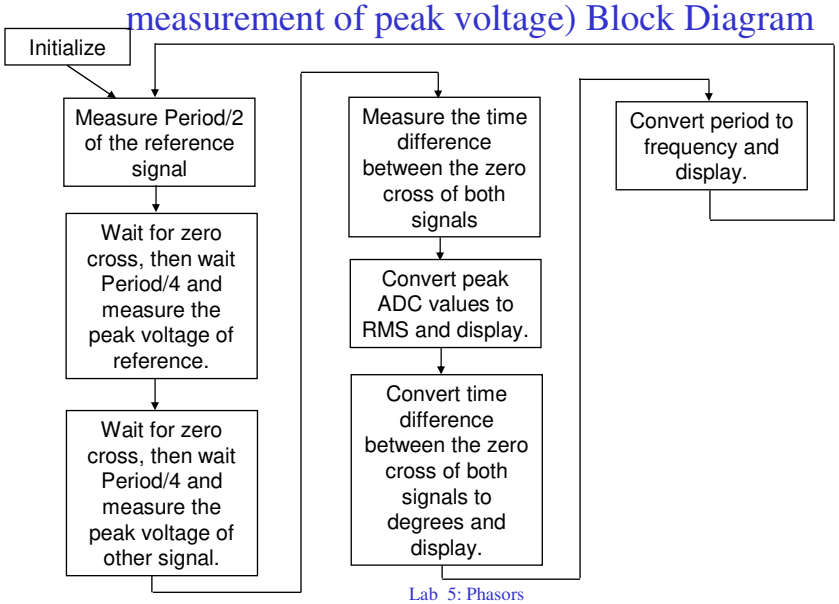
Measuring Half Period Using Timer 0

```
// Measure half period at pin P1.0 using timer 0
TR0=0; // Stop timer 0
TMOD=0B_0000_0001; // Set timer 0 as 16-bit timer
TH0=0; TL0=0; // Reset the timer
while (P1_0==1); // Wait for the signal to be zero
while (P1_0==0); // Wait for the signal to be one
TR0=1; // Start timing
while (P1_0==1); // Wait for the signal to be zero
TR0=0; // Stop timer 0
// [TH0,TL0] is half the period in multiples of 12/CLK, so:
Period=(TH0*0x100+TL0)*2; // Assume Period is unsigned int
```

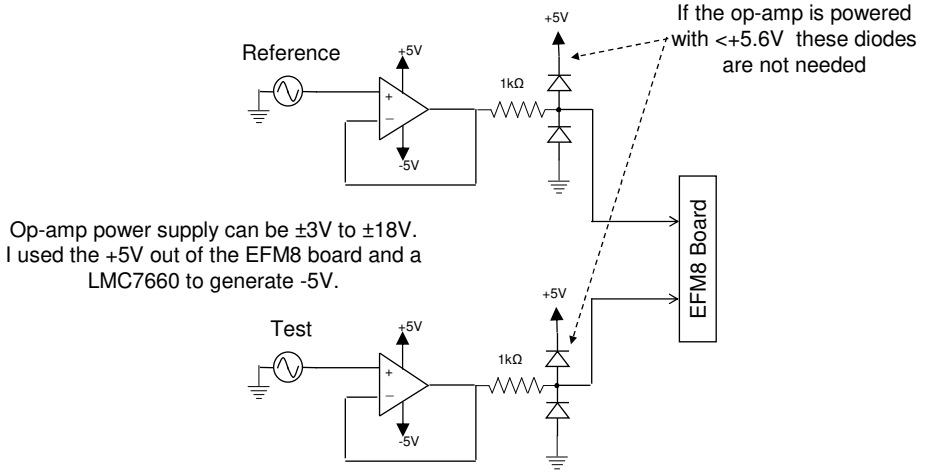

Half Period of the Reference Signal.

- After the code in the previous slide is executed we have half the period in [TH0, TL0] in units of 12/CLK. For the EFM8LB1 CLK can be 12.5MHz, 24MHz, 48MHz, or 72MHz.
- We can do two things with half the period:
  - Multiply it by two to obtain the period, then get its inverse to obtain the frequency
  - Divide it by two to find where the peak of the sine wave occurs.

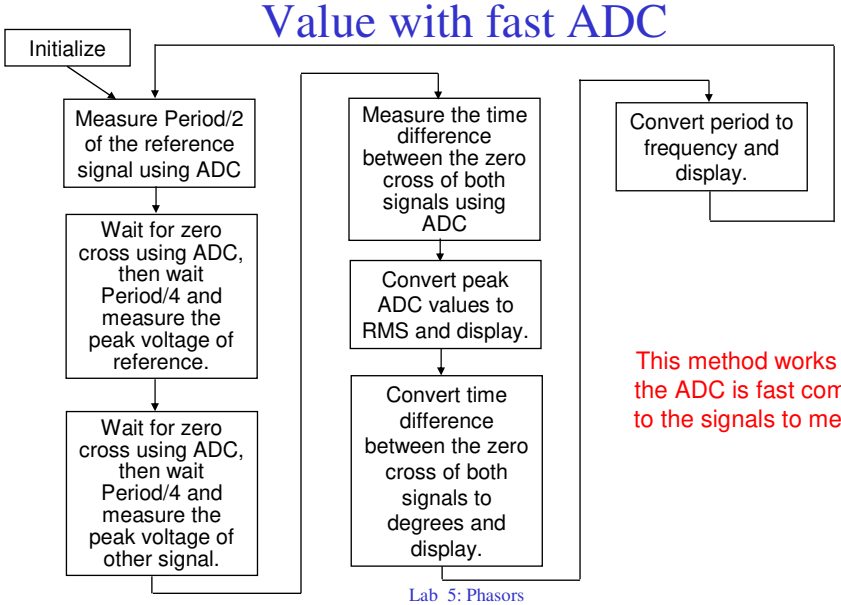
Phasor Measurement (using zero cross and one shot measurement of peak voltage) Block Diagram



One Shot Measurement of the Peak Value with fast ADC



One Shot Measurement of the Peak Value with fast ADC



This method works only if the ADC is fast compared to the signals to measure.

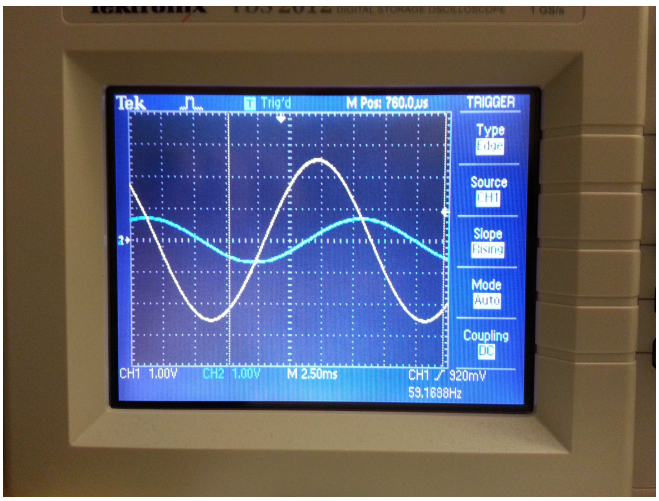
Measure half period using the Fast ADC in the EFM8

```
// Start tracking the reference signal
AMX0P=LQFP32_MUX_P1_7;
ADINT = 0;
AD0BUSY=1;
while (!ADINT); // Wait for conversion to complete
// Reset the timer
TL0=0;
TH0=0;
while (Get_ADC()!=0); // Wait for the signal to be zero
while (Get_ADC()==0);  // Wait for the signal to be positive
TR0=1; // Start the timer 0
while (Get_ADC()!=0); // Wait for the signal to be zero again
TR0=0; // Stop timer 0
half_period=TH0*256.0+TL0; // The 16-bit number [TH0-TL0]

// Time from the beginning of the sine wave to its peak
overflow_count=65536-(half_period/2);
etc.
```

GetADC()

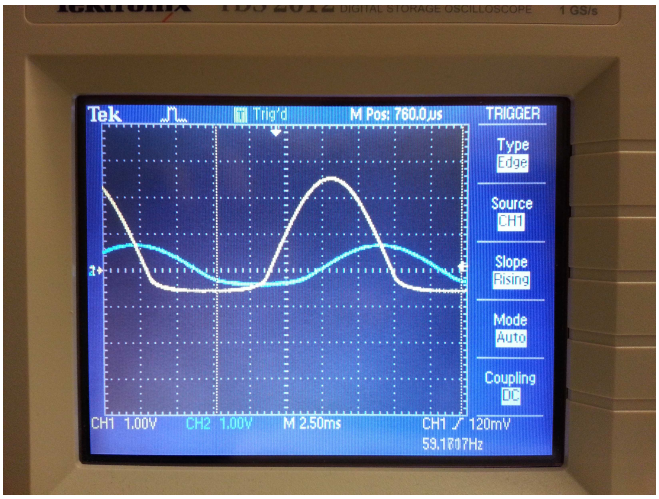
```
unsigned int Get_ADC (void)
{
    ADINT = 0;
    AD0BUSY = 1;
    while (!ADINT); // Wait for conversion to complete
    return (ADC0);
}
```


The Reference and Test Signals



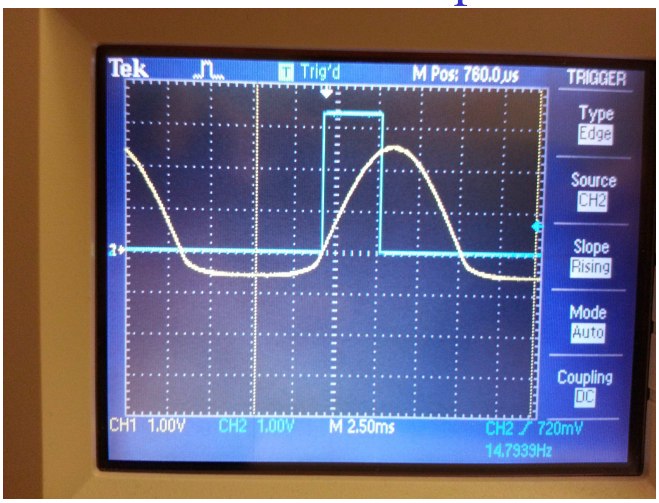
Reference: CH1 (yellow)
Test: CH2 (blue)

Signals connected to the EFM8 board



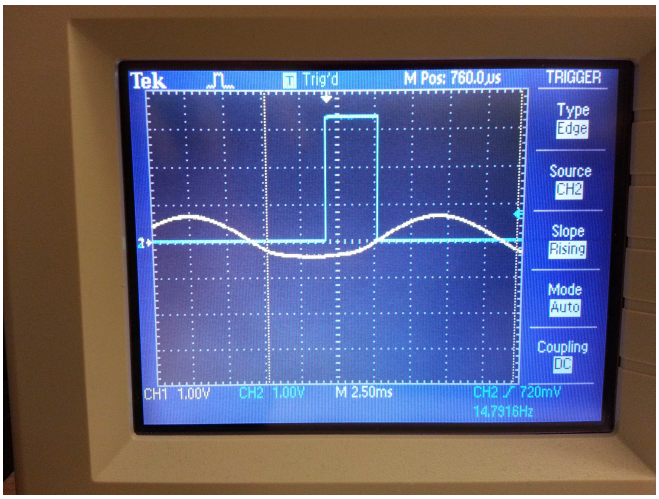
Only the positive part of the sine waves are correct!

Using P0.0 to debug 'times' using the oscilloscope



The debug pin (CH2, P0.0) is set to one when the 'Reference' wave crosses zero and clear to zero when the 'Test' signal crosses zero

Using P0.0 to debug using the scope

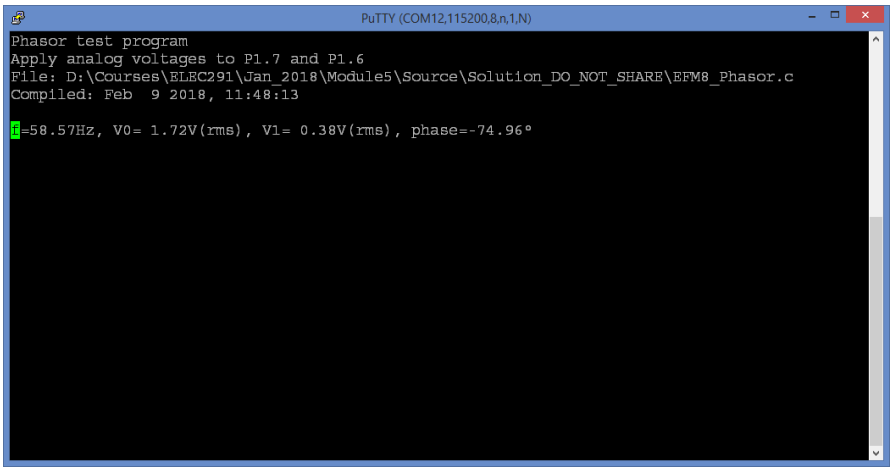


The debug pin (CH2, P0.0) is set to one when the 'Reference' wave crosses zero and clear to zero when the 'Test' signal crosses zero

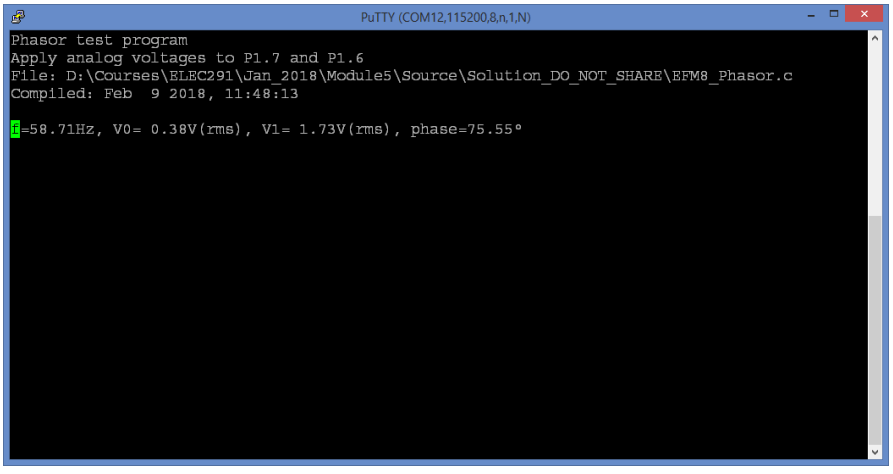
How to compute the phase difference in degrees

- One complete period of the wave is equivalent to 360°.
- The width of the pulse from the previous slide is about 3.5ms, and the period of the wave is about 16.9ms.
- Therefore the phase difference is 3.5*(360°/16.9) =74.56°.

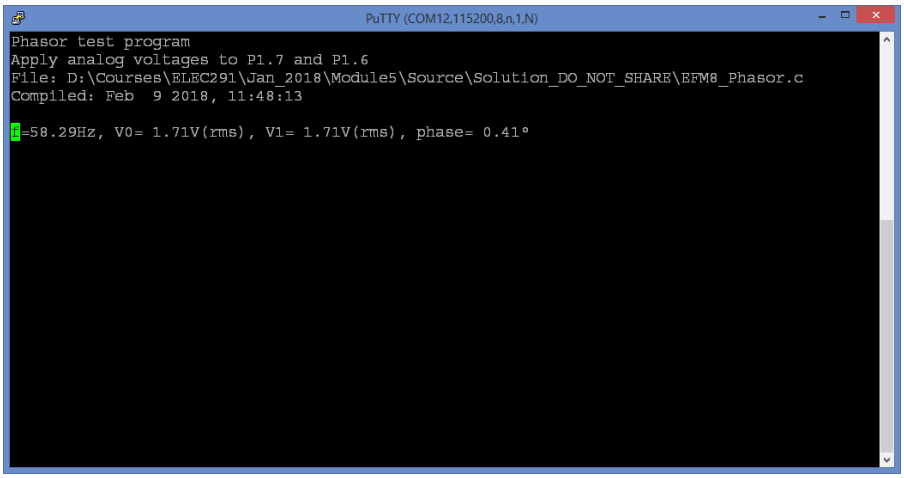
Output (Original Circuit)



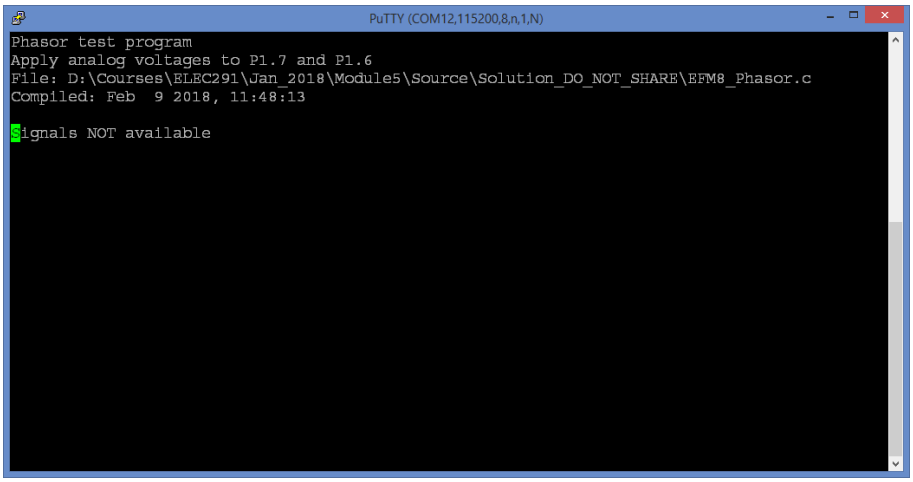
Output (Signals Swapped)



Output (Same Signal Both Inputs)



Output (no signals connected)



Working With Makefiles

- Industry standard.
- Easy compilation and linking of multiple files.
- Works for any processor, compiler, or operating system.
- Allows to automate tasks.
- Permits the execution of external programs, for example the flash loader.

Working With Makefiles in CrossIDE

To work with Makefiles for the EFM8 board we need these programs:

- CrossIDE V2.26 (or newer) & GNU Make V4.2 (or newer). In CrossIDE installation.
- CALL51 Toolchain for Windows. In CrossIDE installation.
- EFM8 Flash Loader: EFM8_prog. Available on Canvas or in the same zip as the examples.
- PuTTy.

Available on Canvas

- All the Makefile projects as a zip file.
- The "EFM8 Microcontroller System" with the instructions on how to use the Makefiles.
- Instructions and Makefiles for the ATmega328, PIC32MX130, LPC824, and STM32L051, and MSP430 processors. (Needed for lab #6 and project #22)

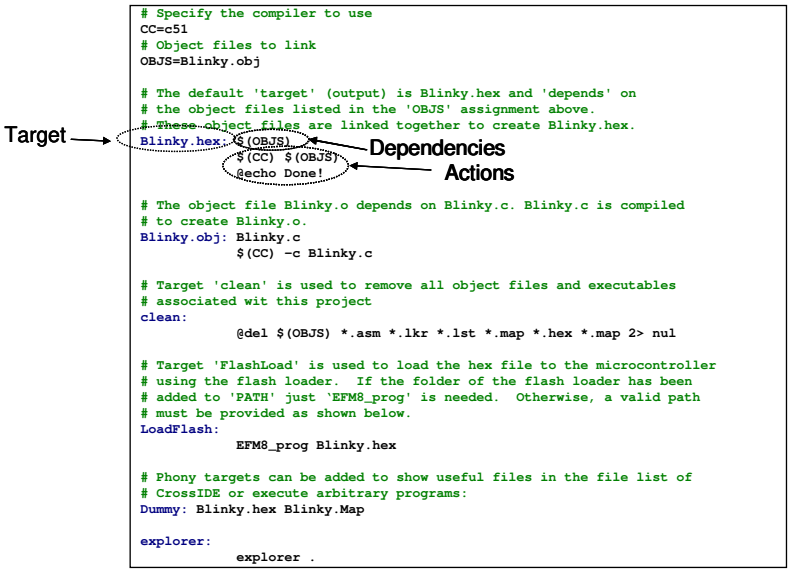
Setting the Path for the Programs

- The folder of these programs: GNU make, the compiler, the flash loader, and PuTTy must be available in the PATH environment variable of your computer.
- Also there is a little program called "wait.exe" that is used by some of the Makefiles. Its folder must be in the PATH as well.
- Instructions to set the PATH are available here:
  http://www.computerhope.com/issues/ch000549.htm

CrossIDE Workflow With Makefiles

- Creation and Maintenance of Makefiles.
- Using Makefiles with CrossIDE: Compiling, Linking, and Loading.
- Testing.

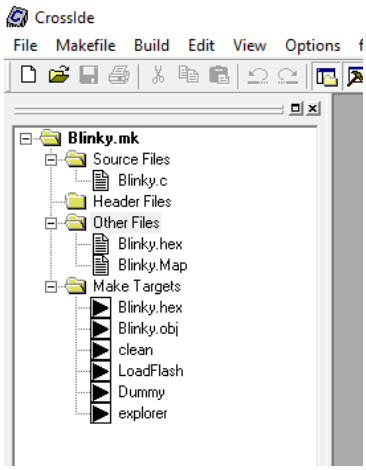
Example Makefile



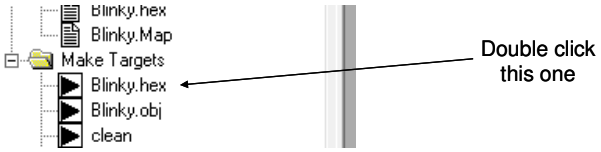

```
# Specify the compiler to use
CC=c51
# Object files to link
OBJS=Blinky.obj

# The default 'target' (output) is Blinky.hex and 'depends' on
# the object files listed in the 'OBJS' assignment above.
# These object files are linked together to create Blinky.hex.
Blinky.hex: $(OBJS)
            $(CC) $(OBJS)
            @echo Done!

# The object file Blinky.o depends on Blinky.c. Blinky.c is compiled
# to create Blinky.o.
Blinky.obj: Blinky.c
            $(CC) -c Blinky.c

# Target 'clean' is used to remove all object files and executables
# associated wit this project
clean:
            @del $(OBJS) *.asm *.lkr *.lst *.map *.hex *.map 2> nul

# Target 'FlashLoad' is used to load the hex file to the microcontroller
# using the flash loader.  If the folder of the flash loader has been
# added to 'PATH' just `EFM8_prog' is needed.  Otherwise, a valid path
# must be provided as shown below.
LoadFlash:
            EFM8_prog Blinky.hex

# Phony targets can be added to show useful files in the file list of
# CrossIDE or execute arbitrary programs:
Dummy: Blinky.hex Blinky.Map

explorer:
            explorer .
```

Makefile as a CrossIDE Project

To open a Makefile in CrossIDE, click "Makefile"→"Open" and select the Makefile to open.

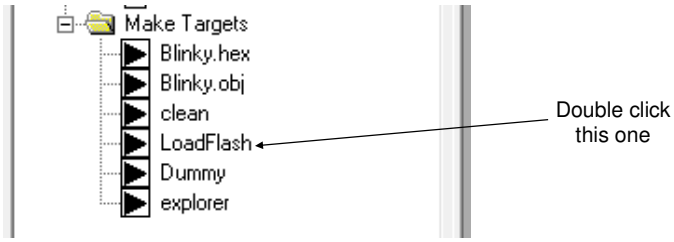


Compiling/Linking




```
---------------- CrossIde - Running Make ----------------
c51 -c Blinky.c
c51 Blinky.obj
Done!
```

Load Flash Memory




```
----------------- CrossIde - Running Make -----------------
EFM8_prog.exe -ft230 -r Blinky.hex
Serial flash programmer for the EFM8LB1. (C) Jesus Calvino-Fraga (2012-2017)
Connected to COM12
Blinky.hex: loaded 208 bytes
.Found EFM8LB12F64E_QFP32.  Id: 0x3442
Sending 'setup' command... Done.
Erasing flash memory...
############################################################
############################################################
##### Done.
Writing flash memory...
## Done.
Verifying...  Done.
Running program...  Done.
Actions completed in 2.2 seconds.
```


CrossIDE Makefile examples for the EFM8 board (may not be complete)

- **7SegDisp**: Uses a timer interrupt to handle a multiplexed three 7-segment display.
- **ADC**: Reads four channels of the built in 14-bit ADC (P2.2, P2.3, P2.4, and P2.5) and displays the result using printf() via PuTTy.
- **All_timers**: Uses the six timers and the five channels of the Programmable Counter Array (PCA) with their corresponding Interrupt Services Routines (ISRs) to generate eleven different frequencies at 11 pins.
- **Autotest**: Test that all the solder joints in the EFM8 board are correct.
- **Blinky**: Toggles an LED connected to pin P2.1. This is the same project used in the examples above.
- **BlinkyISR**: Similar to "Blinky" but instead of using a delay loop, it uses a timer interrupt. Timer 0 and its corresponding interrupt service routine (ISR) are used in this example.

CrossIDE Makefile examples for the EFM8 board (may not be complete)

- **DAC**: Demonstrates how to use the 12-bit DAC of the EFM8LB1. Also shows how to generate different wave forms: sine, ramp, triangle, and square using the DAC.
- **EFM8_Servo**: Shows how to control a standard servo motor (HS-422) using a timer interrupt to generate the required 50 Hz PWM signal. The pulse width can be changed via the serial port and PuTTy to arbitrary values between 0.6 ms and 2.4 ms.
- **EFM8_UUID**: Shows how to read the "Universally Unique Identifier" available in the EFM8LB1 microcontroller. The identifier is displayed via the serial port and PuTTy.
- **Frequency**: Shows how to measure frequency using one of the counters in the EFM8LB1 microcontroller.
- **HelloWorld**: Uses printf() to display "Hello, World!" via the serial port and PuTTy.
- **I2C_24C02**: Shows how to use the built-in I2C/SMB controller to access and test a 24C02 I2C EEPROM.
- **I2C_Nunchuck**: Shows how to use the built-in I2C/SMB controller to access and use the Nintendo nunchuck that came with the WII game console. A nunchuck and connector are required to use this program.

CrossIDE Makefile examples for the EFM8 board (may not be complete)

- **LCD**: Shows how to configure and use a Hitachi compatible 16 x 2 LCD in four bit mode.
- **Period**: Shows how to measure period using one of the timers in the EFM8LB1 microcontroller and an arbitrary input pin.
- **Quadrature_Encoder**: Uses a timer interrupt to read and process the input of a 2-phase quadrature encoder. The encoder is used to increment/decrement a variable that is printed via the serial port to PuTTy.
- **SPI_ADC**: Shows how to use the built-in SPI controller to access the MCP3008 10-bit SPI ADC.
- **SPI_EEPROM**: Shows how to use the built-in SPI controller to access and test the FT93C66A SPI EEPROM.
- **Tetris**: Implements the game Tetris via PuTTy. Sorry, no next move preview!
- **Tone**: Shows how to produce a square wave at one of the pins of the microcontroller. The frequency is selected by using the scanf() function.
- **Uart1**: Shows how to configure and use the second uart (UART1) available in the EFM8LB1 microcontroller.