Implementation of a

Network Analyzer

using the TI C54x

Randy Dimmett
Josh Horton

Abstract:

EE 301 Semester Project: Network Analyzer
Authors: Randy Dimmett and Josh Horton
Date: December 17th, 1999
Affiliation: Department of Electrical Engineering, University of Missouri-Rolla
Contact: dimmett@umr.edu

In this project, a network analyzer was designed and simulated in Matlab, and then implemented on Texas Instruments' C54x DSP chip. The DSP chip was programmed to generate a stepped chirp signal on the DSP output, and calculate the frequency response of the DSP input.

1. Introduction

The goal of this project was to design a network analyzer to measure the frequency response of a filter, using the Texas Instruments C54x DSP DSK board. This was accomplished by generating a stepped sinusoid at the output of the D/A, and reading in the value generated by that output into the A/D. This value was used in the calculations for the frequency response of the filter.

2. Theory

The theory for calculation of the frequency and phase response is shown below, where the output of the test filter is used as the input for the calculation of both the phase and frequency response of the filter. This input can be assumed to take the form of

$$A\cos(2\mathbf{p}t + \mathbf{f}) \qquad 3.1$$

The frequency response information is contained in the amplitude term. This information can be recovered by first multiplying Equation 3.1 by a cosine with the phase set to zero and a sine term at the same frequency with the phase also set to zero.

$$A\cos(2\mathbf{p}t + \mathbf{f})\cos(2\mathbf{p}t) \qquad 3.2$$

$$A\cos(2\mathbf{p}t + \mathbf{f})\sin(2\mathbf{p}t) \qquad 3.3$$

Simplifying Equations 3.2 and 3.3 yields:

$$\frac{A\cos(4\mathbf{p}t + \mathbf{f}) + A\cos(\mathbf{f})}{2} \qquad 3.4$$

$$\frac{A\sin(4\mathbf{p}t + \mathbf{f}) + A\sin(\mathbf{f})}{2} \qquad 3.5$$

To eliminate the high frequency terms a low pass filter is applied to Equations 3.4 and 3.5. This filter can be achieved by integrating Equations 3.4 and 3.5 over the period of the generated sinusoids.

$$\frac{A}{2}\int_0^T (\cos(4\mathbf{p}t + \mathbf{f}) + \cos(\mathbf{f}))dt = \frac{TA}{2}\cos(\mathbf{f}) \qquad 3.6$$

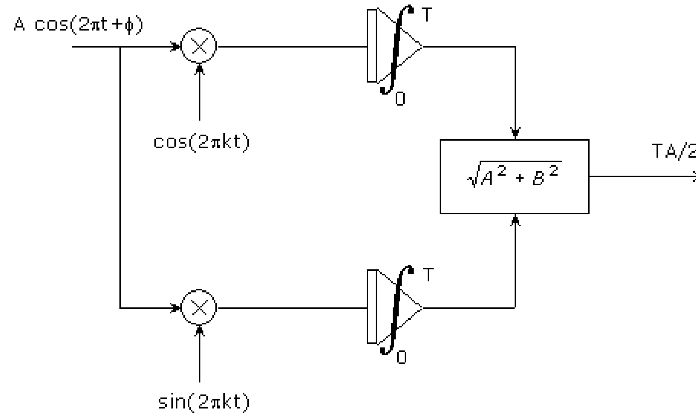
$$\frac{A}{2}\int_0^T (\sin(4\mathbf{p}t + \mathbf{f}) + \sin(\mathbf{f}))dt = \frac{TA}{2}\sin(\mathbf{f}) \qquad 3.7$$

The square root of the sum of the squares of Equations 3.6 and 3.7 yields the amplitude response of the filter at the given input frequency.

$$\sqrt{\left(\frac{TA}{2}\cos(\mathbf{f})\right)^2 + \left(\frac{TA}{2}\sin(\mathbf{f})\right)^2} = \frac{TA}{2} \qquad 3.8$$

This algorithm is show graphically in Figure 1.

Figure 1: Calculation of Amplitude Response



The phase response of the test filter can be measured by taking the inverse tangent of Equations 3.6 and 3.7.

$$\boldsymbol{q} = \tan^{-1}\left(\frac{\cos(\boldsymbol{f})}{\sin(\boldsymbol{f})}\right) \qquad 3.9$$

---

3. Design

The design of the network analyzer was broken into three components. The first was a system model developed in Matlab, which was used to test the program theory and confirm that the design approach would be a viable solution. The second was the generation of the stepped chirp at the output of the D/A of the DSK board. The third was the processing of the input data generated by the stepped sinusoid to calculate the frequency response.

3.1 Designing the Matlab Simulation

At the beginning of the design process, a Matlab script was developed to check the system design and to work out possible bugs before any DSP code was written. The generation of the stepped sinusoid in Matlab was a trivial process. The main function of the Matlab code was to simulate the calculation of the frequency response from the output of a filter if a stepped sinusoid was given as the input to the test filter.

The Matlab code first set up a filter at a given cutoff frequency based on a sampling frequency that was equal to 9.259kHz, the sampling rate of the A/D on the DSK board. Next, a stepped frequency was set up to vary from 100Hz to 3kHz. The generated array of terms of the sinusoid was then sent into the filter. The output array of the filter was then multiplied by the input frequency array and the 90° phase shift of that frequency array to generate the in-phase and quadrature components of the signal. These terms were then summed to find the average value of the sinusoid at the given input frequency. This value was then written to an array, which

contained the frequency response of the filter. The Matlab code for this algorithm can be found in *Appendix* C.

This code performed as predicted and produced the expected frequency response of a known lowpass filter. Even with a frequency step of 100Hz the characteristics of the filter could be seen. A step of 10Hz or smaller yielded the best results. The phase response of the filter was also seen as expected, but it did exhibit a triangular oscillation at frequencies lower than about 40Hz with a step size of 10Hz. Using smaller step frequencies reduced the amplitude of the oscillation.

3.2 Design of the Stepped Sine Wave

The stepped sinusoid designed was basically a choppy chirp function. The generated sinusoid was initialized to a frequency of 300 Hz because our DSP chip makes frequencies below this fairly undetectable. The frequency was designed to step up at 100 Hz increments, until reaching a maximum frequency of 3 Khz**.**

The sampling frequency of the DSP was set at 9259 Hz, so one sample from the 256-sample sine table was read every .108 msec. The generated sinusoid needed to have many cycles at each frequency so an accurate integration could be done (see *Figure 1*). In the DSP code, integration was carried out through 300 cycles at each frequency.

The total number of samples used for each frequency was found by the following relationship:

$$\#\text{ of samples at } f_n = \frac{9259\,(\text{samples / sec})}{f_n\,(\text{cycles / sec})} * (300\text{ cycles})$$

From this equation, at 300 Hz, 9259 samples were used from the lookup table. At 3 Khz, 926 samples were used. This was important to note since the integration was done by summing up all the values retrieved at one frequency. Care had to be taken not to overflow the accumulator with all the additions.

One whole stepping process took about 19 seconds before the frequency was reset to 300 Hz again. Since higher frequency sinusoids do not need as many samples from the sine lookup table as the lower frequency sinusoids, most of this time was spent at lower frequencies. The sinusoid sent to the output can therefore be modeled by the following equation:

$$\text{Stepped sinusoid} = \cos(2\pi f_0 \Delta t_1) + \cos(2\pi (f_0+f_\Delta)\Delta t_2) + \cos(2\pi (f_0+2f_\Delta)\Delta t_3) + \ldots + \cos(2\pi f_e \Delta t_e)$$

Where f_0=300 Hz, f_e = 3 kHz, $f_\Delta \approx$ 100 Hz, and Δt_1, Δt_2, Δt_3,... are time intervals which each start where the previous interval ended and end just before the next interval. The duration of each sinusoid decreases linearly as the frequency increases because the number of samples does and therefore the processing time, which is directly related to the duration of each sinusoid.

The stepped sinewave algorithm can be seen in detail as a flowchart in *Appendix A.*

3.3 Frequency Response Calculation

The frequency response calculation began by reading the value generated at the filter output by using the stepped sinusoid as the input. This value was stored in the A accumulator and multiplied by the cosine term that generated it. This output was shifted down by 17 bits to prevent overflow and was then stored in a temporary variable for use in the next sum calculation. The same procedure was repeated for the input value except it was multiplied by a corresponding sine instead of a cosine.

A check was then performed to see if enough cycles at a given frequency had been generated. If this was false then the program resumed generating the stepped sinusoid. If this was true then the square of the sum of the squares was calculated. Since the square root is a complex calculation, it is not implemented directly with a DSP command. An approximation was used for this calculation.

$$\sqrt{A^2 + B^2} \approx 0.5 \times \min(|A|, |B|) + \max(|A|, |B|) \qquad 3.3.1$$

The output of this calculation produced the frequency response of the filter at the input frequency. This value was then stored in the DSK memory. A detailed flow chart of this algorithm is located in *Appendix* B.

4. Performance

The DSP code to perform the algorithms discussed in Sections 3.2 and 3.3 can be seen in *Appendix* D. The performance of these algorithms in the DSP board was hard to verify. The completed code would calculate and display the frequency response of a test filter, but there was a lot of distortion in frequency response that made it appear very noisy. Some frequency steps would give a smooth response and others would cause a wild fluctuation in the calculated values. This problem was corrected somewhat by turning on sign extension, but it still did not clean up the entire signal.

At lower frequencies the calculated value of the frequency response was smoother than at higher frequencies. A probable cause for this is that at lower frequencies more samples of the filter response are stored and summed. For example, at 300 Hz the number of samples stored for each pass through the lookup table was approximately 30, while at 3 kHz the number of samples stored for each pass was only 3. This could cause major problems for the calculation of the frequency response at higher frequencies and result in the noise that was seen.

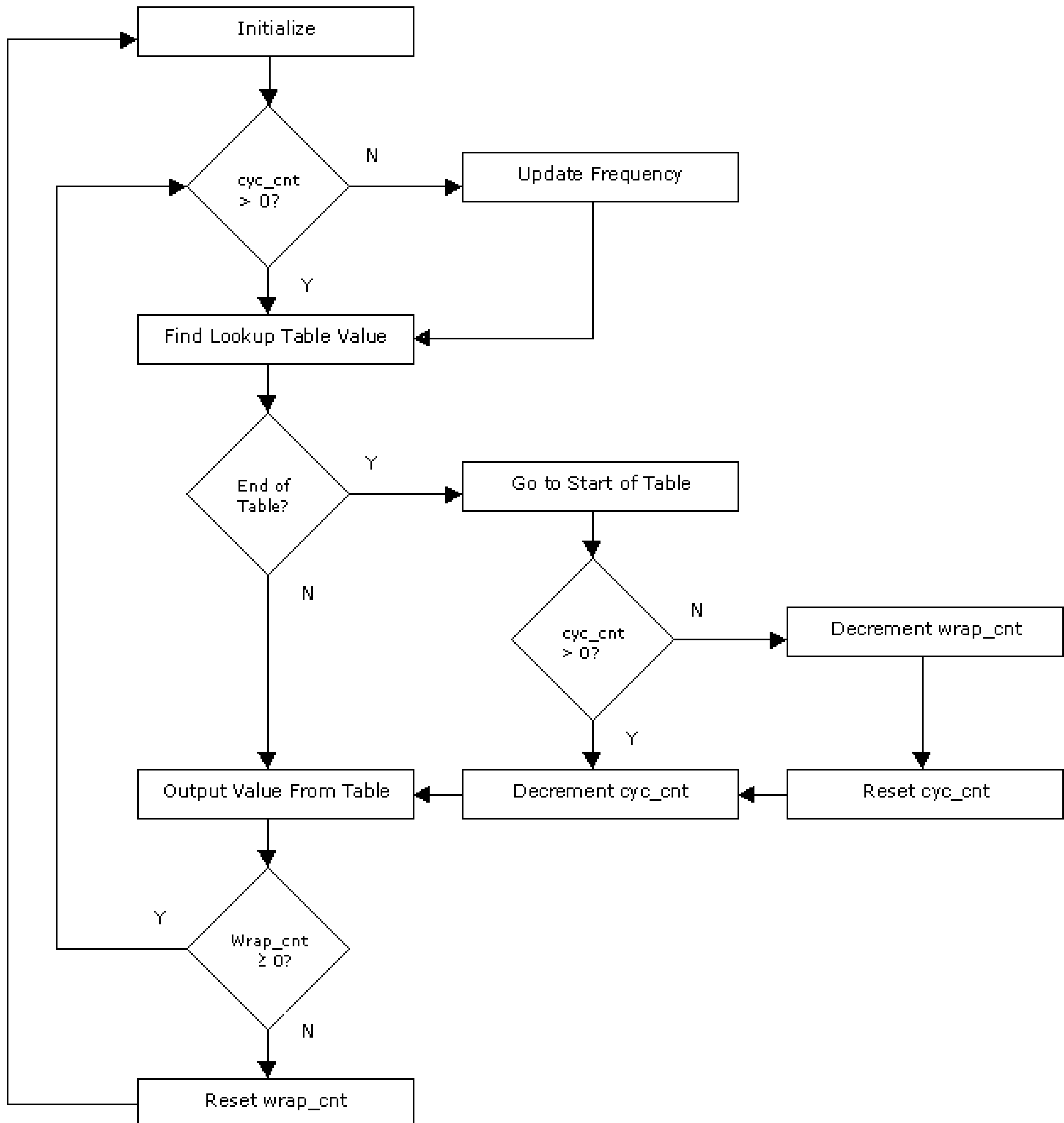
It is also theorized that part of the problem is caused by a limitation of the precision in the DSP board. Using fixed-point math with a limited number of bits causes the calculated values to become truncated through either overflow or underflow. This would generate a noisy output of the frequency response calculations.

5. Conclusion

This project was a complete success in terms of theory and simulation, but had a few bugs in the DSP code that prevented the expected performance. The frequency response of a filter was calculated and displayed with the expected results plus unwanted noise. The general characteristics of the filter such as cut off and roll off could be seen, but exact measurements could not be made because of the excessive noise in the signal. Once this noise is removed and a cleaner signal is achieved this would function as a very nice, cheep network analyzer that could be used for either home projects or other electronic tinkering.

Appendix A

Flow chart of stepped sinewave generation.

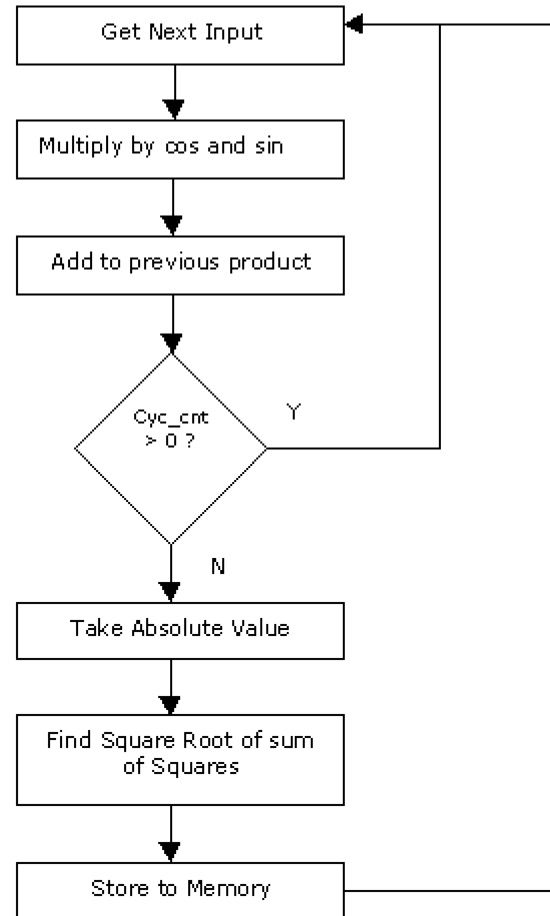


Notes:

"cyc_cnt > 0?" is equivalent to the condition "keep generating more cycles of the current frequency?"
"wrap_cnt ³ 0?" is equivalent to the condition "keep generating a stepped sinewave without resetting the frequency?"

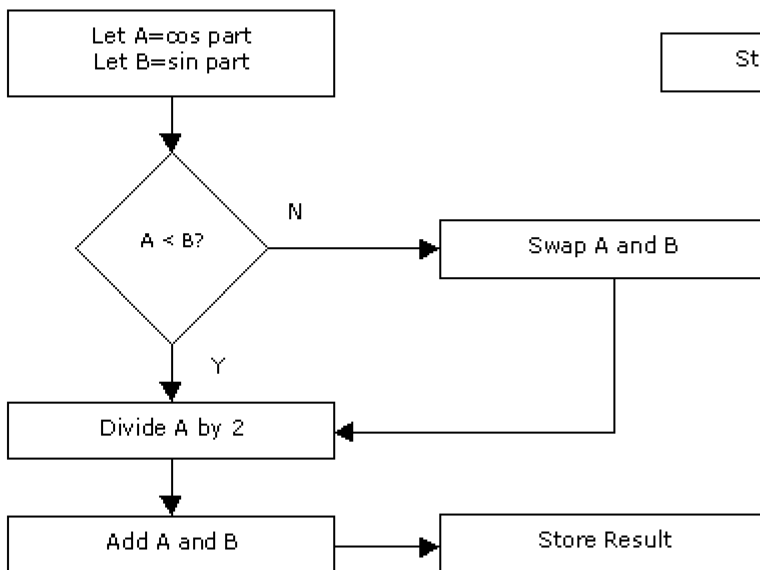
Appendix B

The following flowcharts demonstrate how the amplitude response was determined from the input. Refer to *Figure 1* in the theory section.



The flowchart below shows how the Square root of the sum of the sqaures calculation was done.

$$\sqrt{A^2 + B^2} \approx \frac{1}{2}\min(|A|,|B|) + \max(|A|,|B|)$$



Appendix C

Matlab simulation code

```
% Network Analyzer Simulation
%
% Josh Horton
% Randy Dimmett

close all;
clear all;

fs=9259;
fc=1500;    % cuttoff freq.
Wn=fc/(fs/2)

[B,A]=butter(2,Wn,'high');
%freqz(B,A)

t=0:1/fs:.05;
t1=t(length(t));
p=1; % for linear
Envout=[];
Phaseout=[];

for freq=0:5:3000;
   beta = (freq).*(t1.^(-p));

  c = cos(2*pi * ( beta./(1+p).*(t.^(1+p))));
  s = sin(2*pi * ( beta./(1+p).*(t.^(1+p))));

  ADinput=filter(B,A,c);
  ADI = ADinput .* c;
  ADQ = ADinput .* s;

SUMI=abs(sum(ADI));
SUMQ=abs(sum(ADQ));

realOut=sqrt(SUMI^2+SUMQ^2);
ApproxOut=max([SUMI SUMQ])+min([SUMI SUMQ])/2;
Envout=[Envout ApproxOut];
Phase=atan(SUMI./SUMQ);
Phaseout=[Phaseout Phase];

end

%sound(ADinput);
plot(10*log10(Envout));
figure
freqz(B,A);
figure
plot(Phaseout);
```

Appendix D

Algebraic Assembly Code

```
      .title  "EE301C Project"
      .list
      .width 86
      .length 32000
*
****************************************************************
* Program:  EE301C Project -- Network Analizer
* Authors: Josh Horton and Randy Dimmett
* This File:      main.asm
* External Files:       isrv.asm, acinit.asm script.asm
* Lables Defined: start, loop, ad_isr, da_isr, sine
*                 done, no_wrap
* External Labels Used: acinit
* Label at Entry Point: start
*
****************************************************************
* Set up the chrip
****************************************************************
*
*
fr_start    .set        100   ; Set to starting frequency
*
****************************************************************
* Sections
****************************************************************
*
      .setsect ".text",0x1800,0     ; Program
      .setsect "data",0x0200,1      ; Data
      .setsect "intvecs",0x0180,0   ; Interrupt Vectors
*
****************************************************************
* Set memory locations to store program data
****************************************************************
*
      .sect "data"
deltah            .word       0     ; Stores current delta step value
deltal            .word       0
inch        .word       0     ; Stores current increment value
incl        .word       0
wrapcnt           .word       0     ; Wrap counter
cyc_cnt           .word       0     ; Count number of cycles before
                              ; step increase
prvcos            .word       0     ; Previous cosine value
prvsin            .word       0     ; Previous sine value
cossum            .word       0     ; Temp sum of cosines
sinsum            .word             0     ; Temp sum of sines
tempA       .word             0     ; Temp storage
*
****************************************************************
* Interrupt Service Routine Vectors
****************************************************************
*
```

```
      .sect "intvecs"
      .copy "isrv.asm"
*
***************************************************************
* Main Program
***************************************************************
*
      .sect ".text"
start:
      intm = 1                      ; Turn off interrupts
      DP = #deltah
      @(wrapcnt) = #(1000)
      @(inch) = #0
      @(incl) = #0
      @(deltah) = #(1159603*fr_start/327680);
      @(deltal) = #(1159603*fr_start/10-1159603*fr_start/10);
      @(cyc_cnt) = #10;
      @(prvcos) = #0
      @(prvsin) = #0
      @(cossum) = #0
      @(sinsum) = #0
      @(tempA) = #0
      SP = #0ff0h
      dcall acinit
      DP = #0
      nop
      PMST = #01a0h
      SXM = #1          ; Turn on sine extension mode
      IMR = #240h
      OVM = 0
      ASM = #0
      AR2=#(0x2680)     ; Start of look-up table
      AR4=#(0x2000)     ;Starting address output freq resp.
      repeat(#0xFF)     ; Fill look-up table with 256 words
      (*AR2+)=prog(0xFE00)    ; Move from prog ROM to data RAM
      repeat(#128)      ; Fill look-up table with 256 words
      (*AR2+)=prog(0xFE00)    ; Move from prog ROM to data RAM
      AR2=#(0x2680)     ; Start of look-up table
      intm = 0
loop: nop
      goto loop
*
***************************************************************
* Initialization Routine for Analog Interface Chip
***************************************************************
*
      .copy "acinit.asm"
*
***************************************************************
* A to D Interrupt Service Routine
***************************************************************
*
ad_isr:
*
* These are the registers affected by the ISR -- push them on the stack
* to keep them from disturbing the main program
*
```

```
      push(AR1)
      push(AL)
      push(AH)
      push(AG)
      push(BL)
      push(BH)
      push(BG)
      DP= #0
      nop
      nop
      A = @(TRCV) ; Clear out receive interrupt
      DP = #deltah;
      nop
      nop;
        T=@(prvcos)     ; For multiplying by cosine
        @(tempA)=A
        A=T*@(tempA)
      A=A>>15
      A=A>>2
      B=@(cossum) ; B contains previous sum
        A=A+B
        @(cossum)=A
        T=@(prvsin)     ; For multiplying by sine
        B=T*@(tempA)
      B=B>>15
      B=B>>2
        A=@(sinsum)
        B=B+A
        @(sinsum)=B
*
* Now the sum is finished and stored
* Now do sqrt of the sum of the squares,
*  but only if we have enough cycles stored up.
*
        A = @(cyc_cnt)
      if (AGT) goto donothing
        A=@(cossum)
        B=@(sinsum)
        A=|A|
        B=|B|
*
* Find the max and min values
*
        A=A-B
        if(ALT) goto hypot;
        A=@(sinsum)
        B=@(cossum)
hypot:            ; calculate sqrt of sum of sqrs
        A=@(cossum)
        A=|A|
        B=|B|
*
* Now A should contain minimum
* B should contain maximum
*
```

```
      A=A>>1            ; Divide min by two
        A=A+B;          ; Add 1/2 min and max
        (*AR4+)=A
        nop
        @(cossum)=#0
        @(sinsum)=#0
donothing:
        A=#0
        B=#0
*
***************************************************************
* Chirp
***************************************************************
*
chirp:
      DP = #deltah            ;  Set to chirp page
      B = @(deltah) << 15
      B = B + @(deltal)
      A = @(cyc_cnt)
      if (AGT) goto samestep
      B = B + #0x7fff << 4    ;  Increase step freq
samestep:
      A = B & #0x7FFF
      @(deltal) = A
      A = B << (-15)
      A = A & #0x7FFF
      @(deltah) = A
      A = @(inch) << 15 ; Put the high and low part of the
      A = A + @(incl)         ; increment into the accumulator
      A = A + B
*
* Are we at the end of the table if so wrap around if not don't wrap
*
      B = A << -16;
      B = B - #(0x3FFF);
      if (BLEQ) goto no_wrap
        B = B << 15;
        B = B << 1;
        A = A & #(0x0FFFF);
        A = A | B;
        B = @(cyc_cnt)
        if (BGT) goto cycnorst
          @(wrapcnt) -= 10    ; Increment sample counter
        @(cyc_cnt)=#300;      ; Reset cycle counter
cycnorst:     @(cyc_cnt) -= 1;
no_wrap:                ; If we are not wrapping save counter value
      B = A & #0x7FFF
      @(incl) = B
      B = A << (-15)
      B = B & #0x7FFF
      @(inch) = B
* Find the output value
      B = A << (-11)
      B = B << (-11)
      A = #0x2680 + B   ; Add interpolated value to starting phase
(broken)
      AR2 = A           ; Load the contents of AR2 into A
```

```
      A = *AR2
      nop
      nop
      @(prvcos)=A
      nop
      nop
      AR3=AR2
      B=AR3
      B=B+#64            ; Find the 90 degree shift value
      AR3=B
      B=*AR3
      nop
      nop
        @(prvsin)=B
      nop
      nop
      A = A & #0xFFFC    ; Make 2 LSB 00 for AC01
      TDXR = A           ; Send lower 16 bits to D/A
*
      B=@(wrapcnt)
      if (BGEQ) goto no_reset       ;  Skip if samples < tot_samp
        AR4=#(0x2000);
        @(wrapcnt) = #(1000)  ;  Reset frequency, samp counter
        @(deltah) = #(1159603*fr_start/327680)
        @(deltal) = #(1159603*fr_start/10-1159603*fr_start/327680)
*
no_reset:
*           Return altered CPU registers to original value
*
      BG = pop()
      BH = pop()
      BL = pop()
      AG = pop()
      AH = pop()
      AL = pop()
      AR1 = pop()
*
      return_enable     ; End of ISR routine
      nop
      nop
*
***************************************************************
* D to A Interrupt Service Routine
***************************************************************
*
da_isr:
      return_enable   ; Return to waiting loop
      nop
      nop
*
***************************************************************
* End of File
***************************************************************
*
      .end
```