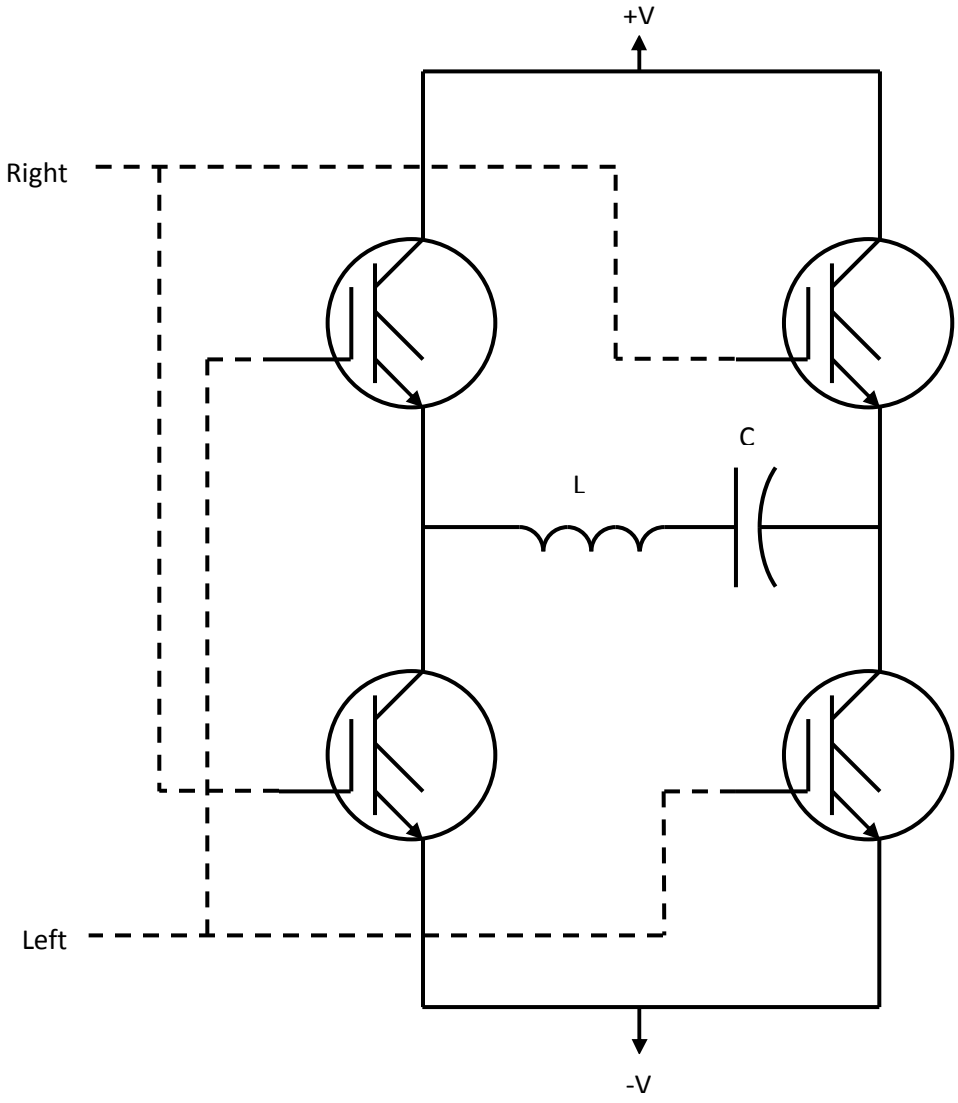
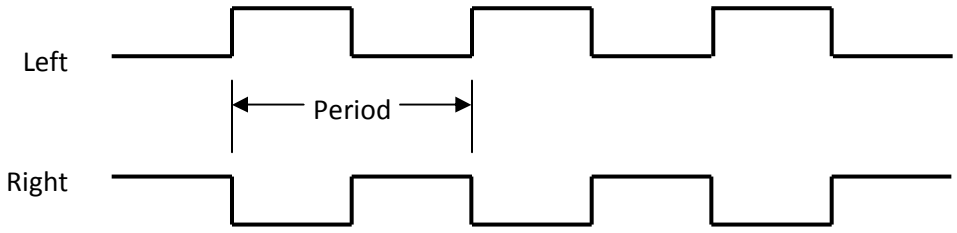
Digital Synthesis of a Sine Wave

This design uses a Freescale microcontroller with an embedded PWM circuit to drive an H-bridge. The microcontroller is the MC9S12E128. Freescale calls this embedded circuit Pulse Width Modulator with Fault Protection (PMF). In this discussion, PWM and PMF are used interchangeably.

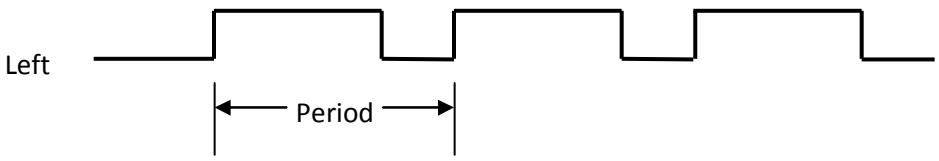


This figure shows a simplified H-bridge circuit. The labels Left and Right are there for discussion purposes. The signals on the Left and Right are complementary. In other words, when Left is positive, Right is negative and vice versa.

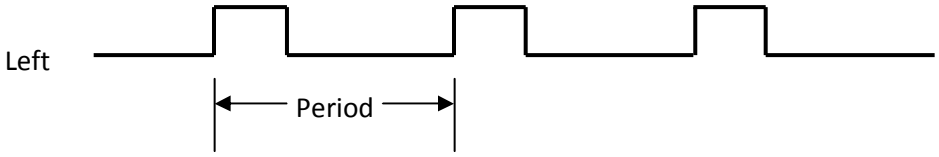


Digital Synthesis of a Sine Wave

A square wave with equal high and low periods will result in 0 volts across capacitor C. In order to produce voltage across C, we have to modulate the square wave.

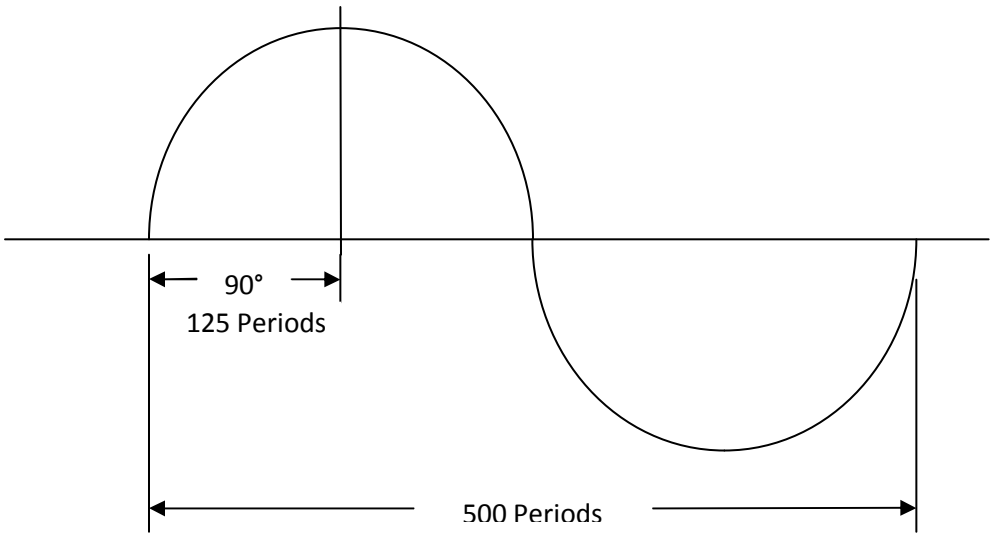


As the high part of the square wave increases, so does the voltage across C.



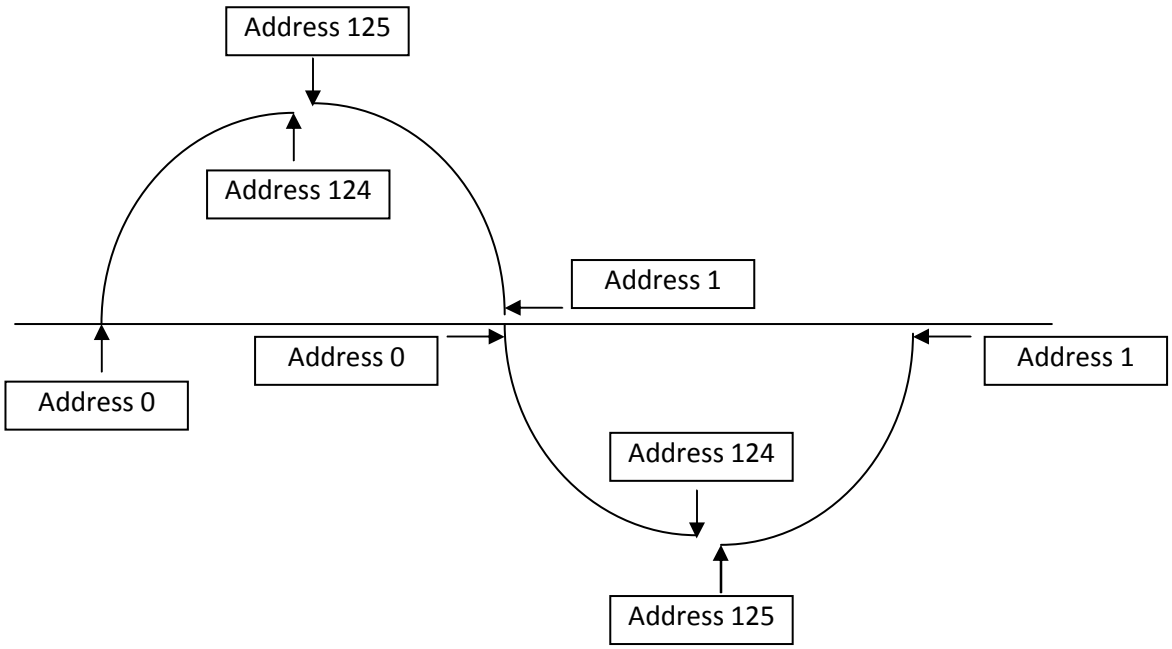
As the high part of the square wave decreases, the voltage across C reverses polarity.

In order to create a smooth sine wave across C, we need to modulate the square wave drive signal using a sine amplitude table. Only values for 90 degrees need to be saved in the table. Each 90 degree segment of the sine wave will consist of 125 pulse periods. Therefore, for one cycle of the sine wave, we will have 125 x 4 = 500 periods. The sine table values are calculated using the spreadsheet **SineTable.xls**. Since there are 125 periods for each 90 degrees, one period represents 90/125 = 0.72 degrees.



The microprocessor accesses values from the sine table using sequential addresses starting at address 0. Each time a value is retrieved from the sine table, the address is immediately incremented. After incrementing the address, a check is made to see if the end of the table has been reached. The end of the table has been reached when the address is at 125. At this point the sine table is accessed with

addresses from 125 to 1 in order to form a decreasing voltage. When the address is decremented to zero, the end of the sine table has been reached and the process starts all over again but the slope remains negative.



The microprocessor operates at 24MHz bus frequency. The internal PWM module is set to use ½ this bus frequency as its input clock. Therefore, to calculate how many bus clocks we need to make one PWM period, we use this formula: 24,000,000 / 2 / 500 / 60 = 400. This will be the Modulus count and determines the PWM period.

This PMF circuit is different from most pulse width modulators in that it can interrupt the microcontroller after each pulse. The microcontroller can then make adjustments to the pulse width before reloading the value for the next pulse. It also contains protection inputs that will shut it down, but I haven't yet made use of this feature in this version.

Each time the PMF interrupts the microcontroller, a calculation is made to determine the value that will be loaded for the next pulse. The value from the sine wave look-up table is multiplied by the desired amplitude (Amp) to come up with the final value. Amp can have values between 0 and 255. Therefore, Amp/255 is a fraction which has values from 0 to 1.

For a positive slope the following calculation is made:

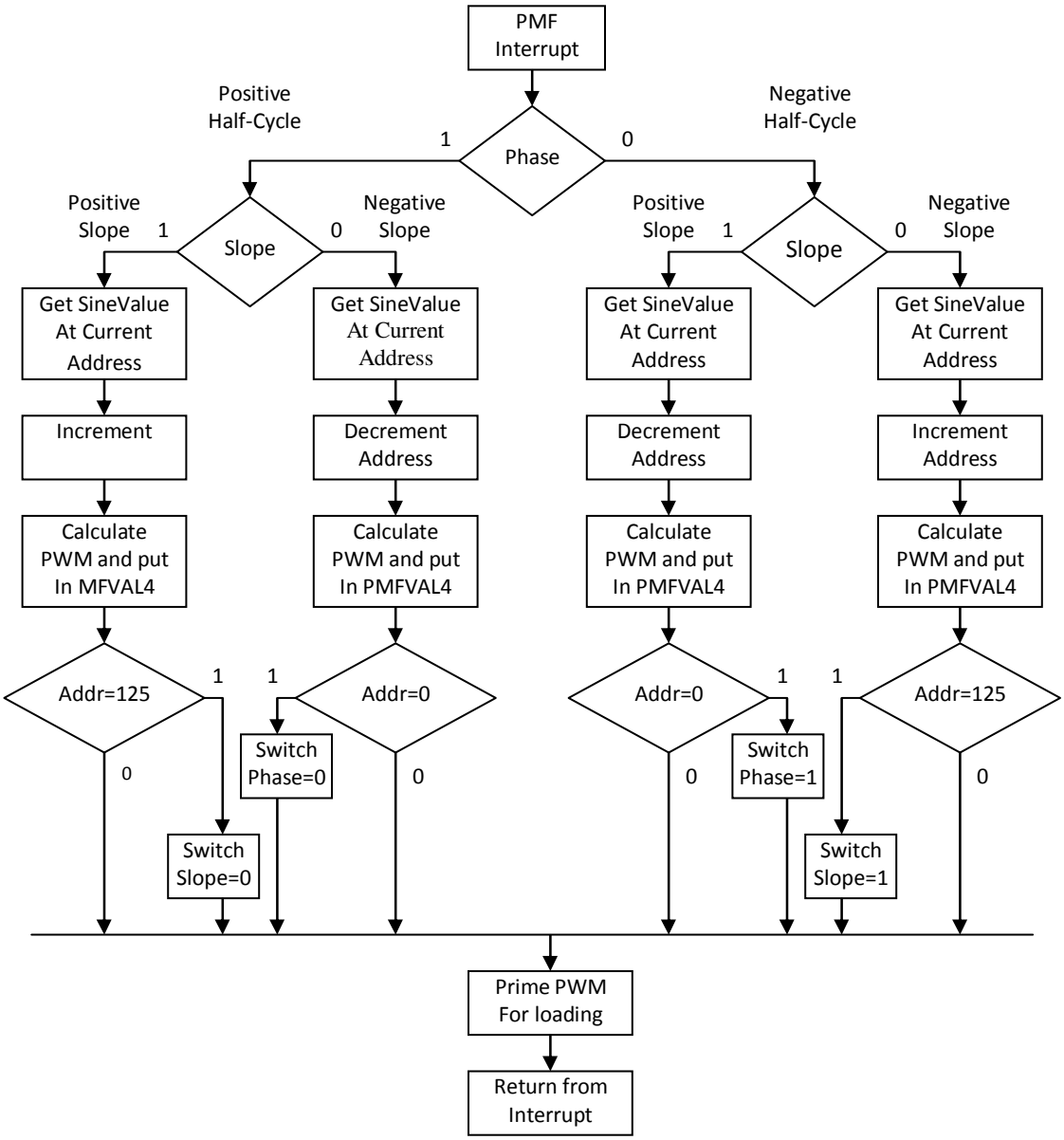
PMFVAL4 = (Modulus/2)+(SineValue*Amp/255);

And for the negative slope:

PMFVAL4 = (Modulus/2)-(SineValue*Amp/255);

Note: The Amp value can be set to go out of limits and will produce undesirable results. Therefore always use the IncAmplitude subroutine which checks the limits before incrementing the Amp value.

The following flowchart implements the interrupt routine for the pulse with modulator.

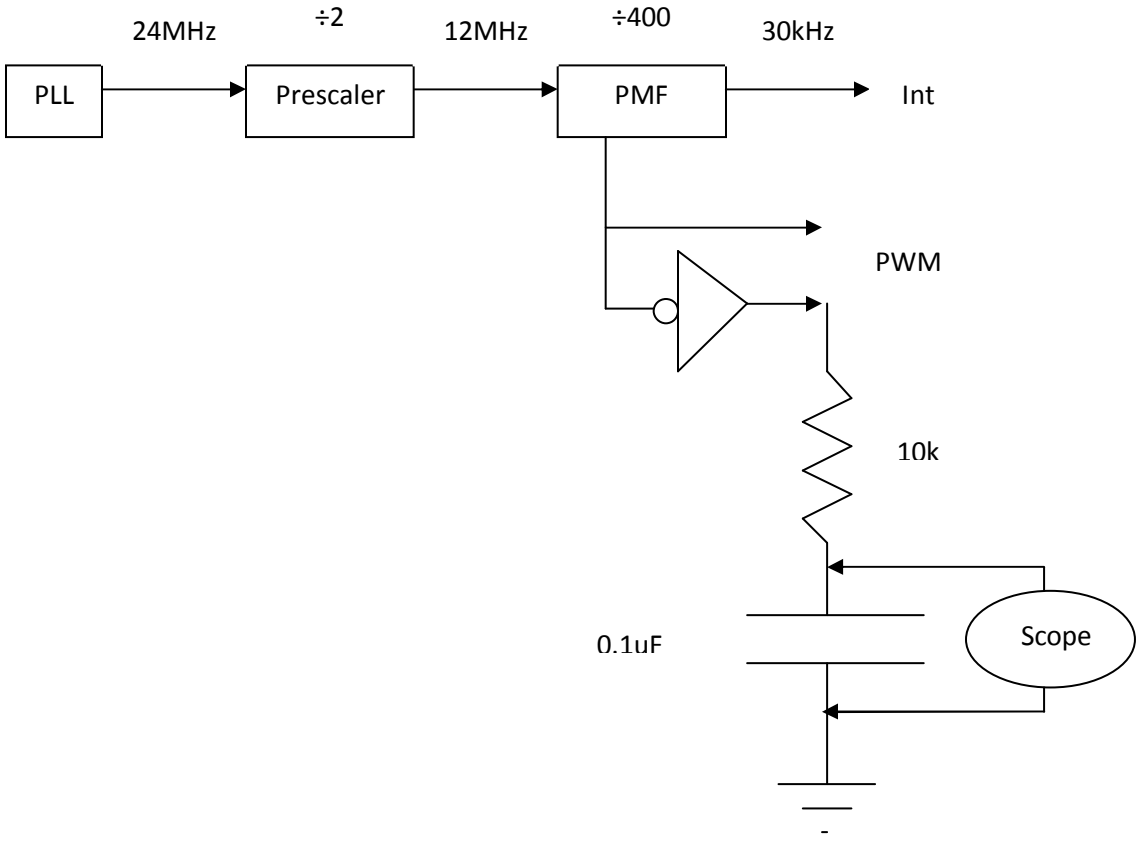


```
/*****************************************************************************
   PMF.c

*****************************************************************************/

// Include Files
#include <MC9S12E128.h>         // Derivative information
#include <stdio.h>

#include "PMF.h"
#include "ATD.h"
#include "Buffers.h"
#include "SCI0.h"
#include "Datatypes.h"


// Constants
// One-quarter of a sine wave (90 degrees). Only first 126 values are used.
const byte Sine[128] =
{
  0,  3,  5,  8, 10, 13, 15, 18, 20, 23, 25, 28, 30, 33, 35, 37,
 40, 42, 45, 47, 50, 52, 55, 57, 59, 62, 64, 67, 69, 71, 74, 76,
 78, 81, 83, 85, 87, 90, 92, 94, 96, 99,101,103,105,107,109,111,
113,116,118,120,122,124,126,127,129,131,133,135,137,139,141,142,
144,146,148,149,151,152,154,156,157,159,160,162,163,165,166,168,
169,170,172,173,174,175,176,178,179,180,181,182,183,184,185,186,
187,188,189,189,190,191,192,192,193,194,194,195,195,196,196,197,
197,198,198,198,199,199,199,199,200,200,200,200,200,200,200,200
};


// Global Variables
BOOL     Phase;      // Positive and negative half-cycle (2 per cycle)
BOOL     Slope;      // Positive and negative slope (2 positive and 2 negative per
cycle)
word     Angle;      // Phase angle of a sign wave
byte     Amp;        // Amplitude of sine wave generated
int      Adj;        // Correction for zero crossing

word     SineValue;  // Value of the sine wave from table


/*****************************************************************************/
#pragma CODE_SEG NON_BANKED
interrupt void PMFC_Reload_Int(void)
{
     if (Phase)      // Positive half-cycle
     {
          if (Slope) // Positive slope
          {
               SineValue = Sine[Angle++];// Get sine wave value from table
               PMFVAL4 = (Modulus/2)+(SineValue*Amp/255);     // Calculate PWM
               if (Angle == 125)
               {
                    switch(Adj)
                    {
                         case +1:        // Add an extra angle cycle
                              Angle--;   // Will do 124 over again
                              Adj = 0;
                              break;

                         case 0:              // In sync, no need to adjust
                              Slope = 0; // Change to negative slope
```

```
                              break;

                        case -1:        // Subtract one angle cycle
                              Slope = 0; // Change to negative slope
                              Angle--;   // Will not do 125
                              Adj = 0;
                              break;
                  }
            }
      }
      else // Negative slope
      {
            SineValue = Sine[Angle--];// Get sine wave value from table
            PMFVAL4 = (Modulus/2)+(SineValue*Amp/255);      // Calculate PWM
            if (Angle == 0)
                  Phase = 0; // Change to negative half-cycle
      }
}
else // Negative half-cycle
{
      if (!Slope)     // Negative slope
      {
            SineValue = Sine[Angle++];// Get sine wave value from table
            PMFVAL4 = (Modulus/2)-(SineValue*Amp/255);      // Calculate PWM
            if (Angle == 125)
            {
                  switch(Adj)
                  {
                        case +1:        // Add an extra angle cycle
                              Angle--;  // Will do 124 over again
                              Adj = 0;
                              break;

                        case 0:               // In sync, no need to adjust
                              Slope = 1; // Change to positive slope
                              break;

                        case -1:        // Subtract one angle cycle
                              Slope = 1; // Change to positive slope
                              Angle--;   // Will not do 125
                              Adj = 0;
                              break;
                  }
            }
      }
      else // Positive slope
      {
            SineValue = Sine[Angle--];// Get sine wave value from table
            PMFVAL4 = (Modulus/2)-(SineValue*Amp/255);      // Calculate PWM
            if (Angle == 0)
                  Phase = 1; // Change to positive half-cycle
      }
}
PMFENCC_LDOKC = 1;         // Allow PWM register load
PMFFQCC_PWMRFC = 1;        // Clear PWM Reload Flag C
}
#pragma CODE_SEG DEFAULT


/*****************************************************************************/
void InitPMF(void)
{
/* Use multiple timebase generators. C, B, and A are edge-aligned. C is
```

```
   complementary pair, B and A are independent. */
     PMFCFG0 = PMFCFG0_MTG_MASK | PMFCFG0_EDGEC_MASK | PMFCFG0_EDGEB_MASK |
                      PMFCFG0_EDGEA_MASK |/* PMFCFG0_INDEPC_MASK |*/
PMFCFG0_INDEPB_MASK |
                      PMFCFG0_INDEPA_MASK;

// Prescaler C
     PMFFQCC_PRSCC = 1;    // PWM clock frequency is f(bus)/2

// Load frequency
     PMFFQCC_LDFQC = 0;    // Reload every PWM opportunity

// Modulo of PWM C
     PMFMODC = Modulus;    // See PMF.h

// Pulse width of PWM C
     PMFVAL4 = Modulus/2; // Equal positive and negative periods produce 0 volts

// Deadtime of PWM C
     PMFDTMC = 2;          // Insert this many f(bus) cycles for dead time

// Enable generator C
     PMFENCC_LDOKC = 1;
     PMFENCC_PWMENC = 1;

     Phase = TRUE;         // 1 = top half, 0 = bottom half
     Slope = TRUE;         // 1 = going up, 0 = going down
     Angle = 0;            // Start at zero address is sine table
     Amp = 0;              // Starting amplitude is zero

     PMFENCC_PWMRIEC = 1; // Enable PWM C interrupts
}


/*****************************************************************************/
void IncAmplitude(void)
{
     if (Amp < 255)
          Amp++;
}


/*****************************************************************************/
void DecAmplitude(void)
{
     if (Amp > 0)
          Amp--;
}


/*****************************************************************************/
void DisplayAmplitude(void)
{
     char* BufNum;

     BufNum = GetBuffer();           // Get a free buffer
     (void)sprintf(BufNum,"Amplitude: %d\r\n",Amp);
     SCI0_PutQueue(BufNum);
}
```

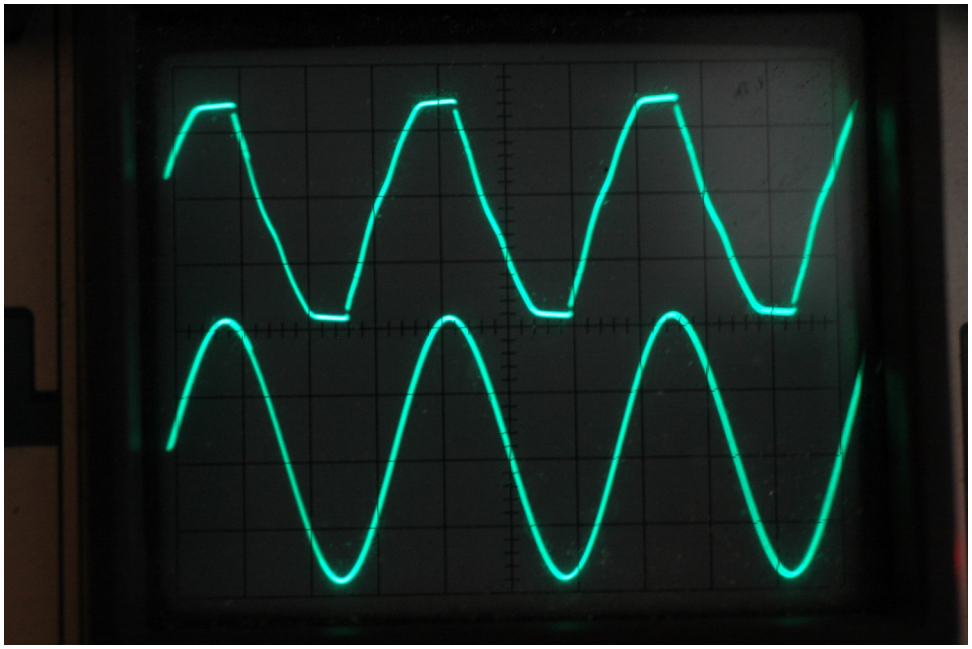
PWM



This microcontroller has limitations in its ability to do math. It can multiply an 8-bit number by another 8-bit number to get a 16-bit product. It can also divide a 16-bit number by an 8-bit number to get an 8-bit quotient. That is why I had to use the prescaler and adjust the sine table so that the values would fit into a single byte.

30kHz is within the capabilities of most IGBTs. It also divides evenly by 500 periods per sine wave giving an exact 60Hz frequency.

Scope



The upper trace is the power coming into the house from the mains. The bottom trace is the PWM generated across a small filter as in the diagram above.