

AD-A252 494



1

CSDL-T-1131

**AN IMPLEMENTATION OF A
FAULT-TOLERANT, MULTIDIRECTIONAL,
DIGITAL INTERPOLATION BEAMFORMER**

by
W. Brad Bogan

June 1992

Master of Science Thesis
Massachusetts Institute of Technology

DTIC
S **ELECTE** **D**
JUL 01 1992
A

*Original contains color
plates: All DTIC reproductions
will be in black and
white*

This document has been approved
for public release and sale; its
distribution is unlimited.

92-17223



92



The Charles Stark Draper Laboratory, Inc.
555 Technology Square, Cambridge, Massachusetts 02139-3563

An Implementation of a Fault-Tolerant, Multidirectional, Digital Interpolation Beamformer

by
Ensign W. Brad Bogan, United States Navy

B.S., Systems Engineering
United States Naval Academy (1990)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements
for the degree of
Master of Science

at the
Massachusetts Institute of Technology

June 1992

©W. Brad Bogan, 1992

Accession For	
NTIS	CRA&I ☒
DTIC	IAB ☐
Unannounced ☐	
Justification	
By <i>perform 50</i>	
Distribution	
Availability	
Dist	Availability
A-1	Special

The author hereby grants M.I.T. permission to reproduce and to
distribute copies of this thesis document in whole or in part.

Signature of Author *W. Brad Bogan*
Department of Electrical Engineering and Computer Science
May 6, 1992

Certified by *Amnon Aliphas*
Amnon Aliphas, C.S. Draper Lab Supervisor

Certified by *Thomas F. Knight*
Thomas F. Knight, Principal Research Scientist

Accepted by *Campbell L. Searle*
Campbell L. Searle
Chairman, Department Committee on Graduate Students

An Implementation of a Fault-Tolerant, Multidirectional, Digital Interpolation Beamformer

by
Ensign W. Brad Bogan, United States Navy

Submitted to the Department of Electrical Engineering
and Computer Science on May 6, 1992 in partial
fulfillment of the requirements for the degree of
Master of Science

Abstract

In recent years, methods of incorporating fault-tolerance in computer architectures have shifted from the exclusive focus on traditional modular redundancy techniques to include more efficient, analytical means of achieving the same result. Song and Musicus (1990) describe such a method of analytic fault-tolerance by using a statistical test and minimal redundancy to protect linear operations. Aliphas, Wei, and Musicus (1991) in turn developed a hardware prototype to test this method of statistical fault-tolerance. Using this hardware prototype, a multidirectional, digital interpolation sonar beamformer was implemented. Whereas the original prototype used multiple processors performing the same task on different input data, this implementation uses a different architecture whereby the same data is sent to multiple processors performing different tasks. The digital beamformer relaxes the sampling requirement typically required by sampling the incoming waveform at the Nyquist rate and upsampling to form the beam. Implementing the beamformer on the current architecture offered double fault detection and single fault correction capabilities across a range of look angles. In conjunction with previous results, the results of this thesis verify that the same fault detection and correction algorithm may be used for architectures both with processors performing different operations on the same data, and with processors performing the same operation on different data.

Thesis Supervisor: Thomas F. Knight
Title: Principal Research Scientist

Index Terms: Fault-Tolerant Computers, Digital Signal Processing Architectures, Multidirectional Beamformers, Digital Interpolation Beamforming

Acknowledgements

Thanks are due to several people who helped in one way or another towards the research, development, and writing of this thesis.

Completion of this project could not have been possible without the groundwork laid by Amnon Aliphas, Alex Wei, and Bruce Musicus on the hardware prototype that was to become my development vehicle. In addition, Amnon Aliphas helped out in no small amount with all aspects of the work along the way.

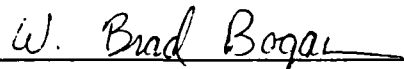
Thanks are due to several other individuals as well. Tom Martin of Valley Enterprises was an immense help with his programming experience and knowledge of the hardware. Paul Rosenstrach helped with the finer points of sonar beamforming, and supplied the data with which to test the system. John Trent and Dave Mackovjak helped by proofreading my initial draft, and offered useful comments on the system during development and testing as well. For his help with LaTeX, and on countless other topics from finance to used cars, I give Art Zemke a very big thanks. His friendship helped make this experience much more memorable.

Lastly, thanks must also be given to my fiancée Cynthia, whose moral support sparked my motivation to finish so that we could start our lives together.

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under IR&D project 345.

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

I hereby assign my copyright of this thesis to The Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts.


W. Brad Bogan

Permission is hereby granted by The Charles Stark Draper Laboratory, Inc., to Massachusetts Institute of Technology to reproduce any or all of this thesis.

Contents

1	Introduction	9
2	Multi-Input Single Function (MISF) Fault-Tolerance	11
2.1	Aspects of MISF fault-tolerance	11
2.1.1	Adding redundancy	12
2.1.2	Selecting the weights	15
2.1.3	Fault detection and correction	17
2.2	Maximum likelihood estimation	18
2.3	Finite precision considerations	21
2.4	Hardware prototype	22
3	Single Input Multi-Function (SIMF) Fault Tolerance	27
3.1	The structure	27
3.2	The algorithm	29
3.3	Possible applications	30
3.3.1	Same nature data	30
3.3.2	Different nature data	30
4	Beamformer Application	32
4.1	Multidirectional fault-tolerant beamforming	32
4.2	Digital interpolation beamforming	33
4.2.1	Second-order sampling	35
4.2.2	Interpolation	37
4.3	Implementing the beamformer	41

4.3.1	Calculating the time delays	41
4.3.2	Polyphase filter implementation	43
4.3.3	Forming the checksum responses	47
4.4	Results	51
4.5	System challenges	54
4.6	Theoretical overhead	57
5	Conclusions and Recommendations for Future Work	58
A	Source Code	62

List of Figures

2.1	Triple Modular Redundancy for a single processor	12
2.2	MISF fault-tolerant architecture	12
2.3	Weight vectors in 2D syndrome space with fault decision regions. . .	16
2.4	Graphical interpretation of MLE algorithm	19
2.5	MISF algorithmic fault-tolerant testbed	23
2.6	Valley VE-32C processor board with four DSP32C's, VME bus inter- face, and FIFO I/O bus.	24
2.7	Custom I/O board with FIFO's and bus shunt.	25
2.8	Layout of the 16 DSP32C's with the I/O board.	26
3.1	SIMF fault-tolerant architecture.	28
3.2	SIMF fault-tolerant architecture for composite data.	31
4.1	Multidirectional beamformer with N look angles.	33
4.2	Analog beamformer and sensor array of N_e hydrophones	34
4.3	Two methods of performing second-order sampling, (a) by delaying the signal, and (b) by delaying the sampling gate	36
4.4	Filter impulse response for $L = 7$, $N_e = 63$	39
4.5	Filter magnitude response for $L = 7$, $\Delta = 50\mu s$	39
4.6	Task layout for $N = 10$, $C = 3$	42
4.7	Determination of time delays for a signal θ° off broadside	43
4.8	Digital interpolation beamforming for complex input sequences. . . .	44
4.9	Polyphase interpolation filter bank	45
4.10	Determination of fractional delay	45

4.11 Data bin structure with $q + b$ point overlap region and encircled filter window.	47
4.12 Structure of checksum operators for the first batch of data and a single sensor.	50
4.13 Beamformer response when all processors working.	52
4.14 Faulty output with processor 6 zeroed and corrected output	55

List of Tables

4.1	Parameters for WQS-1 implementation	38
4.2	Subfilters used for $N = 10$, $C = 3$ beamformer application	46
4.3	Measured relative likelihoods with threshold for $N = 10$, $C = 3$	53
A.1	Source files listed by processor, with dependencies	62

Chapter 1

Introduction

In a conventional N -modular redundant computer architecture, N copies of processors, memory, and I/O units are driven with the same program and the same data. The outputs of these N units are then compared to verify that they are operating correctly. Although it is possible for these particular schemes to reliably detect or even correct up to $(N - 1)/2$ errors, a large amount of additional hardware must be dedicated to monitoring any potential faults, and the synchronization required between the processors is very demanding. Traditional triple-modular redundancy, then, is capable of single error correction, but at the expense of at least 200% overhead.

One method of incorporating fault-tolerance relies upon low redundancy, arithmetic coding to protect linear computations against failure [2, 9]. This approach uses a generalized likelihood ratio test (GLRT) in order to statistically produce the best possible fault decisions in applications where the calculations are subject to truncation or rounding errors. The architecture utilizes a parallel bank of signal processors performing the same linear function, where each processor operates on its own unique input. This method of fault-tolerance was tested with a hardware prototype using commercial processors, and successfully demonstrated [2].

A new approach to incorporating such analytic fault-tolerance has been taken that creates the opportunity for an entirely new class of applications. By effectively transposing the previous architecture, a design is achieved wherein a common input is broadcast to a bank of signal processors each performing its own unique function.

Since one of the beneficial properties of linear operations is commutativity, it will be seen that the same generalized likelihood ratio test will apply to the error detection and correction of this new architecture. In addition, the opportunities for combining the two architectures exist, and open the door to a host of signal processing applications.

To demonstrate the principles of this signal processing architecture, a multidirectional sonar beamformer was chosen. Real sonar data was collected, and tested using the beamformer programmed with existing hardware. Because of the low overhead involved in arithmetic redundancy, such a sonar system is ideally suited for small, unmanned applications where both power and size are constrained. This application is only one, however, and certainly several others exist that could effectively utilize such analytic fault-tolerance.

This thesis is broken into four remaining chapters. Chapter 2 reviews the derivation of analytic fault-tolerance for a single function architecture with multiple inputs. Chapter 3 begins the work done for this thesis by extending the derivations of Chapter 2, and arriving at the new signal processing architecture with a single input and multiple functions. Chapter 4 discusses the aspects of digital interpolation beamforming, and the implementation of a digital interpolation beamformer onto this single input, multi-function architecture. Chapter 5 offers conclusions, and recommendations for future work.

Chapter 2

Multi-Input Single Function (MISF) Fault-Tolerance

2.1 Aspects of MISF fault-tolerance

To meet the demands of many computationally intensive signal processing applications, massively parallel multiprocessor architectures are often used [8]. Some applications may require processors operating in parallel, and performing the same task. For these types of applications, it is desirable to send different data to the different processors in order to increase the total processing throughput. If N processors are used in such an architecture, the total processing throughput is thus approximately N times that of a single processor.

To protect these N processors from failure using triple modular redundancy would require an additional $2N$ processors, as well as multiple voters to compare the outputs of all processors. Figure 2.1 shows a triplicately modular redundant system for a single processor. Each voter declares a processor's output faulty if it fails to agree with the majority of all processor outputs. Majority here means two, so triple modular redundancy is capable of isolating a single error. In theory, a single voter can isolate an error from such a triplicated processor scheme. Additional voters are needed, however, in the event that one fails. How the output of these triplicated voters are then used depends upon the particular application. An alternative to this method of

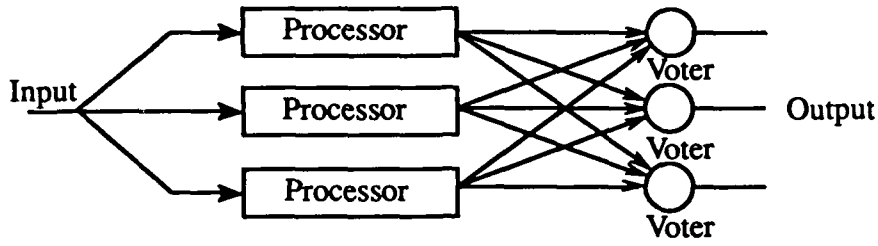


Figure 2.1: Triple Modular Redundancy for a single processor

fault-tolerance is to incorporate the redundancy analytically.

2.1.1 Adding redundancy

Figure 2.2 shows how redundancy is incorporated arithmetically with minimal overhead.

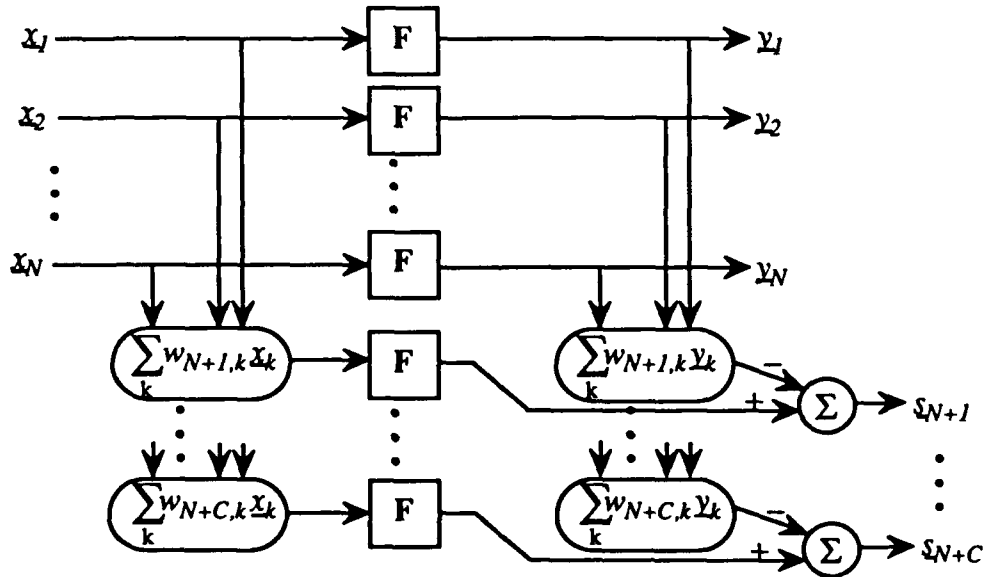


Figure 2.2: MISF fault-tolerant architecture

In a manner analogous to adding parity bits to a data word, C “checksum” processors are added to protect N “working” processors. Let $\underline{x}_k(m)$ represent a p length data vector, or packet, sent to processor k at time m , where $1 \leq k \leq N$. Each of the

C checksum processors then computes a different weighted linear combination of the packets at time m , forming new packets [2]:

$$\underline{x}_j(m) = \sum_{k=1}^N w_{j,k} \underline{x}_k(m) \quad \text{for } j = N+1, \dots, N+C \quad (2.1)$$

where the $w_{j,k}$ are scalar weights. The same linear function $\mathbf{F}$ is then applied to the inputs of all $N+C$ processors, producing r length output packets $\underline{y}_k(m)$:

$$\underline{y}_k(m) = \mathbf{F} \underline{x}_k(m) \quad \text{for } k = 1, \dots, N+C \quad (2.2)$$

Thus $\mathbf{F}$ is an $r \times p$ matrix. The output error signals can then be computed by subtracting the same weighted linear combinations of the outputs from the N working processors, or channels, from the outputs of the checksum channels:

$$\underline{s}_j(m) = \underline{y}_j(m) - \sum_{k=1}^N w_{j,k} \underline{y}_k(m) \quad \text{for } j = N+1, \dots, N+C \quad (2.3)$$

For the ease of discussion, the time index m will be omitted from here on. Since these error signals may be used for fault diagnosis, they will be referred to as syndromes. If all processors have computed their results properly the syndromes should be zero, since

$$\mathbf{F} \sum_{k=1}^N w_{j,k} \underline{x}_k = \sum_{k=1}^N w_{j,k} \mathbf{F} \underline{x}_k \quad \text{for } j = N+1, \dots, N+C$$

In actuality, the processors do not have infinite precision, and thus truncation or rounding errors can be expected. These errors are typically small, and are uniformly distributed over the range of the smallest quantization level used by the processors. Furthermore, these errors due to limited precision occur randomly, and may be assumed to be independent. By the central limit theorem then, in the absence of a processor failure, the syndromes, which consist of a sum of uniformly distributed random variables, may be approximated as low intensity white Gaussian noise.

To calculate the syndromes in the event of a processor failure, assume that the output from processor k differs from its correct value by the composite error $\underline{\phi}_k$ [9].

In this event, equation (2.3) becomes:

$$\underline{s}_j = \underline{\phi}_j - \sum_{k=1}^N w_{j,k} \underline{\phi}_k + \text{noise} \quad \text{for } j = N+1, \dots, N+C \quad (2.4)$$

This equation can be simplified by defining the weights in the checksum processors to be:

$$w_{N+i, N+j} = \begin{cases} -1 & \text{for } i = j = 1, \dots, C \\ 0 & \text{for } i \neq j \end{cases} \quad (2.5)$$

so that

$$\underline{s}_j = - \sum_{k=1}^{N+C} w_{j,k} \underline{\phi}_k + \text{noise} \quad \text{for } j = N+1, \dots, N+C \quad (2.6)$$

It is assumed that any hardware failures occur independently in the $N+C$ processors, so that at most one working or one checksum processor fails at a given time. Suppose then that only the k th processor fails, so that only $\underline{\phi}_k$ is nonzero. Given this, the syndromes will have values:

$$\mathbf{s} = \begin{pmatrix} \underline{s}_{N+1} \\ \vdots \\ \underline{s}_{N+C} \end{pmatrix} = - \begin{pmatrix} w_{N+1,k} \mathbf{I} \\ \vdots \\ w_{N+C,k} \mathbf{I} \end{pmatrix} \underline{\phi}_k + \text{noise} \quad (2.7)$$

where

$$\begin{pmatrix} w_{N+1,k} \mathbf{I} \\ \vdots \\ w_{N+C,k} \mathbf{I} \end{pmatrix}$$

is called the k th weight vector, and will be referred to as $\mathbf{W}_k$.

Note that a checksum channel failure can be distinguished from a working channel failure if at least two of the weights in the set $\{w_{N+1,k}, \dots, w_{N+C,k}\}$ are nonzero for all $1 \leq k \leq N$ [2]. Thus, a checksum channel failure results in only one non-zero syndrome, whereas a working channel failure will result in at least two non-zero syndromes.

2.1.2 Selecting the weights

A block weight matrix can be defined, composed of the $N + C$ weight vectors:

$$\mathbf{W} = \begin{pmatrix} w_{N+1,1}\mathbf{I} & \cdots & w_{N+1,N}\mathbf{I} & -\mathbf{I} & \cdots & \mathbf{0} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ w_{N+C,1}\mathbf{I} & \cdots & w_{N+C,N}\mathbf{I} & \mathbf{0} & \cdots & -\mathbf{I} \end{pmatrix} \quad (2.8)$$

where each block is $r \times r$. Naturally, the weights should be chosen so as to reduce the difficulty of computing the checksum inputs and syndromes. Small integers $(0, \pm 1, \dots)$ make particularly good choices, since no multiplication is needed then. A desirable property is that the columns of $\mathbf{W}$ form a "complete set", wherein if $(\alpha_1 \alpha_2 \cdots \alpha_C)^T$ is a weight vector, then every vector of the form $(\pm \alpha_1 \pm \alpha_2 \cdots \pm \alpha_C)^T$ is also either a weight vector or the negative of a weight vector [9]. This causes the weight vectors to exhibit a useful symmetry.

There are various selection criteria for choosing the scalar weights $w_{j,k}$, but they should be chosen so that it is possible to distinguish between failures in the different working processors. In case of failure on the k th channel, equation (2.7) implies that the syndromes should lie on a line defined by the k th weight vector, $\mathbf{W}_k$. If there was no rounding or truncation noise present, the syndromes would fall perfectly on that line. Due to errors of finite precision, however, they would not fall directly on the line, but should fall close by. In this case, the processor closest in signal space to the syndromes would be declared faulty. Hence it is desirable to maximize the angular separation between weight vectors, in order to reduce the chance of a misdiagnosis. If the inner product between any two weight vectors is defined as:

$$r_{k,m} = \sum_{j=N+1}^{N+C} w_{j,k}^* w_{j,m} \quad (2.9)$$

then the angle $\theta_{k,m}$ between any two weight vectors k and m may be defined in the usual manner:

$$\cos \theta_{k,m} = \frac{|r_{k,m}|}{\sqrt{r_{k,k} r_{m,m}}} \quad (2.10)$$

To illustrate this method of geometric projection fault diagnosis, a simple two dimensional weight matrix may be formed:

$$W = \begin{pmatrix} C_1 & C_3 & -I & 0 \\ S_1 & S_3 & 0 & -I \end{pmatrix} \quad (2.11)$$

where

$$C_n = \cos\left(\frac{\pi n}{4}\right) \quad S_n = \sin\left(\frac{\pi n}{4}\right)$$

Figure 2.3 shows the 4 columns of (2.11) assuming a block size of $q = 1$, where the arrows indicate the weight vectors in two dimensional space. The angular separation

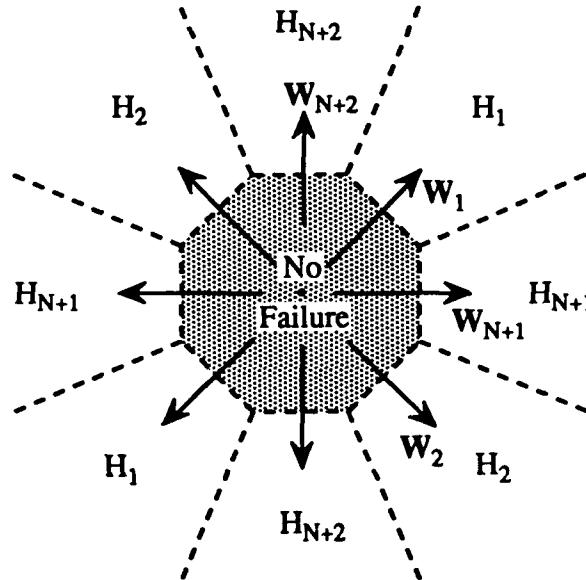


Figure 2.3: Weight vectors in 2D syndrome space with fault decision regions.

between weight vectors here is 45° . Syndromes falling far from the origin naturally have higher energies, and probably correspond to a processor failure. Syndromes falling near the origin have smaller energies, and probably correspond to mere rounding noise. Letting H_k be the hypothesis that the k th processor has failed, figure 2.3 shows the fault decision regions for the four different processors. The angular separation of the weight vectors may be further increased by using complex weights, but at the cost of increased complexity. Furthermore, adding checksum processors

increases the syndrome space thereby allowing greater separation as well, but at a cost of increased redundancy.

2.1.3 Fault detection and correction

If only one checksum processor was used in figure 2.2, then only one syndrome would be formed. In this case, a single error could be detected, since that syndrome would be zero except in the event a processor has failed. However, it is possible to distinguish between a failure on processor k_1 and processor k_2 only if weight vectors $\mathbf{W}_1$ and $\mathbf{W}_2$ are linearly independent. This requires a minimum of two checksum processors. Hence it is possible to detect and correct a single error with $C = 2$.

In general, in order to detect L failures, every possible set of L weight vectors must be linearly independent, for otherwise it would be possible for a combination of L failures to produce all-zero syndromes. This requires L dimensional weight vectors, and therefore L checksum processors. In order to detect *and* correct up to K failures, the hyperplane defined by all possible linear combinations of a given set of K weight vectors cannot intersect with the hyperplane defined by all possible linear combinations of any other set of K weight vectors, except at the origin [9]. In figure 2.3, if the hyperplanes are taken to be the lines represented by the weight vectors, it is clear to see that these lines intersect only at the origin. So here, $K = 1$, which is the dimension of a line. Detection and correction of up to K faults thus requires a $2K$ syndrome space, or equivalently, $2K$ checksum processors.

Therefore, given $C = 2K + L$ additional checksum processors, up to $K + L$ simultaneous failures may be detected, and, if no more than K have occurred, can be corrected, provided that every set of $2K + L$ weight vectors are linearly independent. Note that the fault detection and correction capacity is independent of the number of working processors. The work done so far has concentrated on a particular choice

of $\mathbf{W}$, where $N = 10$ and $C = 3$:

$$\mathbf{W} = \begin{pmatrix} \mathbf{I} & \mathbf{I} & \mathbf{I} & \mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{I} & \mathbf{I} & \mathbf{I} & \mathbf{I} & -\mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{I} & \mathbf{I} & -\mathbf{I} & -\mathbf{I} & \mathbf{I} & \mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{I} & -\mathbf{I} & \mathbf{0} & -\mathbf{I} & \mathbf{0} \\ \mathbf{I} & -\mathbf{I} & \mathbf{I} & -\mathbf{I} & \mathbf{I} & -\mathbf{I} & \mathbf{I} & -\mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & -\mathbf{I} \end{pmatrix} \quad (2.12)$$

and therefore this configuration is capable of double fault detection, and single fault correction.

2.2 Maximum likelihood estimation

A reasonable approach for diagnosing the failure is to measure the total syndrome energy and compare it with a fixed threshold, γ [8].

If it is above the threshold, i.e. if

$$\sum_{j=N+1}^{N+C} \|\underline{s}_j\| \geq \gamma$$

then a failure has probably occurred. To diagnose the failure, hypothesize that processor k has failed, with fault value $\underline{\phi}_k$. If this is the case, then each syndrome $\underline{s}_j$ should have the value $-w_{j,k}\underline{\phi}_k$, according to equation (2.7). The energy E_k in the difference between the observed syndromes and these expected values would then be:

$$E_k(\underline{\phi}_k) = \sum_{j=N+1}^{N+C} \|\underline{s}_j + w_{j,k}\underline{\phi}_k\|^2 \quad (2.13)$$

If processor k has actually failed, and $\underline{\phi}_k$ is the value of the actual failure, then E_k will be on the order of the round-off noise. Otherwise, E_k will be large. Therefore, for all $N + C$ processors, estimate the failure $\hat{\underline{\phi}}_k$ which best explains the observed syndromes in terms of a failure on processor k in a least squares sense, assuming all processor failures are equally likely:

$$\hat{\underline{\phi}}_k \leftarrow \min_{\underline{\phi}_k} E_k(\underline{\phi}_k) \quad (2.14)$$

Then the processor $\hat{k}$ most likely to have failed is the one that yields the smallest difference in energy between the observed syndromes and their estimated values:

$$\hat{k} \leftarrow \min_k E_k(\hat{\phi}_k) \quad (2.15)$$

Correcting the output from processor $\hat{k}$ using the estimated fault value gives

$$\hat{y}_k = \underline{y}_k - \hat{\phi}_k \quad (2.16)$$

Figure 2.4 shows this procedure graphically for an assumed batch size of $q = 1$. The vector $-\mathbf{W}_k \hat{\phi}_k$ is the projection of the syndromes $\mathbf{s}$ onto the k th weight vector line, and the squared distance from $\mathbf{s}$ to the weight vector line is $E_k(\hat{\phi}_k)$.

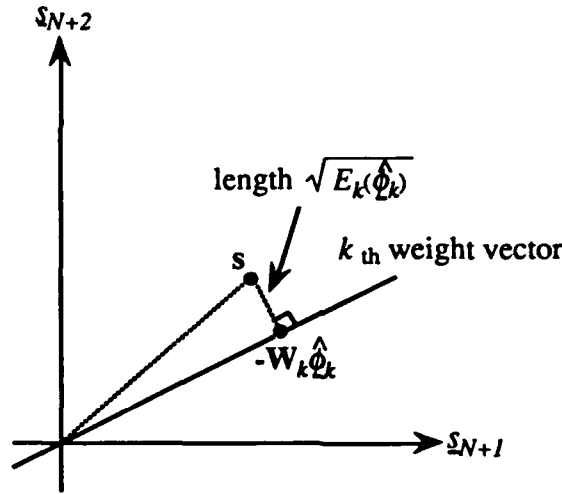


Figure 2.4: Graphical interpretation of MLE algorithm

This procedure can be shown [2, 9] to be equivalent to the following computationally efficient algorithm:

Compute the cross-correlations $\rho_{j,k}$ between the syndromes

$$\rho_{j,k} = \underline{s}_k^H \underline{s}_j \quad \text{for } j, k = N+1, \dots, N+C$$

Compare syndrome energy with threshold

$$\text{If } \sum_{k=N+1}^{N+C} \rho_{k,k} \geq \gamma, \text{ then FAILURE}$$

To correct the failure, compute relative likelihoods

$$L'_k = \frac{1}{r_{k,k}} \mathbf{W}_k^H \begin{pmatrix} \rho_{N+1,N+1} & \cdots & \rho_{N+1,N+C} \\ \vdots & \ddots & \vdots \\ \rho_{N+C,N+1} & \cdots & \rho_{N+C,N+C} \end{pmatrix} \mathbf{W}_k$$

$$\text{where } r_{k,k} = \sum_{j=N+1}^{N+C} |w_{j,k}|^2$$

Choose the processor k with the largest L'_k , (call it $\hat{k}$).

Correct the fault :

$$\hat{y}_{\hat{k}} = \underline{y}_{\hat{k}} + \frac{1}{r_{\hat{k},\hat{k}}} \sum_{j=N+1}^{N+C} w_{j,\hat{k}}^* s_j \quad (2.17)$$

where γ is a constant threshold which can be set to achieve a desired probability of false alarm, and the L'_k are proportional to the negative of the errors E_k plus a constant. The reason for calling them relative likelihoods is explained below.

It is noteworthy to point out that this least squares algorithm can be shown [9] to be optimal under some relatively simple conditions. It is already known that errors due to finite precision arithmetic have been modeled as white Gaussian noise, with statistics that may be precomputed based upon the linear function $\mathbf{F}$, and the weights $w_{j,k}$. If, additionally

1. Rounding noise on each sample is uncorrelated with the signal and with rounding noise on any other sample.
2. There is either no rounding when computing the input checksums, or else the linear function is an orthogonal transform.
3. The weight matrix forms a complete set.

then under these conditions, samples of the syndromes are all independent and iden-

tically distributed [2]. It will be seen that the beamformer implemented with this analytic fault tolerance satisfies these conditions. Additionally, this least squares algorithm is virtually identical to the statistically optimal Generalized Likelihood Ratio Test (GLRT), which becomes a maximum likelihood criterion under the assumption that all processors may fail with equal probability. The L'_k are *relative* because they are proportional to the relative likelihood of a failure on processor k compared with no failure at all. Furthermore, because the fault tolerance is statistical in nature, the probabilities of error detection, false alarm, and misdiagnosis may be computed in terms of the syndrome variances due to rounding and truncation noise.

2.3 Finite precision considerations

Any system designed to use the fault-tolerant algorithm described above utilizing current signal processing technologies would benefit from employing floating-point arithmetic, although the original algorithm considered the effects of fixed-point arithmetic [9]. Floating-point processing carries with it its own implications. If the checksums are computed with floating-point arithmetic, minimal rounding will occur in comparison to fixed-point arithmetic, and the energies of the syndromes will typically be many times smaller. Therefore, hardware failures become easier to discover, and the rounding noise in practice does not cause a measurable false alarm or misdiagnosis rate.

Errors due to limited precision uncover a subtle disadvantage to using equation (2.17) for error correction. In theory, once the faulty processor has been identified, a simple linear combination of the syndromes is sufficient to estimate the fault; subtracting this linear combination from the faulty processor's output yields the correct data. In practice however, a fault can cause the floating-point processor's output to have extremely large values, near the extreme limits of its dynamic range. These large values are then carried over into the calculation of the syndromes, which will then be very large as well. A weighted sum of these large valued syndromes may then cause an overflow, causing trouble in the fault correction step. A more numerically reliable

approach to fault correction is to estimate the correct processor output directly from the observed outputs of the other working processors. Combining equations (2.3) and (2.17) yields:

$$\begin{aligned}
\hat{y}_k &= y_k + \frac{1}{r_{k,k}} \sum_{j=N+1}^{N+C} w_{j,k}^* y_j - \frac{1}{r_{k,k}} \sum_{j=N+1}^{N+C} \sum_{m=1}^N w_{j,k}^* w_{j,m} y_m \\
&= y_k - \frac{1}{r_{k,k}} \sum_{j=N+1}^{N+C} \sum_{m=N+1}^{N+C} w_{j,k}^* w_{j,m} y_m - \frac{1}{r_{k,k}} \sum_{m=1}^N r_{k,m} y_m \\
&= y_k - \frac{1}{r_{k,k}} \sum_{m=1}^{N+C} r_{k,m} y_m \\
&= -\frac{1}{r_{k,k}} \sum_{\substack{m=1 \\ m \neq k}}^{N+C} r_{k,m} y_m
\end{aligned} \tag{2.18}$$

where $r_{k,m}$ is defined in (2.9). This formula is numerically more robust than (2.17), although it involves about $(N+C)/C$ times more computation.

2.4 Hardware prototype

The primary advantage of analytic fault-tolerance is that the computational overhead is generally far less than the 200% overhead required by traditional triple modular redundancy. This method, however, has its disadvantages as well. Because this technique relies upon a statistical test to determine which processor most likely failed, and the value of that fault, mistakes can occur in the presence of unusually high rounding noise. Faults may be ignored, may be declared when they do not exist, or may be misdiagnosed as another processor's error. Fortunately, a misdiagnosis occurs for the most part when the error is small, and therefore any mistaken "correction" will most likely be small as well. An additional problem is that a failed processor's output is necessarily corrected from the syndromes, which are themselves corrupted with round-off noise. Thus the reconstructed output contains a noisy version of the missing output.

Additional questions arise when an actual implementation is considered. Consideration must be given to moving data in and out of the $N+C$ processors, as well as

performing the checksum and syndrome calculations, and shipping the data to the MLE processor for final correction. Substantial control and processor synchronization is required throughout. In order to test this least squares algorithm, a hardware prototype was constructed [2] as illustrated in figure 2.5, using commercially available components.

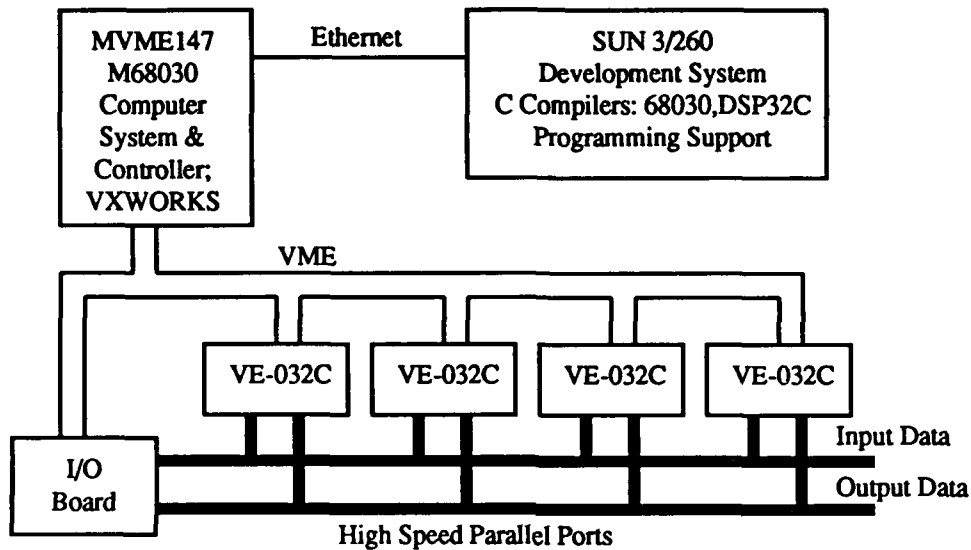


Figure 2.5: MISF algorithmic fault-tolerant testbed

The prototype was built using four commercial VE-32C boards from Valley Enterprises, each housing four floating-point DSP32C digital signal processors from AT&T, for a total of sixteen processors. Each processor is theoretically capable of 25MFlops, yielding a theoretical total of 400MFlops. Figure 2.6 depicts one of the processor boards. All boards have VME interfaces, which are used for downloading code into the processors, for passing synchronization messages between the host and the processors, and for other board control. A 10MHz, 16-bit parallel bus provides input to all four boards, feeding data into an input FIFO on each board which can be read by any processor. A separate 10MHz bus can be driven with output from any processor on any board.

A customized I/O board, illustrated in figure 2.7, configured to shunt the output bus directly to the input bus, allows any processor to send data to up to four processors

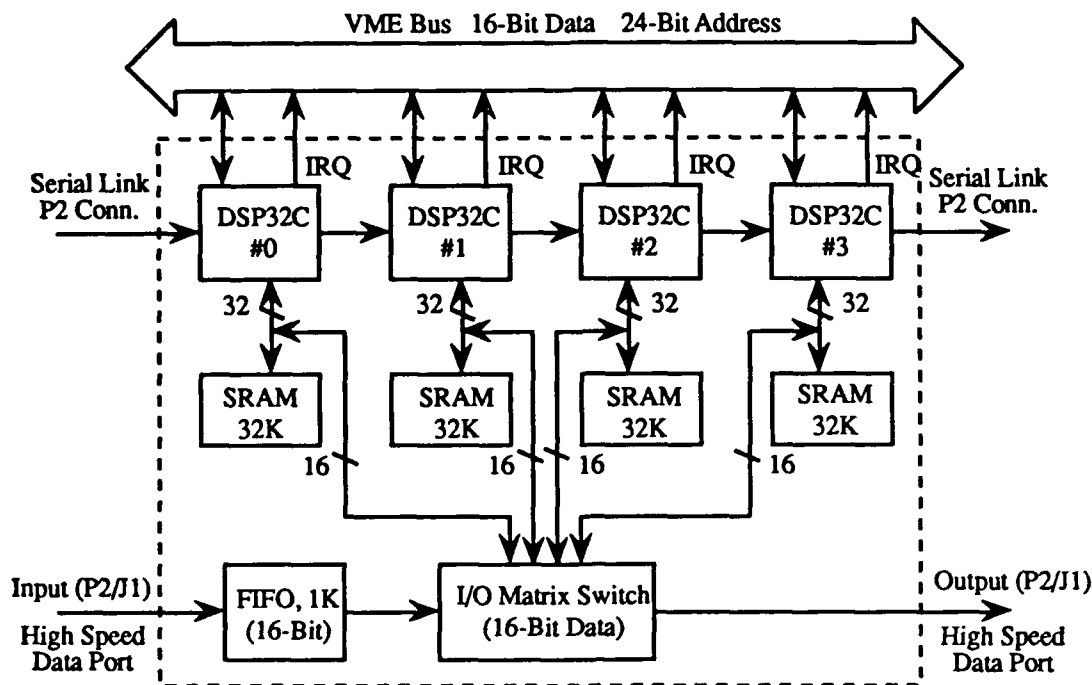


Figure 2.6: Valley VE-32C processor board with four DSP32C's, VME bus interface, and FIFO I/O bus.

on four different boards. It can also be used to buffer input or output data between the host and the fast bus systems via the VME bus. A Motorola MVME147 VME board with a MC68030 processor acts as host to the system, and is connected via Ethernet to a Sun 3/260 development system. The host utilizes the VXWORKS operating system, which interfaces smoothly with the SUN UNIX environment. C code can be compiled for either the host or the DSP32C's, and downloaded using VXWORKS and some custom commands.

Because of the limitations involved with shipping the data between processors, the actual layout of the tasks becomes very important. A generic layout showing the placement of the DSP32C's on the separate VE-32C boards in conjunction with the I/O board is shown in figure 2.8. As it will be seen, the actual task distribution will, with such an architecture, strongly influence the selection of the weights.

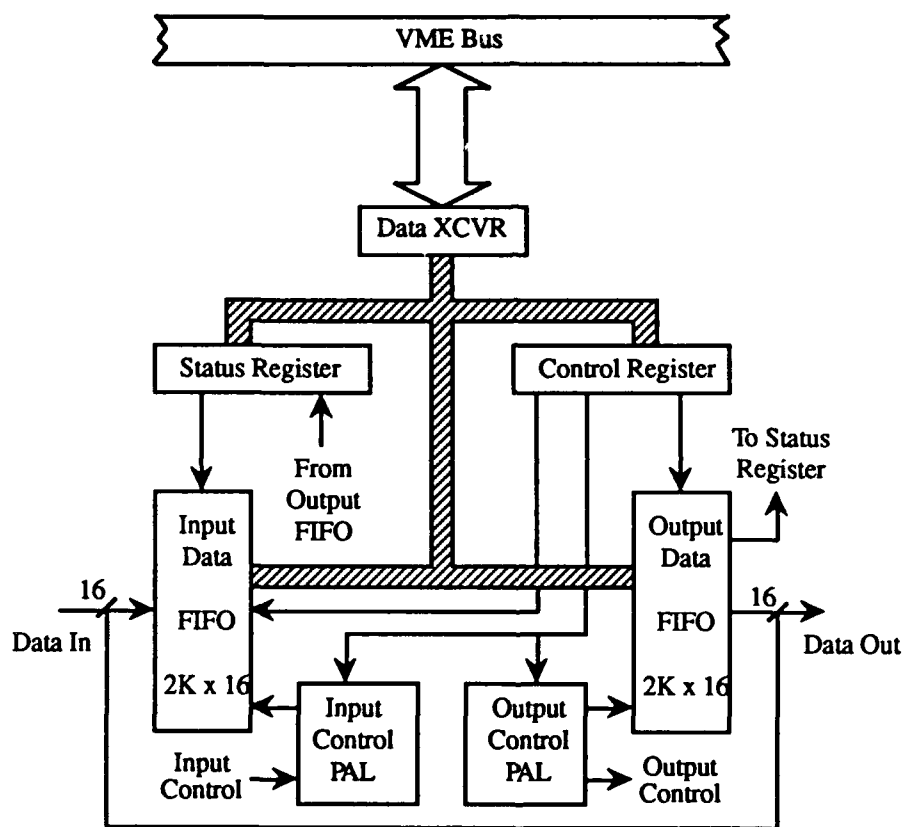


Figure 2.7: Custom I/O board with FIFO's and bus shunt.

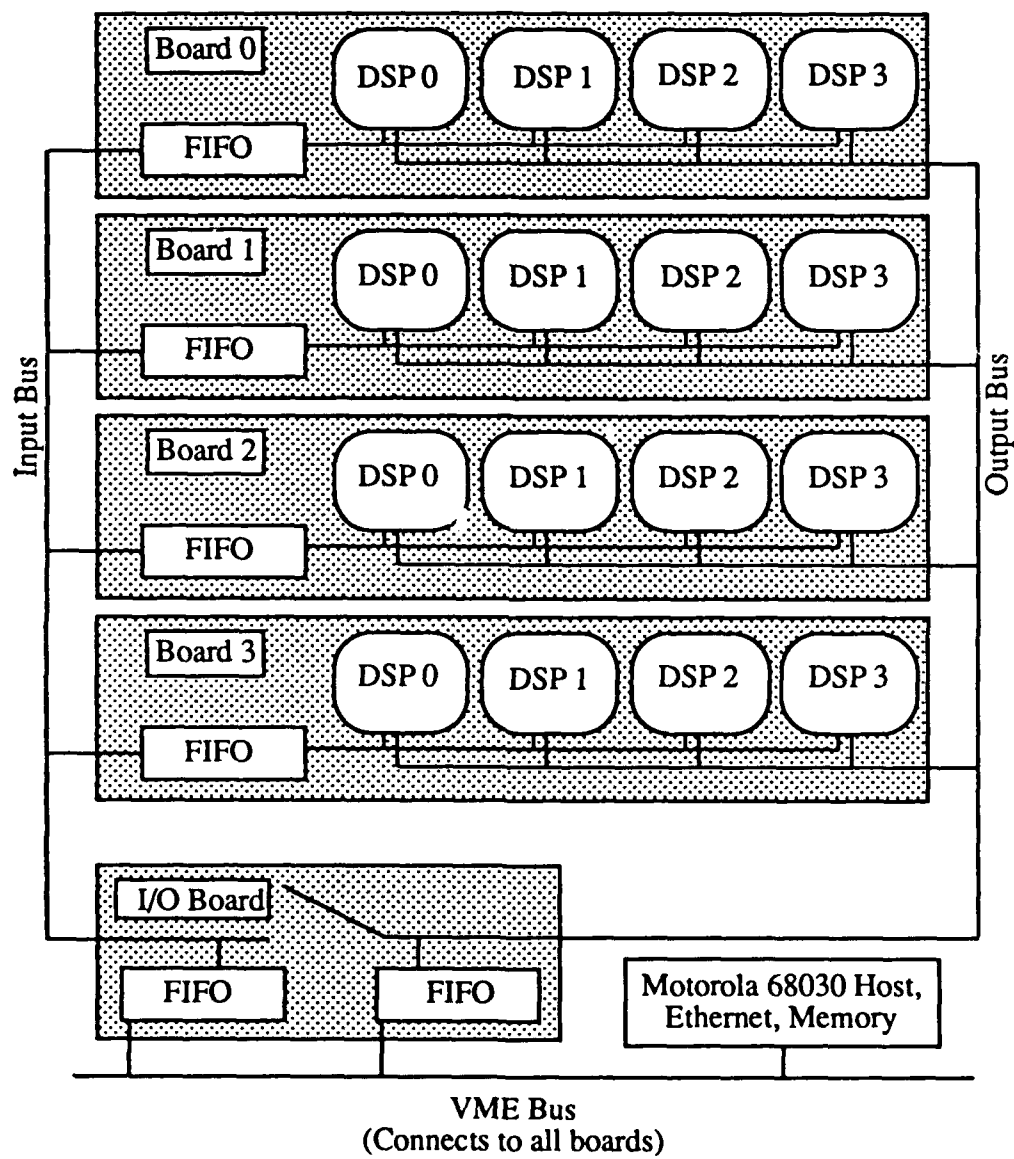


Figure 2.8: Layout of the 16 DSP32C's with the I/O board.

Chapter 3

Single Input Multi-Function (SIMF) Fault Tolerance

3.1 The structure

Representing a fundamentally different signal processing architecture, single input multi-function (SIMF) fault-tolerance, introduced in [1], enables a wealth of applications previously unattainable with MISF fault-tolerance. As the name implies, SIMF fault-tolerant architectures have a common input to all channels, with each processor performing a different linear task on the data. Figure 3.1 illustrates such a system.

As figure 3.1 shows, SIMF fault-tolerance is merely a transposition of a MISF architecture. The checksums now compute weighted linear combinations of the linear functions, as opposed to the data. Letting $\mathbf{F}_k$ be the linear operator in the k th processor for $1 \leq k \leq N$ yields:

$$\mathbf{F}_j = \sum_{k=1}^N w_{j,k} \mathbf{F}_k \quad \text{for } j = N+1, \dots, N+C \quad (3.1)$$

where again, the $w_{j,k}$ are scalar weights. The same p length data vector $\underline{x}(m)$ is then sent through all $N+C$ processors, producing r length output vectors $\underline{y}_k(m)$:

$$\underline{y}_k(m) = \mathbf{F}_k \underline{x}(m) \quad \text{for } k = 1, \dots, N+C \quad (3.2)$$

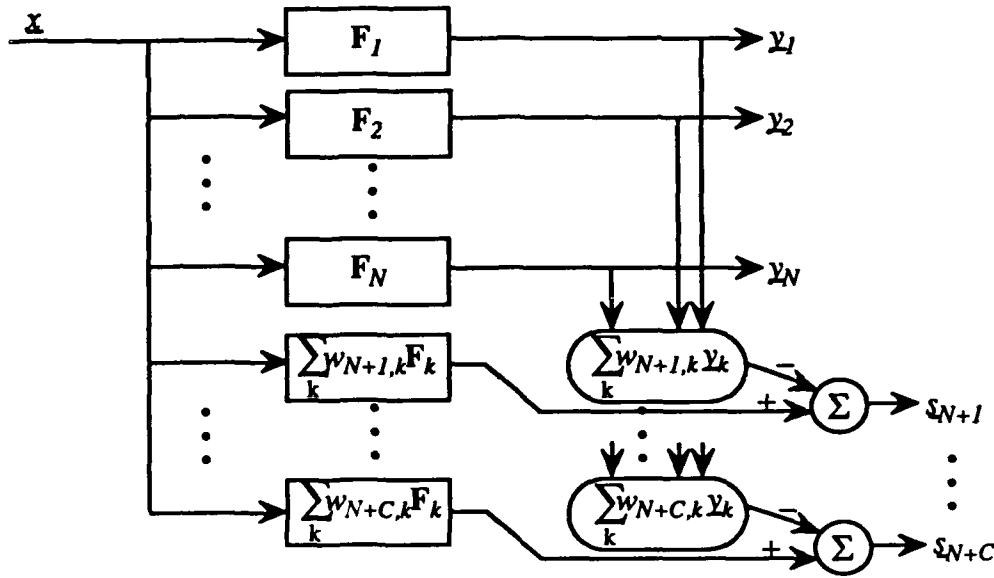


Figure 3.1: SIMF fault-tolerant architecture.

The syndromes then remain:

$$s_j(m) = y_j(m) - \sum_{k=1}^N w_{j,k} y_k(m) \quad \text{for } j = N+1, \dots, N+C \quad (3.3)$$

A SIMF configuration enjoys at least one advantage over similar MISF architectures when it comes to balanced computations through the checksum channels. MISF systems, as figure 2.2 illustrates, must first compute weighted sums of the input packets before processing in the checksum channels can begin. A SIMF architecture, by comparison, is capable of processing data through the checksum channels without delay. It may turn out, however, that the checksum channel operators in equation (3.1) involve significantly more computations than those in the working channels. In this case, the lack of delay before processing in the checksum channels begins may be overshadowed by the delay in the processing itself.

This is not the case, however, if the structure of the operators in the working channels are all similar. As in chapter 2, if the input and output vectors have length p and r respectively, then the linear operator F_k may be represented as an $r \times p$

matrix. The structure of the individual $\mathbf{F}_k$'s, not surprisingly, will determine the structure of the checksum channel operators. For example, assume that $p = r = 2$, and the matrices $\mathbf{F}_k$ all have a diagonal structure such that:

$$\mathbf{F}_k = \begin{pmatrix} a_k & 0 \\ 0 & b_k \end{pmatrix} \quad \text{for } k = 1, \dots, N \quad (3.4)$$

where a_k, b_k are arbitrary scalar variables. In this case, using equation (3.1), the checksum channels become

$$\mathbf{F}_j = \begin{pmatrix} \sum_{k=1}^N w_{j,k} a_k & 0 \\ 0 & \sum_{k=1}^N w_{j,k} b_k \end{pmatrix} \quad \text{for } j = N + 1, \dots, N + C \quad (3.5)$$

and thus after precomputing equation (3.5) for all C checksums, the checksum operators will require the same number of multiplies and additions when applied to the data as the working channels, and the processing will be balanced. When the working channel operators, $\mathbf{F}_k$, do not all have the same matrix structure, there is no guarantee that the checksum channel operators, $\mathbf{F}_j$, will require the same number of multiplications and additions. In this case the processing is not balanced. It should be noted that if the linear operators in the working channels are time varying, $\mathbf{F}_k(t)$, then the checksum channel operators can be precomputed only if a closed-form, analytic solution exists for equation (3.1).

3.2 The algorithm

The maximum likelihood estimation and correction algorithm of section 2.2 computed cross-correlations from the syndromes, compared the sum of the auto-correlations against a fixed threshold to check for failure, and computed relative likelihoods based upon the correlations and the weight vectors of equation (2.12). In other words, the entire algorithm depended only upon the syndromes. Since the syndromes for a SIMF configuration are computed exactly as they are for a MISF configuration, the method remains unchanged. Therefore, both architectures can be protected by the same fault

detection and correction algorithm.

3.3 Possible applications

3.3.1 Same nature data

The most conceivable applications for SIMF fault-tolerance, are ones in which the functions differ only slightly, and the nature of the operation the working channels are implementing is the same. In other words, the matrix structure of the linear operators are the same. Two examples of this would include both radar and sonar processing. In these types of signal processing for instance, the inputs from several sensors are delayed in a predetermined manner, and summed to determine the incident signal strength from a predetermined direction. The working channels in these cases then differ only by the delays given the signals of the various input sensors.

3.3.2 Different nature data

Not all of the data broadcast to the bank of processors in figure 3.1 need be used by every processor. Each processor may apply its own window to select the portion of the input composite data that is applicable to the linear operation being performed. Figure 3.2 illustrates this case. The configuration depicted in this figure combines the aspects of both MISF and SIMF. Not all processors are using the same input packet, although the same composite data vector, $\mathbf{x}$, is sent to all processors. In this case, the checksum channels would receive the entire composite data vector, and the various windows would then be applied in the weighted sum of the various $\mathbf{F}_k$'s. Furthermore, the matrix structure of the various $\mathbf{F}_k$'s may or may not be the same. If the matrix structure of the various $\mathbf{F}_k$'s differs from working channel to working channel, it is quite likely that the checksum channel operators will require additional multiplications and additions, and the computation will not be balanced throughout the processors.

A potential application for the system of figure 3.2 can again be taken from radar

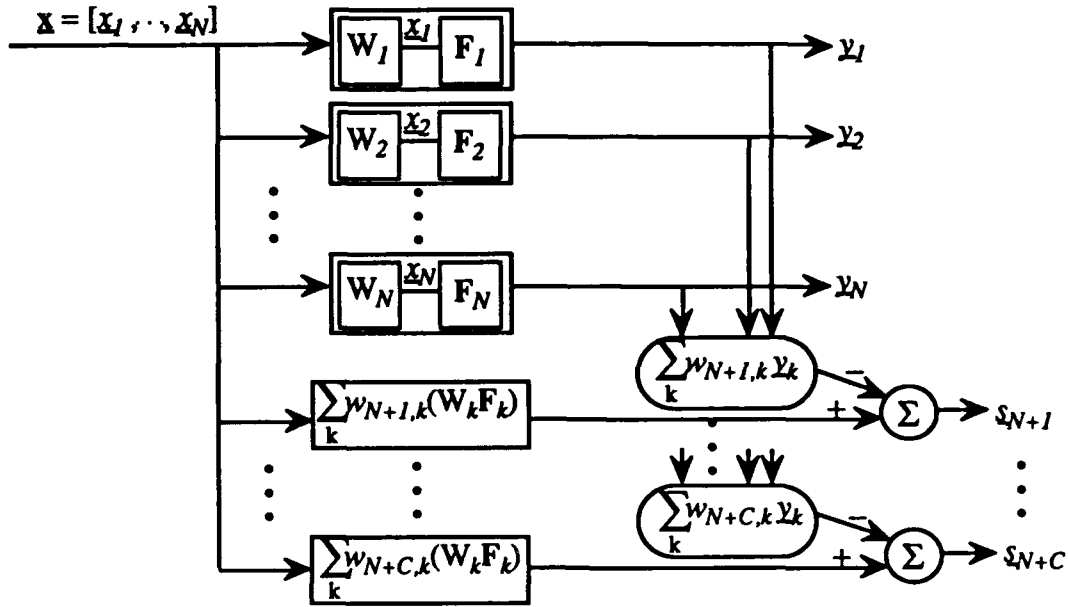


Figure 3.2: SIMF fault-tolerant architecture for composite data.

and sonar processing. For very large arrays, sections of sensors are typically grouped together into smaller sub-arrays in order to ease the processing burden, where $\underline{x}_k$ represents data from the k th sub-array. The outputs of various processors can then be combined to reconstruct the full array. It is also possible, assuming the data bandwidths are the same, to perform radar processing in one or more processors, sonar processing in others, and protect them all with the same set of checksum processors. In a like manner, systems that were previously separate can be joined to reduce the amount of redundancy required to make them fault-tolerant.

Chapter 4

Beamformer Application

4.1 Multidirectional fault-tolerant beamforming

Single input, multi-function fault-tolerance is particularly well suited for multidirectional sonar beamforming, where the linear operator has the same structure in all channels, and the data need not be windowed prior to processing. Beamformers are essentially spatial filters, getting their output by taking the weighted sum of signals from an array of sensors. Rather than consider a channel to consist of a single sensor, a channel will instead consist of the entire array of sensors. In this way, different channels then implement different time delays between the sensors, thereby steering the beam in a different direction. Figure 4.1 illustrates this point.

Most digital beamformers already work in this manner; rather than “paint” the area of interest, the beams are computed in parallel to give an instantaneous beam pattern. Ordinary digital beamformers, however, require very high sampling rates and thus place severe demands upon the hardware, and do not have the benefits of arithmetic redundancy.

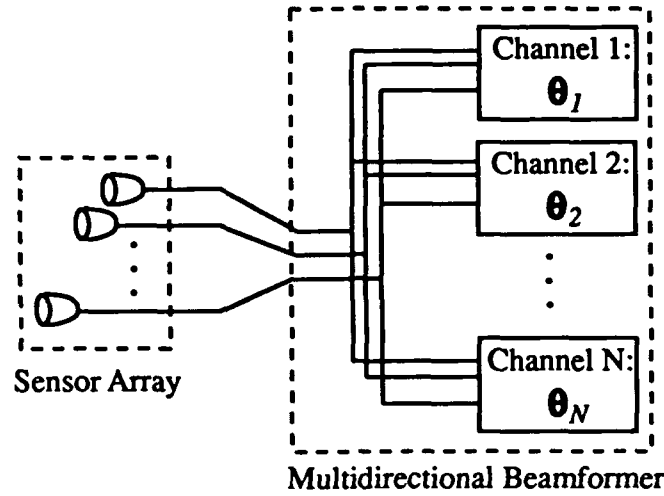


Figure 4.1: Multidirectional beamformer with N look angles.

4.2 Digital interpolation beamforming

Conventional analog time-domain beamforming may be represented as:

$$b(t) = \sum_{n=1}^{N_e} a_n x_n(t - \tau_n) \quad (4.1)$$

where n is the sensor number, N_e is the number of sensors in the array, a_n is the n th shading coefficient of the spatial window applied to the array, and τ_n represents the time delay required for the n th sensor in the array in order to form the beam, $b(t)$. Figure 4.2 illustrates this conventional beamformer. In this particular application, no spatial window was applied, and so $a_n = 1$ always.

Digital time-domain beamforming involves sampling equation (4.1) but places considerable demands upon system components by requiring sampling periods several times smaller than those required for mere waveform reconstruction. This is so that the time delays needed to form the beam, typically much smaller than the Nyquist sampling period, can be implemented. Digital interpolation beamforming relaxes this requirement by allowing the data to be sampled at the Nyquist period, Δ , and then interpolating the data prior to beamformation. The data is thus upsampled by a

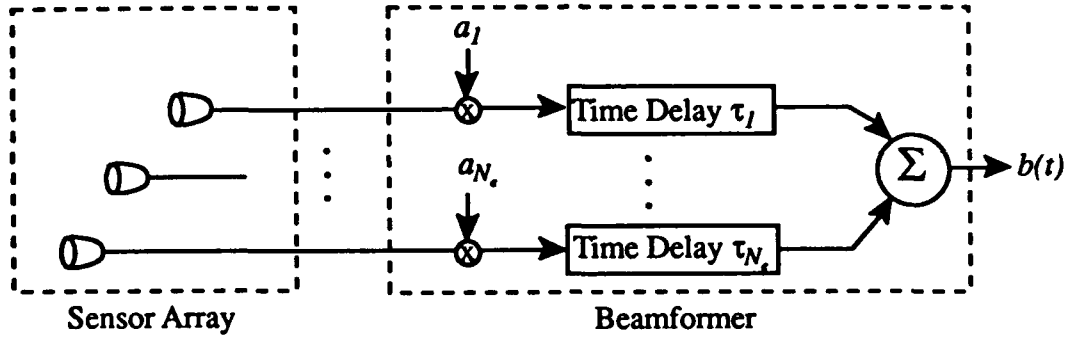


Figure 4.2: Analog beamformer and sensor array of N_e hydrophones

factor of L to a new sampling period, δ , such that $\delta = \Delta/L$. The time delays needed for beamformation are then approximated by integer delays, M_n , of the upsampled period, δ , for each sensor in the array. The accuracy of this approximation depends upon how accurately $M_n\delta$ approximates τ_n , where M_n is the rounded integer part of the quotient $\tau_n\delta^{-1}$. A beam for which

$$\tau_n = M_n\delta, \quad n = 1, \dots, N_e \quad (4.2)$$

is called synchronous [12]. Due to the linear nature of this equation, it is expected that a synchronous beam would only be possible with a linear array, and indeed this is the case. Any bandpass signal with bandwidth W centered on f_o and limited to the frequency interval

$$(f_o - W/2) \leq |w| \leq (f_o + W/2) \text{ for } W \leq f_o \quad (4.3)$$

may be represented as

$$x(t) = x_I(t) \cos \omega_o t - x_Q(t) \sin \omega_o t \quad (4.4)$$

where $x_I(t)$ and $x_Q(t)$ are the in-phase and quadrature components of x , respectively, band limited to the frequency interval $|f| \leq W/2$, and ω_o is the angular center

frequency of the bandpass signal [5]. As in the case of the signal waveform, the beam output may be expressed as

$$b(t) = b_I(t) \cos \omega_o t - b_Q(t) \sin \omega_o t \quad (4.5)$$

where $b_I(t)$ and $b_Q(t)$ are the in-phase and quadrature components of b , respectively. Combining equations (4.1) and (4.4) yields:

$$b(t) = \sum_{n=1}^{N_e} [x_{I_n}(t - \tau_n) \cos \omega_o(t - \tau_n) - x_{Q_n}(t - \tau_n) \sin \omega_o(t - \tau_n)] \quad (4.6)$$

where $x_{I_n}(t)$ and $x_{Q_n}(t)$ are the in-phase and quadrature components, respectively of $x_n(t)$. Equating (4.5) and (4.6) yields the following:

$$b_I(t) = \sum_{n=1}^{N_e} [x_{I_n}(t - \tau_n) \cos \omega_o \tau_n + x_{Q_n}(t - \tau_n) \sin \omega_o \tau_n] \quad (4.7)$$

$$b_Q(t) = \sum_{n=1}^{N_e} [x_{I_n}(t - \tau_n) \sin \omega_o \tau_n - x_{Q_n}(t - \tau_n) \cos \omega_o \tau_n] \quad (4.8)$$

Hence, the in-phase and quadrature components of the beam output can be obtained from the in-phase and quadrature components of the sensor outputs.

4.2.1 Second-order sampling

Sampling equations (4.7) and (4.8) requires sampled versions of $x_I(t)$ and $x_Q(t)$ for each sensor. Rather than use a complex demodulation scheme, these samples may be obtained directly from the bandpass signal without need for analog multipliers or filters via second-order sampling. With this method, the quadrature components are obtained from the complex envelope in pairs, with each sample period yielding two samples separated by one quarter of the carrier wavelength. In this manner, one sample corresponds to the in-phase component, and the other corresponds to the quadrature component. Which component is sampled first depends upon the way in which the second-order sampling is implemented.

Figure 4.3 illustrates the second-order sampling method for obtaining quadrature components.

This sampling method requires a sampling period of Δ where

$$\Delta = \frac{\ell}{2f_o} \leq W^{-1} \quad \text{for } \ell = 1, 2, \dots \quad (4.9)$$

In Figure 4.3a, the signal is delayed in the lower path and both samples are taken simultaneously, yielding [12]

$$\begin{aligned} x_1(m\Delta) &= x(m\Delta) \\ &= x_I(m\Delta) \cos(\omega_o m\Delta) - x_Q(m\Delta) \sin(\omega_o m\Delta) \\ &= (-1)^{m\ell} x_I(m\Delta) \end{aligned} \quad (4.10)$$

and

$$\begin{aligned} x_2(m\Delta) &= x(m\Delta - \frac{1}{4f_o}) \\ &= x_I(m\Delta - \frac{1}{4f_o}) \cos[\omega_o(m\Delta - \frac{1}{4f_o})] - x_Q(m\Delta - \frac{1}{4f_o}) \sin[\omega_o(m\Delta - \frac{1}{4f_o})] \\ &= (-1)^{m\ell} x_Q(m\Delta - \frac{1}{4f_o}) \end{aligned} \quad (4.11)$$

Here an implicit approximation is made by assuming $x_Q(m\Delta - \frac{1}{4f_o}) = x_Q(m\Delta)$. This approximation is valid for highly narrowband signals which can be expected with the

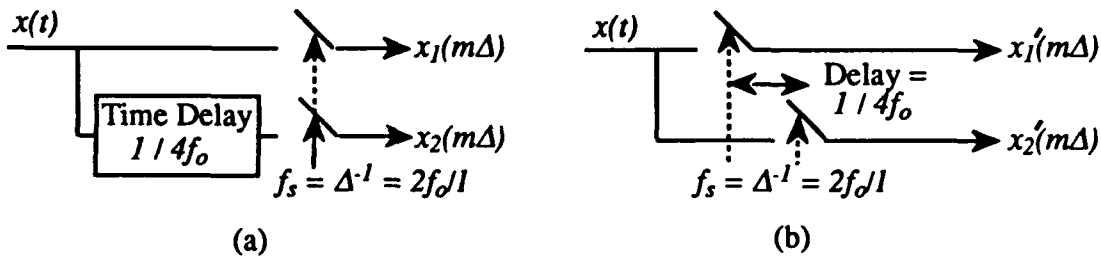


Figure 4.3: Two methods of performing second-order sampling, (a) by delaying the signal, and (b) by delaying the sampling gate

sonar system considered here [10], since this would mean a complex envelope that varies slowly relative to the modulating frequency. This application, however, uses samples obtained as in Figure 4.3b, where instead of delaying the signal, the lower sampling gate is delayed. This effectively interchanges the order of the components so that $x'_1(m\Delta) = (-1)^{m\ell} x_Q(m\Delta)$ and $x'_2(m\Delta) = (-1)^{m\ell} x_I(m\Delta)$, again under the same narrowband approximation. Sampling equations (4.7) and (4.8) with period Δ therefore yields :

$$b_I(m\Delta) = \sum_{n=1}^{N_e} [x_{I_n}(m\Delta - \tau_n) \cos \omega_o \tau_n + x_{Q_n}(m\Delta - \tau_n) \sin \omega_o \tau_n] \quad (4.12)$$

$$b_Q(m\Delta) = \sum_{n=1}^{N_e} [x_{I_n}(m\Delta - \tau_n) \sin \omega_o \tau_n - x_{Q_n}(m\Delta - \tau_n) \cos \omega_o \tau_n] \quad (4.13)$$

Henceforth it will be assumed that $x_I(m\Delta)$ and $x_Q(m\Delta)$ are available.

4.2.2 Interpolation

If Δ represented a sampling rate high enough for digital beamformation, equations (4.12) and (4.13) would suffice. That is not the case, however, and the samples must be interpolated to a higher sampling rate so that the proper delays may be implemented by shifting the signals forwards and backwards by discrete sample bins. Because they have a limited number of states, and because they have linear phase, FIR filters are chosen for this task. Their absence of recursion allows for a reduction in the required computation, by making it possible to only compute the output points necessary. If the filter was IIR, this would not be possible, due to the feedback of the outputs.

Since no FIR filter can ever have ideal low-pass filter characteristics, error will be introduced at the beamformer output. This finite error affects the time-filtered outputs of each sensor, which in turn affects the space-filtered beamformer output, and is induced because the frequency response of the filter deviates from the desired "box-car" shape. With proper filter design, this error can be minimized, but in general the beam response will still be less than ideal because the number of sensors

is typically limited.

The actual sonar chosen for this application was the WQS-1 obstacle avoidance sonar with a cylindrical aperture. The necessary specifications for the WQS-1 are listed in table 4.1. Other sonar implementations have utilized interpolation ratios of

Table 4.1: Parameters for WQS-1 implementation

Specification	Variable	Value
frequency of modulation	f_o	200 kHz
radius of array	$WRAD$	6.875 in.
angular spacing of sensors	$ASPC$	3°
assumed signal bandwidth	W	20 kHz
complex sampling period	Δ	50 μs
number of sensors used	N_e	15
interpolation ratio	L	7

4 and 10 [10, 12], and so an intermediate value was chosen for this application. With the use of the Remez exchange algorithm [7], an optimal equiripple FIR filter, $h(m\delta)$ was designed with the following characteristics:

$$f_u = \frac{5}{140} \cdot \frac{1}{\delta} \quad f_l = \frac{15}{140} \cdot \frac{1}{\delta}$$

where f_u and f_l are the upper passband edge and lower stopband edge, respectively. Equal weight was given to both the passband and the stopband errors. The ripple in the passband and the stopband can be minimized by adding more coefficients to the filter, at the cost of increased computational complexity. The number of coefficients, N_c , used for this application was 63. The reason behind choosing N_c as a multiple of the interpolation ratio will become apparent in the next section. The impulse and magnitude responses are shown in figures (4.4) and (4.5) respectively.

The first step to take in interpolating any sequence is to zero-pad it. Given an interpolation ratio of L , the zero-padded data sequence may be represented by :

$$v_{I_n}(m\Delta) = \begin{cases} x_{I_n}\left(\frac{m\Delta}{L}\right) & m = 0, \pm L, \pm 2L, \dots \\ 0 & \text{otherwise} \end{cases} \quad (4.14)$$

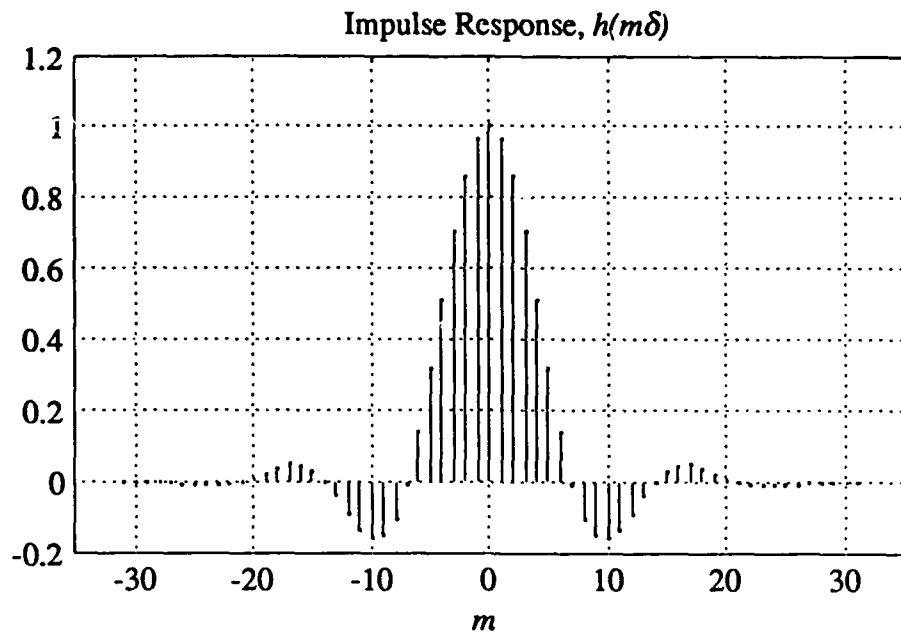


Figure 4.4: Filter impulse response for $L = 7$, $N_c = 63$.

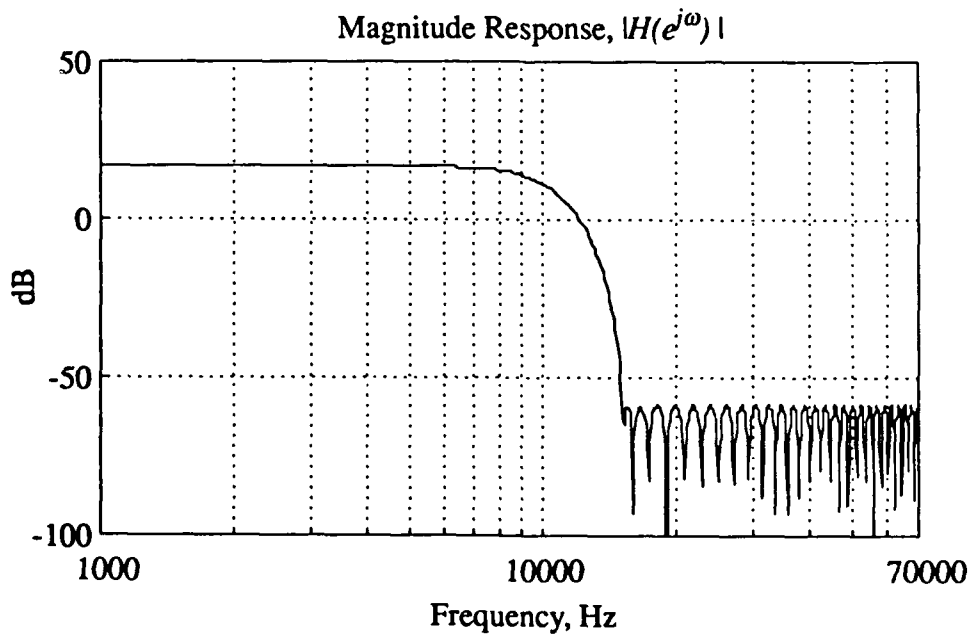


Figure 4.5: Filter magnitude response for $L = 7$, $\Delta = 50\mu\text{s}$.

where, again, Δ is the sampled period, and $\delta = \Delta/L$ is the interpolated period. An analogous construction may be made for $v_{Q_n}(m\Delta)$ as well. Interpolating the data sequences uses a discrete convolution so that

$$\tilde{x}_{I_n}(m\delta) = \sum_{k=0}^{N_c-1} h(k\delta)v_{I_n}[(m-k)\delta] \quad \text{for } n = 1, \dots, N_e \quad (4.15)$$

where again, $\tilde{x}_{V_n}(m\delta)$ is formed in an analogous manner. Given that the time delays, τ_n , required for beamformation are approximated by integer multiples of the upsampled data period, equations (4.12) and (4.13) can be approximated appropriately by shifting the interpolated sequences:

$$b_I(m\Delta) = \sum_{n=1}^{N_e} \{ \tilde{x}_{I_n}[(mL - M_n)\delta] \cos \omega_o \tau_n + \tilde{x}_{Q_n}[(mL - M_n)\delta] \sin \omega_o \tau_n \} \quad (4.16)$$

$$b_Q(m\Delta) = \sum_{n=1}^{N_e} \{ \tilde{x}_{I_n}[(mL - M_n)\delta] \sin \omega_o \tau_n - \tilde{x}_{Q_n}[(mL - M_n)\delta] \cos \omega_o \tau_n \} \quad (4.17)$$

and by combining equations (4.16) and (4.17) with (4.15), the following equations result:

$$\tilde{b}_I(m\Delta) = \sum_{n=1}^{N_e} \left[\sum_{k=0}^{N_c-1} h(k\delta)v_{I_n}(m\Delta - M_n\delta - k\delta) \cos \omega_o \tau_n + \sum_{k=0}^{N_c-1} h(k\delta)v_{Q_n}(m\Delta - M_n\delta - k\delta) \sin \omega_o \tau_n \right] \quad (4.18)$$

$$\tilde{b}_Q(m\Delta) = \sum_{n=1}^{N_e} \left[\sum_{k=0}^{N_c-1} h(k\delta)v_{I_n}(m\Delta - M_n\delta - k\delta) \sin \omega_o \tau_n - \sum_{k=0}^{N_c-1} h(k\delta)v_{Q_n}(m\Delta - M_n\delta - k\delta) \cos \omega_o \tau_n \right] \quad (4.19)$$

where $\tilde{b}_I(m\Delta)$ and $\tilde{b}_Q(m\Delta)$ are the approximated beam sums.

To simplify these results, it is possible to write the interpolated data values as a single complex point $v_n(m\Delta)$, such that $v_n(m\Delta) = v_{I_n}(m\Delta) - jv_{Q_n}(m\Delta)$ where here $v_n(m\Delta)$ is baseband. The conjugation here is necessary to ensure the proper signs in

the output equations. Using Euler's equation leads to

$$\begin{aligned} v_n(m\Delta)e^{j\omega_o\tau_n} = & [v_{I_n}(m\Delta)\cos\omega_o\tau_n + v_{Q_n}(m\Delta)\sin\omega_o\tau_n] \\ & + j[v_{I_n}(m\Delta)\sin\omega_o\tau_n - v_{Q_n}(m\Delta)\cos\omega_o\tau_n] \end{aligned} \quad (4.20)$$

and with a similar baseband complex representation for the in-phase and quadrature components of the beam, $\tilde{b}(m\Delta) = \tilde{b}_I(m\Delta) + j\tilde{b}_Q(m\Delta)$, so that equations (4.18) and (4.19) can be reduced to

$$\tilde{b}(m\Delta) = \sum_{n=1}^{N_e} \sum_{k=0}^{N_e-1} h(k\delta)e^{j\omega_o\tau_n} v_n(m\Delta - M_n\delta - k\delta) \quad (4.21)$$

4.3 Implementing the beamformer

Using the hardware described in section 2.4, a multi-directional digital interpolation beamformer was chosen to demonstrate the principles of SIMF fault-tolerance. The particular layout of the beams is as illustrated in Figure 4.6, where there are $N = 10$ working channels, and $C = 3$ checksum channels. The weights chosen for the application were the same as in equation (2.12).

4.3.1 Calculating the time delays

Using the specifications of table 4.1, it is possible to determine the time delays and their integer approximations based upon the geometry of the array. Figure 4.7 helps illustrate how these values are determined. It is reasonable to assume that an incident waveform may be approximated as a plane. The steering direction, θ , is then normal to the incident waveform at the point of tangency. To steer the beam to this direction then, requires a series of time delays, τ_n , for each sensor. These time delays can be found by projecting the individual sensors from a mirror image of the array onto the incident waveform. Hence, the τ_n 's are determined from the cosine of the angle between the normal to the incident waveform, and the sensor of interest. With this approach, the time delays would be zero at the geometric limits of the array, and

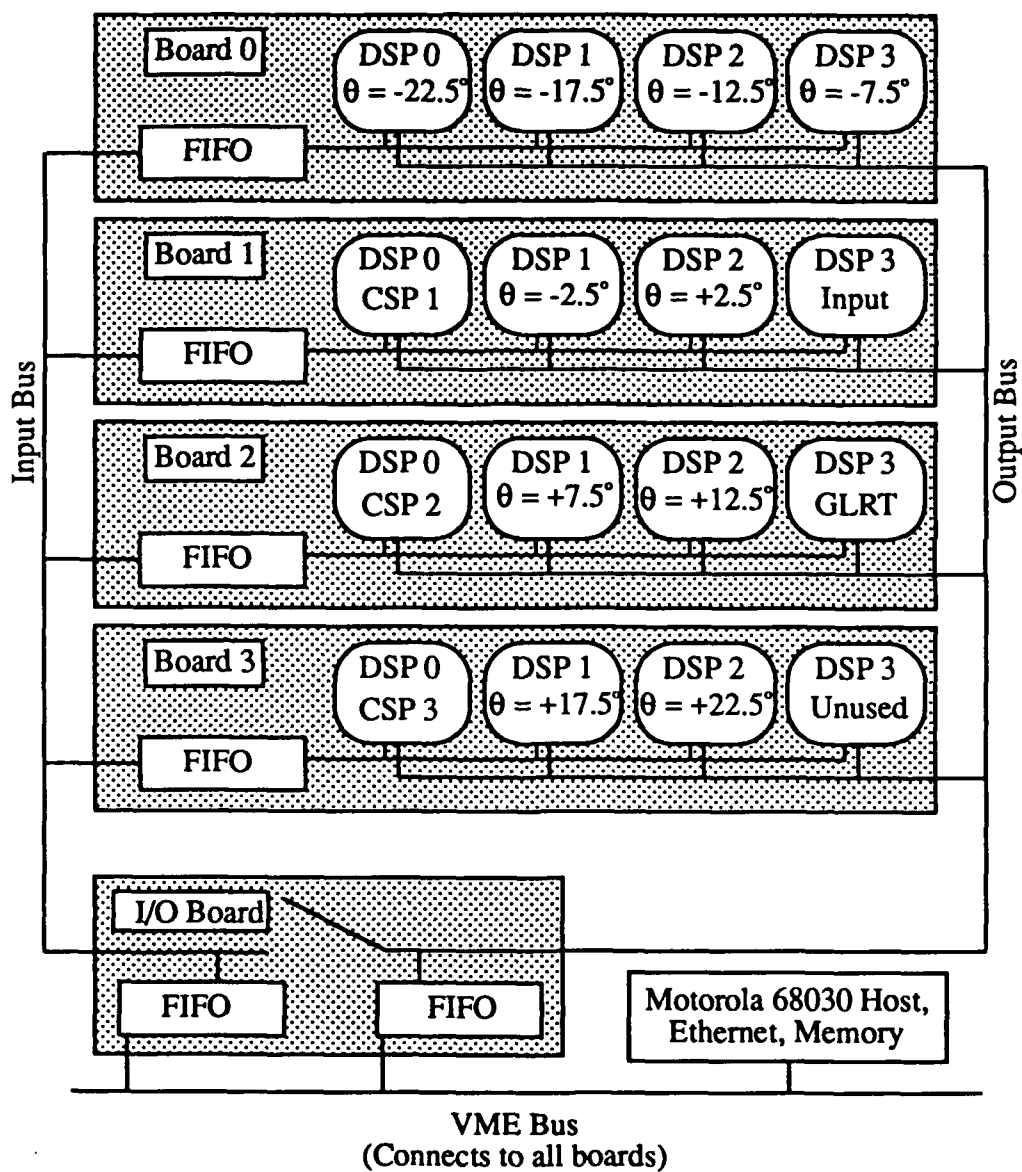


Figure 4.6: Task layout for $N = 10, C = 3$

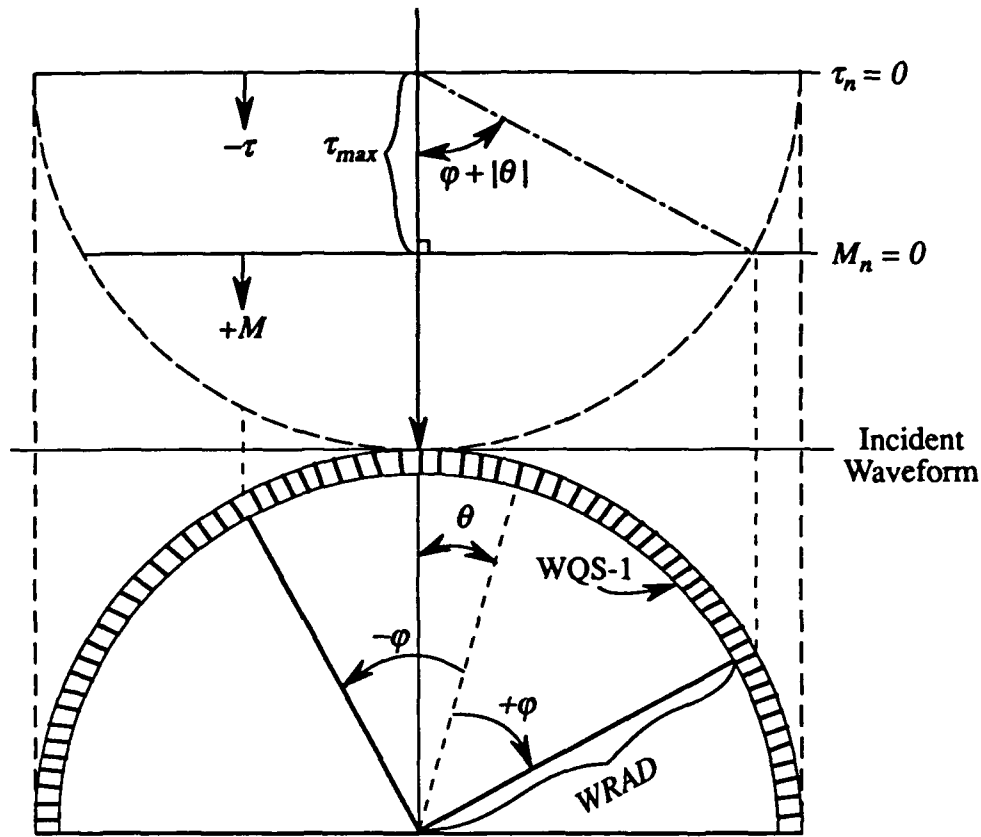


Figure 4.7: Determination of time delays for a signal θ° off broadside

would be maximized at the point of tangency. For this application, however, these values of τ_n are negated, so that the sensor closest in angular distance to the steered direction will have the most *negative* value. This is done so that the values of M_n may be subtracted from the least negative integer approximation used in the array; therefore $M_n = 0$ corresponds to $\lceil \tau_{max} \delta^{-1} \rceil$. By doing this, the values of M_n will be strictly nonnegative, and will be referenced to the limits of those sensors in the array used in the calculation of the beam ($\pm\varphi$). By comparison, the τ_n 's are referenced to the geometrical limitations of the array ($\theta \pm 90^\circ$ for a cylindrical array such as this).

4.3.2 Polyphase filter implementation

With limited memory on each of the VE-32C boards (8K 4-byte words per DSP, or 32K bytes) zero-padding the data is not a viable option. Fortunately, this is

not necessary. Equation (4.21) indicates that the output sequence has the same sampling period as the original input sequences. Figure 4.8 gives a block diagram representation of equation (4.21). The original complex signals, sampled with period Δ , are upsampled by a factor of L , then filtered by a complex FIR filter. This is the interpolation and sinusoid weighting step combined. Each sensor's signal is then uniquely delayed by an integer number of samples at the upsampled data period. After the signals have been delayed, they are then downsampled to the previous period Δ , and summed to form the beam. The interpolation and decimation ratios need not be identical, but since they are, they may be removed altogether by using a collection of polyphase filters. Given a full order, linear phase FIR filter h of order $N_c - 1$, a series of smaller order polyphase filters $g_\beta(m\Delta)$ can be constructed [3] such that

$$g_\beta(m\Delta) = h(m\delta L + \beta), \quad \text{for } \beta = 0, 1, \dots, L - 1 \text{ and } \forall m \quad (4.22)$$

Figure 4.9 shows the implementation of a polyphase filter bank. Here, the output of the β th path, $y_\beta(m\Delta)$, corresponds to the interpolation output sample $y(m\delta L + \beta)$ obtained by applying a single input $x(m\Delta)$ to the filter bank. In other words, for a particular $x(m\Delta)$, each of the L branches of the filter bank contributes one nonzero output (or vector of outputs) corresponding to its family of output samples. Since the integer delays, M_n , remain constant for a given sensor and a given steering angle, only one branch of the polyphase filter bank per hydrophone is required for a desired steering direction. This makes it possible to not only remove the padded zeros in the data sequence, but to implement the convolution with a filter $1/L$ times as long,

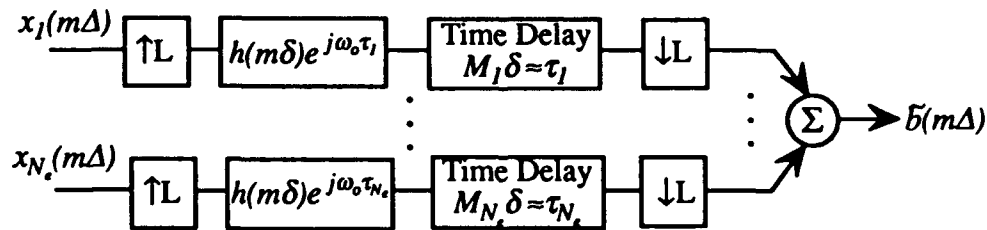


Figure 4.8: Digital interpolation beamforming for complex input sequences.

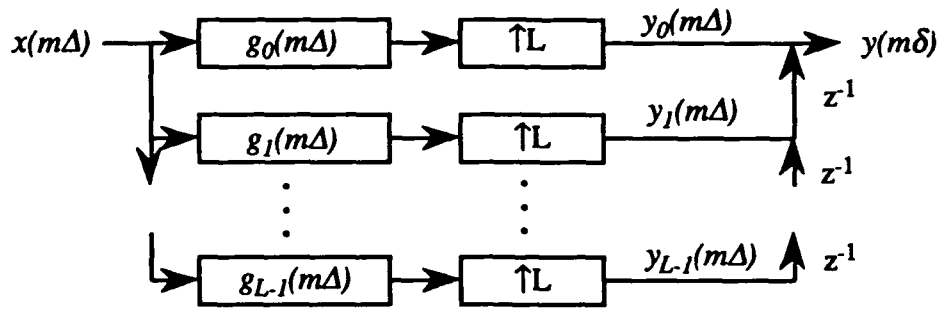


Figure 4.9: Polyphase interpolation filter bank

thereby reducing total computation *per sensor* by an order of L^2 for each steering angle.

To obtain an expression for β_n in terms of M_n , assume for the moment that $M_n < L$. Figure 4.10 illustrates this case. The choice of M_n is based upon the number of steps *back* in time needed for a particular sensor to properly delay its interpolated data and form the beam. Conversely, the outputs of the polyphase filter bank steps *forward* in time with each sample sent through the network. Thus, for the case where $M_n < L$, it must be that $M_n + \beta_n = L$. The case where $M_n \geq L$ corresponds to a combined whole integer delay P_n , plus the fractional delay β_n . Accordingly, a modulo L division must first be performed on M_n . This yields the following equations:

$$P_n = \left\lfloor \frac{M_n}{L} \right\rfloor \quad (4.23)$$

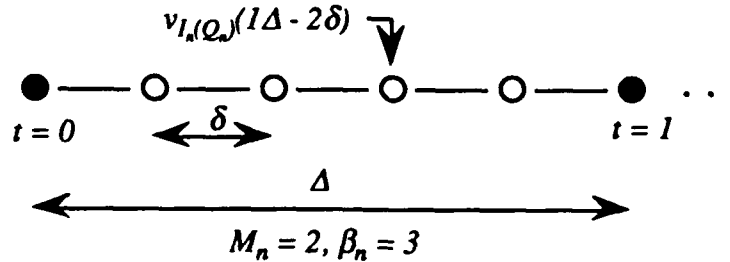


Figure 4.10: Determination of fractional delay

Table 4.2: Subfilters used for $N = 10$, $C = 3$ beamformer application

Values of β_n for all N working processors										
n	θ in degrees									
	-22.5	-17.5	-12.5	-7.5	-2.5	+2.5	+7.5	+12.5	+17.5	+22.5
1	3	4	4	5	0	0	0	0	0	0
2	3	4	4	5	0	0	0	6	0	0
3	3	4	4	5	6	0	6	6	6	6
4	3	4	4	5	6	6	6	5	6	6
5	3	4	4	5	6	6	6	5	6	5
6	4	4	4	5	6	6	5	5	5	5
7	4	4	4	5	6	6	5	5	5	5
8	4	5	4	5	6	6	5	4	5	4
9	5	5	5	5	6	6	5	4	4	4
10	5	5	5	5	6	6	5	4	4	4
11	5	6	5	6	6	6	5	4	4	3
12	6	6	5	6	6	6	5	4	4	3
13	6	6	6	6	0	6	5	4	4	3
14	0	0	6	0	0	0	5	4	4	3
15	0	0	0	0	0	0	5	4	4	3

$$\beta_n = [(P_n + 1) \cdot L - M_n] \bmod L \quad (4.24)$$

The values of β_n used in this application are given in table 4.2.

Using this fact, the inner summation term of equation (4.21) may be rewritten so that

$$\sum_{k=0}^{N_c-1} h(k\delta) v_n(m\Delta - M_n\delta - k\delta) = \sum_{k=0}^q g_{\beta_n}(k\Delta) x_n(m\Delta - k\Delta) \quad (4.25)$$

where, as with $v_n(m\delta)$, $x_n(m\Delta) = x_{I_n}(m\Delta) - jx_{Q_n}(m\Delta)$, and q is the order of the β_n th subfilter. By having selected N_c as a multiple of the interpolation ratio, q is constant for all β_n , $1 \leq n \leq N_c$.

Now applying this result to equation (4.21), the same results are achieved without zero-padding the data.

$$\bar{b}(m\Delta) = \sum_{n=1}^{N_c} \sum_{k=0}^q g_{\beta_n}(k\Delta) e^{j\omega_o \tau_n} x_n[(m - k)\Delta] \quad (4.26)$$

Thus each output point is the sum over all sensors in the array of the resulting discrete

time convolution between each sensor's complex data and a complex FIR filter. Since all values are now referenced to the same sampling period Δ , it will be omitted for simplicity.

4.3.3 Forming the checksum responses

An FIR filter can be implemented in several ways, but to aid in the representation of input/output relationships, it is perhaps easiest to view an FIR filter as a matrix multiplication. To do this, it is first necessary to define the data on which the filter is to be applied.

Figure 4.11 shows the layout for a data bin of a single sensor for the first batch of data. Here N_s is the number of complex data samples per sensor sent in a given batch

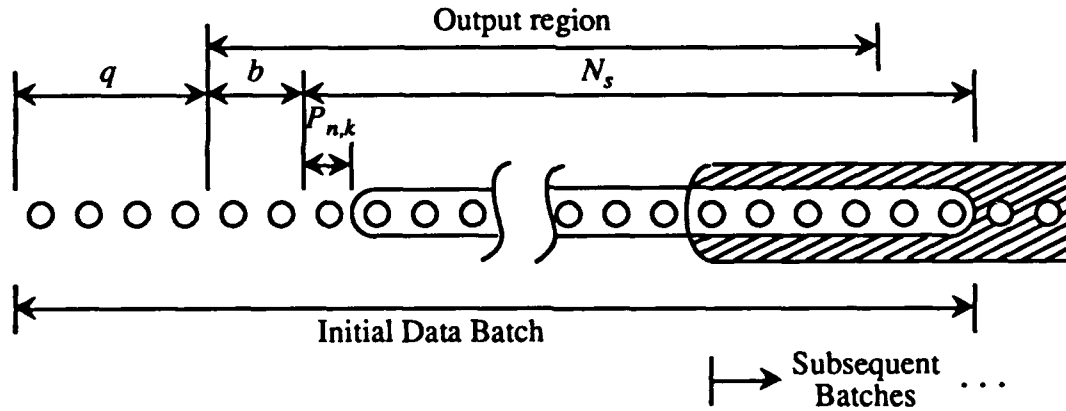


Figure 4.11: Data bin structure with $q + b$ point overlap region and encircled filter window.

of data, q is the order of the subfilters as before, $P_{n,k}$ is the number of whole integer delays for sensor n in working channel k , and b is the maximum number of whole integer delays required across the sensors to form a particular beam. It is important to note that since this application involves $N = 10$ different beams, b must be chosen as the largest of the maximum delays required in each working channel so that the checksum channel responses may be properly formed. In this manner, the size of the data bin remains constant for all $N + C$ processors, assuming a fixed set of look

angles.

Even though only N_s points get sent with each batch of data, the additional pad points in the data bin, where $pad = q + b$, are necessary to maintain a continuous output. In the absence of any whole integer delays, the last q points would be cycled to maintain a continuous output through the use of overlap-save convolution [9]. When whole integer delays are introduced, the FIR filter is implemented over a delayed set of data points, initiated on the first batch of data. If a particular sensor had no whole integer delays, and only the last q points were cycled, there would be precisely b points of convolution overlap unaccounted for in the next filtering operation for that sensor, since a given sensor applied to a given beam will always use the same subfilter. Including b in the length of the data bin allows for the necessary delay in the filtering operation. Including b in the number of data points needed to be cycled after each convolution guarantees a continuous output from each sensor.

As Figure 4.11 illustrates, there is no delay compensation necessary after the first batch of data, and hence the coefficients required in the checksum channels after the first data batch are merely a weighted sum of the filter coefficients in the working channels on a per sensor basis. The necessary delay comes in the first data batch, and must be accounted for to properly form the checksum channel beams. The data in this $maxlen$ point data bin, where $maxlen = N_s + pad$, can be represented by means of the vector $\underline{x}_n(m)$,

$$\underline{x}_n(m) = [x_n(mN_s - pad) \dots x_n(mN_s + (N_s - 1))]^T \quad (4.27)$$

where m is the batch number, and $x_n(t) = 0$ for $t < 0$. Correspondingly, a similar output vector from a single sensor and a single working channel, $\tilde{\underline{b}}_{n,k}(m)$, can be defined:

$$\tilde{\underline{b}}_{n,k}(m) = [\tilde{b}_{n,k}(mN_s - pad) \dots \tilde{b}_{n,k}(mN_s + (N_s - 1))]^T \quad (4.28)$$

where output values preceding $mN_s - b$, and following $mN_s + (N_s - b - 1)$, are invalid due to the overlap-save convolution being performed. Thus, each $\underline{x}_n(m)$ generates valid output samples $\tilde{b}_{n,k}(mN_s - b)$ through $\tilde{b}_{n,k}(mN_s + (N_s - b - 1))$ contained within

$\tilde{b}_{n,k}(m)$. Written as a discrete time convolution, the input/output relationship is as follows:

$$\tilde{b}_{n,k}(t) = \sum_{k=0}^q g_{\beta_{n,k}}(k) x_n(t-k) \quad \text{for } t = mN_s - pad, \dots, mN_s + (N_s - 1) \quad (4.29)$$

where the phase terms have been temporarily removed for convenience. This representation, however, fails to account for the shift applied to the first data batch for sensors with nonzero whole integer delays. This delay can be accounted for with a matrix multiplication representation. Define

$$\mathbf{G}_{n,k} = \begin{pmatrix} g_{\beta_{n,k}}(0) & 0 & \dots & 0 \\ \vdots & g_{\beta_{n,k}}(0) & 0 & \\ g_{\beta_{n,k}}(q) & \vdots & g_{\beta_{n,k}}(0) & \ddots & \vdots \\ 0 & g_{\beta_{n,k}}(q) & \ddots & 0 & \\ \vdots & \ddots & \ddots & g_{\beta_{n,k}}(0) & 0 \\ 0 & \dots & 0 & g_{\beta_{n,k}}(q) & \dots & g_{\beta_{n,k}}(0) \end{pmatrix} \quad (4.30)$$

where $\mathbf{G}_{n,k}$ is a $(maxlen - P_{n,k}) \times (maxlen - P_{n,k})$ band diagonal matrix operating on the desired window of data from the n th sensor on the k th processor with $P_{n,k}$ defined in equation (4.23). Let $\mathbf{0}_{P_{n,k}}$ be a $P_{n,k} \times P_{n,k}$ matrix of all zeros. Then the input-output for the n th sensor on the k th channel can be written as

$$\mathbf{F}_{n,k}(m) = \begin{pmatrix} \mathbf{0}_{P_{n,k}} \\ \mathbf{G}_{n,k} e^{j\omega_o \tau_{n,k}} \end{pmatrix} \quad (4.31)$$

so that $\tilde{b}_{n,k}(m) = \mathbf{F}_{n,k}(m) \cdot \underline{x}_n(m)$. It now becomes a simple matter to find the total beam response of a given channel, $\tilde{b}_k(m)$, rather than just one sensor's contribution. A block matrix, $\mathbf{F}_k(m)$, may be formed so that $\mathbf{F}_k(m) = [\mathbf{F}_{1,k}(m), \dots, \mathbf{F}_{N_e,k}(m)]$, and a block data vector, $\underline{x}(m)$ may be formed as $\underline{x}(m) = [\underline{x}_1^T(m), \dots, \underline{x}_{N_e}^T(m)]^T$. Then equation (4.26) with whole integer delays properly accounted for becomes

$$\tilde{b}_k(m) = \mathbf{F}_k(m) \cdot \underline{x}(m) \quad (4.32)$$

As figure 4.11 illustrates, $P_{n,k}$ may be taken as zero after the first batch of data without problem, since the delay has by that time already been implemented. This means that $F_k(m)$ is constant after the first batch of data, and thus the checksum channel operators may be pre-calculated, allowing the checksum channel calculations on-line to match those of the working channels. The structure of the $F_{n,k}(0)$ matrices becomes very valuable in determining the proper filter coefficients required by the C checksum channels. Figure 4.12 shows a weighted sum of the $F_{n,k}(0)$'s, for a given n , over all N working channels. Again, the first *pad* rows(columns) are zero for this

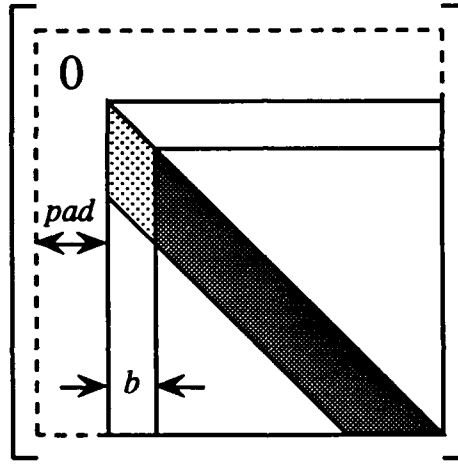


Figure 4.12: Structure of checksum operators for the first batch of data and a single sensor.

batch since the signal is assumed to be causal. Note that the only shift needed is at the beginning since the last b points of $\tilde{b}_{n,k}(m)$ are not output. Here, the separate blocks represent the various individual matrices $G_{n,k}e^{j\omega_0\tau_{n,k}}$ that operate on specific windows of data for a given steering angle.

Note that the sum of the separate $F_{n,k}(0)$'s is band diagonal as well. The lightly shaded region corresponds to the n th sensor in one or more of the N working channels having anything but a maximal number of whole integer delays. This region has, at most, b nonzero columns for a particular sensor in the checksum channels. The heavily shaded area is the region within which *all* channels (for a given sensor) contribute to the checksum output. Hence, for the first batch of data, it is necessary to store $b + 1$

sets of complex filter coefficients, per sensor, in each of the C checksum channels. For subsequent batches, only the last one of these $b + 1$ sets will be used.

4.4 Results

The beamformer was tested with forward-looking sonar data collected at Lake Champlain [4]. Ten ribbed steel drums of length 72 in. and diameter 21 in. were used as test targets. The acquired data consisted of 16 synchronous receive staves on the AN/WQS-1 active sonar, of which 15 were used.

Figure 4.13 shows the return from a single ping, where the look angles of the $N = 10$ processors have been chosen so as to center the beams on the target return. This image shows the actual return from the 10 processors when they are all properly working. Since each processor computes only one discrete look angle, and the processor's are all separated by 5° , it was necessary to spatially interpolate the outputs in order to get a smooth image. As figure 4.13 shows, the largest returns come from $+2.5^\circ$, corresponding to processor 6. The target, in fact, was located at approximately 3° off broadside. As is shown, the largest return occurs at approximately 26.2 ms. The smaller returns prior to this resulted from some assorted cinder blocks that were in the water near the deployed target.

For the purposes of testing the maximum likelihood algorithm of section 2.2, a processor was chosen to appear as though it had failed by injecting an error at its output. Three types of constant errors were allowed, representing separate types of failures. The options consisted of adding a constant to the data, setting the data to a constant value, or adding a constant to a single sample. For the purposes of detecting the faulty processor, the latter of these error types was needed. The threshold, γ , was set so that a single bit transient error on a single processor would not go undetected. Once detected, the error was corrected using the numerically robust equation (2.18). Once the threshold is set to detect a single bit error, detecting more severe errors becomes easier. Table 4.3 shows the relative likelihoods of all processors when various types of errors are injected into the first working processor.

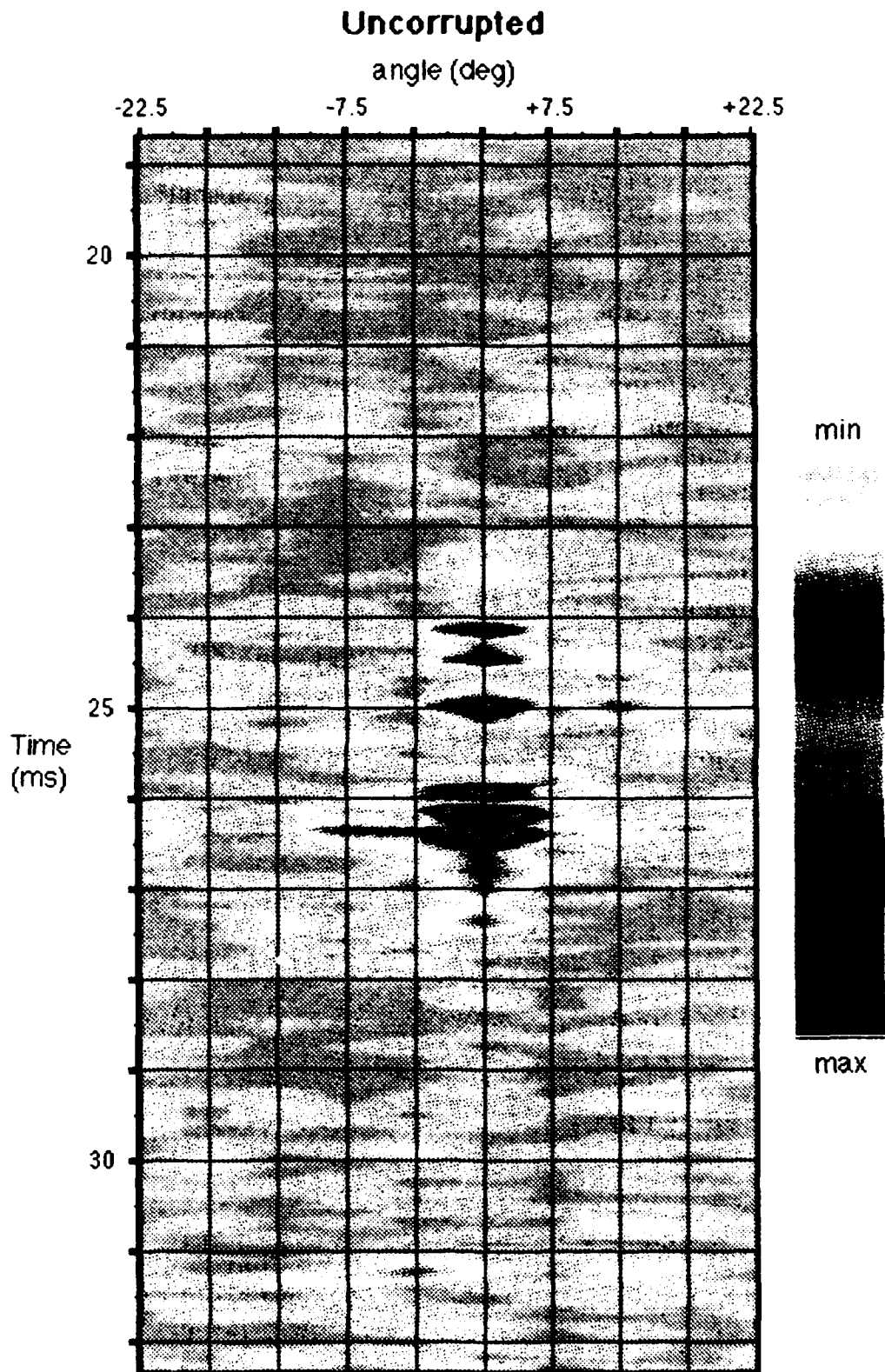


Figure 4.13: Beamformer response when all processors working.

Table 4.3: Measured relative likelihoods with threshold for $N = 10, C = 3$
(boldface indicates largest likelihood)

Task	Likelihoods			
	No fault	1-bit fault, sample 3, θ_1	1-bit fault, all samples, θ_1	Set real=0, imaginary=0, θ_1
Threshold	4.085e-04	4.085e-04	4.085e-04	4.085e-04
θ_1	3.113e-12	4.578e-05	1.56e-03	2.51e+00
θ_2	1.98e-11	5.08e-06	1.73e-04	2.79e-01
θ_3	3.47e-12	5.09e-06	1.73e-04	2.79e-01
θ_4	3.84e-11	5.09e-06	1.73e-04	2.79e-01
θ_5	1.90e-11	3.05e-05	1.04e-03	1.67e+00
θ_6	5.34e-12	8.19e-12	2.04e-10	5.96e-08
θ_7	2.68e-12	3.05e-05	1.04e-03	1.67e+00
θ_8	4.14e-11	4.09e-11	5.82e-10	5.96e-08
θ_9	7.31e-12	3.05e-05	1.04e-03	1.67e+00
θ_{10}	2.15e-11	3.27e-11	5.53e-10	1.19e-07
CSP1	2.43e-11	1.53e-05	5.19e-04	8.36e-01
CSP2	4.52e-12	1.53e-05	5.19e-04	8.36e-01
CSP3	1.98e-11	1.53e-05	5.19e-04	8.36e-01

Given that a transient, single bit error would be difficult to pinpoint on an image intensity plot, a much more catastrophic error was chosen to demonstrate the fault-tolerance of the beamformer. In case of a real processor failure, a sonar operator might expect that a component failure has occurred somewhere. In either case, the output would be erroneous. To simulate this type of error, the output of processor 6 was replaced with all zeros. Figure 4.14 shows the faulty, or corrupted, output alongside the output corrected on the fly after the computation of every $N_s = 34$ output points.

A final option on the program enables the injection of a *clipping* error, where it is assumed that the most significant bits in the output registers of a processor have failed, clipping the output at a certain level. Whereas the dramatic error of figure 4.14 may be easy to recognize because it essentially paints a stripe down a would be operator's display, a clipping error does not remove background noise, but may be enough to remove any peak returns. Such an error would be much more difficult for an operator to catch, and further illustrates the value of the system.

4.5 System challenges

The original prototype, built to test the concepts of MISF fault-tolerance, had a number of constraints due to the use of off-the-shelf components [2]. Naturally, by utilizing the same hardware the same constraints existed and, in addition, several programming challenges appeared throughout the development of the beamformer.

Although the DSP32C's are themselves very fast, they are constrained by the 16-bit, 1K-word FIFO's on the VE-32C boards, which limit the batch size, and hence the data rate. As a result, with $N_s = 15$ sensors per channel, only 34 complex data samples can be sent for each sensor with a given batch. With the addition of the extra buffer space needed in external memory on the VE-32C's for each DSP32C to perform overlap save convolution, the overall length of the complex data buffers is 42, of which 8 points ($q + b = 8$) must be cycled after each batch. Hence, nearly 20% of each sensor's data buffer must be cycled before new data may be read in. This

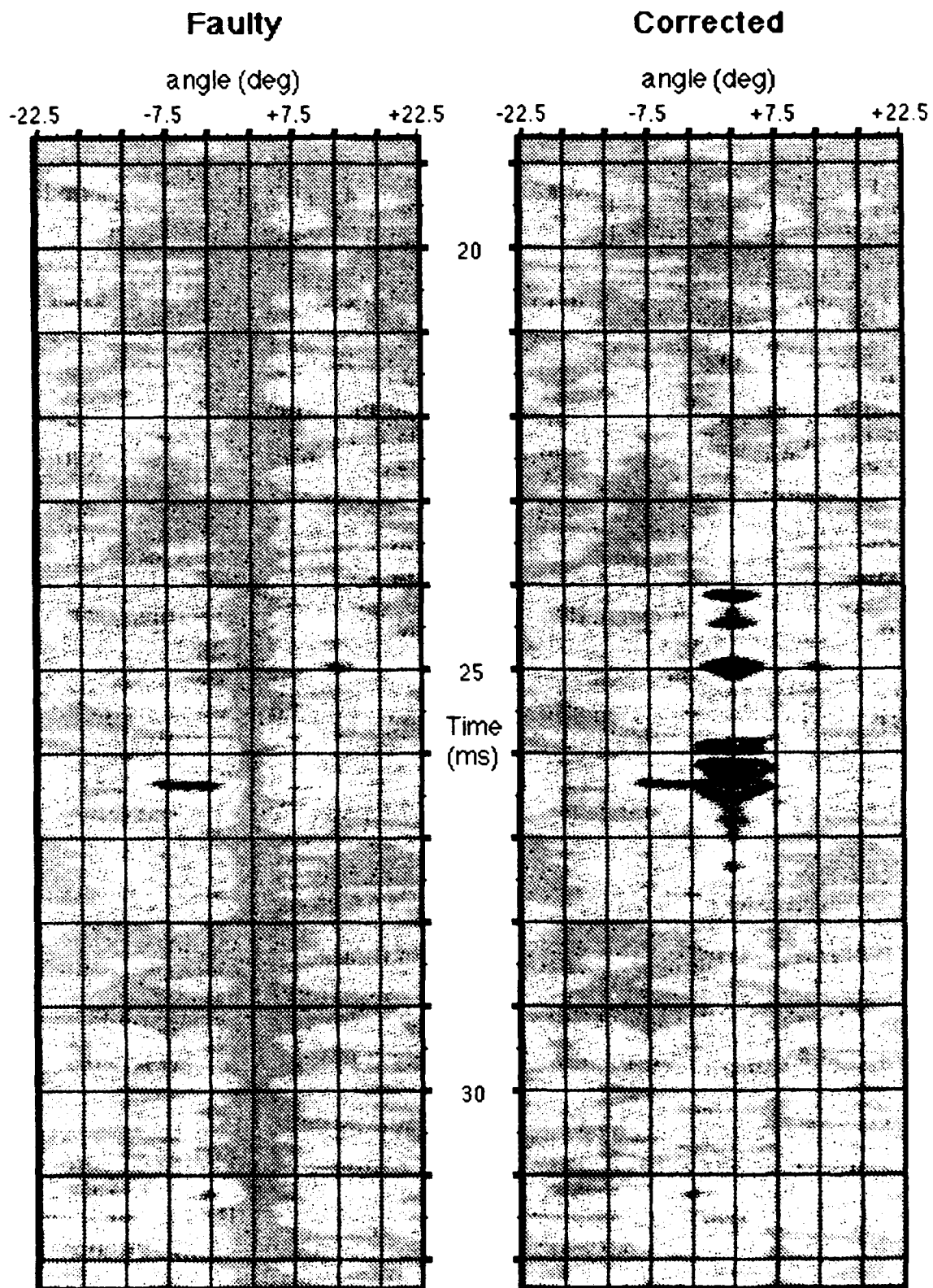


Figure 4.14: Faulty output with processor 6 zeroed and corrected output

reduces the efficiency of the beamformer.

An additional problem with the VE-32C architecture is that only one DSP32C per board can read data from the input FIFO at a time. The placement of the working processors and the checksum processors in figure 4.6 is capable of overcoming this problem when configured as a MISF fault-tolerant architecture. In this case, advantage is taken of the zero weights; if weight $w_{j,k} = 0$, then checksum processor j does not need to read working processor k 's input. This elegant layout is not possible using SIMF fault-tolerance, however, since all processors receive the same input. In this case, the same input must be transmitted four times. Many problems were encountered in the initial development of the MISF fault-tolerant prototype when attempting to read from and write to the FIFO's. It was found that these problems resulted most directly from a lack of synchronization, more precisely when a DSP tried to read from a FIFO at the same time another was writing to it. To combat these problems, the 68030 microprocessor host was set up to control all transfers.

Several programming problems appeared throughout the implementation. The DSP32C's come with an application library complete with several functions, though several of them could not be used. The version of the DSP32C's used for this application had a problem utilizing a particular two-way communication link between them and the 68030 host. The parallel I/O interrupt register (PIR) is set up to "flag" the DSP's when an operation must be performed. Unfortunately, the DSP's were capable of writing to this register, but could not read from it. This made it impossible to use some of the subroutines in the application library, including *_firc* used to implement a complex FIR filter. To overcome this communications problem, a section of memory was reserved for sending and receiving commands between the host and the DSP's. In addition to these problems, the compiler used for the DSP32C's was an earlier version, and had a relatively small symbol table. Errors resulting from exceeding the size of this table were particularly difficult to catch, and required the programs to be split into smaller, separately compiled pieces.

4.6 Theoretical overhead

In an idealized system with no extraneous overheads for memory access, communication, synchronization, or scheduling, it is not difficult to calculate the percentage of overhead dedicated to fault-tolerance [2, 8].

Using weight matrix (2.12) with a batch of p real data points per hydrophone, calculation of the syndromes requires $p(N+C)C = 39p$ multiply-adds. Using symmetry, computation of the cross-correlations requires $p(C+1)C/2 = 6p$ multiply-adds. Summing the auto-correlations to test for failure accounts for 2 adds, and each likelihood requires $C^2 - 1 = 8$ additions and 1 scaling. In addition, there are 13 comparisons between the likelihoods. Correcting the output, in theory, requires up to $Cp = 3p$ additions, and p scalings. The beamformers implement a complex FIR filter on complex data prior to summing, and hence require $qN_e p = 120p$ multiply-adds, and $(N_e - 1)p = 14p$ additions. There are $C = 3$ processors forming redundant beams. Since the GLRT computations are not protected by this configuration, they would have to be performed in triplicate. The ratio of work done by the $N = 10$ processors is thus about:

$$\frac{3}{10} + \frac{24p + 3(6p + 13 \cdot 9 + 15 + 4p)}{10 \cdot 134p} \quad (4.33)$$

Because beamforming is very computationally intensive, the theoretical overhead is fairly minimal. In fact, for $p = 2N_s = 68$, the overhead is a mere 36%.

Chapter 5

Conclusions and Recommendations for Future Work

The purpose of this multidirectional sonar beamformer implementation was to utilize the principles of multiple input, single function fault-tolerance, and apply them to a different configuration consisting of a single input with multiple functions for an actual application. This application was then to demonstrate that the usefulness of analytic fault-tolerance is indeed much greater than previously acknowledged, as it clearly did. In addition, this application opened the window to a more generic use of analytic fault-tolerance, where entirely different applications may be joined to reduce the overhead necessary for error detection and correction. Combined with earlier results [2], the results from this thesis show that the maximum likelihood algorithm of section 2.2 can be used to protect a combination of SIMF and MISF configurations.

The digital interpolation beamformer chosen for this application is not closed to improvements. Several of the shorter subroutines could be replaced with more efficient assembler macros, and the C-code itself has not been optimized. The beamformer architecture implemented here itself is not fault-tolerant, but serves to demonstrate the principles of SIMF fault-tolerance. In actuality, if, say, one of the VE-32C boards were to fail, that would account for 4 processor failures, which the current configuration

could not handle.

As a topic of future research, the concept of analytic fault tolerance, either for MISF or SIMF (or a combination of both), may be extended to nonlinear systems through the use of homomorphic analysis [10] or linearization, for example. As a further extension, then, these nonlinear systems may be combined in a particular configuration with linear systems, offering a very broad range of analytic fault-tolerance.

Bibliography

- [1] A. Aliphas. "IR&D Project No. 345 Proposal Report #92031398". C. S. Draper Laboratory, January 1991.
- [2] A. Aliphas, A. J. Wei, and B. R. Musicus. "A 16-Processor Prototype for a Fault Tolerant Parallel Digital Signal Processor". Technical Report CSDL-P-3039, C. S. Draper Laboratory, Cambridge, MA, March 1991.
- [3] R. E. Crochiere and L. R. Rabiner. "Interpolation and Decimation of Digital Signals—A Tutorial Review". *Proceedings of the IEEE*, 69:300–331, March 1981.
- [4] D. Eyring and P. Rosenstrach. "Lake Champlain Forward-Look Sonar Data Collection". Technical Report EJC 91-1106, C. S. Draper Laboratory, Cambridge, MA, October 1991.
- [5] O. D. Grace and S. P. Pitt. "Quadrature Sampling of High-Frequency Waveforms". *Journal of the Acoustic Society of America*, 44(5):1453–1454, 1968.
- [6] W. C. Knight, R. G. Pridham, and S. M. Kay. "Digital Signal Processing for Sonar". *Proceedings of the IEEE*, 69:1451–1506, November 1981.
- [7] The MathWorks, Inc. *Pro-Matlab User's Guide*, 3.5 edition.
- [8] B. R. Musicus, A. Aliphas, and A. J. Wei. "A Prototype for a Fault Tolerant Parallel Digital Signal Processor". In *Proceedings of the Conference on Application Specific Array Processors*, pages 518–529. IEEE Computer Society, September 1990.

- [9] B. R. Musicus and W. S. Song. "A Fault-Tolerant Multiprocessor Architecture for Digital Signal Processing Applications". Technical Report RLE-TR-552, Massachusetts Institute of Technology, Cambridge, MA, March 1990.
- [10] A. V. Oppenheim and R. W. Schaffer. *Discrete-Time Signal Processing*. Prentice Hall, New Jersey, 1989.
- [11] J. Paradiso. "Sonar Application of a Quadrature-Sampled Interpolation Beamformer". Technical Report EJB 90-191, C. S. Draper Laboratory, Cambridge, MA, December 1990.
- [12] R. G. Pridham and R. A. Mucci. "A Novel Approach to Digital Beamforming". *Journal of the Acoustic Society of America*, 63:425-434, February 1978.
- [13] R. G. Pridham and R. A. Mucci. "Digital Interpolation Beamforming for Low-Pass and Bandpass Signals". *Proceedings of the IEEE*, 67:904-919, June 1979.
- [14] R. G. Pridham and R. A. Mucci. "Shifted Sideband Beamformer". *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-27:713-722, December 1979.

Appendix A

Source Code

Various programs, subprograms, and header files were written for the DSP32C's and the 68030 host. The usage, along with any dependencies, are listed in table A.1 for clarity with the actual source code following.

Table A.1: Source files listed by processor, with dependencies

file	processor(s)	dependencies and/or include files
genb.c	input	genb32c.h, libb.h
mod_genb.c	input	dsp32c.h
beam.c	beam	filter.c, dsp32c.h
csp.c	csp	filter.c, dsp32c.h
filter.c	beam, csp	my_macros.h
write_coefX.c	beam	dsp32c.h, libb.h, fircoef.h, myP.h
write_cspZ.c	csp	dsp32c.h, libb.h, fircoef.h, myP.h
beam_mle.c	mle	dsp32c.h
write_weights.c	mle	dsp32c.h
beamtrans.c	68030	rw.c, util.c, dspstruct.h, macros.h, scrn.h
dsp32c.h	beam, csp, input, mle	my_macros.h

Because they were written prior to work on this application began to facilitate memory access and debugging, rw.c and util.c are not listed in this appendix. In addition, genb32c.h, fircoef.h, and myP.h are not listed. The former differs from dsp32c.h by a single constant redefinition, and the latter two merely contain the matrices of subfilter coefficients, and whole integer delays for each sensor in all channels, respectively.

/* GENB.C - generates data for other processors

This starts by waiting for the host to send Ns via COMBUF.
It generates a block of $2N_e N_s$ samples of fixed-point complex data
(in external memory, so the 68030 can examine it if necessary).
It then waits for a GO_XMIT from the 68030, and sends the data,
1 block at a time, to its output.
It then loops back and repeats forever.
Acknowledges are returned to the 68030 via PIR as soon as possible.

COMMAND BUFFER USE:

COMBUF0: commands

COMBUF1: Ns (# complex data samples/sensor to generate)

COMBUF2: most recent error code

COMBUF3: count of sets of packets computed so far

COMBUF4: count of packets transmitted so far

COMBUF5: b (max # whole integer delays in working channels)

*/

#include "genb32c.h"

#include <libb.h>

/* Define Communications buffer assignments */

#define COM_CMD 0

#define COM_Ns 1

#define COM_ERR 2

#define COM_SET 3

#define COM_PACK 4

#define COM_b 5

extern void put_PIR(); /* external subroutine to comm. with 68030 */

#define Theta 0 /* steered angle off broadside */

main()

{

int count, Ns, Nout;

register int i, j, nes;

register unsigned short *Port, command, *Combuf;

register short int *In, *Ptr;

float cond, ti, tq, tau, tht;

register float s_arg;

/* Initialize buffer pointers */

In = (short int *)IBUF;

Port = (unsigned short *)PORT;

Combuf = (unsigned short *)COMBUF0;


```

/* Initialize PIR, COMBUF areas */
put_PIR(0);
Combuf[COM_CMD] = 0;
Combuf[COM_Ns] = 0;
Combuf[COM_ERR] = 0;
Combuf[COM_SET] = 0;
Combuf[COM_PACK] = 0;
Combuf[COM_b] = 0;
Nout = 0;
Ns = 0;
tht = (float)(Theta*M_PI/180);
nes = (int)(Ne/2);
count = 5;

/* Enter the main loop, waiting for commands from the host */
for(;;){

/* Check if new data is needed */
if(count > 4) {
    Ptr = In;
    for(count = 1; count<=4 ; count++) {
        i = Ns;
        while(i-->0) { /* condition the data */
            if (mod((float)(LS*(Ns-1-i)),2.0)==0.0)
                cond = 1.0;
            else
                cond = -1.0;
            /* quadrature sampled 1st */
            tq = (float)((Ns-1-i)*DELTA);
            /* in-phase sampled 2nd */
            ti = (float)(tq - (1.0/(4.0*Fc)));
            for (j = -nes; j <= nes; j++)
            {
                s_arg = (float)((ASPC*j)-tht-M_PI_2);
                tau = (float)(-mod_sin(s_arg)*WRAD/C);
                s_arg = -mod_sin((float)(W0*(ti-tau)));
                *Ptr++ = (short int)(2048.0 * s_arg * cond); /* in-phase */
                s_arg = mod_sin((float)(W0*(tq-tau)));
                *Ptr++ = (short int)(2048.0 * s_arg * cond); /* quad. */
            }
        }
    }
    count = 1;
    Ptr = In;
}

```

```

        if(Ns > 0) {
            Combuf[COM_SET]++;
        }
    }

/* Main dispatch loop - wait for command in COMBUF0 */
for(;;) {

    /* Extract command code, dispatch to handler */
    do{} while((command = *Combuf) == 0 || command == ACK_ERR);
    i = (command & 0xff00);
    *Combuf = 0;

    /* Do the dispatch */
    if(i == GO_N) {
        Ns = (int) Combuf[COM_Ns];
        Nout = (int)(2*Ns*Ne);
        put_PIR(command);
        count = 5;
        break;

    } else if(i == GO_XMIT) {
        for (j=0; j++ < Nout;)
            *Port = (short int)(*Ptr++);
        put_PIR(command);
        Combuf[COM_PACK]++;
        if(count++ == 4)
            break;

    } else {
        Combuf[COM_ERR] = command;
        put_PIR(ACK_ERR);
    }
}
}
}

```

/* MOD_GENB.C - transmits data for other processors

This starts by waiting for the host to send Ns via COMBUF.
 It then waits for a GO_XMIT from the 68030, and sends the data,
 1 block at a time, to its output.
 It then loops back and repeats forever.
 Acknowledges are returned to the 68030 via PIR as soon as possible.

```

COMMAND BUFFER USE:
COMBUF0: commands
COMBUF1: Ns (# complex data samples/sensor to generate )
COMBUF2: most recent error code
COMBUF3: count of sets of packets computed so far
COMBUF4: count of packets transmitted so far
COMBUF5: b (max # whole integer delays in working channels)
*/

```

```

#include "dsp32c.h"

```

```

/* Define Communications buffer assignments */

```

```

#define COM_CMD  0
#define COM_Ns   1
#define COM_ERR  2
#define COM_SET  3
#define COM_PACK 4
#define COM_b    5
extern void put_PIR();

```

```

main()

```

```

{
    int batch,count,Ns,Nout;
    register int i,j;
    register unsigned short *Port,command,*Combuf;
    register short int *In,*Ptr;

```

```

/* Initialize buffer pointers */

```

```

    In = (short int *)IBUF;
    Port = (unsigned short *)PORT;
    Combuf = (unsigned short *)COMBUF0;

```

```

/* Initialize PIR, COMBUF areas */

```

```

    put_PIR(0);
    Combuf[COM_CMD] = 0;
    Combuf[COM_Ns] = 0;
    Combuf[COM_ERR] = 0;
    Combuf[COM_SET] = 0;
    Combuf[COM_PACK] = 0;
    Combuf[COM_b] = 0;
    Nout = 0;
    Ns = 0;
    count = 1;
    batch = -1;

```

```

/* Enter the main loop, waiting for commands from the host */
for(;;){
    count = 1;
    if(Ns > 0) {
        batch += 1;
        Combuf[COM_SET]++;
    }

/* Main dispatch loop - wait for command in COMBUF0 */
    for(;;) {

/* Extract command code, dispatch to handler */
        do{} while((command = *Combuf) == 0 || command == ACK_ERR);
        i = (command & 0xff00);
        *Combuf = 0;

/* Do the dispatch */
        if(i == GO_N) {
            Ns = (int) Combuf[COM_Ns];
            Nout = (int)(2*Ns*Ne);
            put_PIR(command);
            break;

        } else if(i == GO_XMIT) {
            Ptr = In + batch*Nout;
            for (j=0; j++ < Nout;)
                *Port = (short int)(*Ptr++);
            put_PIR(command);
            Combuf[COM_PACK]++;
            if(count++ == 4)
                break;

        } else {
            Combuf[COM_ERR] = command;
            put_PIR(ACK_ERR);
        }
    }
}

/* BEAM.C(CSP.C) - Computes a beam response of the input data
   in the working(checksum) processors, sends to output.

```

This starts by waiting for the host to send Ns, b via the COMBUF.
It waits for GO_RCV from the host, then reads multiple blocks of

2Ne samples of fixed-point data from its input FIFO. (Puts data in external memory, so the 68000 can examine it if necessary). It converts to floating point, interpolates, then finds the on-axis value. If it is chosen as the faulty processor, it then injects the const. error into the output buffer. It performs overlap-save convolution by maintaining a circular data buffer. It then waits for a GO_XMIT from the 68000, and sends the data in floating point format, 16-bits at a time, to its output FIFO. It then loops back and repeats forever. Acknowledges are sent to the 68000 as soon as possible

COMMAND BUFFER USE:

COMBUF0 : commands
 COMBUF1 : Ns (# complex data samples/sensor for each batch)
 COMBUF2 : last error command
 COMBUF3 : count of sets of packets computed so far
 COMBUF4 : fault type, sent from host
 COMBUF5 : b (max # of whole integer delays in working channels)
 COMBUF6 : 1 if faulty processor, 0 otherwise
 COMBUF7 : failure sample, sent from host
 COMBUF0f: real fault value
 COMBUF1f: imaginary fault value

*/

```
#include "dsp32c.h"
```

```
#include <libb.h>
```

```
/* Define communications buffer assignments */
```

```
#define COM_Ns      1
#define COM_ERR     2
#define COM_SET     3
#define COM_FAULT   4
#define COM_b       5
#define COM_PROC    6
#define COM_SAMP    7
extern void put_PIR();
```

```
main()
```

```
{
    short int errproc, errflag, samp;
    int      Ns, b, nes, maxlen, pt, pad, Nout, points;
    register unsigned short int *Port, command, *Combuf;
```

```

register short int In, *beam_io;
register int start, n, m;
COMPLEX *Data, *Beam, *magn;
float *Mag, *freal, *fimag;

/* Initialize pointers */
Port = (unsigned short int *)PORT;
Data = (COMPLEX *)IBUF;
Beam = (COMPLEX *)BEAM;
magn = (COMPLEX *)BBUF;
Mag = (float *)MBUF;
freal = (float *)COMBUF0f;
fimag = (float *)COMBUF1f;
Combuf = (unsigned short *)COMBUF0;

/* Initialize PIR, COMBUF areas */
put_PIR(0);

for (n=0; n<12; n++)
    Combuf[n] = 0;

errflag = samp = 0;
nes = (int)((int)(Ne)/2);
errproc = Nout = maxlen = points = start = pad = 0;

/* Main dispatch loop - wait for command in COMBUF0 */
for(;;) {

    /* Extract command code, dispatch to handler */
    do{} while((command = *Combuf) == 0 || command == ACK_ERR);
    m = (command & 0xff00);
    *Combuf = 0;

    /* New Value of Ntot */
    if(m == GO_N) {
        Ns = (int) Combuf[COM_Ns];
        b = (int) Combuf[COM_b];
        errproc = Combuf[COM_PROC];
        errflag = Combuf[COM_FAULT];
        samp = Combuf[COM_SAMP];
        Nout = (int)(Ns*4);
        pad = (int)(ORDER + b);
        maxlen = (int)(Ns + pad);
        if (Combuf[COM_SET] == 0) {
            points = (int)(Ns);
        }
    }
}

```

```

        start = (int)(pad);
        for (m = 0; m < pad; m++) {
            Beam[m].real = 0.0;
            Beam[m].imag = 0.0;
        }
    }
    put_PIR(command);

/* Receive and process new packet, Convert to Floating Point */
} else if(m == GO_RCV) {
    for (m = 0; m < Ns; m++) {
        for (n = 0; n < Ne; n++) {
            pt = (int)(n*maxlen + pad + m);
            In = (short int)(*Port);
            Data[pt].real = (float)(In);
            In = (short int)(*Port);
            Data[pt].imag = -(float)(In);    /* X = Xr - jXi */
        }
    }

    /* Digital Interpolation Beamforming, Data Cycling */

    for (n = 0; n < Ne; n++) {        /* Interp. & Form Beam */
        pt = (int)(n*maxlen);
        filter(&Data[pt+start], &Beam[start], points, n, b, Combuf[COM_SET]);
        for (m = 0; m < pad; m++)    /* Cycle data */
            Data[pt] = Data[Ns+(pt++)];
    }

    if (errproc > 0) {
        for (m = 0; m < Ns; m++) {
            if ((errflag==1) || ((samp==m) && (errflag==3))) {
                magn[m].real += *freal; magn[m].imag += *fimag;
            } else if (errflag==2) {
                magn[m].real = *freal; magn[m].imag = *fimag;
            }
        }
    }

    /* Subsequent filters over all maxlen pts. */
    points = (int)(maxlen);
    start = 0;
    put_PIR(command);
    Combuf[COM_SET]++;

/* Transmit finished packet, 16 bits at a time */
} else if(m == GO_XMIT) {

```

```

    beam_io = (short int *) (BBUF);
    *Mag = 0.0;
    for (m=0; m < Ns; m++) {
        *Mag += (float)((magn[m].real * magn[m].real +
                        magn[m].imag * magn[m].imag)/Ns);
        for (n=0; n < 4; n++)
            *Port = *beam_io++;
    }
    put_PIR(command); /* acknowledge quickly */

} else {
    Combuf[COM_ERR] = command;
    put_PIR(ACK_ERR);
}
}
}

#include "my_macros.h"

void filter(in,out,N,sensor,b,batch) /* subroutine to implement */
COMPLEX *in, *out;                  /* complex FIR filter.      */
int N, sensor, b;
unsigned short batch;
{
    COMPLEX *fir, *coef, sum, *data = in;
    register int t, k, n = 0;
    int step;

    step = (int)(FILTBUF + sensor*(b+1)*LEN*8); /* 8 bytes/complex pt. */
    fir = (COMPLEX *) (step);

    do {
        if (sensor == 0) {
            out->real = 0.0; /* set output to 0 if 1st hydrophone */
            out->imag = 0.0;
        }
        in = data + n; /* ...reset data ptr */
        k = (int)(min(n,(ORDER+b)));
        t = (int)(max(0,(n-ORDER)));
        sum.real = 0.0; sum.imag = 0.0;
        if (n < (ORDER+b))
            do {
                if ((k >= b) || (batch > 0))
                    coef = fir + b*LEN + (n-k);
                else

```



```

        coef = fir + k*LEN + (n-k);
        sum.real += (float)((((in->real * coef->real) -
                                (in->imag * coef->imag)));
        sum.imag += (float)((((in->real * coef->imag) +
                                (in->imag * coef->real)));

        in--;
    } while (k-- > t);
else {
    coef = fir + b*LEN;
    do {
        sum.real += (float)((((in->real * coef->real) -
                                (in->imag * coef->imag)));
        sum.imag += (float)((((in->real * coef->imag) +
                                (in->imag * coef->real)));

        in--; coef++;
    } while (k-- > b);
}
out->real += (float)(sum.real/Ne); /* normalize by Ne */
out++->imag += (float)(sum.imag/Ne);
} while (++n < N);
}

/* WRITE_COEFX.C : Computes complex filter coefficients needed for
                   working processor #X, and writes to external
                   memory to be used by BEAM.C
*/

#include "dsp32c.h"
#include <libb.h>
#include "fircoef.h" /* matrix of subfilter coefficients */
#include "myP.h"      /* matrix of whole integer delays */

/* Define communications buffer assignments */

#define T X /* selects look angle #X: theta = -22.5 + T*5 */

main()
{
    int diff, step, nes, Q, M, M_max;
    register int j, n, m;
    register float tau, arg;
    COMPLEX *filt, *Fir, *Coef, phase;
    float mx, Theta;

```

```

/* Initialize pointers */
Coef = (COMPLEX *)FILTBUF;

Theta = (float)(theta[T]*M_PI/180);
nes = (int)((int)(Ne)/2);
step = (int)(b + 1);

mx = (float)(ASPC*nes) + (float)(abs(Theta)) - (float)(M_PI_2);
M_max = round((float)(WRAD*mod_sin(mx)/(C*DEL)));
for (n = 0; n < Ne; n++) {
    Fir = Coef + n*step*LEN;
    arg = (float)((n - nes)*ASPC - Theta - M_PI_2);
    tau = (float)((WRAD/C)*mod_sin(arg));
    arg = (float)(W0*tau);
    phase.real = -mod_sin(arg-M_PI_2); /* = cos(arg) */
    phase.imag = mod_sin(arg);
    M = M_max - round(tau/DEL);
    if ((Q = (int)((P[T][n]+1)*L - M)) == (int)(L))
        Q = 0;
    for (j = 0; j < step; j++) {
        filt = Fir + j*LEN;
        diff = (int)(j - P[T][n]);
        if ((diff >= 0) && (diff <= b)) {
            for (m = 0; m < LEN; m++) {
                filt->real = (float)(fircoef[Q][m] * phase.real);
                filt->imag = (float)(fircoef[Q][m] * phase.imag);
                filt++;
            }
        }
        else {
            for (m = 0; m < LEN; m++) {
                filt->real = 0.0;
                filt->imag = 0.0;
                filt++;
            }
        }
    }
}

/* WRITE_CSPZ.C :      calculates the summed complex filter
                        coefficients for CSP #Z, and writes
                        them to external memory for use by CSP.C
*/
#define N 10          /* number of working channels */

```

```

/* weights used for Zth CSP */
int wt[N] = {1, 1, 1, 1, 0, 0, 1, 1, 1, 1};
#include "dsp32c.h"
#include <libb.h>
#include "fircoef.h" /* matrix of subfilter coefficients */
#include "myP.h"      /* matrix of whole integer delays */

main() {
    int nes, Q, M, M_max, step, diff;
    register int j, k, m, n;
    register float tau, arg;
    COMPLEX *fir, *coef, *filt, phase;
    float mx, tht;

    coef = (COMPLEX *)FILTBUF;
    nes = (int)((int)(Ne)/2);
    step = (int)(b + 1);

    for (n = 0; n < Ne; n++) {
        fir = coef + n*step*LEN;
        for (k = 0; k < N; k++) {
            tht = (float)(theta[k] * M_PI/180);
            mx = (float)(ASPC*nes) + (float)(abs(tht)) - (float)(M_PI_2);
            M_max = round((float)(WRAD*mod_sin(mx)/(C*DEL)));
            arg = (float)((n - nes)*ASPC - tht - M_PI_2);
            tau = (float)((WRAD/C)*mod_sin(arg));
            arg = (float)(WO*tau);
            phase.real = -mod_sin(arg-M_PI_2); /* = cos(arg) */
            phase.imag = mod_sin(arg);
            M = M_max - round(tau/DEL);
            if ((Q = (int)((P[k][n] + 1)*L - M)) == (int)(L))
                Q = 0;
            for (j = 0; j < step; j++) {
                filt = fir + j*LEN;
                if (k == 0) {
                    for (m = 0; m < LEN; m++) {
                        filt->real = 0.0; filt->imag = 0.0;
                        filt++;
                    }
                    filt = fir + j*LEN;
                } /* END if (k == 0) ... */
                diff = (int)(j - P[k][n]);
                if ((diff >= 0) && (diff <= b))
                    for (m = 0; m < LEN; m++) {
                        filt->real += (float)(fircoef[Q][m] * phase.real * wt[k]);

```

```

        filt->imag += (float)(fircoef[Q][m] * phase.imag * wt[k]);
        filt++;
    };
    } /* END for (j = 0; j < step; j++) ... */
    } /* END for (k = 0; k < N; k++) ... */
    } /* END for (n = 0; n < Ne; n++) ... */
} /* END main */

```

/* BEAM_MLE.C - Computes syndromes, syndrome cross-correlations, likelihoods, picks the most likely failure ... and corrects it. (N = 10, C = 3)

This starts by waiting for the host to send Ns to determine batch size via COMBUF1.

It collects 13 input packets from the FIFO. For each packet, it waits for GO_RCV from the host, then reads Ns complex floating-point data points from its input FIFO, storing them in IBUF. It accumulates the syndromes as the packets arrive, scaling by 1/16 to avoid overflow.

After computing the syndromes, it calculates the syndrome cross-correlations, storing them in the rho variables.

It then computes the likelihoods, and picks the maximum.

It computes a threshold set to 1/2 the expected energy for a one-bit error on the input. Depending on whether or not the likelihood is larger than this, it corrects the error.

Acknowledges are sent to the 68030 as soon as possible.

COMMAND BUFFER USE:

COMBUF0: commands

COMBUF1: Ns (from host: # complex samples per hydrophone)

COMBUF2: last error command

COMBUF3: count of sets of packets computed so far

COMBUF4: count of packets computed so far

COMBUF6: Most likely failed processor, or no fault (0)

COMBUF7: Most likely failed processor

*/

#include "dsp32c.h"

/* Define communication buffer assignments */

#define COM_CMD 0

#define COM_Ns 1

#define COM_ERR 2

#define COM_SET 3

#define COM_PACK 4

```

#define COM_DECIDE 6
#define COM_MLE 7

extern void put_PIR();
float scale = 1.0;

main() {
    register unsigned short int *Port,command,*Combuf;
    register short int *Inptr;
    register float maxL, *Mag;
    COMPLEX *In, *Out, *s1, *s2, *s3;
    float *R, *Gamma, *Wt, *FixW, *Rho, *Likely, test;
    register int i, j, k, m;
    int Ns, count, proc;
    void accum(), innerprod();

    /* Initialize Pointers */
    Port = (unsigned short int *)PORT;
    Combuf = (unsigned short int *)COMBUF0;
    In = (COMPLEX *)IBUF;
    Out = (COMPLEX *)CBUF;
    s1 = (COMPLEX *)OBUF;
    Rho = (float *)RBUF;
    Likely = (float *)LBUF;
    Gamma = (float *) (LBUF + 13*4);
    Wt = (float *)WBUF;
    R = (float *) (WBUF + 4*39);
    FixW = (float *) (WBUF + 4*52);
    Mag = (float *)MBUF;

    /* Initialize PIR, COMBUF areas */
    put_PIR(0);
    Combuf[COM_CMD] = 0;
    Combuf[COM_Ns] = 0;
    Combuf[COM_ERR] = 0;
    Combuf[COM_SET] = 0;
    Combuf[COM_PACK] = 0;
    Ns = 0;
    count = 0;
    test = 0.0;

    /* Main Dispatch Loop -- wait for command in COMBUF0 */
    for(;;) {

        /* extract command code, dispatch to handler */

```

```

do{} while ((command = *Combuf) == 0);
m = (command & 0xff00);
*Combuf = 0;

/* New Value of N */
if (m == GO_N) {
    Ns = (int)Combuf[COM_Ns];
    put_PIR(command);
    s2 = s1 + Ns; s3 = s2 + Ns;

/* Receive and process new packet */
} else if (m == GO_RCV) {
    m = 4*Ns;
    Inptr = (short int *) (IBUF + count*8*Ns);
    while (m-- > 0)
        *Inptr++ = (short int)*Port;
    Combuf[COM_PACK]++;
    put_PIR(command);
    i = (int)(3*count);
    j = (int)(count*Ns);
    /* Compute the syndromes, */
    /* initializes if needed */
    accum(-Wt[i], &In[j], s1, Ns, count);
    accum(-Wt[++i], &In[j], s2, Ns, count);
    accum(-Wt[++i], &In[j], s3, Ns, count);

    if (++count >= 13) { /* If all 13 packets here, do GLRT */
        k = m = 0;
        *Gamma = (float)(1.0/(scale*2448.0));
        innerprod(s1, s1, &Rho[0], Ns); /* calculate the rho's */
        innerprod(s1, s2, &Rho[1], Ns);
        innerprod(s1, s3, &Rho[2], Ns);
        innerprod(s2, s2, &Rho[4], Ns);
        innerprod(s2, s3, &Rho[5], Ns);
        innerprod(s3, s3, &Rho[8], Ns);
        /* invoke symmetry ... */
        Rho[3] = Rho[1]; Rho[6] = Rho[2]; Rho[7] = Rho[5];
        /* sum of autocorrelations */
        test = (Rho[0]+Rho[4]+Rho[8]);
        proc = 0; maxL = 0.0;
        do { /* compute the likelihoods */
            Likely[k] = 0.0;
            for (m = 0; m < 9; m++) {
                i = 3*k + m/3; j = 3*k + m%3;
                Likely[k] += Wt[j]*Wt[i]*Rho[m]*R[k];
            }
        } while (k++ < 10);
    }
}

```

```

    }
    if (Likely[k] > maxL) { /* choose faulty proc. */
        maxL = Likely[k];
        proc = k+1;
    }
} while (++k < 13);
if (test > *Gamma) /* indicate failed proc., if any */
    Combuf[COM_DECIDE] = (short int)proc;
else
    Combuf[COM_DECIDE] = 0;
Combuf[COM_MLE] = (short int)proc;

j = 0;
do { /* Correct the output */
    accum(FixW[(proc-1)*13+j], &In[j*Ns], Out, Ns, j);
} while (++j < 13);

innerprod(Out, Out, Mag, Ns);
/* Avg. mag^2 of corrected output */
*Mag = (float)(*Mag/Ns);

count = 0; test = 0.0;
Combuf[COM_SET]++;
put_PIR(GO_DONE); /* Signal that Likelihoods may be read */
}

} else {
    Combuf[COM_ERR] = command;
    put_PIR(ACK_ERR);
}
}
}

void accum(weight, data, sum, N, flag)
float weight;
COMPLEX *data, *sum;
int N, flag;
{
    register int i = 0;
    if (flag != 0)
        flag = 1;
    do {
        sum[i].real = (float)(sum[i].real*flag + data[i].real*weight);
        sum[i].imag = (float)(sum[i].imag*flag + data[i].imag*weight);
    } while (++i < N);
}

```

```

}

void innerprod(v1,v2,out,N)
COMPLEX *v1,*v2;
float *out;
int N;
{
    register int i = 0;
    *out = 0.0;
    do {
        *out += ((v1[i].real*v2[i].real)+(v2[i].imag*v1[i].imag))/scale;
    } while (++i < N);
}

/* WRITE_WEIGHTS.C : stores the weight matrix at WBUF in external
   memory on BEAM_MLE, prior to running. It is loaded and run at
   the same time as write_coef*, and write_csp*.
*/

#include "dsp32c.h"
#define total 13
#define check 3
float wt[check][total] =
    {{1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0,-1.0, 0.0, 0.0},
     {1.0, 1.0,-1.0,-1.0, 1.0, 1.0, 0.0, 0.0, 1.0,-1.0, 0.0,-1.0, 0.0},
     {1.0,-1.0, 1.0,-1.0, 1.0,-1.0, 1.0,-1.0, 0.0, 0.0, 0.0, 0.0,-1.0}};
#define ONETHIRD 0.3333333333333333
#define TWOTHIRD 0.6666666666666666
float rkk[total] = {ONETHIRD, ONETHIRD, ONETHIRD, ONETHIRD, 0.5,
                    0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0};
/* The following matrix is used when correcting faults */
float fixW[total][total] = {
    {0., -ONETHIRD, -ONETHIRD, ONETHIRD, -TWOTHIRD, 0., -TWOTHIRD, 0.,
     -TWOTHIRD, 0., ONETHIRD, ONETHIRD, ONETHIRD},
    {-ONETHIRD, 0., ONETHIRD, -ONETHIRD, 0., -TWOTHIRD, 0., -TWOTHIRD,
     -TWOTHIRD, 0., ONETHIRD, ONETHIRD, -ONETHIRD},
    {-ONETHIRD, ONETHIRD, 0., -ONETHIRD, 0., TWOTHIRD, -TWOTHIRD, 0.,
     0., -TWOTHIRD, ONETHIRD, -ONETHIRD, ONETHIRD},
    {ONETHIRD, -ONETHIRD, -ONETHIRD, 0., TWOTHIRD, 0., 0., -TWOTHIRD,
     0., -TWOTHIRD, ONETHIRD, -ONETHIRD, -ONETHIRD},
    {-1., 0., 0., 1., 0., 0.,-0.5, 0.5,-0.5, 0.5, 0., 0.5, 0.5},
    { 0.,-1., 1., 0., 0., 0., 0.5,-0.5,-0.5, 0.5, 0., 0.5,-0.5},
    {-1., 0.,-1., 0.,-0.5, 0.5, 0., 0.,-0.5,-0.5, 0.5, 0., 0.5},
    { 0.,-1., 0.,-1., 0.5,-0.5, 0., 0.,-0.5,-0.5, 0.5, 0.,-0.5},
    {-1.,-1., 0., 0.,-0.5,-0.5,-0.5,-0.5, 0., 0., 0.5, 0.5, 0.},

```



```

    { 0., 0., -1., -1., 0.5, 0.5, -0.5, -0.5, 0., 0., 0.5, -0.5, 0.},
    { 1., 1., 1., 1., 0., 0., 1., 1., 1., 1., 0., 0., 0.},
    { 1., 1., -1., -1., 1., 1., 0., 0., 1., -1., 0., 0., 0.},
    { 1., -1., 1., -1., 1., -1., 1., -1., 0., 0., 0., 0., 0.}
};
main() {

    float *weight, *Rkk, *Fix;
    register int i, j;

    weight = (float *)WBUF;
    Rkk = (float *) (WBUF + 4*39);
    Fix = (float *) (WBUF + 4*52);

    for (i = 0; i < total; i++) {
        *Rkk++ = rkk[i];
        for (j = 0; j < total; j++) {
            if (j < check)
                *weight++ = (float)(wt[j][i]/16.0);
                /* write weights in column major order */
            *Fix++ = fixW[i][j];
                /* write fixW in row major order      */
        }
    }
}

/*****
/*
/*   DSP32c.h  :: Include file for the DSP's
/*
/*
*****/

/* The FIFO port address */
#define PORT      0xAC000

/* Define the codes for communication with the DSP32C's
   Upper 8-bits = code
   Lower 8-bits = transaction number
*/
#define GO_N      0x0100
#define GO_XMIT 0x0200
#define GO_RCV   0x0300
#define GO_DONE  0x0400

#define ACK_ERR  0xffff

```

```

/* External memory data buffer addresses */
/* Define the communication buffer area DSP32C addresses */
/* Shorts */
#define COMBUF0  0xffffe0
#define COMBUF1  0xffffe2
#define COMBUF2  0xffffe4
#define COMBUF3  0xffffe6
#define COMBUF4  0xffffe8
#define COMBUF5  0xfiffea
#define COMBUF6  0xffffec
#define COMBUF7  0xffffee
/* Floats */
#define COMBUF0f 0xfffff0
#define COMBUF1f 0xfffff4
#define COMBUF2f 0xfffff8
#define COMBUF3f 0xfffffc

#include "my_macros.h"
#define ASPC      0.0523598776 /* angle between elements (radians) */
#define Fc        200000.0      /* carrier frequency in Hz */
#define DELTA     5.00e-5       /* quad. sampling period (sec) */
#define DEL       7.142857143e-6 /* interp. samp. period (sec) */
#define WRAD      0.5729166667 /* array radius in feet */
#define C         5000.0        /* speed of sound in feet/sec */
#define W0        1.2566370614e+6 /* Sonar freq. in radians */
#define IBUF      0x008000      /* Stores incoming quadrature data */

/* These defines for beam, csp channels only */
#define BBUF      0x00b040      /* Beam Output, starts at BEAM + ORDER */
#define BEAM      0x00b000      /* Interpolation results */
#define MBUF      0x00d800      /* average magnitude over results */

/* These defines for beam_mle channel only */
#define OBUF      0x00a000      /* Syndrome buffers in beam_mle */
#define RBUF      0x00b000      /* Syndrome cross-corr. in beam_mle */
#define CBUF      0x00c000      /* Corrected proc. output in beam_mle */
#define LBUF      0x00d000      /* Likelihoods and thresh. in beam_mle */
#define WBUF      0x00e000      /* Weights used by beam_mle */

float theta[] = {-22.5,-17.5,-12.5,-7.5,-2.5,2.5,7.5,12.5,17.5,22.5};

/* MY_MACROS.H */
#define M_PI      3.14159265358979323846

```

```

#define M_PI_2  1.57079632679489661923
#define M_1_PI  0.31830988618379067154
#define L      7          /* interpoltaiion ratio      */
#define LEN    9          /* length(PolyPhase Filter) */
#define ORDER  8          /* LEN - 1                  */
#define Ne     15         /* number of hydrophones    */

#define min(e1,e2)  (float)(e1) <= (float)(e2) ? (float)(e1):(float)(e2)
#define max(e1,e2)  (float)(e1) >= (float)(e2) ? (float)(e1):(float)(e2)
#define abs(e1)     (float)(e1) >= 0.0 ? (float)(e1):(float)(-e1)
#define sign(e1)    (float)(e1) >= 0.0 ? (int)(1):(int)(-1)

#define FILTBUF 0x00c000      /* Filter Coefficient Storage */

typedef struct {
    float real;
    float imag;
} COMPLEX;

/* LIBB.H */
extern float mod(), mod_sin(), sqrt();
extern int round();
extern void filter();

/* BEAMTRANS.C - The main subroutine which coordinates the fault
   tolerance application for N=10, C=3.

   The program runbeam(start,stop) sequences all activity
*/

#define DEBUG      1
#define b_max      0 /* maximum number of whole integer delays */
#define maxbat 16    /* maximum number of data batches allowable */
#define endbat 47    /* last batch for which there is data */
#include <stdio.h>
#include <math.h>
#include "dspstruct.h"
#include "macros.h"
#include "scrn.h"

int taberr_short();
static void magnitudes();
struct tabtrans transferD[],transferB[];

/* Help subroutine for this file */

```

```

void tt_help()
{
    printf("FAULT TOLERANCE APPLICATION SEQUENCER (tabtrans.c)\n");
    printf(" runbeam() - run all trans. once\n");
    printf(" runbeam(nrun) - run all trans. nrun times\n");
    printf(" runbeam(start,stop) - run trans. start through stop\n");
    printf(" readcom() - disp. and interpret all comm. buffers\n");
    printf(" readbeam(board,proc) - disp. buffers for beam.c proc.\n");
    printf(" readbeam_all() - disp. buffers for all beam.c procs.\n");
    printf(" readcsp(board) - disp. buffers for given csp.c proc.\n");
    printf(" readcsp_all() - disp. buffers for all csp.c procs.\n");
    printf(" readbeam_mle() - disp. buffers for beam_mle.c proc.\n");
    printf(" weights() - prints weight matrix from beam_mle proc.\n");
    printf("\n");
    return;
}

/* Define struct which keeps track of which application has been
   loaded into the DSP32C's
*/
int Nappl = -1;
struct appl apples[] = {
    {"N=1, C=0 ",
     transferD, 1,          /* transactions table and length */
     1,                  /* number of packets transmitted by input */
     1, 3,               /* input (board,proc) */
     0,                  /* number of packets input by beam_mle */
     2, 3,               /* beam_mle (board,proc) */
     0,                  /* number of beam processors */
     {0,0,0,0, 0,0, 0,0, 0,0, 0,0}, /* beam boards in order */
     {0,0,0,0, 0,0, 0,0, 0,0, 0,0}, /* beam procs in order */
     0,                  /* number of csp processors */
     0,                  /* number of packets input by each csp */
     {0,0,0,0},          /* csp boards in order */
     {0,0,0,0}},         /* csp procs in order */

    {"N=10, C=3 (weights 0,+ -1)",
     transferB, 17,       /* transactions table and length */
     4,                  /* number of packets transmitted by input */
     1, 3,               /* input (board,proc) list */
     13,                 /* number of packets input by beam_mle */
     2, 3,               /* beam_mle (board,proc) list */
     10,                 /* number of beam processors */
     {0,0,0,0, 1,1, 2,2, 3,3, -1,-1}, /* beam boards in order */
     {0,1,2,3, 1,2, 1,2, 1,2, -1,-1}, /* beam procs in order */

```

```

        3,                /* number of csp processors */
        1,                /* number of packets input by each csp */
        {1,2,3,-1},      /* csp boards in order */
        {0,0,0,-1}},     /* csp procs in order */

        {NULL, NULL, 0, 0}
    };

/* Define hardware addresses for the 16 DSP32C's PIO reg. banks */
#define PIO0_0 (struct dspstruct *) (board0+0x00)
#define PIO0_1 (struct dspstruct *) (board0+0x40)
#define PIO0_2 (struct dspstruct *) (board0+0x80)
#define PIO0_3 (struct dspstruct *) (board0+0xc0)

#define PIO1_0 (struct dspstruct *) (board1+0x00)
#define PIO1_1 (struct dspstruct *) (board1+0x40)
#define PIO1_2 (struct dspstruct *) (board1+0x80)
#define PIO1_3 (struct dspstruct *) (board1+0xc0)

#define PIO2_0 (struct dspstruct *) (board2+0x00)
#define PIO2_1 (struct dspstruct *) (board2+0x40)
#define PIO2_2 (struct dspstruct *) (board2+0x80)
#define PIO2_3 (struct dspstruct *) (board2+0xc0)

#define PIO3_0 (struct dspstruct *) (board3+0x00)
#define PIO3_1 (struct dspstruct *) (board3+0x40)
#define PIO3_2 (struct dspstruct *) (board3+0x80)
#define PIO3_3 (struct dspstruct *) (board3+0xc0)

/* Define table of data packet transfers */
/* struct tabtrans {
    unsigned short int IOboard, SWboard0,SWboard1,SWboard2,SWboard3;
    struct dspstruct *pPIOtrans;
    struct dspstruct *pPIOrcv0, *pPIOrcv1, *pPIOrcv2, *pPIOrcv3;
    int transboard;
    char *comment;
}; */

/* Define table of data packet transfers for N=1, C=0 application */
static struct tabtrans transferD[] = {

{ IO_Brd0,
  SW_In0, SW_Out3 | SW_OutEn | SW_LED, NULL, NULL,
  PIO1_3, PIO0_0, NULL, NULL, NULL, 1,
  "Input data from DSP[1][3] to DSP[0][0]"}

```

```
};
```

```
/* This is the table defining the transfers for N=10, C=3 */
```

```
static struct tabtrans transferB[] = {
```

```
{ IO_Brd0 & IO_Brd1 & IO_Brd2 & IO_Brd3,  
  SW_In0,  SW_In0 | SW_Out3 | SW_OutEn | SW_LED,  SW_In0,  SW_In0,  
  PIO1_3, PIO0_0, PIO1_0, PIO2_0, PIO3_0, 1,  
  "Input data from DSP[1][3] to DSP[0][0], DSP[1][0], DSP[2][0], DSP[3][0]"},  
{ IO_Brd0 & IO_Brd1 & IO_Brd2 & IO_Brd3,  
  SW_In1,  SW_In1 | SW_Out3 | SW_OutEn | SW_LED,  SW_In1,  SW_In1,  
  PIO1_3, PIO0_1, PIO1_1, PIO2_1, PIO3_1, 1,  
  "Input data from DSP[1][3] to DSP[0][1], DSP[1][1], DSP[2][1], DSP[3][1]"},  
{ IO_Brd0 & IO_Brd1 & IO_Brd2 & IO_Brd3,  
  SW_In2,  SW_In2 | SW_Out3 | SW_OutEn | SW_LED,  SW_In2,  SW_In2,  
  PIO1_3, PIO0_2, PIO1_2, PIO2_2, PIO3_2, 1,  
  "Input data from DSP[1][3] to DSP[0][2], DSP[1][2], DSP[2][2], DSP[3][2]"},  
{ IO_Brd0,  
  SW_In3,  SW_Out3 | SW_OutEn | SW_LED,  NULL,  NULL,  
  PIO1_3, PIO0_3, NULL, NULL, NULL, 1,  
  "Input data from DSP[1][3] to DSP[0][3]"},  
  
{ IO_Brd2,  
  SW_Out0 | SW_OutEn | SW_LED, NULL, SW_In3, NULL,  
  PIO0_0, PIO2_3, NULL, NULL, NULL, 0,  
  "Output data from DSP[0][0] to DSP[2][3]"},  
{ IO_Brd2,  
  SW_Out1 | SW_OutEn | SW_LED, NULL, SW_In3, NULL,  
  PIO0_1, PIO2_3, NULL, NULL, NULL, 0,  
  "Output data from DSP[0][1] to DSP[2][3]"},  
{ IO_Brd2,  
  SW_Out2 | SW_OutEn | SW_LED, NULL, SW_In3, NULL,  
  PIO0_2, PIO2_3, NULL, NULL, NULL, 0,  
  "Output data from DSP[0][2] to DSP[2][3]"},  
{ IO_Brd2,  
  SW_Out3 | SW_OutEn | SW_LED, NULL, SW_In3, NULL,  
  PIO0_3, PIO2_3, NULL, NULL, NULL, 0,  
  "Output data from DSP[0][3] to DSP[2][3]"},  
  
{ IO_Brd2,  
  NULL, SW_Out1 | SW_OutEn | SW_LED, SW_In3, NULL,  
  PIO1_1, PIO2_3, NULL, NULL, NULL, 1,  
  "Output data from DSP[1][1] to DSP[2][3]"},  
{ IO_Brd2,
```

```

        NULL, SW_Out2 | SW_OutEn | SW_LED, SW_In3, NULL,
        PIO1_2, PIO2_3, NULL, NULL, NULL, 1,
        "Output data from DSP[1][2] to DSP[2][3]"},
{ IO_Brd2,
    NULL, NULL, SW_Out1 | SW_In3 | SW_OutEn | SW_LED, NULL,
    PIO2_1, PIO2_3, NULL, NULL, NULL, 2,
    "Output data from DSP[2][1] to DSP[2][3]"},
{ IO_Brd2,
    NULL, NULL, SW_Out2 | SW_In3 | SW_OutEn | SW_LED, NULL,
    PIO2_2, PIO2_3, NULL, NULL, NULL, 2,
    "Output data from DSP[2][2] to DSP[2][3]"},
{ IO_Brd2,
    NULL, NULL, SW_In3, SW_Out1 | SW_OutEn | SW_LED,
    PIO3_1, PIO2_3, NULL, NULL, NULL, 3,
    "Output data from DSP[3][1] to DSP[2][3]"},
{ IO_Brd2,
    NULL, NULL, SW_In3, SW_Out2 | SW_OutEn | SW_LED,
    PIO3_2, PIO2_3, NULL, NULL, NULL, 3,
    "Output data from DSP[3][2] to DSP[2][3]"},
{ IO_Brd2,
    NULL, SW_Out0 | SW_OutEn | SW_LED, SW_In3, NULL,
    PIO1_0, PIO2_3, NULL, NULL, NULL, 1,
    "Checksum Output from DSP[1][0] to DSP[2][3]"},
{ IO_Brd2,
    NULL, NULL, SW_Out0 | SW_In3 | SW_OutEn | SW_LED, NULL,
    PIO2_0, PIO2_3, NULL, NULL, NULL, 2,
    "Checksum Output from DSP[2][0] to DSP[2][3]"},
{ IO_Brd2,
    NULL, NULL, SW_In3, SW_Out0 | SW_OutEn | SW_LED,
    PIO3_0, PIO2_3, NULL, NULL, NULL, 3,
    "Checksum Output from DSP[3][0] to DSP[2][3]"}
};

```

```

/* This is the routine which sequences all activity */
/* Basic communication protocol:

```

Runbeam writes command into location COMBUF0 in DSP32C, together with any auxiliary data in locations COMBUF1-7. Upper 8 bits is the command, lower 8 bits is a unique transaction number.

DSP32C waits for a new command to appear in memory. It overwrites the command with -1 to indicate that it read it, and executes the command. When ready, it copies the command into PIR to acknowledge the action. It then returns to the beginning to await a new command.

```

*/

/* Global error injection variables */
float dataval[68]; /* Uncorrupted data */
float errval[2]; /* Injected error values (constant) */
int errtrans; /* Inject error just before this
               transaction (if any) */
int errboard; /* board number to be affected */
int errproc; /* proc number to be affected */
short int errflag; /* indicates error type */
int ncorrect[4]; /* number of correct/incorrect diagnoses
                  above/below threshold. [0]=correct above;
                  [1]=correct below; [2]=incorrect above;
                  [3]=incorrect below */
int histogram[15]; /* histogram of error decisions */

/* For use with writing/reading actual sonar data -- MATLAB format */
char varout[] = "B00";
char filein[] = "/usr/usr/ftsp/wbb1545/beam/input/Xxx.mat";
char fileout[] = "/usr/usr/ftsp/wbb1545/beam/batch/batch00.mat";
char mskfile[] = "/usr/usr/ftsp/wbb1545/beam/input/mask.mat";

void runbeam(start,stop)
int start,stop;
{
    int i,j,ti,i1,i2, board, proc;
    int nrun,nr,flag,dataflag,mskflag,rdflag,ldflag;
    short int err, k;
    unsigned short int *IOCRptr;
    unsigned short int *Sptr;
    struct tabtrans *thistable;
    register struct tabtrans *ptrTab;
    register int safety,command;
    register struct dspstruct *pPIOxmit;
    register struct dspstruct *pPIOrcv0,*pPIOrcv1,*pPIOrcv2,*pPIOrcv3;
    void rd(),wd();
    FILE *mfp, *fopen(); /* Masking data for fault option 2 */
    long dummy; /* Matlab */
    char *buf; /* file header */
    double temp; /* parameters */

    Sptr = (unsigned short int *)IOSTATREG;
    IOCRptr = (unsigned short int *)IOCONREG;

    /* Initialize error statistics */

```



```

errtrans = -1;
errboard = -1;
errproc = -1;
errflag = 0;
rdflag = 0;
mskflag = 0;

/* Figure out which application has been loaded */
findappl();
if(Nappl < 0) return;
printf("Running application %s\n",apples[Nappl].comment);
thistable = apples[Nappl].table;

/* Parse arguments */
if(stop == 0) {
    if(start == 0) {
        nrun = 1;
    } else {
        nrun = start;
    }
    start = 0;
    stop = apples[Nappl].ntrans;
} else {
    nrun = 1;
}

/* Main host data packet transfer loop */

/* Setup PCR registers -
Run, regular map, DMA, 16-bit, non-autoincrement */
for(i=0 ; i < NBOARD ; i++) {
    for(j=0 ; j<4 ; j++) {
        i1 = ptrDSP[i][j] -> PCR;
        ptrDSP[i][j] -> PCR = ((i1 & PCR_RUN) | PCR_REG | PCR_DMA | PCR_PIO16);
    }
}
readPIO();
if(askynq() == -1)
    return;

/* select level of status printout during run */
flag = 1;
printf("Choose level of printout during run:\n");
printf(" 0 - print avg. magnitudes for each run without pause\n");
printf(" 1 - print avg. magnitudes, stop if error above thresh.\n");

```

```

printf(" 2 - print avg. magnitudes for each run with pause\n");
printf(" 3 - print avg. magnitudes and trans. details\n");
printf(" 4 - print avg. magnitudes, trans. details, PIO reg.\n");
askint("Choice: ",&flag);

dataflag = askprompt("Use actual sonar data? ");
if (dataflag < 0) return;
if (dataflag > 0) {
    ldflag = askprompt("Load data into INPUT DSP? ");
    if (ldflag > 0) {
        askint(" starting from batch: ",&i);
        i1 = (int)(i-1+maxbat)>(int)endbat ? (int)(endbat-i+1):(int)maxbat;
        nrun = (int)i1;
        for(j=0; j < i1; j++)
            wd(j,i);
    } else if (ldflag == 0) nrun = (int)maxbat;
    else return;
    printf("Select output datafile save option:\n");
    printf(" 0 - None\n");
    printf(" 1 - Save uncorrected (corrupted) output\n");
    printf(" 2 - Save corrected output\n");
    askint("Choice: ",&rdflag);
}

/* Initialize error statistics */
for(i=0 ; i<4 ; i++) {
    ncorrect[i] = 0;
}
for(i=0 ; i<=apples[Nappl].nbeam + apples[Nappl].ncsp ; i++) {
    histogram[i] = 0;
}

/* Determine whether to screw up any output buffers in the DSP32C's */
i1 = askprompt("Force error in BEAM or CHECKSUM processor? ");
if(i1 < 0) return;
if(i1 > 0) {
    errboard = 0;
    errproc = 0;
    if(askint("Channel proc. (1) or checksum proc. (2): ",&i1)) {
        switch(i1) {
            case 1:
                printf("There are channel procs. 1 to %d\n",apples[Nappl].nbeam);
                askint("Which channel proc.? ",&i1);
                errboard = apples[Nappl].beam_bd[--i1];
                errproc = apples[Nappl].beam_pr[i1];

```

```

        break;

    case 2:
        printf("There are checksum procs. 1 to %d\n",apples[Nappl].ncsp);
        askint("Which checksum proc.? ",&i1);
        errboard = apples[Nappl].csp_bd[--i1];
        errproc = apples[Nappl].csp_pr[i1];
        break;
    }
}
if(askint("Constant Error (1) or Masking Error (2): ",&mskflag)) {
    errtrans = apples[Nappl].input_pack;
    if (mskflag==1) {
        printf("Options:\n");
        printf(" 1 - Add constant to data\n");
        printf(" 2 - Set data to constant\n");
        printf(" 3 - Add constant to single sample\n");
        errval[0] = 0.0; errval[1] = 0.0;
        if(askint("Choose: ",&i1)) {
            errflag = i1;
            if (i1 < 3) {
                askfloat(" Real value? ",&errval[0]);
                askfloat(" Imaginary value? ",&errval[1]);
            } else {
                askint(" Which sample? ",&i1);
                askfloat(" Real value? ",&errval[0]);
                askfloat(" Imaginary value? ",&errval[1]);
            }
        }
    }
}
else if (mskflag==2) {
    mfp = fopen(mskfile,"rb");
    /* Prepare matlab data file */
    fread(&dummy,sizeof(long),1,mfp); /* eat machine ID      */
    fread(&dummy,sizeof(long),1,mfp); /* eat # rows          */
    fread(&dummy,sizeof(long),1,mfp); /* eat # cols          */
    fread(&dummy,sizeof(long),1,mfp); /* eat imag. data flag */
    fread(&dummy,sizeof(long),1,mfp); /* var name length     */
    buf = (char *)calloc(dummy,sizeof(char));
    fread(buf,sizeof(char),dummy,mfp); /* eat var name        */
    /* Now data points may be read in */
}
}
}

```

```

i = apples[Nappl].nbeam;
for (j = 0; j < i+apples[Nappl].ncsp; j++) {
    if (j < i) {
        board = apples[Nappl].beam_bd[j];
        proc = apples[Nappl].beam_pr[j];
    } else {
        board = apples[Nappl].csp_bd[j-i];
        proc = apples[Nappl].csp_pr[j-i];
    }
    k = 0;
    if ((errboard == board) && (errproc == proc) && (mskflag == 1))
        k = 1;
    wr_s(board,proc,COMBUF6,1,&k); /* Faulty Proc. (const. fault) */
    k = (short int)(errflag);
    wr_s(board,proc,COMBUF4,1,&k); /* Fault Type (const. fault) */
    k = (short int)(i1);
    wr_s(board,proc,COMBUF7,1,&k); /* Faulty Samp. (if applicable) */
    wr_fdsp(board,proc,COMBUF0f,1,&errval[0]); /* Real fault val. */
    wr_fdsp(board,proc,COMBUF1f,1,&errval[1]); /* Imag fault val. */
}

/*
Write Ns, b_max into COMBUF 1, 5. Code GO_N into COMBUF0
Wait to make sure that each pro. returns a GO_N acknowledgement
*/

if(start == 0) {
    for(i=0 ; i<NBOARD ; i++) {
        for(j=0 ; j<4 ; j++) {
            pPIOxmit = ptrDSP[i][j];
            pPIOxmit -> PIR = 0; /* Init PIR reg. to default value */
            pPIOxmit -> PARE = COMBUFh;
            pPIOxmit -> PAR = COMBUF1;
            pPIOxmit -> PDR = (short int)(Ns);
            pPIOxmit -> PAR = COMBUF5;
            pPIOxmit -> PDR = (short int)(b_max);
            pPIOxmit -> PAR = COMBUF0;
            pPIOxmit -> PDR = GO_N;
            safety = 0;
            while(pPIOxmit->PIR != GO_N) {
                if(safety++ > SAFENUM) {
                    safety = 0;
                    /* readPIO(); */
                    printf("Waiting for (%d,%d) to acknowledge new Ns value.\n",
                        i,j);
                }
            }
        }
    }
}

```

```

        printf(" Current PIR=%x. Expecting %x. Wait ",
               pPIOxmit->PIR,GO_N);
        i1 = askynq();
        if(i1 == 0) {
            break;
        } else if(i1 == -1) {
            return;
        }
    }
}
if(flag >= 3) {
    printf("Acknowledge code (PIR): = %x (#tries=%d)\n",
           pPIOxmit->PIR,safety);
}
}
}
}

```

```

dispinit0();
/* LOOP THROUGH PROGRAM NRUN TIMES */
for(nr = 1 ; nr <= nrun ; nr++) {
    printf("*** BATCH %d ***\n",nr);

    for(i = start ; i < stop ; i++) { /* DO ALL TRANSACTIONS */
        ptrTab = &thistable[i];

        if(flag >= 3) {
            printf("\nTransaction: %d\n" ,i );
            printf(" %s\n" ,ptrTab -> comment );
        }

        /* Inject a random error, if desired, before this transaction */
        if ((i==errtrans) && (mskflag==2)) {
            printf("INJECTING ERROR INTO PROCESSOR %d,%d",
                   errboard,errproc);
            for (i1=0; i1 < 2*Ns; i1++) {
                fread(&temp,sizeof(double),1,mfp);
                dataval[i1] = (float)temp;
            }
            wr_fdsp(errboard,errproc,BBUF,2*Ns,dataval);
        }

        /* Setup the IO board */
        *IOCrptr = IOCR = ptrTab -> IOboard;
    }
}

```

```

/* Setup the FIFO switch registers on each DSP32C board */
ptr0DSP0 -> PARE = DSPSWREGh;
ptr0DSP0 -> PAR = DSPSWREGl;
ptr0DSP0 -> PDR = DSPCR[0] = ptrTab -> SWboard0;

ptr1DSP0 -> PARE = DSPSWREGh;
ptr1DSP0 -> PAR = DSPSWREGl;
ptr1DSP0 -> PDR = DSPCR[1] = ptrTab -> SWboard1;

ptr2DSP0 -> PARE = DSPSWREGh;
ptr2DSP0 -> PAR = DSPSWREGl;
ptr2DSP0 -> PDR = DSPCR[2] = ptrTab -> SWboard2;

ptr3DSP0 -> PARE = DSPSWREGh;
ptr3DSP0 -> PAR = DSPSWREGl;
ptr3DSP0 -> PDR = DSPCR[3] = ptrTab -> SWboard3;

/* Send code to start the transmission */

pPIOxmit = ptrTab -> pPIOtrans;
if(flag >= 3) {
    printf("prior transmitter PIR = %x\n",pPIOxmit->PIR);
}
pPIOxmit -> PARE = COMBUFh;
pPIOxmit -> PAR = COMBUF0;
command = GO_XMIT+i;
pPIOxmit -> PDR = command;

/* wait until acknowledge code from DSP in PIR register */
/* (verify that DSP is has begun transmission) */

safety = 0;
while(pPIOxmit -> PIR != command) {
    if(safety++ > SAFENUM) {
        readPIO();
        safety = 0;
        for(i1=0 ; i1<NBOARD ; i1++) {
            for(i2=0 ; i2<4 ; i2++) {
                if(ptrDSP[i1][i2] == pPIOxmit) {
                    printf("Transaction %d: Waiting for transmitter (%d,%d).\n",
                        i,i1,i2);
                    break;
                }
            }
        }
    }
}

```

```

    }
    printf(" Currently PIR=%x; Expecting %x. Wait ",
           pPIOxmit->PIR,GO_XMIT+i);
    i1 = askynq();
    if(i1 == 0) {
        break;
    } else if(i1 == -1) {
        return;
    }
}
}
if(flag >= 3) {
    printf("Transmitter: PIR = %x (#tries=%d)\n",
           pPIOxmit->PIR,safety);
}

/* Send GO_RCV code to start reception */
command = GO_RCV+i;

if ((pPIOrcv0 = ptrTab -> pPIOrcv0) != 0)
{
    if(flag >= 3) {
        printf("Write GO_RCV in rcv0 (code %x)\n",command);
    }
    pPIOrcv0 -> PARE = COMBUFh;
    pPIOrcv0 -> PAR = COMBUF0;
    pPIOrcv0 -> PDR = command;

    if ((pPIOrcv1 = ptrTab -> pPIOrcv1) != 0)
    {
        if(flag >= 3 ) {
            printf("Write GO_RCV in rcv1 (code %x)\n",command);
        }
        pPIOrcv1 -> PARE = COMBUFh;
        pPIOrcv1 -> PAR = COMBUF0;
        pPIOrcv1 -> PDR = command;

        if ((pPIOrcv2 = ptrTab -> pPIOrcv2) != 0)
        {
            if(flag >= 3) {
                printf("Write GO_RCV in rcv2 (code %x)\n",command);
            }
            pPIOrcv2 -> PARE = COMBUFh;
            pPIOrcv2 -> PAR = COMBUF0;
            pPIOrcv2 -> PDR = command;

```

```

    if ((pPIOrcv3 = ptrTab -> pPIOrcv3) != 0)
    {
        if(flag >= 3) {
            printf("Write GO_RCV in rcv3 (code %x)\n",command);
        }
        pPIOrcv3 -> PARE = COMBUFh;
        pPIOrcv3 -> PAR = COMBUF0;
        pPIOrcv3 -> PDR = command;
    }
}
}
}

/* Wait for ack. from all rcvrs, starting with the first */
if (pPIOrcv0 != 0)
{
    safety = 0;
    while(pPIOrcv0 -> PIR != command){
        if(safety++ > SAFENUM) {
            readPIO();
            safety = 0;
            for(i1=0 ; i1<NBOARD ; i1++) {
                for(i2=0 ; i2<4 ; i2++) {
                    if(ptrDSP[i1][i2] == pPIOrcv0) {
                        printf("Transaction %d: Waiting for receiver (%d,%d).\n",
                            i,i1,i2);
                        break;
                    }
                }
            }
            printf(" Current PIR=%x, expecting %x. Wait ",
                pPIOrcv0->PIR,command);
            i1 = askynq();
            if(i1 == 0) {
                break;
            } else if(i1 == -1) {
                return;
            }
        }
    }
    if(flag >= 3) {
        printf("receiver 0: PIR = %x (#tries=%d)\n",
            pPIOrcv0->PIR,safety);
    }
}

```



```

if (pPIOrcv1 != 0)
{
    safety = 0;
    while(pPIOrcv1 -> PIR != command){
        if(safety++ > SAFENUM) {
            readPIO();
            safety = 0;
            for(i1=0 ; i1<NBOARD ; i1++) {
                for(i2=0 ; i2<4 ; i2++) {
                    if(ptrDSP[i1][i2] == pPIOrcv1) {
                        printf("Transaction %d: Waiting for receiver (%d,%d).\n",
                            i,i1,i2);
                        break;
                    }
                }
            }
            printf(" Current PIR=%x, expecting %x. Wait ",
                pPIOrcv1->PIR,command);
            i1 = askynq();
            if(i1 == 0) {
                break;
            } else if(i1 == -1) {
                return;
            }
        }
    }
    if(flag >= 3) {
        printf("receiver 1: PIR = %x (#tries=%d)\n",
            pPIOrcv1->PIR,safety);
    }
}

if (pPIOrcv2 != 0)
{
    safety = 0;
    while(pPIOrcv2 -> PIR != command){
        if(safety++ > SAFENUM) {
            readPIO();
            safety = 0;
            for(i1=0 ; i1<NBOARD ; i1++) {
                for(i2=0 ; i2<4 ; i2++) {
                    if(ptrDSP[i1][i2] == pPIOrcv2) {
                        printf("Transaction %d: Waiting for receiver (%d,%d).\n",
                            i,i1,i2);
                        break;
                    }
                }
            }
        }
    }
}

```

```

    }
    }
    }
    printf(" Current PIR=%x, expecting %x. Wait ",
           pPIOrcv2->PIR,command);
    i1 = askynq();
    if(i1 == 0) {
        break;
    } else if(i1 == -1) {
        return;
    }
    }
    }
    if(flag >= 3) {
        printf("receiver 2: PIR = %x (#tries=%d)\n",
               pPIOrcv2->PIR,safety);
    }

    if (pPIOrcv3 != 0)
    {
        safety = 0;
        while(pPIOrcv3 -> PIR != command){
            if(safety++ > SAFENUM) {
                readPIO();
                safety = 0;
                for(i1=0 ; i1<NBOARD ; i1++) {
                    for(i2=0 ; i2<4 ; i2++) {
                        if(ptrDSP[i1][i2] == pPIOrcv3) {
                            printf("Transaction %d: Waiting for receiver (%d,%d).\n",
                                    i,i1,i2);
                            break;
                        }
                    }
                }
                printf(" Current PIR=%x, expecting %x. Wait ",
                       pPIOrcv3->PIR,command);
                i1 = askynq();
                if(i1 == 0) {
                    break;
                } else if(i1 == -1) {
                    return;
                }
            }
        }
        if(flag >= 3) {

```

```

        printf("receiver 3: PIR = %x (#tries=%d)\n",
               pPIOrcv3->PIR,safety);
    }
}
}
}

/* Turn off the output buffer on the transmitter */

pPIOxmit -> PARE = DSPSWREGh;
pPIOxmit -> PAR = DSPSWREGl;
DSPCR[ptrTab -> transboard] = pPIOxmit -> PDR = SW_LED;

if(flag >= 4) {
    readPIO();
    i1 = askynq();
    if(i1 == 0) {
        break;
    } else if(i1 == -1) {
        return;
    }
}

} /* end transact */

/* print results of the GLRT test */
if(stop == apples[Nappl].ntrans ) {
    /* Wait until beam_mle() is finished */
    if((i = apples[Nappl].mle_bd) >= 0) {
        j = apples[Nappl].mle_pr;
        safety = 0;
        while(ptrDSP[i][j] -> PIR != GO_DONE){
            if(safety++ > SAFENUM) {
                readPIO();
                safety = 0;
                printf("Waiting for beam_mle (%d,%d) to finish\n",i,j);
                printf(" Current PIR=%x, expecting %x. Wait ",
                       ptrDSP[i][j]->PIR,GO_DONE);
                i1 = askynq();
                if(i1 == 0) {
                    break;
                } else if(i1 == -1) {
                    return;
                }
            }
        }
    }
}

```

```

    }
}
if(flag >= 3) {
    printf("Beam_mle() is done: PIR = %x (#tries=%d)\n",
        ptrDSP[i][j]->PIR,safety);
}
i = taberr_short(&err);
if(i >= 0) {
    ncorrect[i]++;
    histogram[err]++;
}
if(flag == 1 && (i==0 || i==2)) break;
if(flag >= 2 && nr < nrun && askynq() <= 0) break;
}
}
if (rdflag > 0)
    rd(nr,rdflag); /* read output data, write to .MAT data file */
} /* END NRUN */
if (mskflag == 2)
    fclose(mfp);
if(stop == apples[Nappl].ntrans) {
    printf("***DIAGNOSES:  above threshold  below threshold\n");
    if(errtrans >= 0) {
        printf("      Correct:  %15d  %15d\n",
            ncorrect[0],ncorrect[1]);
        printf("      Incorrect: %15d  %15d\n",
            ncorrect[2],ncorrect[3]);
    } else {
        printf("      count:  %15d  %15d\n",
            ncorrect[2],ncorrect[3]);
    }
    printf("\n***HISTOGRAM:\n Hyp:      ");
    j = apples[Nappl].nbeam;
    i1 = j+apples[Nappl].ncsp;
    for(i=0 ; i<j ; i++) {
        printf(" beam");
    }
    for(; i<i1 ; i++) {
        printf("  csp");
    }
    printf("\n      none");
    for(i=0; i<j ; i++) {
        printf("  %d,%d",
            apples[Nappl].beam_bd[i],apples[Nappl].beam_pr[i]);
    }
}

```

```

    for( ; i<i1 ; i++) {
        printf(" %d,%d",
            apples[Nappl].csp_bd[i-j],apples[Nappl].csp_pr[i-j]);
    }
    printf("\n  #:");
    for(i=0 ; i<=i1 ; i++) {
        printf("%5d",histogram[i]);
    }
    printf("\n");
}
return;
}

/* Display the communications buffer area */
void readcom()
{
    printf("COMBUF0: Command\n");
    reads_all(COMBUF0,1);
    printf("COMBUF1: Ns\n");
    reads_all(COMBUF1,1);
    printf("COMBUF2: Last Erroneous Command\n");
    reads_all(COMBUF2,1);

    if(!askyes()) return;

    printf("COMBUF3: Number of data batches processed\n");
    reads_all(COMBUF3,1);
    printf("COMBUF4: INPUT, BEAM_MLE: # separate data packets processed\n");
    printf("          BEAM, CSP: Fault type\n");
    reads_all(COMBUF4,1);
    printf("COMBUF5: BEAM, CSP : Maximum # whole integer delays, b\n");
    reads_all(COMBUF5,1);

    if(!askyes()) return;

    printf("COMBUF6: BEAM_MLE: GLRT Thresholded decision\n");
    printf("          BEAM, CSP: Faulty processor \n");
    reads_all(COMBUF6,1);
    printf("COMBUF7: BEAM_MLE: Non-thresholded decision\n");
    printf("          BEAM, CSP: Faulty sample number\n");
    reads_all(COMBUF7,1);

    if(!askyes()) return;

    printf("COMBUF0f: BEAM, CSP: Real fault value\n");

```

```

readfdsp_all(COMBUF0f,1);
printf("COMBUF1f: BEAM, CSP: Imaginary fault value\n");
readfdsp_all(COMBUF1f,1);
printf("MBUF: Average squared magnitude of output\n");
readfdsp_all(MBUF,1);

return;
}

/* Display all the data buffers for BEAM.C */
void readbeam(board,proc)
int board,proc;
{
    short int N;

    /* Read value of Ns from beam.c */
    rd_s(board,proc,COMBUF1,1,&N);
    debug1(" Ns=%d\n",N);

    /* Display the output buffer */
    printf("\nBEAM(%d,%d) - output beam response buffer\n",board,proc);
    readfdsp(board,proc,BBUF,N*2);

    return;
}

/* Display all the data buffers for all BEAM.C */
void readbeam_all()
{
    int i;

    /* Figure out which application has been loaded */
    findappl();
    if(Nappl < 0) return;
    printf("Assuming application %s was recently run\n",
           apples[Nappl].comment);

    /* Find next beam processor, display its status */
    for(i=0 ; i<apples[Nappl].nbeam ; i++) {
        readbeam(apples[Nappl].beam_bd[i],apples[Nappl].beam_pr[i]);

        if(askprompt("More BEAM buffers?") <= 0) return;
    }
}

```

```

    }
    return;
}

/* Display the data buffers for CSPx.C */
void readcsp(board,proc)
int board,proc;
{
    short int N;

    /* Read value of N from csp.c */
    rd_s(board,proc,COMBUF1,1,&N);
    debug1(" Ns=%d\n",N);

    /* Display the output buffer */
    printf("\nCSP(%d,%d) - output checksum beam response buffer\n",
        board,proc);
    readfdsp(board,proc,BBUF,N*2);

    return;
}

/* Display the data buffers for all CSPx.C */
void readcsp_all()
{
    int i;

    /* Figure out which application has been loaded */
    findappl();
    if(Nappl < 0) return;
    printf("Assuming application %s was recently run\n",
        apples[Nappl].comment);

    /* Display the status of all the checksum processors */
    for(i=0 ; i<apples[Nappl].ncsp ; i++) {
        readcsp(apples[Nappl].csp_bd[i],apples[Nappl].csp_pr[i]);
        if(askprompt("More CSP buffers?") <= 0) return;
    }
    return;
}

/* Display the data buffers for the BEAM_MLE.C processor */
void readbeam_mle()

```

```

{
    short int N;
    short err;
    int i,i1,board,proc;

    /* Figure out which application has been loaded */
    findappl();
    if(Nappl < 0) return;
    printf("Assuming application %s was recently run\n",
           apples[Nappl].comment);
    board = apples[Nappl].mle_bd;
    proc = apples[Nappl].mle_pr;

    /* Read value of N from beam_mle.c */
    rd_s(board,proc,COMBUF1,1,&N);
    debug1(" Ns=%d\n",N);

    /* Diagnosed failure */
    rd_s(board,proc,COMBUF6,1,&err);
    if(err > 0) {
        printf("BEAM_MLE - failure diagnosed in processor %d ",err);
        i = apples[Nappl].nbeam;
        if(err <= i) {
            printf("(%d,%d)\n",apples[Nappl].beam_bd[err-1],
                   apples[Nappl].beam_pr[err-1]);
        } else {
            printf("(%d,%d)\n",apples[Nappl].csp_bd[err-i-1],
                   apples[Nappl].csp_pr[err-i-1]);
        }
    }
    } else {
        printf("BEAM_MLE - no failure detected\n");
    }

    /* Display the input buffers */
    for(i=1; i<=apples[Nappl].nbeam + apples[Nappl].ncsp ; i++) {
        if(i <= apples[Nappl].nbeam) {
            printf("\nBEAM_MLE - input buffer, BEAM processor %d\n",i);
        } else {
            printf("\nBEAM_MLE - input buffer, CHECKSUM processor %d\n",i);
        }
        readfdsp(board,proc,IBUF+(i-1)*8*N,2*N);
        i1 = askprompt("More input buffers?");
        if(!i1) {
            break;
        }
    }
}

```



```

    } else if(i1 == -1) {
        return;
    }
}

/* Display the syndrome buffers */
for(i=1; i<=apples[Nappl].ncsp ; i++) {
    printf("\nBEAM_MLE - syndrome buffer %d\n",i);
    readfdsp(board,proc,OBUF+(i-1)*8*N,2*N);
    i1 = askprompt("More syndrome buffers?");
    if(!i1) {
        break;
    } else if(i1 == -1) {
        return;
    }
}

/* Display the syndrome cross-correlations */
printf("\nBEAM_MLE - syndrome cross-correlations\n");
i = apples[Nappl].ncsp;
i = i*i;
readfdsp(board,proc,RBUF,i);

/* Display the likelihoods */
printf("\nBEAM_MLE - Likelihoods\n");
i = apples[Nappl].nbeam + apples[Nappl].ncsp;
readfdsp(board,proc,LBUF,i);
if(!askyes()) return;

/* Display the computed threshold */
printf("\nBEAM_MLE - Threshold\n");
readfdsp(board,proc,LBUF+i*4,1);
if(!askyes()) return;

/* Display the fixed data */
if(err > 0) {
    printf("\nBEAM_MLE - processor %d fixed data\n",err);
    readfdsp(board,proc,CBUF,2*N);
}
return;
}

/* Returns code of which processor failed in *error
Returns 0 if correct diagnosis above threshold, 1 if correct
diagnosis below threshold, 2 if incorrect above threshold, 3

```

```

        if incorrect below threshold, -1 if 'q' typed during printout
        or if error.
*/
static int taberr_short(error)
short *error;
{
    short int N;
    short err;
    int i,i1,errb,errp;
    int board,proc;
    int retval;

    /* Find board, proc location of outmle.c */
    board = apples[Nappl].mle_bd;
    proc = apples[Nappl].mle_pr;

    /* Read value of N from outmle.c */
    rd_s(board,proc,COMBUF1,1,&N);
    debug1(" Ns=%d\n",N);

    /* Display the magnitudes */
    magnitudes();

    /* Diagnosed failure */
    rd_s(board,proc,COMBUF6,1,&err);
    if(err > 0) {
        *error = err;
        i = apples[Nappl].nbeam;
        if(err <= i) {
            errb = apples[Nappl].beam_bd[err-1];
            errp = apples[Nappl].beam_pr[err-1];
        } else {
            errb = apples[Nappl].csp_bd[err-i-1];
            errp = apples[Nappl].csp_pr[err-i-1];
        }
    }

    /* Correct failure diagnosis */
    if(errtrans >=0 && errb == errboard && errp == errproc) {
        printf("CORRECT DIAGNOSIS: processor (%d,%d) failed\n",
            errb,errp);
        retval = 0;
    }

    /* Failure occurred, but incorrectly diagnosed */
    } else if(errtrans >=0) {
        printf("INCORRECT DIAGNOSIS: processor (%d,%d) actually failed\n",

```

```

        errboard,errproc);
    printf(" processor (%d,%d) declared as failed",errb,errp);
    retval = 2;

    /* No failure, but GLRT declared a failure anyway */
} else {
    printf("FALSE ALARM:\n");
    printf(" no failure, but processor (%d,%d) declared faulty\n",
        errb,errp);
    retval = 2;
}

} else {
    *error = apples[Nappl].nbeam + apples[Nappl].ncsp + 1;
    if(errtrans >= 0) { /* Failure, missed by GLRT */
        printf("FAILURE NOT DIAGNOSED:\n");
        printf(" processor failure in (%d,%d) missed\n",
            errboard,errproc);
        retval = 3;
    } else {
        printf("CORRECT DIAGNOSIS: ALL PROCESSORS WORKING\n");
        retval = 1;
    }
}
return(retval);
}

```

```

/* This subroutine uses the information in the apples descriptor
   to figure out where all the beam and csp programs are supposed
   to be, reads the complex outputs from outmle, computes the average
   magnitudes, and then prints a listing of the magnitudes in a
   4 by 4 square, matching the magnitude to the processor location.
*/

```

```

static void magnitudes()
{
    int board,proc,i,j,nf, mleb, mlep;
    float faulty_mag[4][4],correct_mag[4][4],max = -1.0;

    /* Init the Mag array */
    for(board = 0 ; board < NBOARD ; board++) {
        for(proc = 0 ; proc < 4 ; proc++) {

```

```

        faulty_mag[board][proc] = -1.0;
        correct_mag[board][proc] = -1.0;
    }
}
mleb = apples[Nappl].mle_bd;
mlep = apples[Nappl].mle_pr;

/* Read magnitudes, allocate to the positions in the array */

for(i= 0 ; i<apples[Nappl].nbeam; i++) {
    board = apples[Nappl].beam_bd[i];
    proc = apples[Nappl].beam_pr[i];
    rd_fdsp(board,proc,MBUF,1,&faulty_mag[board][proc]);
    if ((board != errboard) || (proc != errproc))
        correct_mag[board][proc] = faulty_mag[board][proc];
    else
        rd_fdsp(mleb,mlep,MBUF,1,&correct_mag[board][proc]);
    if (max-faulty_mag[board][proc]<0) max=faulty_mag[board][proc];
}
for(i=0 ; i<apples[Nappl].ncsp; i++) { /* used only for scaling */
    board = apples[Nappl].csp_bd[i];
    proc = apples[Nappl].csp_pr[i];
    rd_fdsp(board,proc,MBUF,1,&faulty_mag[board][proc]);
    if (max-faulty_mag[board][proc]<0) max=faulty_mag[board][proc];
    faulty_mag[board][proc] = -1.0;
}
disp0(faulty_mag,correct_mag,max);

/* Print the magnitudes */
return;
}

dispinit0()
{
    int i,j,r1,r2;
    float tht;

    home_cursor();
    clear_screen();
    move_cursor(1,30);
    printf("Average squared magnitudes");
    move_cursor(2,18);
    printf("uncorrected");
    move_cursor(2,45);
    printf("|");

```

```

move_cursor(2,57);
printf("corrected");
set_scroll(16,24);
inverse_video();
for (i=1; i<= 10; i++) {
    move_cursor(i+2,3);
    tht = -22.5 + (i-1)*5.0;
    if (tht < 0.0) printf("%5.1f",tht);
    else printf("+%4.1f",tht);
}
regular_video();
move_cursor(16,0);
}

disp0(Mag1,Mag2,max)
float Mag1[4][4],Mag2[4][4],max;
{
    int board,proc,i, barend, line;
    float ratio,samp;

    ratio = 33.0/max;
    for (i = 3; i < 13; i++) {
        move_cursor(i,10);
        erase_line();
        move_cursor(i,45);
        printf("|");
    }
    for (i = 1; i <= 2; i++) {
        for(board = 0 ; board <NBOARD ; board++) {
            for(proc=0 ; proc<4 ; proc++) {
                if (i==1) samp = Mag1[board][proc]; /* faulty output */
                else samp = Mag2[board][proc]; /* corrected output */
                if(samp >= 0) {
                    barend=(int)(samp*ratio)+10+(i-1)*37;
                    if ((barend > 43) && (i == 1))
                        barend = 43;
                    else if ((barend > 80) && (i == 2))
                        barend = 80;
                    if (board==0){
                        line = proc+3;
                    }
                    else if (board==1 && 0<proc && proc<3){
                        line = proc+6;
                    }
                    else if (board==2 && 0<proc && proc<3){

```

```

        line = proc+8;
    }
    else if (board==3 && 0<proc && proc<3){
        line = proc+10;
    }
    else break;

    barch( line , 10+((i-1)*37) , barend );
    if (samp == max){
        inverse_video();
        printf("X");
        regular_video();
    }
}
}
}
}
}
move_cursor(16,0);
} /* end disp0 */

/* This subroutine tries to determine the application loaded.
   If Nappl is -1, then asks user for help, after displaying which
   DSP32C programs appear to have been loaded.
   If Nappl >= 0, then simply returns
*/
void findappl()
{
    int i;

    if(Nappl == -1) {
        printf("Cannot tell which application has been loaded.\n");
        layout();
        printf("Which application is this? \n");
        for(i=0 ; apples[i].comment ; i++) {
            printf("  %d: %s\n",i,apples[i].comment);
        }
        askint("Choose? ",&Nappl);
    }
    return;
}

/* This subroutine reads the Wmat matrix from outmle(), and
   formats this weight matrix for display
*/

```

```

void weights()
{
    int i,j,k,board,proc;
    int nf,nc;
    short sdata[9];
    float Wmat[64];
    float W[4][16];

    /* Find the application */
    findappl();
    if(Nappl < 0) return;

    /* Count number of FFT and CSP processors */
    nf = apples[Nappl].nbeam;
    nc = apples[Nappl].ncsp;

    /* Read the weight matrix */
    rd_fdsp(apples[Nappl].mle_bd,apples[Nappl].mle_pr,WBUF,nc*(nf+nc),
            Wmat);

    /* Distribute the weights among the boards */
    for(i=0 ; i<4; i++) {
        for(j=0 ; j<16 ; j++) {
            W[i][j] = 0.;
        }
    }
    for(i=0 ; i<nf+nc ; i++) {
        if(i < nf) {
            board = apples[Nappl].beam_bd[i];
            proc = apples[Nappl].beam_pr[i];
        } else {
            board = apples[Nappl].csp_bd[i-nf];
            proc = apples[Nappl].csp_pr[i-nf];
        }
        for(k=0 ; k<nc ; k++) {
            W[board][proc+4*k] = Wmat[i+(nf+nc)*k];
        }
    }

    /* Format for display */
    for(i=0 ; i<NBOARD ; i++) {
        printf("(%d,*): ",i);
        for(j=0 ; j<4 ; j++) {
            rd_s(i,j,DSPNAME,8,sdata);
            sdata[8] = 0;

```

```

        printf("%15s  ",(char*)sdata);
    }
    printf("\n");
    for(k=0 ; k<nc; k++) {
        printf("      ");
        for(j=0 ; j<4 ; j++) {
            printf("%15.7g  ",W[i][j+4*k]);
        }
        printf("\n");
    }
    if(i<NBOARD-1) printf("\n");
}
return;
}

void rd(batch,readflag)
int batch,readflag;
{
    MAT B;
    int i,board,proc,mem;
    float Data[10*Ns][2];
    double translate;
    FILE *fopen(), *fd;

    varout[1] = batch/10 + '0'; fileout[38] = batch/10 + '0';
    varout[2] = batch%10 + '0'; fileout[39] = batch%10 + '0';
    if ((fd=fopen(fileout,"wb")) == NULL)
        printf("Can't open output file\n");
    lseek(fd,0L,0);

    board = apples[Nappl].mle_bd;
    proc = apples[Nappl].mle_pr;
    B.type = (long)1000;
    B.mrows = (long)Ns; B.ncols = (long)10;
    B.namlen = (long)4; B.imagf = (long)1;
    if (readflag==1) { /* read corrupted data */
        rd_fdsp(board,proc,IBUF,(10*2*Ns),Data);
    } else { /* read corrected data */
        for (i=0; i<10; i++) {
            mem = IBUF + i*8*Ns;
            if ((errboard == apples[Nappl].beam_bd[i])
                && (errproc == apples[Nappl].beam_pr[i]))
                rd_fdsp(board,proc,CBUF,(2*Ns),&Data[i*Ns][0]);
            else
                rd_fdsp(board,proc,mem,(2*Ns),&Data[i*Ns][0]);
        }
    }
}

```



```

    }
}

fwrite(&B,sizeof(MAT),1,fd);
fwrite(varout,sizeof(char),B.namlen,fd);
for (i = 0; i < 10*Ns; i++) { /* write the REAL data */
    translate = (double)Data[i][0];
    fwrite(&translate,sizeof(double),1,fd);
}
for (i = 0; i < 10*Ns; i++) { /* write the IMAG data */
    translate = (double)Data[i][1];
    fwrite(&translate,sizeof(double),1,fd);
}
fclose(fd);
printf("Data written to %s\n",fileout);
}

void wd(batch,start)
int batch,start;
{
    double val;
    long dummy;
    char *buf;
    short i, j, data[Ns][2*Ne];
    int count, board, proc, mem;
    FILE *fopen(), *fd;

    filein[34] = (batch+start)/10 + '0';
    filein[35] = (batch+start)%10 + '0';

    if ((fd=fopen(filein, "rb")) == NULL)
        printf("Can't open %s\n",filein);

    fread(&dummy,sizeof(long),1,fd);    /* eat machine ID      */
    fread(&dummy,sizeof(long),1,fd);    /* eat # rows          */
    fread(&dummy,sizeof(long),1,fd);    /* eat # cols          */
    fread(&dummy,sizeof(long),1,fd);    /* eat imag. data flag */
    fread(&dummy,sizeof(long),1,fd);    /* var. name length    */
    buf = (char *)calloc(dummy,sizeof(char));
    fread(buf,sizeof(char),dummy,fd);   /* eat var. name       */
    /* NOW DATA POINTS MAY BE READ IN */

    for(i = 0; i < Ns; i++) {
        for(j = 0; j < 2*Ne; j++) {
            fread(&val,sizeof(double),1,fd);

```

```

        data[i][j] = (short int)val;
    }
}
fclose(fd);
board = apples[Nappl].input_bd;
proc = apples[Nappl].input_pr;
count = (int)(2*Ns*Ne);
mem = IBUF + batch*2*count;
wr_s(board,proc,mem,count,data);
j = start+batch;
printf("Batch number %d written to 0x%4x on (%d,%d).\n",
        j,mem,board,proc);
}

/* DSPSTRUCT.H : include file for 68030 files */

/* Terminal characteristics */
#define TERMLINES 20
#define SBIG 0xffff

/* Number of boards in the system */
#define NBOARD 4

/* Define the codes for communication with the DSP32C's
   Upper 8-bits = code
   Lower 8-bits = transaction number
*/
#define GO_N      0x0100
#define GO_XMIT   0x0200
#define GO_RCV    0x0300
#define GO_DONE   0x0400

/* Define miscellaneous constants */
#define NULL 0
#define SAFENUM 25000 /* number of times to test for acknowledge */

/* VME BUS HARDWARE REGISTERS */
/* Master Address Modifier Register on VMEchip (8 bit port) */
#define VME_MAMR      0xFFFE200D

/* VMEbus address modifiers */
#define STD_SPR_DAT    0x3D /* Standard Supervisor Data */
#define STD_SPR_COD    0x3E
#define STD_USR_DAT    0x39
#define STD_USR_COD    0x3A

```

```

#define SHORT_SUP      0x2D
#define SHORT_USR      0x29

/* Structure mimicing the PIO register configuration of the DSP32C */
struct dspstruct{
    unsigned short int PAR, junk1, PDR, junk2, EMR, junk3,
    ESR, PCR, PIR, junk4, PCRh, PARE, PDR2;
};

/* Pointers to the PIO registers on the 4 boards, 4 DSP per board */
extern struct dspstruct *ptrDSP[4][4];
/* Pointers to the PIO registers on boards 0, 1, 2, 3 */
extern struct dspstruct *ptr0DSP[4],*ptr1DSP[4],*ptr2DSP[4],*ptr3DSP[4];
/* Pointers to the PIO registers on boards 0, 1, 2, 3, DSP #0 */
extern struct dspstruct *ptr0DSP0, *ptr1DSP0, *ptr2DSP0, *ptr3DSP0;

/* Globals holding the latest contents of read-only registers
    DSPCR - Switch/LED control registers on the 4 boards
    IOCR - IO board control register
*/
extern unsigned short int DSPCR[4], IOCR;

/* Base Address of the Valley boards */
#define base      0xffff0000

/* Addresses of the 4 boards */

#define board0      (base+0x0000400)
#define board1      (base+0x0000800)
#define board2      (base+0x0001800)
#define board3      (base+0x0003400)

/* IO board control/status register addresses */
#define IOCONREG      (base+0x000C000)
#define IOSTATREG      (base+0x000C002)
#define IOOUTFIFO      (base+0x000C004)
#define IOINFIFO      (base+0x000C006)

/* Switch Register DSP32C address for DSP boards */
#define DSPSWREGh      0xa
#define DSPSWREGl      0x8000

#define DSPFIFOh      0xa
#define DSPFIFOl      0xc000

```

```

/* Define bit field codes for the PCR register in the DSP-32C */
#define PCR_RUN    0x01
#define PCR_REG    0x02
#define PCR_ENI    0x04
#define PCR_DMA    0x08
#define PCR_AUTO    0x10
#define PCR_PDF    0x20
#define PCR_PIF    0x40
#define PCR_DMA32 0x100
#define PCR_PIO16 0x200
#define PCR_FLG    0x400

/* Define bit field codes for the I/O board */
#define IO_Brd0 0x0E00
#define IO_Brd1 0x0D00
#define IO_Brd2 0x0B00
#define IO_Brd3 0x0700

#define IO_FifoEn 0x4
#define IO_LedOn 0x8000

/* Define Switch Register bit field codes for the VE-32C boards */
#define SW_In0 0x00
#define SW_In1 0x01
#define SW_In2 0x02
#define SW_In3 0x03
#define SW_InDis 0x04
#define SW_Out0 0x00
#define SW_Out1 0x08
#define SW_Out2 0x10
#define SW_Out3 0x18
#define SW_OutEn 0x20
#define SW_LED 0x80
#define SW_IRQ2MT 0x40

/* Define the communication buffer area DSP32C addresses */
#define COMBUFh 0xff
/* Shorts */
#define COMBUF0 0xffffe0
#define COMBUF1 0xffffe2
#define COMBUF2 0xffffe4
#define COMBUF3 0xffffe6
#define COMBUF4 0xffffe8
#define COMBUF5 0xffffea

```

```

#define COMBUF6  0xffffec
#define COMBUF7  0xffffee
/* Floats */
#define COMBUF0f  0xfffff0
#define COMBUF1f  0xfffff4
#define COMBUF2f  0xfffff8
#define COMBUF3f  0xfffffc

/* Memory buffer locations in DSP32C's */
#define IBUF      0x008000 /* Input buffers in external memory */
#define BBUF      0x00b040 /* Beam output buffers in external memory */
#define OBUF      0x00a000 /* Syndrome buffers in outmle */
#define CBUF      0x00c000 /* Corrected processor output in outmle */
#define RBUF      0x00b000 /* Syndrome cross-correlations in outmle */
#define LBUF      0x00d000 /* Likelihoods and threshold in outmle */
#define MBUF      0x00d800 /* Average magnitude buffer */
#define WBUF      0x00e000 /* Weights used by outmle */
#define FASTBUF   0x000c00 /* Internal RAM buffer */
#define FASTBUF2  0xffff800 /* Internal RAM buffer */
#define DSPNAME   0x00fff0 /* DSP32C prog. name (used by download())*/

/* Define struct defining which application is loaded into DSP32C's */
extern int Nappl; /* Which application was loaded last */
struct appl {
    char *comment; /* explanation of the application */
    struct tabtrans *table; /* Table of data transfers for runbeam */
    int ntrans; /* Length of this table */
    int input_pack; /* number of packets transmitted by input */
    int input_bd,input_pr; /* board, proc location of input program */
    int mle_pack; /* number of packets input by beam_mle */
    int mle_bd,mle_pr; /* board, proc location of GLRT program */
    int nbeam; /* number of beam processors */
    int beam_bd[12],beam_pr[12]; /* board, proc of up to 12 beam programs*/
    int ncsp; /* number of csp processors */
    int csp_pack; /* number of packets input by each csp */
    int csp_bd[4],csp_pr[4]; /* board, proc of up to 4 csp programs */
};
extern struct appl apples[];
typedef struct {
    long type;
    long mrows;
    long ncols;
    long imagf;
    long namlen;

```

```

} MAT;

/* Define table struct defining the data packet transfers */
struct tabtrans {
    unsigned short int IOboard, SWboard0,SWboard1,SWboard2,SWboard3;
    struct dspstruct *pPIOtrans;
    struct dspstruct *pPIOrcv0, *pPIOrcv1, *pPIOrcv2, *pPIOrcv3;
    int transboard;
    char *comment;
};

/* Function type definitions */
/* init.c */
extern void init_help(),init_all(),io_init();
extern void dsp_init(),dsp_board_init();
extern void dsp_proc_init();

/* util.c */
extern void util_help_aux();
extern int askyes(),askynq(),askprompt();
extern int askint(),askshort(),askfloat();

/* rw.c */
extern void rw_help(),rw_help_aux(),help();
extern void readf(),readfdsp(),readfdsp_all();
extern void reads(),reads_all(),readi(),readi_all();
extern float pow2(),ftoieee();
extern void writefdsp(),writes(),writei();
extern void rd_s(),wr_s(),rd_i(),wr_i();
extern void rd_f(),wr_f(),rd_fdsp(),wr_fdsp();

/* tabtrans.c */
extern void tt_help(),runbeam();
extern void readcom(),readbeam(),readbeam_all(),readcsp();
extern void readcsp_all(),readbeam_mle(),findappl(),weights();
extern int taberr();

/* la.c */
extern void la_help(),la(),laB(),laC(),download_all(),layout();
extern int download();

/* wrta.c */
extern void wrta_help(),run(),run_all(),stop();

/* setpcr.c */

```

```

extern void setpcr_help(),readPIO(),readPIO2();
extern void setPCR(),setPCR_all(),setPAR(),setPAR_all();

#define Ne    15      /* number of hydrophones */
#define Ns    34      /* Ns is the # complex samples per hydrophone */

/* MACROS.H */
#ifdef DEBUG
extern int debug();
#define debug0(s) printf(s)
#define debug1(s,x) printf(s,x)
#define debug2(s,x,y) printf(s,x,y)
#define debug3(s,x,y,z) printf(s,x,y,z)
#define debug4(s,w,x,y,z) printf(s,w,x,y,z)
#else
#define debug0(s)
#define debug1(s,x)
#define debug2(s,x,y)
#define debug3(s,x,y,z)
#define debug4(s,w,x,y,z)
#endif

/* SCRN.H */
set_scroll(top,bottom)
/* top and bottom of scrolling region */
int top,bottom;
{
    if ((top < 25) &&
        (top < bottom) &&
        (bottom < 25))
        printf("\033[%d;%dr",top,bottom);
    else printf("\033[;r");
}

barch(line,startcol,stopcol) /* make barchart */
int line, startcol, stopcol;
{
    int i;
    move_cursor(line,startcol);
    inverse_video();
    for(i=startcol ; i< stopcol ; i++) {
        printf("X");
    }
}

```

```

    regular_video();
}

move_cursor(line,col)
int line,col;
{
    printf("\033[%d;%dH",line, col);
}

inverse_video()
{
    printf("\033[7m");
}

regular_video()
{
    printf("\033[0m");
}

clear_screen()
{
    printf("\033[2J");
}

erase_line()
{
    printf("\033[K");
}

home_cursor()
{
    printf("\033[0H");
}

set_vert_axis(row1,row2,col)
int row1,row2,col;
{
    int i;
    for (i=row1; i <= row2 ; i++)
    {
        move_cursor(i,col);
        printf("|");
    }
}

```