Naval Research Laboratory

Washington, DC 20375-5320

NRL/MR/5320--94-7456

Implementation of a Real-Time Data Processing System for Radar Bandwidth Extrapolation and Stretch

ATHENA R. CAUL-TOSKIN

Advanced Radar Systems
Radar Division

June 10, 1994

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY (*Leave Blank*)	2. REPORT DATE June 10, 1994	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE Implementation of a Real-Time Data Processing System for Radar Bandwidth Extrapolation and Stretch	5. FUNDING NUMBERS PE -64211N
6. AUTHOR(S) Athena R. Caul-Toskin	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Research Laboratory Washington, DC 20375-5320	8. PERFORMING ORGANIZATION REPORT NUMBER NRL/MR/5320--94-7456
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research 800 North Quincy Street Arlington, VA 22217-5660	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

An existing system using NRL's Senrad radar is being used to demonstrate the utility of wideband waveforms in radar surveillance. This system processes data using a stretch technique to yield high resolution range profiles of the target. To provide some of the benefits of greater bandwidth without implementing the wideband waveforms, it was suggested to add bandwidth extrapolation processing. The addition of this processing necessitated changes to the existing system for real-time processing. The changes included modifying the Volume Surveillance system hardware, constructing an interface between the existing system and the Volume Surveillance processor and modifying the stretch and bandwidth extrapolation software for a new processor. The changes were implemented and the modified hardware operated correctly. The software modifications were tested successfully with recorded and real-time data. The processing times met the time specifications necessary for real-time operation.

14. SUBJECT TERMS Bandwidth Extrapolation Stretch			15. NUMBER OF PAGES 25
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std 239-18
298-102

CONTENTS

IMPLEMENTATION OF A REAL-TIME DATA PROCESSING SYSTEM FOR RADAR BANDWIDTH EXTRAPOLATION AND STRETCH

Introduction

A program is underway at NRL to demonstrate the utility of wideband waveforms in radar surveillance. Current effort has focused on the development of a demonstration system that implements long range target recognition. The system uses NRL's Senrad radar[1] which performs normal air surveillance. On targets of interest, a wideband waveform replaces a few pulses of the normal dwell. Since the range of interest is limited, these pulses are processed using a "stretch" technique to yield high resolution range profiles of the target. Combining range profiles with tracking data makes it possible to classify uncooperative targets.

Usually, increased bandwidth will improve the performance of target classification. However, in practical radar systems, the amount of available signal bandwith is limited. It has been suggested that bandwidth extrapolation processing can provide some of the benefits of a greater bandwidth without the problems of actually implementing the waveforms.[2] Therefore, the objective of this project was to add real-time processing for bandwidth extrapolation to a stretch data processing system.

Background

It is necessary to explain the operation of the existing system in order to show the necessity of this change and to explain the requirements. The existing system captures radar data from the Senrad receiver and does the processing shown in Fig. 1 using a general purpose Hewlett Packard (HP) computer. The scan rate of the

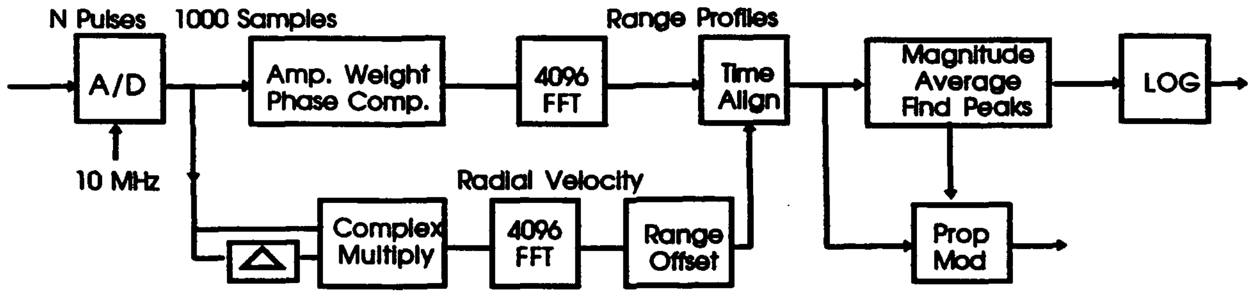

Fig. 1 - Block diagram of stretch data processing

antenna is approximately five seconds and the processing was just completed before data from the next scan was available. Since bandwidth extrapolation requires substantial computation, the system is incapable of running in real-time without some modifications. The modification decided on was to have some of the processing done on a different processor which would interface with the existing system. The new system would capture live data from the radar also and perform the bandwidth extrapolation and stretch processing on this data. The results would

Manuscript Approved April 25, 1994.

be sent to the original HP computer where the processing would be finished and the results would be displayed. The resulting processing times of both the new processor and the HP would allow the process to run in real-time.

The changes necessary to meet this objective included translating the previous code written in the HP Basic programming language by George Linde and his group to the C programming language. The processing was done on a TMS320C30 (C30) digital signal processor (DSP) IC. The introduction of the C30 was most easily achieved by interfacing with the hardware of the Volume Surveillance project. This project is still in progress and there is no published literature on the project to date. The processors used in this system are the same as those used in the Point Defense system.[3,4,5]

The existing hardware system used to do the stretch processing consisted of a NOVA computer, an HP computer and other hardware which will not be discussed here. The NOVA received digitized data from the Senrad receiver and sent it to the HP for processing. The HP performed the stretch processing on a set number of pulses, as well as some other processing, and displayed the results. The information displayed included the range, speed, aspect angle, length, radar cross section and identification of the target, the signal magnitudes of two pulses, range profiles for each pulse, an average range profile, the peak level and range offset from the profiles and the places where propeller modulation was found.

Referring to Fig. 1, a set number of pulses consisting of 1000 samples each is digitized. A complex multiply between the first and last pulse of the data and an FFT are completed. The range offset between these pulses is then determined. The digitized data is also amplitude weighted and phase compensated. An FFT is taken and the range offset found previously is applied to each pulse to time align all of the pulses. To complete the processing, the magnitude average, peaks and any propeller modulation in the range profiles are found. The logarithm of the range profiles is taken and displayed.

Hardware

The hardware for this project consisted basically of three parts: the Volume Surveillance system, the interface and the existing HP system. The Volume Surveillance hardware included a Sun computer, the SP (signal processing) boards and the A/D (analog to digital) converters. The existing HP hardware included the HP computer, NOVA computer and other components which were needed in the existing system but will not be discussed here. Each of the parts will be discussed in turn.

In the Volume Surveillance system, the SP boards (only one of which is used) each contain two SP nodes and a transputer. Each SP node contains one C30. The transputer has four bi-directional communication links and is responsible for routing the data from and to the C30s on the SP board. A separate network of transputers is configured to route data between the Sun computer and the SP boards. The block diagram of the modified Volume Surveillance hardware is shown in Fig. #2. The blocks marked SR (server) are the available SP boards. Only one SP board was used since only one C30 was needed for processing. The blocks marked RT (router) and CS (console) are part of the network of routing transputers.

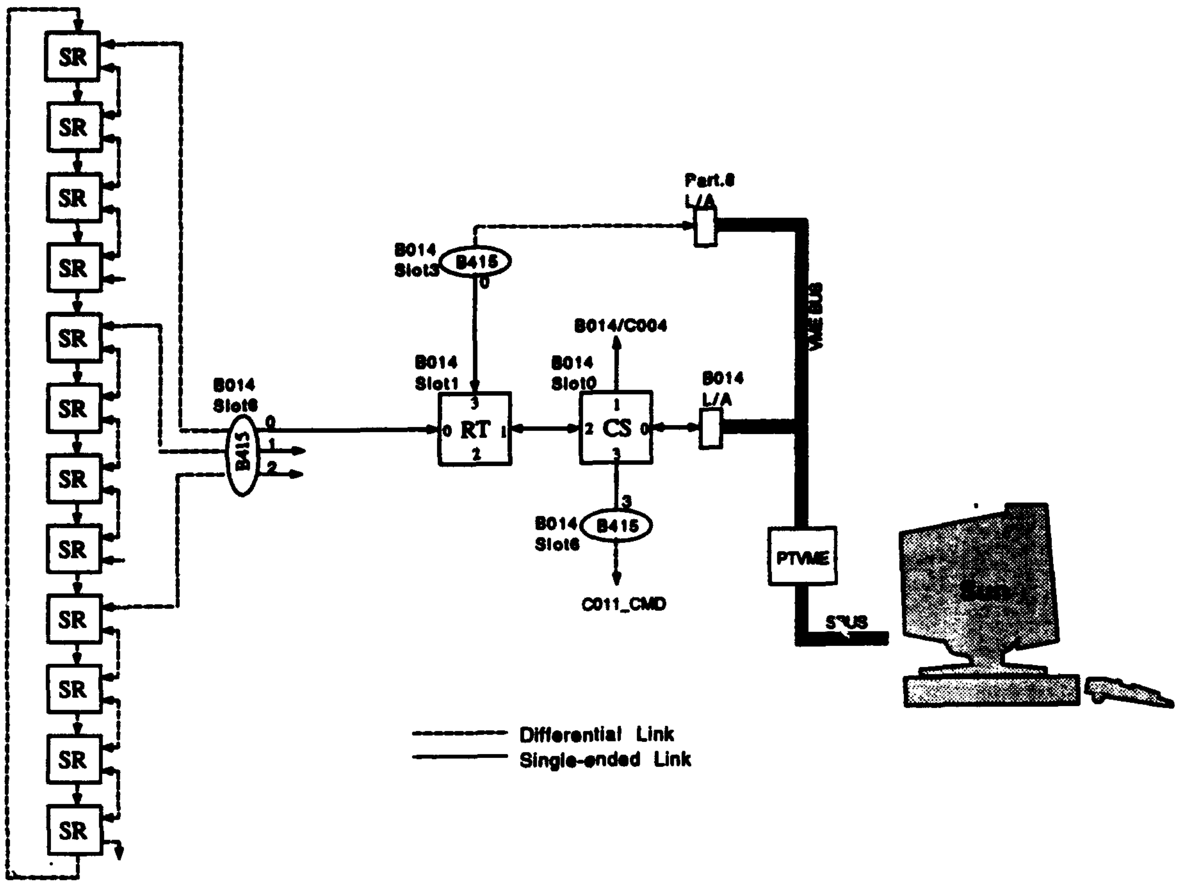

Fig. 2 - Block diagram of modified Volume Surveillance hardware

Some other modifications to the Volume Surveillance hardware were necessary in order to capture the digitized data. The program of the Xilinx field programmable gate array (FPGA) IC on the one SP board used was modified by Mike Livingston. This IC expected an external trigger for each pulse when the radar data was valid. When one of these triggers was received, the Xilinx IC collected and stored 1024 digitized samples from the A/D converters. When the data for all the pulses was ready, the software discussed below was able to directly access the data.

The second modification was a new A/D interface board which was also completed by Mike Livingston. This board buffered the digitized I and Q channel data from the radar and converted it to a 32 bit data format. The hardware used to reset the SP boards was also placed on this board as well as some circuitry to supply the correct clocks to the system.

The interface between the modified Volume Surveillance and the HP hardware consisted of an SBus to IEEE488 (GPIB or HPIB) bus converter installed in the Sun. This converter allowed for direct connection of the Sun and the built-in HPIB bus of the HP through a cable. The HP was set up to be the Controller-in-Charge (CIC) of the bus and, therefore, controlled the data transfer from the Sun. The GPIB bus transfers data at a maximum rate of 1 Mbyte/sec.

The HP still received digitized and other data from the NOVA. Since the C30 only received the digitized data, it could only do a portion of the processing. Therefore, the HP completed the remaining processing. After the C30 completed its processing, it passed the data to be displayed and other data necessary to complete the processing to the HP.

Software

Although the original Basic code was translated to C, some modifications were made for readability, efficiency and to correct for differences between the two languages. Where possible, variables, algorithms and subroutine names were kept the same as in the Basic code. There were also several versions of the software. The only versions covered here will be the original translation and the final working version.

The original bandwidth extrapolation processing translation consisted of two subroutines. The first subroutine is CalcCoeff which calculates the complex filter coefficients. The second subroutine is Lnpred which linearly extends the data in the forward and backward direction to the number of data points needed in the extended waveform. The FFT code used was not a part of the code translated from Basic to C. Two subroutines written by Jim Evins and Russell Scott in C performed the FFT and were used in this processing as well as the stretch processing.

The processing times of the C code routines were found using the C30 emulator. The times for the original bandwidth extrapolation C code translation, which do not include time for data input and output, were as follows:

CalcCoeff	233	ms
Lnpred	164	ms
FFT	4.497	ms
Total Time	401.5	ms

The processing times for the original bandwidth extrapolation code written in Basic were approximately 5 seconds and 3 seconds for the subroutines CalcCoeff and LnPred respectively. Therefore, a substantial improvement was made by the change in hardware and the C language conversion.

The original stretch data processing translation consisted of eight subroutines. The list of subroutines along with a description of what the subroutine does is as follows:

1. Initialize - This subroutine initializes some constants, sets default values and calculates the phase compensation values.
2. Get_File - This subroutine gets the digitized pulse data, performs amplitude weighting and phase compensation on the data and calculates the values of other variables.
3. Signal_Mag - This subroutine calculates the magnitude of two pulses of digitized data and sends this data out to be displayed.
4. Rmotion_Calc - This subroutine performs a complex multiply between the first and last pulse and determines the range offset.
5. Stretch - This subroutine uses the compensated data from Get_File. An FFT is performed on the data and the resulting range profiles are time aligned using the

previously determined range offset. An average range profile is found by magnitude averaging the range profiles of each pulse.

6. Range_Profile - This subroutine sends the logarithm of the range profiles and average profile out to be displayed.
7. Locate_Pmods - This subroutine calculates a threshold used to determine where propeller modulation occurs.
8. Decision - This subroutine is called within Locate_Pmods and produces a target class decision.

The processing times for the original stretch processing C code translation, which do not include time for data input and output, were as follows:

Initialize	26.88	ms
Get_File	45.56	ms
Signal_Mag	4.852	ms
Rmotion_Calc	119.9	ms
Stretch	946.5	ms
Range_Prof	33.85	ms
Locate_Pmods	4.728	ms
Total Time	1.190	s

The processing time for the original stretch processing code written in Basic, which included the eight subroutines listed earlier and other processing, was approximately 5 seconds.

The final version of this software included performing the bandwidth extrapolation on each pulse of data and then the stretch processing on the group of extrapolated pulses. A copy of this software appears in Appendix 1. To meet the time specifications, the processing on each group of pulses must be completed within the time taken for one scan of the radar which is approximately five seconds. For the case of seven pulses, the time taken for processing is equal to (7 * 0.4015 s) + 1.190 s = 4.0005 s.

As mentioned in the hardware section of this report, the C30 only did a portion of the processing. Therefore, some subroutines and code from the original translation are omitted in the final version of the software. Since the C30 does not receive data from the NOVA, some needed data was assumed constant by the C30. The data that was assumed constant was stored in two files, one for all the variables and one for the phase compensation data. These files can be edited on the Sun computer to change the constants. Any changes made to items 1 through 6 or the number of phase compensation values will also necessitate changes within the code. The data and its constant value was as follows:

1. Number of points selected from full waveform	333
2. Number of filter coefficients	150
3. Number of points in extended waveform	1000
4. Number of pulses	7
5. Number of samples per pulse	1000
6. Number of points in FFT	4096
7. Sampling rate (MHz)	10
8. Bandwidth	200
9. Pulse width	100

10. Phase compensation data (1000 floating point numbers)

The final version of the bandwidth extrapolation code did not include taking the FFT. The final version of the stretch processing code did not include any of the code relating to variables that were not available to the C30. The subroutines Locate_Pmods and Decision were removed also. Data was converted to 16 bit two's complement integers before being sent to the displaying HP as a stream of integers. Floating point numbers were converted to integers by multiplying by a scaling factor and truncating the fractional portion of the numbers. The data sent to the HP and the number of integers it was composed of, in parentheses, appears below in the order in which it was sent:

1. Signal magnitudes of two pulses (1000 per pulse = 2000) - scaling factor of 32.
2. Range profile of each pulse (2000 per pulse = 14000) - scaling factor of 32768. (linear data)
3. Average range profile (2000) - scaling factor of 32768. (linear data)
4. Velocity (1) - scaling factor of 100.
5. Magnitude sum of one pulse (1) - scaling factor of 100.
6. Range offset (variable name Pk_at) (1).
7. Peak dwell (1).

Notes:

- Each array (1, 2 and 3) are sent from the lowest to the highest index. The indices for the signal magnitudes are 1 to 1000. The indices for the range profiles and average profile are -999 to 1000.
- Arrays marked as linear data are not ready to be displayed immediately; the logarithm of the array is displayed.

Due to hardware specifications and the method of transmitting data, pairs of 16 bit integers were packed into 32 bit words in the C30. A byte reversal also was performed on the data since the transputer uses "little endian" byte ordering and the HP uses "big endian" byte storage. In other words, the transputer stores the most significant byte of a word in a higher memory address than the least significant byte whereas the HP stores the most significant byte in a lower memory address. During transmission, one byte at a time was sent starting with the most significant byte. Therefore, the receiving system waited to receive two bytes for each integer. The software responsible for transmitting the data across the IEEE 488 bus was run on the Sun. When all of the data was received by the Sun and ready to be sent, the Service Request (SRQ) line was asserted. The HP responded by allowing the Sun to send the data over the bus.

The remaining software to be discussed is the code running on the transputers and the main program to control the processing on the C30. A console program running on the transputer resident in the Sun was responsible for reading the data from the files and sending it out to the C30. Several transputers along the network were responsible for passing the data in the correct path to reach the transputer on the SP board or to pass the data to the GPIB bus converter for transmission to the HP. The main program on the C30 was responsible for waiting to receive the data read from the data files. The C30 polled one of its ports for an indication that the digitized data for the group of pulses was ready. After receiving the digitized data, the bandwidth extrapolation and stretch processing occurred and the necessary data was sent out. The process started again with the C30 waiting for digitized data to be ready and repeats until interrupted externally.

Status

Due to time limitations, this real-time processing system was not fully implemented. The software was fully translated from the Basic language to the C language and was tested for correct performance. The software was ported to the final system located at the Chesapeake Bay Detachment of NRL from the testing site at NRL Washington. The modified Volume Surveillance hardware successfully received digitized data from the Senrad receiver which was routed to the C30 processor. The C30 correctly performed the processing necessary within the time specifications and sent data to the Sun computer to be relayed to the HP computer. Some communication between the Sun and HP computer across the IEEE 488 bus was implemented but was not fully functional. The only remaining task is to modify software on the HP computer which would correct the problems with the interface.

Summary

An existing system using NRL's Senrad radar is being used to demonstrate the utility of wideband waveforms in radar surveillance. This system processes data using a stretch technique to yield high resolution range profiles of the target. To provide some of the benefits of greater bandwidth without implementing the wideband waveforms, it was suggested to add bandwidth extrapolation processing. The addition of this processing necessitated changes to the existing system for real-time processing. The changes included modifying the Volume Surveillance system hardware, constructing an interface between the existing system and the Volume Surveillance processor and modifying the stretch and bandwidth extrapolation software for a new processor. The changes were implemented and the modified hardware operated correctly. The software modifications were tested successfully with recorded and real-time data. The processing times met the time specifications necessary for real-time operation.

Acknowledgements

The author would like to credit those involved in this project as mentioned in this report: George Linde and Mike Livingston as well as Carl Platis and Jim Evins. Carl Platis maintained the original Senrad hardware and made the necessary modifications for the implementation of the real-time system. Special appreciation is due Jim Evins for his continuous technical support.

References

1. Linde, G.J.: The Senrad Experiment System (U), *NRL Report 8615, Naval Research Laboratory*, Washington, D.C. , Sept. 28, 1982.

2. Bowling, S.B., Group 35: Linear Prediction and Maximum Entropy Spectral Analysis for Radar Applications, S.B. Bowling, Group 35; *MIT Lincoln Laboratory, Project Report RMP-122*, May 24, 1977.

3. Alter, J.J., Evins, J.B., Davis, J.L. and D.L. Rooney: A Programmable Radar Signal Processor Architecture, *1991 IEEE National Radar Conference*, March 1991.

4. Evins, J.B., Alter, J.J. and J.P. Letellier: NRL FLEX Processor for Radar Signal Processing, *SPIE's 1991 International Symposium on Optical Applied Science and Engineering*, July 1991.

5. Alter, J.J., Evins, J.B., Popick, G.L., Scott, R.A. and J.P. Letellier: NRL FLEX Radar Signal Processing Architecture - An Update, *SPIE's 1992 International Symposium on Optical Applied Science and Engineering*, July 1992.

Appendix A

Bandwidth Extrapolation and Stretch Data Processing Software

```
/*******************************************************************************
 *      Title:      Stretch Program
 *      File Name:  str.c
 *      Date:       7/28/93
 *
 *      Purpose:    To do stretch processing on some number of pulses of
 *                  1000 samples each. (See block diagram)
 *
 *      Target HW:  TMS320C30 Signal Processor Nodes
 *
 *      Author(s):  George Linde
 *                  Athena Caul (rewrote Basic program in C)  7/28/93
 *
 *      Revision:   Test code reading Chan data from files.
 *******************************************************************************/
/*  Note: This version has arrays numbered starting from 0.
          To be used for creating C30 code                                    */

#include <stdlib.h>
#include <math.h>
#include "hw_sp.h"
#include "dmsg.h"
#include "msg.h"
#include "dp.h"
#include "dpprint.h"
#include "complex.mac"
/* includes declaration:  struct { float i, q; } complex;  */
#include "fft.h"
#include "xilinx.h"
#include "radar.h"

/*****************  DEFAULT VALUES AND CONSTANTS  ************************/
#define NPTS      333           /* # of points selected from full waveform*/
#define NCOEFF    150           /* # of filter coefficients */
#define NEXT     1000           /* # of points of extended array */

#define NPTFFT  4096            /* # of points in FFT */
#define NPULSES    7            /* Default # of pulses */
#define NSAMP   1000            /* # of samples */
#define BW       200            /* signal bandwidth */
#define PW       100            /* radar pulse width */
#define SAMPRT  1.E7            /* sampling rate in Hertz */

#define FORWARD    0            /* Direct Fourier Transform */
#define REVERSE    1            /* Inverse Fourier Transform */

/************ GLOBAL VARIABLES ***********************************************
 *  defined global for use of external functions without these being placed
 *  on stack (don't need to pass, initialized to zero)
 *****************************************************************************/
  int Chan_1[8][1000], Chan_2[8][1000];
  float Phase[1000], Spar[12];
  float Si[1000], Co[1000], I_data[8][1000], Q_data[8][1000];
  int M, N, Ns, Np, Pk_at, Pk_dwell;
  float P2, Pv, Mag_sum, Vel;
```

```
/****************************************************************************
 *   CalcCoeff calculates the complex filter coefficients.
 *         Inputs are: NPTS, NCOEFF, Xi[].
 ****************************************************************************/
static void CalcCoeff(int Npts, int Ncoeff, float Pm[], float *Po,
           complex X[], complex A[])
{
  complex *Aa = (complex *) malloc ( 550 * sizeof(complex) ); /*work array*/
  complex *B1 = (complex *) malloc ( 550 * sizeof(complex) ); /*work array*/
  complex *B2 = (complex *) malloc ( 550 * sizeof(complex) ); /*work array*/
  complex Xnom, Temp, Temp2;
  int It, M, Mm1, Nmm, Nm1;
  float Power, Den;

  *Po = 0.0;
  for (It = 0; It < Npts; It++)
    *Po += ( CMAG2( X[It] ) );
  *Po /= Npts;
  Nm1 = Npts - 1;
  B1[0] = X[0];
  B2[Nm1 - 1] = X[Nm1];
  for (It = 1; It < Nm1; It++) {
    B1[It] = X[It];
    B2[It - 1] = X[It];
  }
  for (M = 1; M <= Ncoeff; M++) {
    Mm1 = M - 1;
    Nmm = Npts - M;
    if ( M != 1 ) {
      for (It = 1; It <= Mm1; It++)
        Aa[It-1] = A[It-1];
      for (It = 1; It <= Nmm; It++) {
        CONJ( Aa[Mm1-1], Temp2 );
        CMULT( Temp2, B2[It-1], Temp );
        CSUB( B1[It-1], Temp, B1[It - 1] );
        CMULT( Aa[Mm1-1], B1[It], Temp );
        CSUB( B2[It], Temp, B2[It-1] );
       }
    }
    Xnom.i = Xnom.q = Den = 0.0;
    for (It = 1; It <= Nmm; It++) {
      CONJ( B1[It-1], Temp2 );
      CMULT( B2[It-1], Temp2, Temp );
      CADD( Xnom, Temp, Xnom );
      Den += ( CMAG2(B1[It-1]) + CMAG2(B2[It-1]) );  /*CADD(Den,Temp,Den)*/
    }
    if ( Den == 0.0 )
      A[Mm1].i = A[Mm1].q = 0.0;
    else {
      CDIVS( Xnom, Den, Temp );
      A[Mm1].i = 2.0 * Temp.i;     /* CMULT( Two, Temp, A[Mm1] ); */
      A[Mm1].q = 2.0 * Temp.q;
    }
    Power = *Po;
    if ( M > 1 )
      Power = Pm[M - 2];
    Pm[Mm1] = Power * ( 1.0 - CMAG2( A[Mm1] ) );
    if ( M != 1 )
      for ( It = 1; It <= Mm1; It++ ) {
        CONJ( Aa[Mm1 - It], Temp2 );
        CMULT( A[Mm1], Temp2, Temp );
        CSUB( Aa[It-1], Temp, A[It-1] );
      }
  }
  free ( Aa );
  free ( B1 );
  free ( B2 );
}
```

```
/*****************************************************************************
 *   Lnpred linearly extends the complex data X from N1 points to N2
 *      points.  The original data positioned in the first N1 elements of
 *      array X, are shifted to the middle of X.  Forward and backward
 *      predictions are done until the total # of data points is N2.
 *   Note: the N2 points contain the original N1 points.
 ****************************************************************************/
static void Lnpred(int N1, int N2, int Ncoeff, complex X[], complex A[])
{
  int I, J, K, L1, L2, N3;
  complex Temp, Temp2;

  L1 = N2/2 - N1/2;                  /* set up limits for loops */
  L2 = N2/2 + N1/2;
  if ( (N1 % 2) == 1 )  {
    L1 += 1;
    L2 += 1;
  }
  for (I = 0; I < N1; I++ ) {       /* shift original data to middle of X[] */
    J = N1 - I - 1;
    K = L2 - I - 1;
    X[K] = X[J];
  }

  N3 = N2 - L2;                      /* Do forward predictic'. '/
  for (I = 0; I < N3; I++ ) {
    J = L2 + I;
    X[J].i = X[J].q = 0.0;
    for (K = 0; K < Ncoeff; K++ ) {
      CMULT( A[K], X[J-K-1], Temp );
      CADD( X[J], Temp, X[J] );
    }
  }

  for (I = 0; I < L1; I++ ) {       /* Do backward prediction */
    J = L1 - I - 1;
    X[J].i = X[J].q = 0.0;
    for (K = 0; K < Ncoeff; K++ ) {
      CONJ( A[K], Temp2 );
      CMULT( Temp2, X[J+K+1], Temp );
      CADD( X[J], Temp, X[J] );
    }
  }
}
```

```
/*************************************************************************
 *   Initialize
 *************************************************************************/
void Initialize( )
{
  int I, J;
  float K, Sl, Ys, Y1, Phr, G[4];
  double log10(), sin(), cos();

  Np = NPULSES;                     /* Number of pulses */
  Ns = NSAMP;                       /* Number of samples */
  Sl = BW / PW;                     /* Chirp slope */
  P2 = (SAMPRT / Sl) /2033.4;       /* Scale in feet */
  M = (int) ( log10((float) NPTFFT) / log10(2.0) + 0.5 );
  N = 1 << M;                       /* N = points in FFT = 2^M */

  G[1] = 0.29265;                   /* phase compensation */
  G[2] = -0.0157838;
  G[3] = 0.00218104;
  for (J = 0; J < Ns; J++ ) {       /* read in backwards */
    K = ((float)J - ((float) Ns)/2.) / ((float) Ns);
    Ys = 0.0;
    Ys = G[1] * cos(K*2*M_PI) + G[2] * cos(K*4*M_PI) + G[3] * cos(K*6*M_PI);
    Y1 = 0.6 + 1.2 * Ys;
    Phr =  (- Phase[J]) * M_PI / 720.0;       /* (-tp)/4 in radians */
    Si[J] = sin(Phr) * Y1;
    Co[J] = cos(Phr) * Y1;
  }
}

/*************************************************************************
 *   Get_File gets the stretch data and performs amplitude weighting
 *       and phase compensation (using values calculated in Initialize).
 *************************************************************************/
void Get_File( )
{
  int I, J, tp1, tp2;
  float S, Sl, Bw, Pw;
  double sqrt();

  Np = Spar[3];
  Ns = Spar[4];
  N = Spar[5];
  S = Spar[6];                      /* sampling rate in MHz */
  Bw = Spar[7];
  Pw = Spar[8];
  P2 = S * 1000000;                 /* sampling rate in Hz */
  Sl = Bw / Pw;                     /* chirp slope */
  P2 = (P2 / Sl) / 2033.4;             /* scale in feet */

  for ( I = 0; I < Np; I++ ) {      /* read all pulses in dwell */
    for (J = 0; J < Ns; J++ ) {     /* read in backwards to correct error */
      tp1 = Chan_1[I][J] / 28.;
      tp2 = Chan_2[I][J] / 28.;
      Chan_1[I][J] = tp1;
      Chan_2[I][J] = tp2;
      I_data[I][J] = ( ((float)tp1) * Si[J] + Co[J] * ((float)tp2) )/1000.0;
      Q_data[I][J] = ( ((float)tp2) * Si[J] - Co[J] * ((float)tp1) )/1000.0;
    }
  }
}
```

```
/***********************************************************************
 *   Signal_Mag calculates the signal magnitude of the I and Q data
 *       and plots it to a file.
 ***********************************************************************/
void Signal_Mag( dmsg )
  DMsg *dmsg;
{
  int  I, Yt, Yp, Y, cnt;
  float *Mag1 = (float *) malloc ( 1000 * sizeof(float) );
  float *Mag2 = (float *) malloc ( 1000 * sizeof(float) );
  double sqrt();

  for (I = 0; I < Ns; I++ ) {
    Yt = Chan_2[3][I];
    Yp = Chan_1[3][I];
    Mag1[I] = (float) (sqrt((float) (Yt*Yt + Yp*Yp)) * 1.41);
    Yt = Chan_2[Np-1][I];
    Yp = Chan_1[Np-1][I];
    Mag2[I] = (float) (sqrt((float) (Yt*Yt + Yp*Yp)) * 1.41);
    if ( I < 20 )
      DP_Printf(" Mag1[%d] = %d\n", I, (int) (Mag1[I] * 32);
  }

  cnt = 0;
  for ( I = 0; I < Ns; I = I + 2 ) {
    Y = (int) (Mag1[I] * 32);
    Yt = ( (Y << 8) & 0xFF00 ) | ( (Y >> 8) & 0x00FF );
    Y = (int) (Mag1[I+1] * 32);
    Yp = ( (Y << 8) & 0xFF00 ) | ( (Y >> 8) & 0x00FF );
    dmsg->sigmag.mag[cnt++] = ( (Yp << 16) | (Yt & 0xFFFF) );
  }
  for ( I = 0; I < Ns; I = I + 2 ) {
    Y = (int) (Mag2[I] * 32);
    Yt = ( (Y << 8) & 0xFF00 ) | ( (Y >> 8) & 0x00FF );
    Y = (int) (Mag2[I+1] * 32);
    Yp = ( (Y << 8) & 0xFF00 ) | ( (Y >> 8) & 0x00FF );
    dmsg->sigmag.mag[cnt++] = ( (Yp << 16) | (Yt & 0xFFFF) );
    if ( I > 996 )
      DP_Printf("sigmag[%d]=%d   sigmag[%d]=%d\n",I,(int)(Mag2[I]*32),
                I+1,(int)(Mag2[I+1] * 32));
  }
  DP_WriteOut( DMSG_TYPE_SIGMAG_DATA,
               sizeof(DMsg_SigmagData), (Msg *)dmsg );

  Mag_sum = 0.0;
  for (I = 0; I < Ns; I++ )
    Mag_sum += Mag1[I];
  Mag_sum /= ((float) Ns);
  free( Mag1 );
  free( Mag2 );
}
```

```
/*************************************************************************
 *    Rmotion_Calc finds the product of two pulses and determines the
 *          range offset.
 *************************************************************************/
void Rmotion_Calc( )
{
  int I, J, M, Lmi;
    /* Subscript of Yr is actually -128 to 127 so subtract 128 */
  float Yoffset, Pvf, X1, X2, Y1, Y2, Tim;
  float *Yr = (float *) malloc ( 257 * sizeof(float) );
  complex *tbl = (complex *) malloc ( 2050 * sizeof(complex) );
    /* Chan is R_float and I_float */
  complex *Chan = (complex *) malloc ( 4096 * sizeof(complex) );
  double log10(), sqrt();

  for ( I = 0; I < 4096; I++ )
    Chan[I].i = Chan[I].q = 0.0;
  for ( I = 3; I < 997; I++ ) {
    X1 = ((float)Chan_1[0][I]) / 1000.;
    X2 = ((float)Chan_1[Np-1][I]) / 1000.;
    Y1 = ((float)Chan_2[0][I]) / 1000.;
    Y2 = ((float)Chan_2[Np-1][I]) / (-1000.);
    Chan[I].i = X1*X2 - Y1*Y2;
    Chan[I].q = X2*Y1 + X1*Y2;
  }

  M = 12;                        /*  N = 2^M points in the FFT = 4096 */
  I = 1 << M;
  fft_init( I, FORWARD, tbl );           /* FFT_Floating(); */
  /* complex array Chan is used for both input and output */
  fft2( Chan, I, FORWARD, tbl );

  for ( I = 128; I <= 255; I++ )        /* from 0 to 127 */
    Yr[I] = 20.0 * ((float) log10( CMAG(Chan[I-128]) + 1.0E-15 ));
  for ( I = 0; I <= 127; I++ )          /* from -128 to -1 */
    Yr[I] = 20.0 * ((float) log10( CMAG(Chan[3968+I]) + 1.0E-15 ) );

  Lmi = 0;                              /* find maximum in Yr[] */
  Yoffset = Yr[0];
  for ( I = 1; I <= 255; I++ )          /* from -128 to 127 */
    if ( Yr[I] > Yoffset ) {
      Yoffset = Yr[I];
      Lmi = I;
    }
  Tim = (Np - 1) * 0.0031284;
  if ( (Lmi > 255) || (Lmi < 2) )
    Lmi = 128;
  X1 = Yr[Lmi-1] - Yoffset;
  X2 = Yr[Lmi+1] - Yoffset;
  Y1 = Yr[Lmi] - Yoffset;
  Pv = ((float)Lmi)-128. + 0.5 * (X1 - X2) / (X1 + X2 - 2*Y1);
  Pvf = Pv * P2 / N;
  Vel = Pvf / Tim * 3600. / 6076.1155;   /* round this to two digits */
  free( Yr );
  free( tbl );
  free( Chan );
}
```

```
/*************************************************************************
 *   Stretch uses the compensated data from Get_file. Stretch performs an
 *       FFT, time alignment using values from Rmotion_Calc and does a
 *       magnitude average on the range profile.
 *************************************************************************/
void Stretch( R_profile, Profile_avg )
  float R_profile[][2401], Profile_avg[];
{
  int I, J, L, M, Last_dwell, Vel_offset, T_offset;
  float max, Dwell_mag, sum, X;
  float *R_tmp = (float *) malloc ( 4096 * sizeof(float) );
  complex *tbl = (complex *) malloc ( 2050 * sizeof(complex) );
  /* data is R_float and I_float */
  complex *data = (complex *) malloc ( 4096 * sizeof(complex) );
  double sqrt(), log10();

  Last_dwell = 0;
  for ( I = 0; I < Np; I++ ) {
    for ( J = 0; J < Ns; J++ ) {
      data[J].i = Q_data[I][J];
      data[J].q = I_data[I][J];
    }
    for ( J = Ns; J < 4096; J++ )
      data[J].i = data[J].q = 0.0;

    M = 12;                            /*  N = 2^M points in the FFT = 4096 */
    L = 1 << M;
    fft_init( L, FORWARD, tbl );       /* FFT_Floating(); */
    /* complex array data is used for both input and output */
    fft2( data, L, FORWARD, tbl );

    /* data[0-4095] = R_tmp[2048-4095,0-2047] originally [1-2048,-2047-0] */
    for ( J = 0; J < 2048; J++ )
      R_tmp[J+2048] = CMAG( data[J] );    /* from 2048 to 4095 */
    for ( J = 2048; J < 4096; J++ )
      R_tmp[J-2048] = CMAG( data[J] );    /* from 0 to 2047 */

    if ( I == 0 ) {                    /* locate peak & velocity offset */
      X = 0;
      max = R_tmp[0];                  /* locate max R_tmp value */
      for ( J = 1; J <= 4095; J++ )    /* from -2047 to 2048 */
        if ( R_tmp[J] > max ) {
          max = R_tmp[J];
          X = J;
        }
      Pk_at = X - 2047;
    }
    Vel_offset = (int) (0.5 + Pv / 6. * (I+1));
    T_offset = 0;
    T_offset = Vel_offset + Pk_at;
    if ( I == 6 )
      DP_Printf("- Str - Voff %d   Pk_at %d\n",Vel_offset, Pk_at);

    for ( J = 0; J <= 2400; J++ )
      R_profile[I][J] = R_tmp[847+T_offset+J];   /* offset data array */
    Dwell_mag = 0.0;
    for ( J = 0; J <= 4095; J++ )
      Dwell_mag += R_tmp[J];
    if ( Dwell_mag > Last_dwell ) {   /* save dwell # of peak dwell to */
      Last_dwell = Dwell_mag;         /*   use in rcs measurement */
      Pk_dwell = I;
    }
  }
  free( R_tmp );
  free( tbl );
  free( data );

  /* averaged - sum total of profiles divided by number of pulses */
  for ( J = 0; J <= 2400; J++ ) {
    sum = 0.0;
    for ( I = 0; I < Np; I++ )
      sum += R_profile[I][J];
    Profile_avg[J] = sum / ((float) Np);
  }
}
```

```
/*************************************************************************
 *   Range_Profile plots the range profile for each pulse separately
 *       and the profile average of all the pulses.
 *************************************************************************/
void Range_Profile( R_profile, Profile_avg, dmsg )
  float R_profile[][2401], Profile_avg[];
  DMsg *dmsg;
{

  int I, J, a,b,c,d;
  float X, Y;
  double log10();

  DP_Printf("PEAK LEVEL : %6.1f dB\n",(float) (20.*log10(Profile_avg[1200])));
  DP_Printf("RANGE OFFSET :  %5d FT\n", (int) (((float)Pk_at) * 0.6) );

  d = 0;
  for ( I = 0; I < Np; I++ )
    for ( J = -999; J <= 1000; J = J + 2 ) {
      c = (int) ( R_profile[I][J+1200] * 32768);
      a = ( (c << 8) & 0xFF00 ) | ( (c >> 8) & 0x00FF );
      c = (int) ( R_profile[I][J+1201] * 32768);
      b = ( (c << 8) & 0xFF00 ) | ( (c >> 8) & 0x00FF );
      dmsg->rgprof.rg[d++] = ( (b << 16) | (a & 0xFFFF) );
    }
  DP_WriteOut( DMSG_TYPE_RGPROF_DATA,
               sizeof(DMsg_RgprofData), (Msg *)dmsg );

  d = 0;
  for ( J = -999; J <= 1000; J = J + 2 ) {
    c = (int) ( Profile_avg[J+1200] * 32768);
    a = ( (c << 8) & 0xFF00 ) | ( (c >> 8) & 0x00FF );
    c = (int) ( Profile_avg[J+1201] * 32768);
    b = ( (c << 8) & 0xFF00 ) | ( (c >> 8) & 0x00FF );
    dmsg->avgprofpl.avg[d++] = ( (b << 16) | (a & 0xFFFF) );
  }
}

/************************************************************************
 *  c30_init is copied from /home/volume/src/c30/main.c
 *  This function removes wait states and enables the cache
 ************************************************************************/
static void c30_init()
{
  *PRIMARY_CTL &=   0x00001f1f;          /* force 0 wait states P.Bus */
  *PRIMARY_CTL &=   0x000000ff;          /* No bank switching */
  *EXPANSION_BUS &= 0X0000001f;          /* force 0 wait states E.Bus */
  *ADDR_S1_XMIT_CNTL_REG = 0x2;  /* Disable Ext Rom, enable Ram */
  asm("   or  800h, st" );       /* enable cache */
}
```

```
main()
{
  static complex Xi[1250];
  static float Pm[550];           /*array of updated error power*/
  static float Po;                /*real variance of the data*/
  static complex A[550];          /*array containing filter coefficients*/
  static int J, N1, Ncoeff, N2, Of,a,b,c;
  static int I, ct1, ct2, ct3, count, cnt;
  static int k, l, i_sample, q_sample, *intptr;
  static float Profile_avg[2401], R_profile[9][2401];
  static Radar_CaptureBuffer *rcb;
  DMsg *dmsg = (DMsg *) malloc( sizeof(DMsg) );
  Msg_Length length;
  Msg_Type type;

  c30_init();
  Xilinx_Init();                            /* Initialize Xilinx interface */

  I = ct2 = count = 0;
  ct1 = ct3 = 999;
  *ADDR_LED_PORT = ~0x03;

  for(;;) {
    switch ( type = DP_ReadIn( &length, (Msg *)dmsg ) ) {

      case DMSG_TYPE_XILINX:
        handle_xilinx ( (DMsg_Xilinx *) dmsg );
        rcb = Radar_Init();                 /* Initiliaze DMA interface */
        DP_Printf( "SP: INFO: Xilinx program received and handled.\n" );
        break;

      case DMSG_TYPE_PHASE_DATA:
        Phase[ct1--] = dmsg->phdata.p;    /* read in backwards temp */
        *ADDR_LED_PORT = ~(++count);
        if ( ct1 < 0 )
        break;

      case DMSG_TYPE_SPAR_DATA:
        Spar[ct2++] = dmsg->spdata.sp;
        *ADDR_LED_PORT = ~(count++);
        if (ct2 == 9) {
          Np = Spar[3];
          Ns = Spar[4];
          DP_Printf("All of the Spar data received\n");
        }
        break;

      case DMSG_TYPE_CHAN_DATA:
        Chan_1[I][ct3] = dmsg->chdata.one;     /* read in backwards temp */
        Chan_2[I][ct3--] = dmsg->chdata.two;
        *ADDR_LED_PORT = ~(count++);
        if ( (ct3 < 0) && (I < Np) ) {
          I++;
          ct3 = 999;
        }

        if ( (ct1 < 0) && (ct2 == 9) && (I == Np) ) {

          N1 = Spar[0];             /* Pick middle N1 points */
          Ncoeff = Spar[1];
          N2 = Spar[2];             /* # of points of extended array */
          DP_Printf("The number of data points is %d\n",NPTS);
          DP_Printf("The number of coefficients is %d\n\n",NCOEFF);
          for ( cnt = 0; cnt < 1; cnt++ ) { /* infinite loop eventually */

            for ( I = 0; I < Np; I++ ) {
              Po = 0.0;
              Of = N1 / 2;
              if ( (N1 % 2) == 1 )
                Of -= 2;             /* To match original waveform location */
              for ( J = 0; J < N1; J++ ) {
                Xi[J].i = (float) (Chan_1[I][ N2/2 - Of + J ] );
                Xi[J].q = (float) (Chan_2[I][ N2/2 - Of + J ] );
              }

              CalcCoeff(NPTS, NCOEFF, Pm, &Po, Xi, A);
              Lnpred( N1, N2, NCOEFF, Xi, A );
              for ( J = 0; J < Ns; J++ ) {
                Chan_1[I][J] = (int) (Xi[J].i);
                Chan_2[I][J] = (int) (Xi[J].q);
              }
            }

            Initialize( );
            Get_File( );
            Signal_Mag( dmsg );
            Rmotion_Calc( );
            Stretch( R_profile, Profile_avg );
            Range_Profile( R_profile, Profile_avg, dmsg );

            c = (int) (Vel * 100);
            a = ( (c << 8) & 0xFF00 ) | ( (c >> 8) & 0x00FF );
            c = (int) ( Mag_sum * 100 );
            b = ( (c << 8) & 0xFF00 ) | ( (c >> 8) & 0x00FF );
            dmsg->avgprofpl.velmag = ( (b << 16) | (a & 0xFFFF) );

            a = ( (Pk_at << 8) & 0xFF00 ) | ( (Pk_at >> 8) & 0x00FF );
            b = ( (Pk_dwell << 8) & 0xFF00 ) | ( (Pk_dwell >> 8) & 0x00FF );
            dmsg->avgprofpl.peaks = ( (b << 16) | (a & 0xFFFF) );
            *ADDR_LED_PORT = ~0x55;
            DP_WriteOut( DMSG_TYPE_AVGPROF_PL_DATA,
                         sizeof(DMsg_AvgprofPlusData), (Msg *)dmsg );

            DP_Printf("\nVel = %8.2f   as int %d\n", Vel, (int)(Vel*100) );
            DP_Printf("Mag_sum = %8.2f   as int  %d\n",Mag_sum,
                      (int)(Mag_sum * 100) );
            DP_Printf("Pk_at = %d\n", Pk_at );
            DP_Printf("Pk_dwell = %d\n", Pk_dwell );
          }
        }
        break;

      default:
        DP_Printf(" ERROR: Bad message type received (t=%02x l=%d).\n",
                  type, length );
        break;
    }
  }
}


static void handle_xilinx( DMsg_Xilinx *xilinx_ptr )
{
  int i;

  if ( !Xilinx_Start() )           /* Start configuration xfr mode */
```

```
    DP_Printf("SP: ERROR: Failed xilinx start.\n");

  Led_Print( 0xF1 );
  /* Send configuration data */
  for ( i = 0; i < xilinx_ptr->n; i++ )
    Xilinx_Write( xilinx_ptr->word[i] );

  Led_Print( 0xF2 );
  /* Make sure xilinx chip is fully configured */
  if ( !Xilinx_Done() )
    DP_Printf("SP: ERROR: Failed xilinx done.\n");
  Led_Print( 0xF0 );
}
```