Sonar Beamforming 15

15.1 OVERVIEW

This chapter describes a real-time digital beamforming system for passive sonar. The design of this system is based on several ADSP-2100s that independently perform the beamforming calculations under the supervision of an ADSP-2100 master processor. The modular architecture allows you to tailor the size of the system to your performance needs. Code listings for the master and slave processors are included.

A sonar system can use two different methods to analyze and evaluate possible targets in the water. The first is called *active* sonar. This method involves the transmission of a well defined acoustic signal which can reflect from objects in water. This provides the sonar receiver with a basis for detecting and locating the targets of interest. The limitations of this method are mainly due to the loss of the signal strength during propagation through the water and reverberation caused by the signal reflections. Simplistically, active sonar can be thought of as the underwater equivalent of radar.

The second method is called *passive* sonar. This one bases its detection and localization on sounds which are emitted from the target itself (machine noise, flow noise, transmissions of its active sonar). Its limitations are due to the imprecise knowledge of the characteristics of the target sources and to the dispersion of the target signals by the water and objects in the water. A generic passive sonar system is shown in Figure 15.1, which can be found on the next page.

Sonar systems have a wide variety of military and commercial uses. Some of the military applications include detection, localization, classification, tracking, parameter estimation, weapons guidance, countermeasures and communications. Some of the commercial applications include fish location, bottom mapping, navigation aids, seismic prospecting and acoustic oceanography. More detailed information about sonar technology can be found in Winder, 1975 and Baggeroer, 1978.

15 Sonar Beamforming

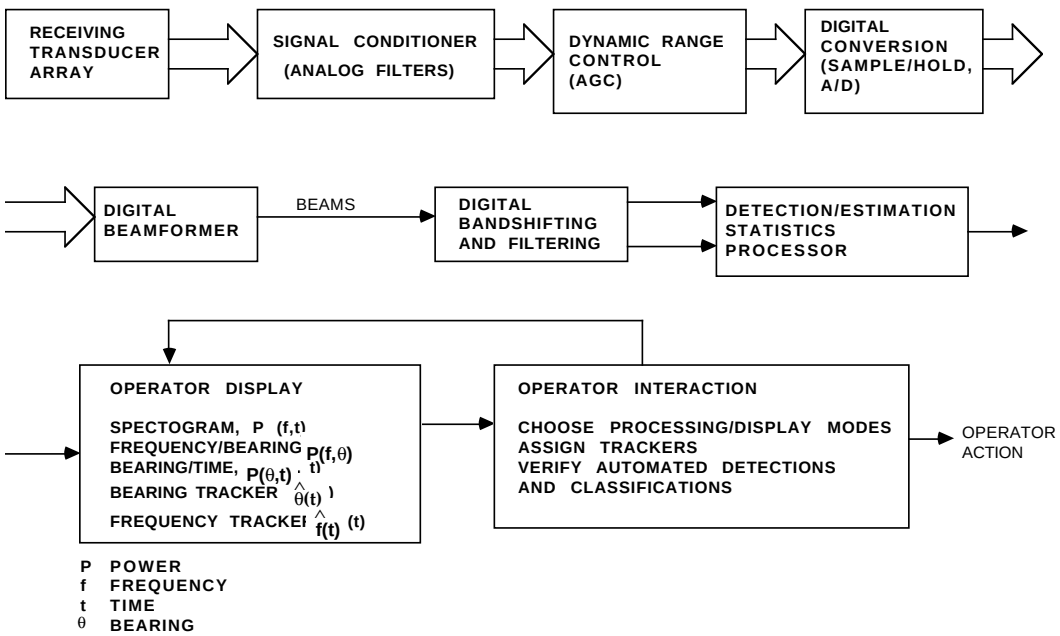

Figure 15.1 Generic Passive Sonar System

15.2 SONAR BEAMFORMING

In its simplest form, sonar beamforming can be defined as "the process of combining the outputs from a number of omnidirectional transducer elements, arranged in an array of arbitrary geometry, so as to enhance signals from some defined spatial location while suppressing those from other sources" (Curtis and Ward, 1980). Thus, a beamformer may be considered to be a spatial filter. It is generally assumed that the waves arriving at the transducers all propagate with the same speed *c*, so that the signals of interest lie on the surface of the cone defined by $\omega = c|k|$ in (k, ω) space. Ideally the passband of the beamformer lies on the intersection of this cone with the plane containing the desired direction vector.

The beamforming operation is accomplished through a series of operations that involve the weighting, delay and summation of the signals received by the spatial elements. The summed output that contains information about a particular direction is called a beam. This output is then sent to a signal processor and/or a display for frequency and temporal discrimination. A beamforming system can employ analog or digital components and techniques; this chapter focuses on a digital beamforming technique.

Sonar Beamforming 15

Beamformers are used both in passive and active sonar systems. In passive sonar, the beamformer acts on the received waveforms. Active sonar also utilizes a conventional beamformer which acts on the waveforms that are reflected from the targets (most active sonars use the same array for receiving and transmitting). There are several well known techniques that can be utilized in forming beams from receiver arrays. The discussion in this chapter focuses on weighted delay-and-sum beamforming technique (also referred as time-delay beamforming) which is very commonly used. Discussion on other techniques, such as FFT beamforming or phase-shift beamforming, may be found in Baggeroer, 1978 and Knight, et al., 1981.

15.2.1 Time-Delay Beamforming

In time-delay beamforming, beams are formed by averaging weighted and delayed versions of the receiver signals. Each receiver has a known location and samples the incoming signals spatially. To steer the beams (i.e. to choose beamforming directions), each receiver's output has to be delayed appropriately relative to the other receivers. The time delays compensate for the differential travel time between sensors for a signal from the desired beam direction.

In order to describe this operation mathematically, let us assume that the array of receivers is composed of a three dimensional distribution of equally weighted omnidirectional sensors. Their spatial locations are specified in the Cartesian coordinate system of Figure 15.2. The

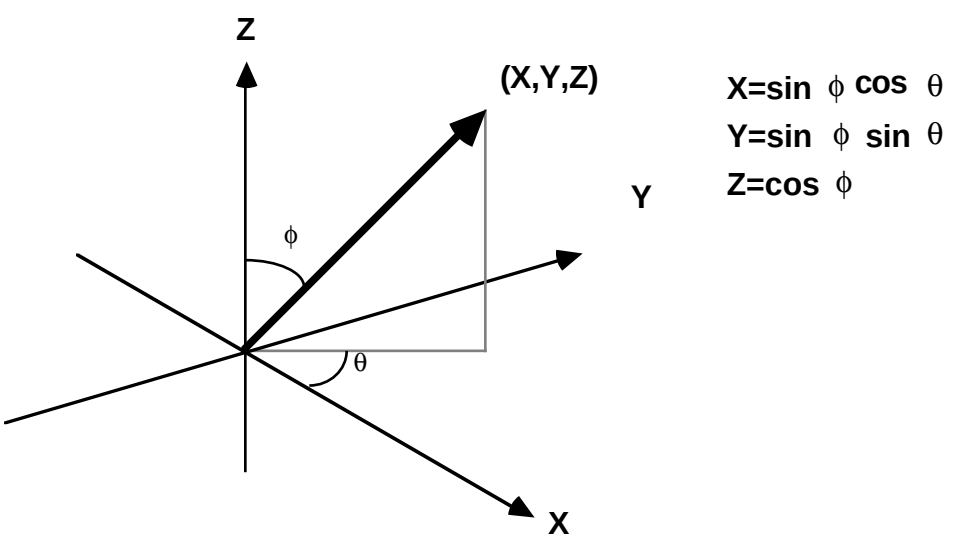

Figure 15.2 Cartesian Coordinate System

beamforming task consists of generating the waveform $b_m(t)$ (or its corresponding sample sequence) for each desired steered beam direction $\mathbf{B_m}$. Each $b_m(t)$ consists of the sum of suitably time delayed replicas of the individual sensor outputs $e_n(t)$.

Let the output of an element located at the origin of coordinates be s(t). Under the assumption of plane wave propagation, a source from direction $\mathbf{S_l}$ (the lth source direction unit vector) produces the following sensor outputs

$$e_n(t) = s(t + ((\mathbf{E_n} \bullet \mathbf{S_l})/v)) \tag{1}$$

where $\mathbf{E_n}$ is the nth element position vector and v is the speed of propagation for acoustic waves in the ocean ($v \approx 1500$ m/s). Delaying each individual sensor output by an appropriate amount to point a beam in the direction $\mathbf{B_m}$ yields the beamformer output

$$b_m(t) = \sum_{n=1}^{N} e_n(t - ((\mathbf{E_n} \bullet \mathbf{B_m})/v)) \tag{2}$$

The operation defined in equation (2) is known as *beamforming*. The beamforming operation is computationally demanding because this summation must be calculated in real time for a large number of sensors and a large number of beams.

For simplicity, the rest of this discussion is limited to one-dimensional (line) arrays with regularly spaced hydrophones (underwater omnidirectional acoustic sensors). This discussion can be generalized to multi-dimensional arrays and line arrays with variable spacing.

15.2.2 Digital Beamforming

Assume that the presence of a plane wave signal $s(t - (\mathbf{E} \bullet \mathbf{B})/v)$ needs to be detected. It is propagating with a known direction **B**, and is measured at **E** in a background of spatially white noise (**B** and **E** are vectors). The line array of Figure 15.3 is used. The signal has the same value at each wavefront and the noise is uncorrelated from sensor to sensor. Thus, in order to enhance the signal from the noise, the sensor outputs are delayed and summed. The delays account for the propagation delay of the wavefront to each sensor. This yields

$$(3) \qquad b(n\Delta) = \quad 1/N \sum_{i=0}^{N-1} X_i(n\Delta + ((\mathbf{E}_i \bullet \mathbf{B})/v))$$

where $X_i(n\Delta)$ is the sampled output of the *ith* sensor. Note that for the *ith* sensor the following relationship holds:

$$(4) \qquad (\mathbf{E}_i \bullet \mathbf{B})/v = -i\,(d/v)\sin\theta$$

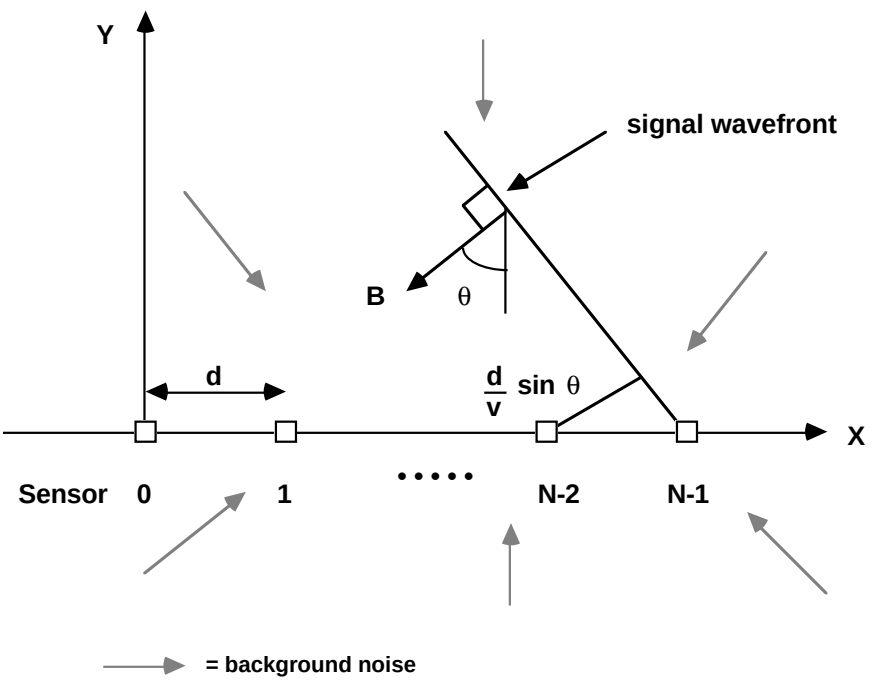

Figure 15.3 Line Array of Equally Spaced Hydrophones

The input to the beamformer is a set of time series. The input is usually one set of time series for each sensor, while the output of the beamformer is another set of time series, generically referred to as beams.

The beamformer is spatially discriminating because for a plane wave with a propagation direction θ, different than θ_0 assumed by the beamformer, the sensor outputs are not coherently combined. This leads to partial

15 Sonar Beamforming

cancellation of the incoming signals with $\theta \neq \theta_0$. Thus, for a plane wave signal

$$X_i(n\Delta) = e^{\,j(\omega n\Delta + (\omega id/v\sin\theta))} \tag{5}$$

and

$$b(n\Delta) = 1/N \sum_{i=0}^{N-1} e^{\,j\omega id(\sin\theta - \sin\theta_0)/v}\; e^{\,j\omega n\Delta} \tag{6}$$

where *d* is the spacing between the sensors. Thus the amplitude of the plane wave arriving from a direction θ at the output of a beamformer steered to θ_0 has an attenuation given by

$$|B(\omega, \theta)| = \frac{\sin[N(\omega/2)\; d(\sin\theta - \sin\theta_0)/v]}{N\sin[(\omega/2)\; d(\sin\theta - \sin\theta_0)/v]} \tag{7}$$

This function is known as the *beam pattern* of the array. An example beam pattern for a line array is shown in Figure 15.4. For a given wave frequency and steering direction θ_0, all plane waves with $\theta \neq \theta_0$ are attenuated, leading to the interpretation of a beamformer as a spatial filter. In most applications, it is desirable to have a beam pattern with a very narrow main lobe and very low level sidelobes for maximum noise rejection.

Increasing ω (the operating frequency) and/or N results in narrower main lobes even though the side lobe level does not change. Increasing the sensor spacing also results in narrower main lobes. But this is limited by the fact that spatial aliasing will occur for $\Delta x > \lambda_{min}/2$, where λ_{min} is the signal wavelength for the highest frequency of interest (Knight, et al., 1981). Spatial aliasing exhibits itself in terms of extra main lobes near the endfire region. In order to reduce the sidelobe levels, the sensor outputs must be weighted. This procedure is known as *shading*. Thus, in equation (6), we replace the 1/N factor by w_k . The corresponding beam pattern is

$$|B(\omega, \theta)| = \sum_{i=0}^{N-1} w_i\; e^{\,j\omega id(\sin\theta - \sin\theta_0)/v} \tag{8}$$

PASTE IN
FIGURE 15.4 HERE

Beam pattern at 0 degrees (perpendicular to line of array) with a sampling rate of 500Hz, for a line of 32 sensors spaced 1m apart

Figure 15.4 Example Beam Pattern

The usual windowing techniques of Fourier transform theory can be used to reduce the sidelobes (Knight, et al., 1981). By employing a Hamming window, for example, the sidelobes may be reduced to –40db at the expense of widening the main lobe.

Another problem that has to be dealt with in a line array with fixed shading is the quantization errors that are introduced from the insertion of the delays. In a digital beamformer, for ideal operation, the beamforming

directions are limited such that the delays t are multiples of the sampling interval. Any other choice of directions (and thus delays) introduces errors onto the beam pattern (Gray, 1985). One method that is used to reduce such errors involves the interpolation of the incoming samples. Further discussion on this topic can be found in Pridham and Mucci, 1978.

An overall block diagram for a conventional time delay digital beamformer is shown in Figure 15.5.

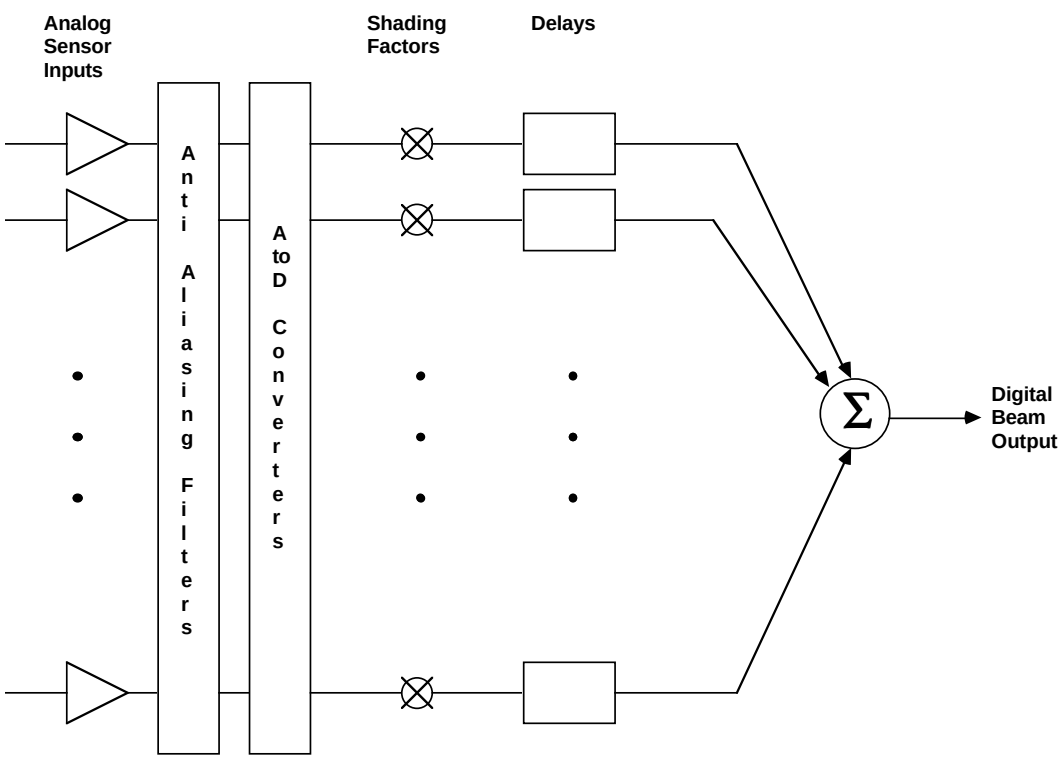

Figure 15.5 Conventional Time-Delay Beamformer

15.3 DIGITAL BEAMFORMER IMPLEMENTATION

The beamforming task can be performed using analog or digital systems. Implementing analog tapped delay lines and forming multiple beams in real time using analog hardware results in big, inflexible and cumbersome systems. A digital beamformer results in a smaller, more accurate, and much more flexible hardware/software unit than an analog technique. In this section, some of the important issues in digital time-delay beamforming systems are discussed.

15.3.1 Computational Power

The computational capacity required by a real-time beamformer can be computed from equation (3) as roughly Nf_s multiplications and additions per second per beam, where N is the number of sensors and f_s is the input sampling frequency. This requirement does not seem by itself very demanding. However, multiple simultaneous beams are usually needed in order to span the space around the sensors and consequently a more realistic requirement is N_bNf_s multiplications and additions per second, where N_b is the number of beams formed. A passive sonar system utilizing ≈30 sensors and requiring ≈30 or more beams at sampling rates of 10kHz or higher needs to perform at least 12 million multiplications and additions per second. Computation demands in the near future will increase rapidly because new generations of quieter sources (e.g., submarines) will require passive sonar to use more sensors (for higher resolution) and more beams.

15.3.2 Memory Usage

Another concern in designing real-time beamformers is the amount of storage that is needed in order to implement the digital delays. For example, in the case of a line array with sensor spacings *d*, the storage necessary to form all the synchronous beams is on the order of N^2f_sd/v. The *synchronous* beams are all the beams that can be formed using delays which are multiples of the input sampling period. In the typical beamforming system that we considered earlier, the size of storage required is on the order of 10000 memory locations. The memory word width chosen in a particular application could be 16 bits or more, which would require at least 20 Kbytes of RAM per beam (unless RAM is shared). The demand for fast accessible storage will be rising in the coming years along with the demand for computation power.

15.3.3 Other Issues

Further discussion on beamforming system issues can be found in Knight, et al., 1981, Janssen, 1987, and Hodgkiss and Anderson, 1981.

15.4 EXAMPLE BEAMFORMER

The example beamformer is able to take inputs from an arbitrary array of up to 32 sensors, form multiple beams in real time, and is user-configurable from an IBM PC personal computer (or compatible). The acoustic frequencies of interest in this example range from 0Hz to 2000Hz. The input sampling rate is 10kHz, which is higher than the Nyquist rate because of the need for a higher resolution in the beamforming directions. The acoustic data is collected by hydrophones and the analog data is

digitized to 12 bits. The gain applied to the analog inputs is manually selectable (1, 10, 100 or 1000). The output beams are sent to another system over a specific parallel interface for further processing. The output beams are also available at analog output points, one at a time, for testing, monitoring and some other signal processing tasks.

15.4.1 System Architecture

Several important issues were considered in selecting an architecture for this beamforming system. One consideration is the need for modularity. A modular system gives a user the ability to start small with the option of expanding the system's capabilities later.

Another consideration is the demand for speed. As discussed earlier, in order to form multiple simultaneous beams, a large number of summations have to be computed in real time. This requirement, along with modularity, led to a distributed processing architecture.

One more consideration is flexibility. A user is able to specify any array configuration and form any beam. Ease of use requires a friendly and interactive user interface, through which the users can specify many different system parameters.

15.4.2 Building Blocks

The beamformer consists of the building blocks that are shown in Figure 15.6. There are several parallel buses in the system for data transfers and communication. A wide common bus is used for the subsystems to communicate and exchange data with each other. Another bus facilitates the communications with the IBM PC. This bus is used to download the user system configuration data from the PC into the beamformer. Finally, a bus is used to send the output beams to a sonar signal processing system which performs further processing on the data.

There are three main types of subsystems in this beamformer; each of these subsystems is implemented as a separate board in the example system:

- The *master* module, which is responsible for controlling the data exchange among all internal modules and the data flow over the I/O buses.

- The *slave* module, which is responsible for the actual beamforming task. Each slave can beamform in multiple directions and more slaves may be added in order to form a larger number of beams.

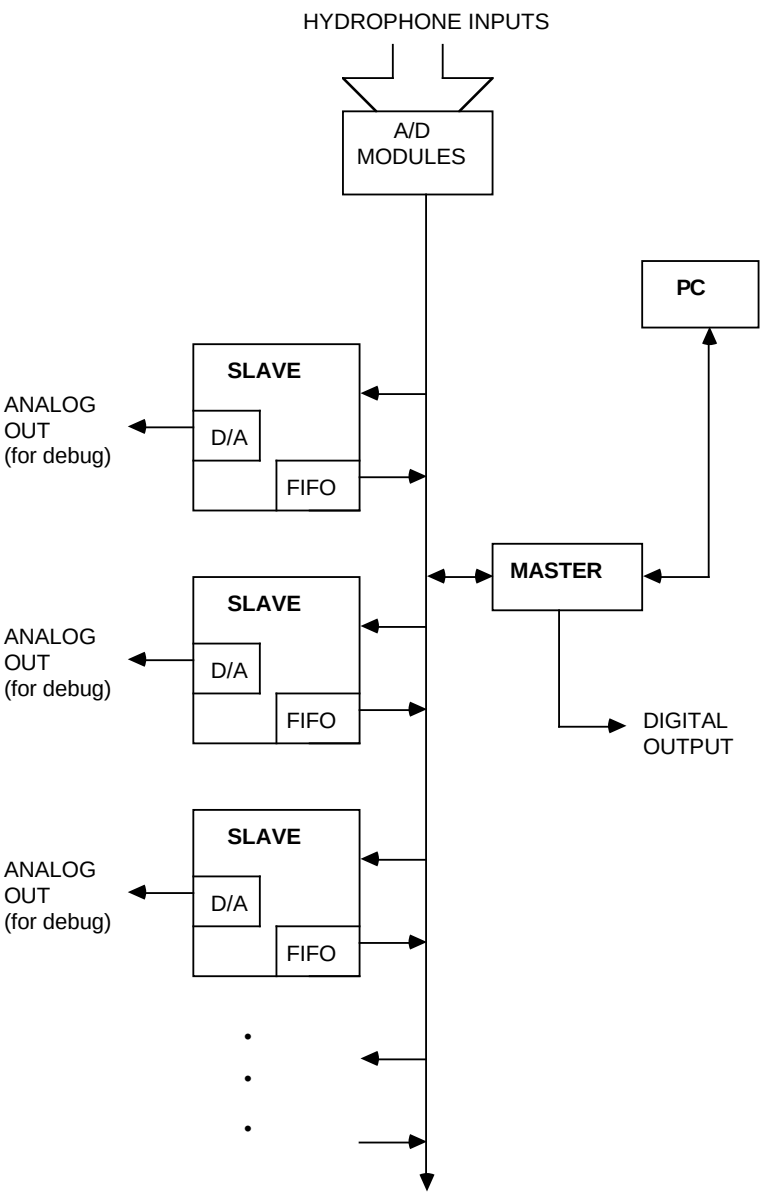

Figure 15.6 Example Beamformer Block Diagram

- The *analog-to-digital conversion* (A/D) module. This module takes the hydrophone outputs as inputs, is responsible for sampling the incoming analog signals and converting them into a digital format. Each A/D module can handle a limited number of hydrophones, but more modules may be attached to the common bus in order to handle a larger number of inputs.

15.4.3 System Operation

The internal operation of the system is controlled by the master. The handling of the incoming samples and the I/O exchanges are also under the master's control. The operation of the system is synchronized to the input sampling clock which has a period of 100 μs (10kHz). This implies that the calculated beam samples have to be sent out every 100μs. Let's call this duration a *system cycle*.

Initially, the master has to accomplish a one-time task of handling of the system configuration data sent from the PC. Thereafter, the master has to go through its duties within one system cycle and be ready to handle the next set of incoming samples. The system events that make up the system configuration (illustrated in Figure 15.7) are as follows:

1. The operator enters system configuration variables (number of sensors, beam directions, shading factors, etc.) through an interactive program on the PC. The same program does some calculations and downloads the data to the master ADSP-2100. Communications between the master and PC are accomplished using a simple protocol over a parallel interface card located in the PC.

2. The master keeps the configuration variables in several of its internal registers. It sends this information to all the slaves so that each of them can identify the beams that they are responsible for.

3. The master waits for a signal from each one of the slaves confirming that they are ready to beamform.

The sampling clock is running during the configuration, but the interrupts initiated by A/D conversions are not recognized until all the slaves are ready. Once each of the slaves has sent the signal that it is ready to beamform, the master starts the cyclic operation of the beamformer.

1. The master responds to the A/D conversion interrupt by reading the results of the A/D conversions, which correspond to a simultaneous snapshot of the incoming waveforms at the hydrophone locations. It reads all the results in sequence and writes them into the memories of all the slaves. Thus, all slaves receive identical copies of the incoming waveform samples.

2. The master initiates an interrupt which orders all the slaves to start beamforming.

MASTER	SLAVE
Power up (or RESET)	Power up (or RESET)
Wait for the user to input the system setup parameters	
PC communications	Idle
Download setup data to slaves	Accept setup data
Idle	Self-prepare using the setup data
Sample all the A/Ds 255 times and send samples to slaves	Receive the first 255 sets of conversion results (fill the sample buffer)
Idle	Beamform
Sample A/Ds, send samples	Receive samples
Read FIFOs, output beams	Beamform
Sample A/Ds, send samples	Receive samples
Read FIFOs, output beams	Beamform
etc.	etc.

Figure 15.7 Sequence of Events

3. Once a slave finalizes a summation (i.e. forms a beam sample), it shifts the result into a FIFO memory for collection by the master. Each slave computes and stores its own beam samples independent of other slaves. Once a slave has completed its assigned set of beams, it waits for the next interrupt (initiated by the master) that orders all the slaves to beamform again.

4. While the slaves are busy forming and storing the current set of beam samples, the master reads the sets of beam samples that were formed during the previous system cycle. After finishing this output duty, the master waits for the next A/D interrupt.

Each slave keeps its input samples in a circular buffer (255 slots) which is located in the slave's data memory space. New incoming samples are put into consecutive locations in this buffer. Once the end of the buffer is reached, the oldest snapshot of samples gets overwritten by the newest samples and this cyclic process goes on.

Each slave performs the beamforming task by reading the appropriate locations in its sample buffer, by multiplying those values with a shading factor and by keeping a running sum of these weighted samples until the summation is finished.

Each slave writes out beam samples to its own dedicated a first-in-first-out memory (FIFO). Only the master can shift the beam samples out of the FIFOs. The master reads and sends each beam sample, one at a time, over its output bus.

The user must realize that the first 255 sets of beam samples produced are invalid. This is due to the fact that the sample buffer is not full until the end of the 255th system cycle. Therefore, during that period, the locations read by the slaves contain meaningless data. Valid system outputs are produced ≈25.5ms after the system starts beamforming.

15.4.4 Timing Issues

There are a number of important operational timing issues due to the length of the system cycles. The number of different beam samples that can be formed by each slave is limited by the system cycle length. This constraint exists because the slaves have to release the control of their individual memory buses in order to allow write operations by the master. Another constraint is that the master needs to read all the incoming samples and also send all the beam samples out within a system cycle. The maximum number of beams that can be formed in this system are directly limited by these timing constraints. The beam allocations per slave must be calculated carefully by the PC during the system configuration phase. Otherwise, incomplete beams and invalid outputs may result because of the master not having enough time to send out all the beam samples or other complications.

15.4.5 Digital Output

The example system provides the ability to send out all the computed beams through a 16-bit parallel output port. This parallel port is located on the master module. It can be used to communicate the beam data to an external signal processor for further processing. The parallel port is

comprised of octal latches, D-flops and address decoding circuitry which allow the master to communicate with the outside world using a simple protocol.

15.4.6 Analog Output

The example system provides the ability to observe some beams through analog outputs. A digital-to-analog (D/A) converter is included on each slave board and the desired analog beam output can be selected using a thumbwheel switch. Each slave sends the desired beam sample to its D/A converter before shifting it into the FIFO. The overhead of this analog port is minimal, and the port is a very useful test point for system debugging.

15.4.7 System Configurations

The minimum system configuration consists of a master, a slave and an A/D module. The maximum possible system configuration is limited by the speed of the internal hardware and the maximum data rate capability of the output port. All buses must be present in any system configuration.

15.5 SYSTEM HARDWARE

The system hardware includes ADSP-2100 DSP processors, high speed hybrid A/D converters and very high speed CMOS and bipolar LSI components. Hardware selection, design, operation and interface issues in the system are discussed in the following sections.

15.5.1 Component Selection

The ADSP-2100 fulfills the high computational requirements of the slave modules. It also fulfills the CPU requirements of the master module by enabling high speed input data transfers between the A/D modules and the slaves as well as the output transfers. It provides easy handling of memory mapped peripherals and can handle four external interrupts. Design time is saved by using the same processor for both master and slave.

The 12-bit fast A/D converters, high precision sample and hold circuits, low noise operational amplifiers used in the input gain section and anti-aliasing filters are also critical for a high performance system. Front-end analog signal conditioning and A/D circuit design and production using discrete components is a difficult task in noisy digital environments such as the one assumed in the example system. A hybrid A/D converter with on-board voltage references along with sample and hold circuits can perform the required duties better than any discrete circuit with similar functionalities.

15 Sonar Beamforming

Analog Devices' AD1332 A/D converter is appropriate and convenient for several reasons:

- The AD1332 integrates several of the necessary components inside. Its central element is a 12-bit 5µs AD7672 A/D converter. A voltage reference for the converter is included. The converter is preceded by an on-board AD585 sample and hold circuit which itself is preceded by an optional 4-pole Butterworth low-pass filter (anti-aliasing filter). The converter output is fed into a 12-bit latch with tristate output buffers (an optional integrated FIFO is also available for the temporary storage of the conversion results).

- The AD1332 is easily addressed from a microprocessor, which makes it ideal for this application.

- Because the front end circuitry must be duplicated for all incoming sensor inputs, the use of a hybrid helps reduce design, prototyping and production times. Multiplexing the sample and hold outputs into fewer converters is not desirable because of system performance considerations.

The selection of the rest of the high speed VLSI and LSI components in the system is not as crucial to system performance. Several levels of address decoding and buffering that are present in the system result in high demands on the memory components. Integrated Device Technologies' (IDT) 2Kx8 CMOS static RAMs with 25ns access times are used as the data memory components on the slaves. The program memories for the master and the slaves also need to be very fast. Cypress Semiconductor's 2Kx8 CMOS EPROMs with 35ns access times are used as the program memory components for all ADSP-2100s.

The slave FIFOs are IDT's 72413L35 64x5 CMOS FIFOs. Analog Devices' AD569 16-bit D/A converters provide analog output ports on the slaves. The rest of the LSI components are off-the-shelf Advanced Schottky (Fairchild's FAST and Texas Instruments' 74AS series), Advanced Low Power Schottky (Texas Instruments' 74ALS series) or very high speed CMOS (IDT's 74FCT series) integrated circuits. More detailed information and specifications for these components are available from their manufacturers.

The interface card between the PC and the master should be a parallel I/O card that can easily be plugged into the PC's backplane and addressed from a high level program. There are a large number of such I/O boards

available. A short development time discourages spending effort on a complicated protocol, and thus a simple protocol and a very flexible I/O card is preferable. Analog Devices' RTI-817 parallel I/O board contains three 8-bit bidirectional ports that accept user-configured directions. These ports are memory mapped and addressable from high level programs. More detailed information about the card can be found in the *RTI-817 User's Manual* published by Analog Devices.

15.5.2 Master Board Hardware

A high level block diagram for the master board is shown in Figure 6.8. The circuitry on the master board is centered around an ADSP-2100 processor running at 8MHz. This master CPU takes its instructions from three CY7291-35 2Kx8 EPROMs (program memory) mapped to the CPU's program memory address space (see Figure 15.9 on the next page). The master board does not contain any data memory. The ADSP-2100's internal registers are sufficient for most operations except during the configuration phase. The details of the configuration operation are discussed later in the firmware section.

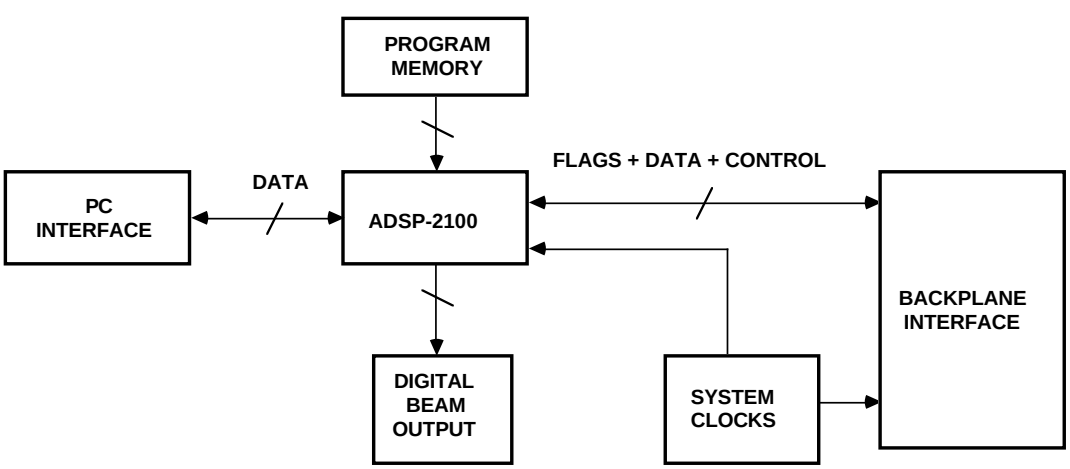

Figure 15.8 Master Module Block Diagram

The master recognizes two interrupts. The first interrupt occurs every 100μs and notifies the master about the availability of new A/D conversion results. This interrupt comes directly from the sampling clock. The second one notifies the master that the external signal processor has received a beam sample and is ready to receive the next one.

15 Sonar Beamforming

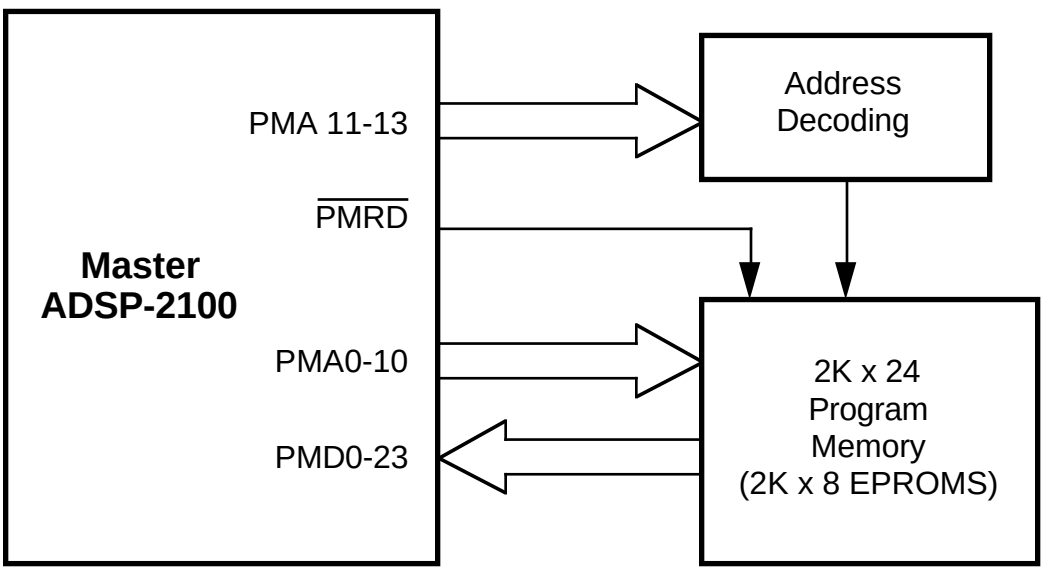

Figure 15.9 Master Module Program Memory Interface

A set of devices generates the sampling clock that is used to sample the analog sensor inputs and to initiate the A/D conversions. These clock signals are sent over the backplane to the A/D boards.

The master board has a number of decoders, latches and buffers that it uses to write to slave data memories, to read slave FIFOs, to send control information to the slaves, to access the PC communication ports and to receive status flags over the backplane bus. See Figure 15.10.

The master board contains a large number of devices dedicated for the CPU's external bus interfaces. These are shown in Figure 15.11. A bank of bus drivers and transceivers provide the necessary buffering for the signals that are traveling over the backplane bus. The communications with the PC are handled through three octal latches that provide a direct interface to the I/O board that is plugged into the PC's backplane. This board also has two inverting octal latches which facilitate the beam sample transfers over the digital output bus.

The interface to the three buses requires four separate connectors: a 96-pin Eurocard connector which is the connection to the backplane, a 50-pin flat cable connector which connects the master and the PC, a 28-pin connector and a 50-pin flat cable connector which are used on the output bus

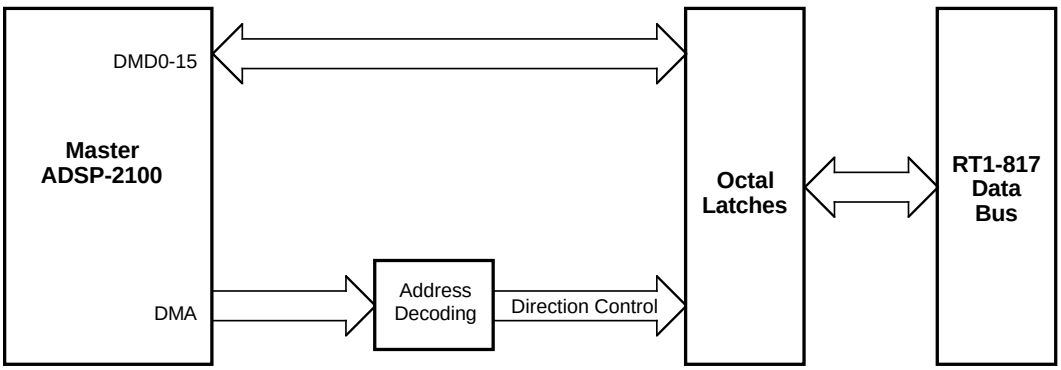

Figure 15.10 Master Module PC Interface

connections between the master and the external signal processor. The master only requires +5V power.

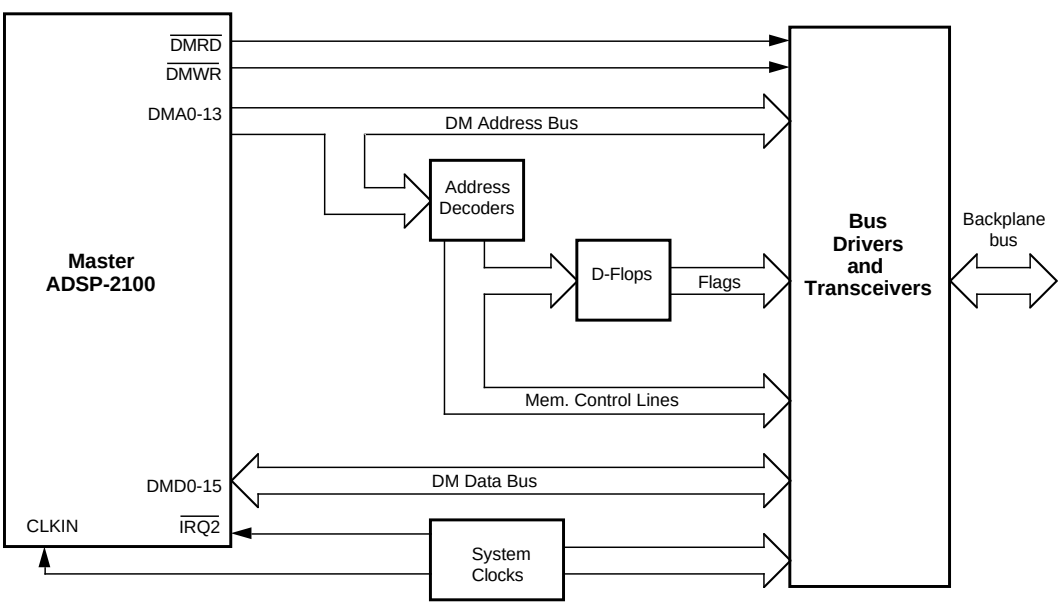

Figure 15.11 Master Module Backplane Interface

15.5.3 Slave Board Hardware

A high level block diagram for the slave board is shown in Figure 15.12. The circuitry on the slave board is centered around an ADSP-2100 processor running at 8MHz. The slave CPU takes its instructions from three CY7291-35 2Kx8 EPROMs (program memory) which are mapped into the program memory address space of the slave CPU (see Figure 15.13). Two 2Kx8, 35ns static RAMs are also mapped into the program memory address space and available for data storage using the upper 16 bits of the ADSP-2100's PMD bus. A decoder provides the necessary address decoding for the PMA lines.

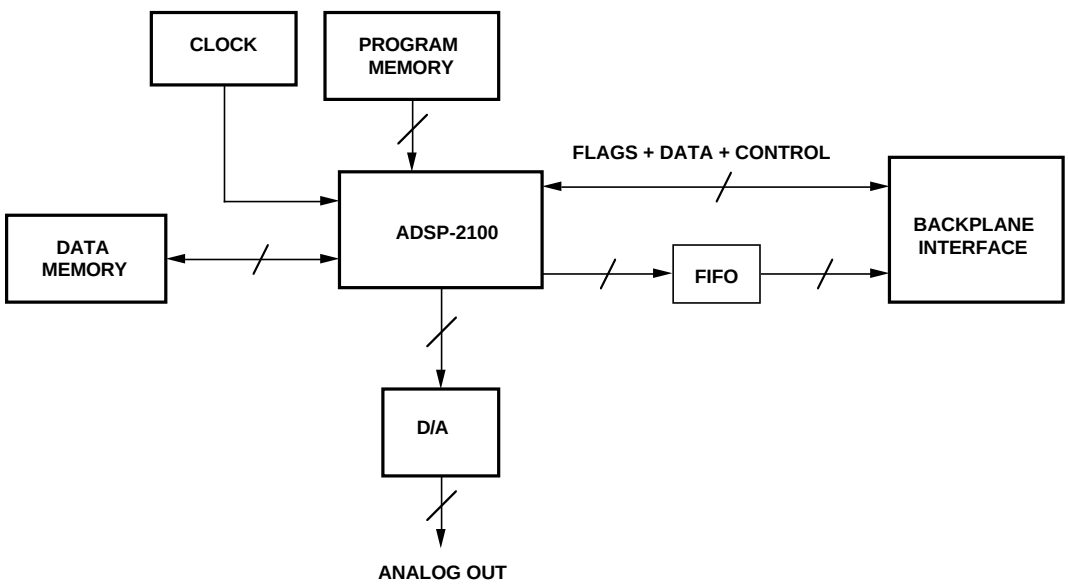

Figure 15.12 Slave Module Block Diagram

There are ten IDT6116LA-25 2Kx8 static RAM chips on the slave board. These devices are mapped onto the data memory address space of the slave CPU (see Figure 15.14). The first 8K locations of this space are dedicated to the circular sample buffer. The remaining 2K locations are used for additional data storage. A decoder provides the necessary address decoding for the static data RAMs as well as the data memory address mapping for some additional devices and flags.

The output FIFO for the slave module consists of four IDT72413L35 64x5 FIFO components. These can be loaded (written to) from the slave DMD bus. The contents of the FIFO can be read from the backplane data bus

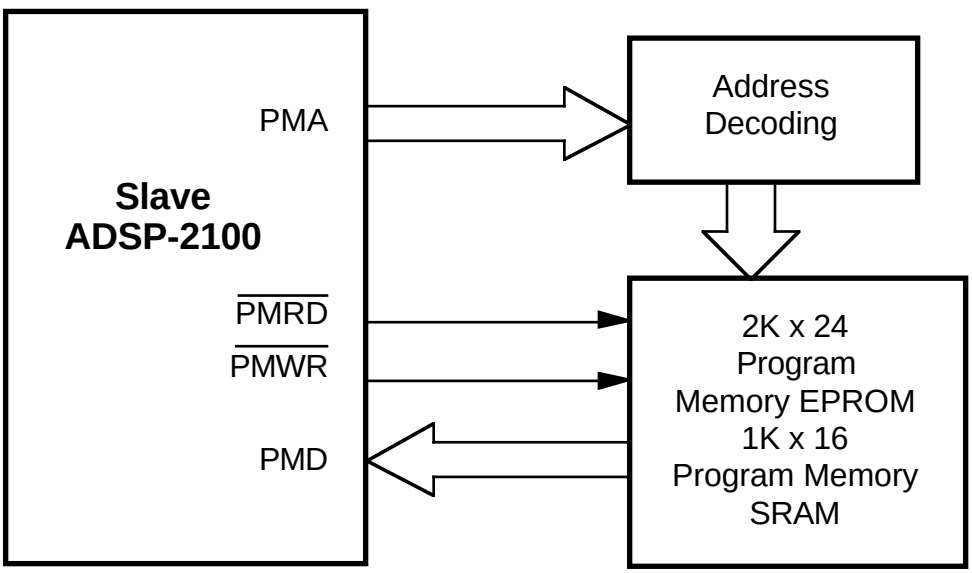

Figure 15.13 Slave Module Program Memory Interface

through buffers (see Figure 15.15). Each FIFO has a unique location in the master's data memory address space; this location is determined by DIP switch settings on the slave board.

The slave CPU only recognizes one external interrupt, the one generated by the master in order to start the beamforming operation after a new sample buffer update. The slave CPU clears this interrupt immediately after it finishes forming its assigned beams.

The slave board, like the master, contains a large number of bus drivers and transceivers for easy interface with the backplane bus. Several lines on the backplane bring control information from the master. Some of these controls cause the slave CPU to halt its operation and surrender the control of its buses to the master. Some status flags are also sent to the master over the backplane.

The AD569 16-bit D/A converter is mapped into the slave data memory address space (see Figure 15.16). An AD588 ±5V voltage reference is used with this D/A converter. Because the D/A converter has a slow access time, the slave write cycle must be extended using the DMACK signal (this is an input to the slave CPU). A small circuit is used to generate DMACK during a write cycle to the D/A converter.

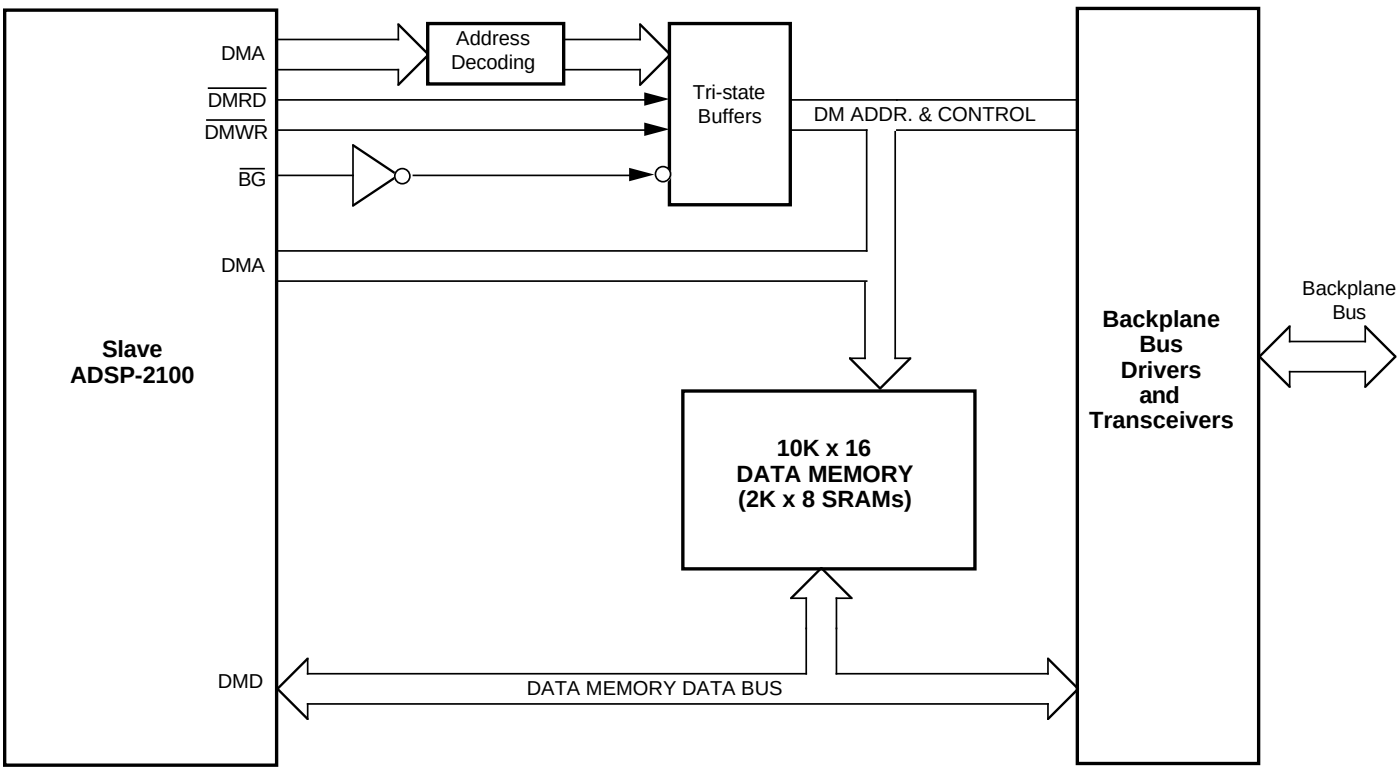

Figure 15.14 Slave Module Data Memory Interface

The slave board has a 16-position thumbwheel switch which selects the beam that is sent out through the D/A converter. A set of four DIP switches give each slave board its own identity. Setting these switches before power-up allows the slave CPU to read them later to determine the beams that are under its responsibility.

There are two connectors on the slave board: a 96-pin Eurocard connector and a male BNC connector. The first is used to interface to the backplane bus, and the second is connected to the analog output of the D/A converter. You can use the BNC connector to send the switch-selected output beam to another device. The slave board requires +5V digital and ±12V analog power supplies.

15.5.4 A/D Board Hardware

A high level block diagram for the A/D board is shown in Figure 15.17. Each A/D board can receive up to four analog inputs ranging between ±5V and can convert them into 16-bit signed fixed-point (1.15 format)

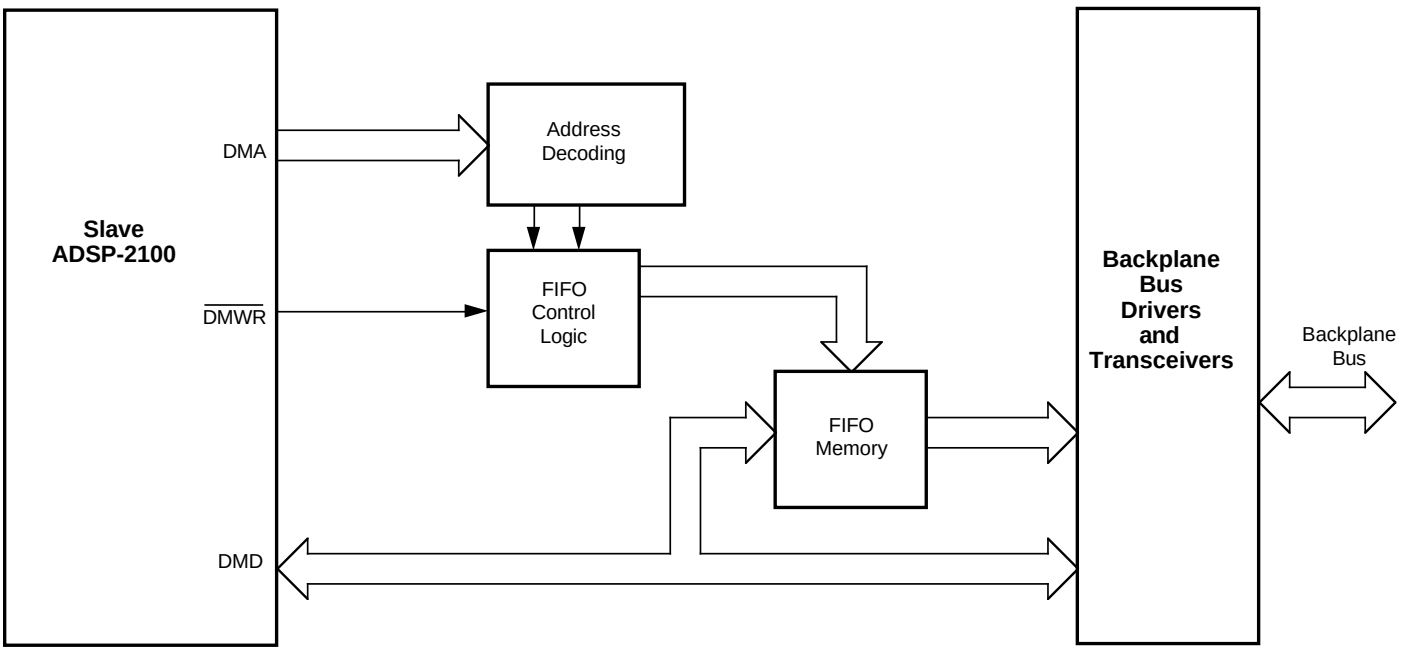

Figure 15.15 Slave Module FIFO Interface

numbers with 12-bit accuracy. The circuitry on the board is designed around four AD1332 12-bit hybrid A/D converters.

For each input, a gain stage which uses ADOP07 operational amplifiers is included on the A/D board to provide the necessary amplification of the

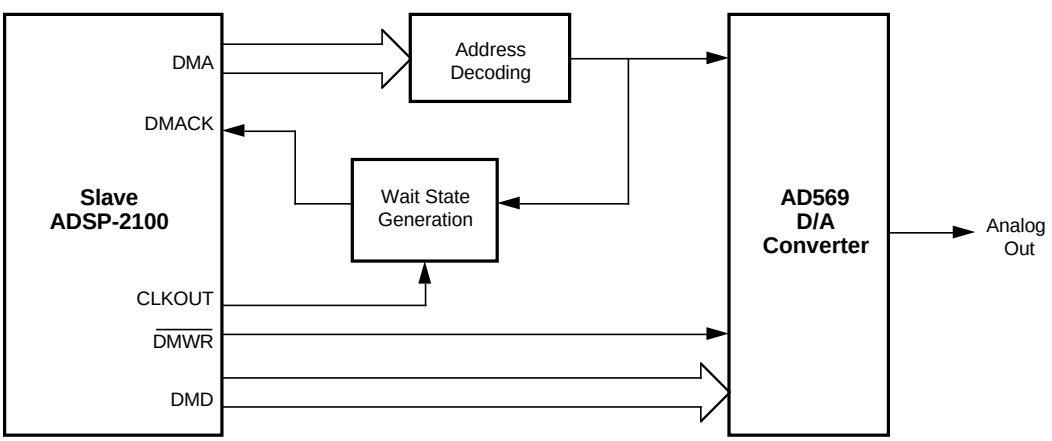

Figure 15.16 Slave Module D/A Converter Interface

hydrophone outputs. The gain is selected (from 1, 10, 100 and 1000) using an external 4-position rotary switch. AD7590 analog CMOS switches select the resistor combination for the amplifier's feedback loop. The CMOS switches are used to allow the future possibility of using CPU-generated signals to make gain selections (i.e., to implement automatic gain control).

The board contains two octal bus drivers to provide adequate buffering for the AD1332 outputs. The outputs of these buffers are tied to the backplane DMD bus (see Figure 15.18). The A/D output, which is in offset binary format, has to have its most significant bit inverted because of the fixed-point format that is used in the system.

A decoder provides a unique location for each AD1332 in the master's data memory address space. The configuration of this decoder must be different in each A/D board for each A/D converter to have a unique location. The A/D board requires +5V digital and ±12V analog power supplies. There are five connectors on the A/D board: a 96-pin male Eurocard connector that is used to interface to the backplane bus and four male BNC connectors that accept the hydrophone outputs.

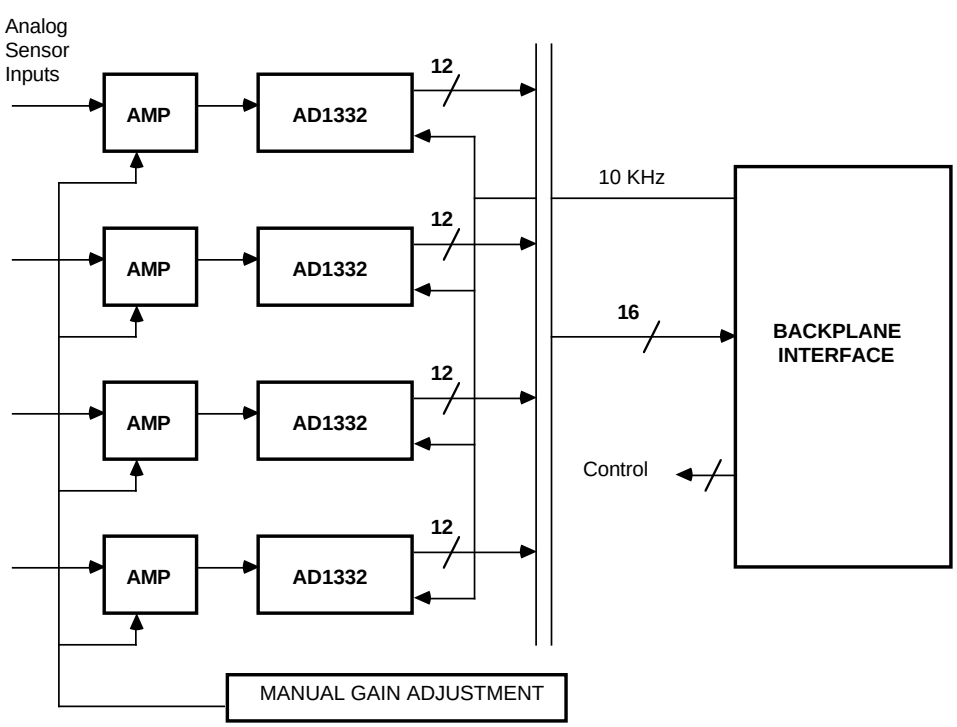

Figure 15.17 A/D Module Block Diagram

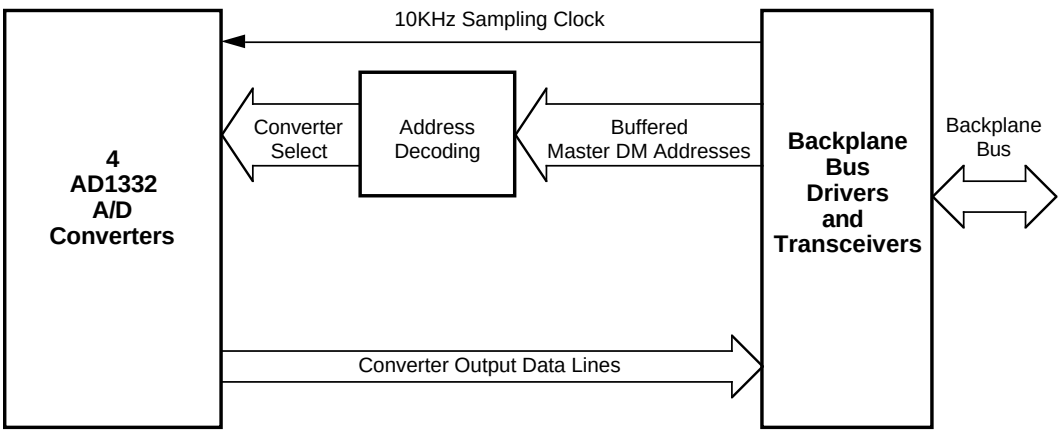

Figure 15.18 A/D Module Backplane Interface

15.6 SYSTEM FIRMWARE

The ADSP-2100 assembly code that is responsible for the master and slave CPUs' operation is discussed in this section.

15.6.1 Master Firmware

The firmware that runs in the master CPU is relatively short. It is assembled using the system specification source file shown in Listing 15.1.

```
.SYSTEM                          master_system;

.SEG/ROM/ABS=0/PM/CODE           rom_program_storage[2048];

.SEG/RAM/ABS=0/DM/DATA           sample_mem[8160];
.SEG/RAM/ABS=8160/DM/DATA        system_info[32];
.SEG/RAM/ABS=8192/DM/DATA        shading_coeff_mem[32];
.SEG/RAM/ABS=8224/DM/DATA        scratch_mem[2016];
.SEG/RAM/ABS=10240/DM/DATA       ad_converters[32];
.SEG/RAM/ABS=14337/DM/DATA       fifos[7];

{The ports declared below are used to set and clear various flags
as well as to communicate with the PC and the external signal
processor}

.PORT/ABS=H#300F                 setbmoutrdy;
.PORT/ABS=H#304F                 clrbmoutrdy;
.PORT/ABS=H#308F                 pcwe;
.PORT/ABS=H#310F                 setsbr;
.PORT/ABS=H#314F                 clrsbr;
```

(listing continues on next page)

```
.PORT/ABS=H#318F                    clrslhalt;
.PORT/ABS=H#31CF                    setslhalt;
.PORT/ABS=H#30C8                    setbpoe;
.PORT/ABS=H#30C9                    clrbpoe;
.PORT/ABS=H#30CA                    clrbmtaken;
.PORT/ABS=H#30CD                    setsmemrd;
.PORT/ABS=H#30CE                    clrsmemrd;
.PORT/ABS=H#30CF                    setslint;
.PORT/ABS=H#3007                    pcrd;
.PORT/ABS=14336                     beamsend;

.ENDSYS;
```

Listing 15.1 System Specification for Master Firmware

The master code, shown in Listing 15.2, only occupies 180 locations in program memory. It can be divided into two sections: the PC communications section and the A/D result handling section.

The PC communications section of the master code starts at the beginning of the file and ends at the *wait* routine. The beginning contains a series of port, variable and interrupt declarations. The program starts by clearing certain flags and then entering the *pc_init_wait* routine which causes the master CPU to wait until the PC is ready to download the system configuration data. Then, the *pc_comm* routine sets some of the internal CPU registers to be used during the communication. At the end of this routine, the PC is notified that the master is ready and the master CPU enters the *pc_wait1* loop.

The first six pieces of data downloaded by the PC are handled differently than the rest. These first six pieces are the following information:

- The number of sensors in the system
- The number of beams to be formed
- The total number of indexes to be used (the indexes are used by the slave CPUs to pick the desired input samples from the circular buffer)
- The number of slaves in the system
- The number of beams assigned to each slave
- The number of beams assigned to the last slave

This data is stored by the *init_sto* routine in the master CPU's data memory (which actually is the same as the slave data memories) in consecutive locations that are declared at the beginning of the program.

Next, the indexes that were calculated by the PC are received and stored in data memory by the *index_store* routine. Then in the *checksum* routine, the master compares the number of pieces of data it received with the number that the PC indicates that it sent (the PC sends a count value to the master in this routine). If these values are equal, that implies that the download was successful.

The *comm_end* routine prepares the master, using the downloaded data, for its next set of tasks (responding to A/Ds, etc.). This routine and the *init_sto* routine are the only times the master CPU reads a value from a slave data memory. This type of read operation is only allowed from a single slave which must be plugged into a designated slot on the backplane.

Another piece of data that is calculated in this routine is the shading coefficient that will be used in the current set of beams. The system currently can only handle a rectangular window with a magnitude of 1. Additional code to handle several different shading windows can be added later.

After configuration, the master CPU issues a signal which commands the slaves to prepare themselves using the recently downloaded data (the slave CPUs were in a TRAP state until now). The master waits for 1000 cycles, more than enough time for the slaves to get prepared. It then enables two nested interrupts and enters the *wait* loop.

The second major section of the code starts at the *wait* loop. The master waits for the "conversion complete" interrupt to occur, then it jumps to the *adcomplete* routine. During this routine, the master reads the output latches (conversion results) of each AD1332 and writes these values into all of the slaves' sample buffers. The master keeps track of a few pointers in order to address the circular buffers and the A/D boards properly.

Next, in the *sendbeam* routine the master reads beam samples from all the slave FIFOs in sequence and writes them to the system output port. Output beam samples are sent one at a time. The handshaking with the external signal processor is executed for each beam sample. The master starts the handshake by asserting the BMOUTRDY signal and then waits for the BMTAKEN interrupt (set by the external processor) that indicates that the external processor is ready to receive the next value. Then the interrupt is cleared by the master and the next beam sample is sent out in the same manner.

15 Sonar Beamforming

For a given number of desired beam directions, it is not always possible to divide the job evenly among the slaves. Thus, the number of beams formed by the last slave may be different than the number formed by each of the rest of the slaves. Consequently, the output FIFO of the last slave is handled by the *oneslave* routine, which serves a similar purpose to the *sendbeam* routine except it only handles the last slave.

Once the beam output tasks are completed, a RTI (return from interrupt) instruction is executed and the program returns to the *wait* loop where the master waits for the next *adcomplete* interrupt.

The master continues its cyclic, double-interrupt-driven operation until the assertion of RESET or a system power-down.

```
.MODULE/ROM/ABS=0    master_code;

{The following are declarations for ports that are used to set
and clear several flags. Data memory mapped D-flipflops are used
to generate flags}

.PORT                setbmoutrdy;  {Beam output ready flag}
.PORT                clrbmoutrdy;
.PORT                pcwe;      {PC output port}
.PORT                setsbr;    {Slave bus request flag}
.PORT                clrsbr;
.PORT                setslhalt; {Slave halt flag}
.PORT                clrslhalt;
.PORT                setbpoe;   {Backplane output enable flag}
.PORT                clrbpoe;
.PORT                clrbmtaken;
.PORT                setsmemrd; {Slave memory read flag}
.PORT                clrsmemrd;
.PORT                setslint;  {Slave interrupt}
.PORT                pcrd;      {PC input port}
.PORT                beamsend;  {Interrupt the external processor}

{The following are variables that contain the system
configuration info.}

.VAR/DM/ABS=8160     sensor_num;
.VAR/DM/ABS=8161     beam_num;
.VAR/DM/ABS=8162     index_num;
.VAR/DM/ABS=8163     slave_num;
.VAR/DM/ABS=8164     beams_per_slave;
.VAR/DM/ABS=8165     last_slave_beam_num;
```

```
{The following is the main body of the master program}

{Interrupt vectors occupy the first four PM locations}
                    JUMP adcomplete;  {Vectored address for
IRQ0}
                    RTI;
                    JUMP beamtaken;   {for IRQ2}
                    RTI;

{Program execution starts here}
                    IMASK=b#0000;
                    ICNTL=b#00000;
                    AY0=h#0FFF;
                    DM(clrbmoutrdy)=MX0;    {Clear all flags by}
                    DM(clrbmtaken)=MX0; {writing a value into}
                    DM(clrsbr)=MX0;     {their respective DM}
                                        {mapped port locations}
                    DM(clrslhalt)=MX0;
                    DM(clrbpoe)=MX0;
                    DM(clrsmemrd)=MX0;
pc_init_wait:       AX0=DM(pcrd);     {Wait for the ready}
                    AR=AX0-AY0;       {message from the PC}
                    IF EQ JUMP pc_comm;
                    JUMP pc_init_wait;

pc_comm:            MX0=h#FF0F ;      {Initial set up before
the}
                    AY0=5;            {PC communications.}
                    AF=AY0+1;   {A message is sent to the PC}
                    AY0=h#1FFF; {at the end of this routine}
                    AX1=h#FFFF; {signaling that the master is}
                    MX1=h#FF0F; {ready}
                    M0=0;
                    I1=8160;
                    L1=6;
                    I2=0;
                    M2=1;
                    L2=8160;
                    M3=-1;
                    I4=8166;
                    M4=1;
                    I5=9000;
                    M5=0;
                    L5=1;
                    I6=0;
                    M6=1;
```

(listing continues on next page)

```
                        L6=2055;
                        DM(setsbr)=AY0; {Set slave Bus Request flag}
                        DM(pcwe)=MX0;   {Send Ready mesage to the
PC}
                        JUMP pc_wait1;

pc_comm_end_check:      AY1=I3;       {Counter register to check
for}
                        AR=AY1-1;     {the end of the index}
                        IF LT JUMP pc_end;      {downloading}

pc_wait1:               AY1=DM(pcrd);     {Wait for FFFF from the
PC}
                        AR=AY1-AX1;
                        IF EQ JUMP pc_read;
                        JUMP pc_wait1;

pc_read:                MODIFY(I6,M6);  {Write the initial 6 pieces}
                        AF=AF-1 ;       {of data and then start}
                        IF LT JUMP index_store; {the index storage}

init_read:              AX0=DM(pcrd);   {Read data sent from the PC}
                        AY1=h#E000;
                        AR=AX0 AND AY1;   {Mask out bottom 13 bits}
                        AY1=h#A000;
                        AR=AR-AY1;
                        IF EQ JUMP init_sto; {Compare to h#A000}
                        JUMP init_read;
init_sto:               AR=AX0 AND AY0;      {Mask out top 3 bits}
                        DM(setbpoe)=MX0;  {Enable backplane drivers}
                        DM(I1,M1)=AR;
                        DM(setsmemrd)=MX0; {Enable reading from}
                                           {slave DM}
                        L4=DM(beam_num);
                        I3=DM(index_num);
                        DM(clrsmemrd)=MX0; {Disable reading from}
                                           {slave DM}
                        L3=I3;
                        DM(clrbpoe)=MX0;  {Disable backplane
drivers}
                        JUMP pc_wait1;

index_store:            AX0=DM(pcrd);     {Read the PC output}
                        AY1=h#E000;
                        AR=AX0 AND AY1;   {Mask out bottom 13 bits}
                        AY1=h#A000;
```

```
                    AR=AR-AY1;
                    IF EQ JUMP sto;   (Compare to h#A000}
                    JUMP index_store;

sto:                AR=AX0 and AY0;
                    DM(setbpoe)=MX0;
                    DM(I2,M2)=AR ;    {Store index}
                    MODIFY(I3,M3);
                    DM(clrbpoe)=MX0;
                    JUMP pc_comm_end_check;
pc_end:             AX1=h#FF0F;
pc_end_loop:        AY1=DM(pcrd);     {Read PC output and
decide}
                    AR=AX1-AY1;    {whether communication
should}
                    IF EQ JUMP checksum;    {be completed or not}
                    JUMP pc_end_loop;
checksum:           AX1=I6;           {Perform cheksum
operation}
checksum_loop:      AY1=DM(pcrd);
                    AR=AX1-AY1;
                    IF EQ JUMP comm_end;
                    JUMP checksum_loop;

comm_end:           MX0=h#F0;         {This routine ends the PC}
                    DM(pcwe)=MX0;  {communications and does
some}
                    DM(setbpoe)=MX1; {reads from the slave DM
in}
                    AY0=h#0201;       {order to prepare for the}
                    AF=AY0-1 ;        {beamforming tasks}
                    AY0=0;
                    DM(setsmemrd)=MX0;
                    SI=DM(sensor_num); {Read #of sensors from}
                    DM(clrsmemrd)=MX0; {slave DM and do the}
                    AY0=SI;     {necessary division to obtain a}
                    AR=AY0-1;   {scaled magnitude value for the}
                    IF EQ JUMP onesensor;   {rectangular shading}
                    AY0=0;                  {window}
                    SR=LSHIFT SI BY 9 (LO);
                    AX1=SR0;
                    ASTAT=0;
                    DIVQ AX1;
                    DIVQ AX1;DIVQ AX1;DIVQ AX1;
                    DIVQ AX1;DIVQ AX1;DIVQ AX1;
                    DIVQ AX1;DIVQ AX1;DIVQ AX1;
```

(listing continues on next page)

```
                        DIVQ AX1;DIVQ AX1;DIVQ AX1;
                        DIVQ AX1;DIVQ AX1;DIVQ AX1;
                        DM(8192)=AY0;     {Shading coefficient}
                        JUMP allsensor;
onesensor:              AY0=h#7FFF; {This is shading coefficient}
                        DM(8192)=AY0; {in case of a single sensor}
allsensor:              DM(setsmemrd)=MX1;
                        AY0=DM(slave_num); {Do the rest of the}
                        AX1=DM(beams_per_slave); {necessary reads}
                        MY1=DM(last_slave_beam_num);
                                    {before releasing slave BR flag}
                        MX0=DM(sensor_num);
                        DM(clrsmemrd)=MX0; {Clear the slave DM read}
                        MY0=255;           {flag and set up some}
                        MR=MX0*MY0(UU);{pointers to be used later
on}
                        SI=MR0 ;          {#of samples to be placed}
                        SR=LSHIFT SI BY -1 (HI);
                                          {into the sample buffer}
                        L2=SR1;
                        AR=AY0-1;
                        DM(clrsbr)=MX1;
                        AX0=AR;
                        I1=10240;
                        L1=MX0;
                        I2=0;
                        M2=1;
                        IF EQ JUMP fix_base;
                        I3=h#3800;        {Base addr. for the FIFOs}
                        AR=AY0+1;         {with multiple slaves}
                        L3=AR;
                        JUMP normal;
fix_base:               I3=h#3801;        {Base addr. for the FIFO}
                        L3=0;                   {with single slave}
normal:                 DM(setslhalt)=MX1; {HALT the slaves, this}
                        DM(clrslhalt)=MX1; {will cause them to get}
                        DM(clrbpoe)=MX1; {out of their TRAP state}
                        CNTR=1000;
                        DO slave_wait UNTIL CE; {Wait until all}
slave_wait:                NOP;           {slaves are ready to go}
                        AR=1;
                        AY1=1;
                        ICNTL=b#00101;
                        IMASK=b#0001;
                                    {Enable sampling interrupt IRQ0}
```

```
wait:               JUMP wait;

adcomplete:         AR=AR-AY1;              {Ignore the first}
                    IF EQ JUMP first_adcomp; {IRQ0 by using
this}
                    DM(setsbr)=MX1;
                    AF=AY0-1;
                    DM(setbpoe)=MX1;  {This routine is used}
                    CNTR=MX0;         {to fill up the sample}
                    DO sample_store UNTIL CE; {memory after
each}
                       MX1=DM(I1,M1);       {A/D conversion}
sample_store:          DM(I2,M2)=MX1;
                    DM(clrsbr)=MX1;
                    DM(setslint)=MX1;
                    DM(clrbpoe)=MX1;

sendbeam:           IF EQ JUMP oneslave; {Read out the beams in}
                    MODIFY(I3,M1);       {a sequential manner
in}
                    CNTR=AX0;   {each FIFO in order of
numbering}
                    DO beamout UNTIL CE; {Send beam outputs via}
                       CNTR=AX1;       {the digital output port}
                       DO fifo_out UNTIL CE;
                          DM(setbpoe)=MX1;
                          MX1=DM(I3,M0);
                          DM(clrbpoe)=MX1;
                          DM(beamsend)=MX1;
                          IMASK=b#0100;{Enable data receive
intr}
                          DM(setbmoutrdy)=MX1;    {Set a flag}
                          CNTR=6;  {for the external processor}
                          DO resp_wait UNTIL CE;
resp_wait:                   NOP;
fifo_out:                 NOP;
beamout:                MODIFY(I3,M1);


oneslave:           CNTR=MY1;
                    DO endfifo UNTIL CE;
                       DM(setbpoe)=MX1; {This routine is for}
                       MX1=DM(I3,M0);   {single slave}
                       DM(clrbpoe)=MX1; {configurations and is}
                                        {also used for handling}
                       DM(beamsend)=MX1; {the last FIFO read
out}
                       IMASK=b#0100;        {It handles the}
```

```
                              DM(setbmoutrdy)=MX1; {irregularity of the}
                              CNTR=6;              {last set of beams,}
                              DO resp_wt UNTIL CE; {i.e. possibly fewer}
resp_wt:                         NOP;              {beams}
endfifo:                      NOP;
first_adcomp:              AR=0;
                           RTI;

beamtaken:                 DM(clrbmoutrdy)=MX1;
                                       {Interrupt routine to handle}
                           DM(clrbmtaken)=MX1;
                                       {the data receive confirmation}
                           RTI;        {from the external processor}

.ENDMOD;
```

Listing 15.2 Master Firmware

15.6.2 Slave Firmware

The slave board requires less firmware than the master. It is assembled using the system specification source file shown in Listing 15.3.

```
.SYSTEM                              slave_system;

.SEG/ROM/ABS=0/PM/CODE               rom_program_storage[2048];
.SEG/RAM/ABS=2048/PM/DATA            index_mem[2048];

.SEG/RAM/ABS=0/DM/DATA               sample_mem[8160];
.SEG/RAM/ABS=8160/DM/DATA            system_info[32];
.SEG/RAM/ABS=8192/DM/DATA            shading_coeff_mem[32];
.SEG/RAM/ABS=8224/DM/DATA            scratch_mem[2016];

{The ports declared below are used to set and clear some flags as
well as to write to the DAC and to read from some hardware
switches}

.PORT/ABS=H#2800                     beamout;
.PORT/ABS=H#3000                     beamdac;
.PORT/ABS=H#3800                     clrslint;
.PORT/ABS=H#3900                     slave_id;
.PORT/ABS=H#3A00                     dac_beam_sel;

.ENDSYS;
```

Listing 15.3 System Specification for Slave Firmware

The slave firmware code occupies only 100 locations in program memory. The slave firmware, shown in Listing 15.4, can be divided into two sections: the system set-up section and the beamforming section.

The set-up section of the code starts at the beginning of the file and ends at the *wait* routine. The beginning contains a series of port, variable and interrupt declarations. Only one interrupt, SLINT, is recognized by the slave; this is the interrupt initiated by the master to start the beamforming operation.

After system parameters are declared, the slave enters a TRAP state. The slave gets reactivated after the master is finished communicating with the PC. The master asserts the SLHALT signal, which wakes up the slave and causes the program execution to continue from the location following the TRAP instruction.

In the first part of the program, the slave moves the indexes from its data memory into its "index memory" which is located in its program memory space (the indexes are in data memory initially because the master has to store them there temporarily). Then, the slave sets up its address registers using the downloaded beamforming information. Next, the slave enters the *wait* loop to wait for the beamforming interrupts issued by the master.

The slave constantly monitors the D/A beam selection switch while it is in the *wait* loop. Since the slave returns to the *wait* loop every 100μs, it can decide, in real time, which beam to send out through the analog port.

The second major section of the program starts at the *wait* loop. As soon as the SLINT interrupt is received, the slave jumps to the *beam_form* routine. The *beam_form* routine contains very tight loops which allows the slave to form a large number of beams. The routine reads the index memory and picks the indexed samples from the sample buffer. These samples are the delayed samples that are needed for the beam summation.

The frame of the circular buffer is rotated every time a new set of samples comes in. Therefore the indexes that are read must be modified before being used, because they are referenced to the absolute origin of the circular buffer.

The samples that are read are multiplied by the shading factor (which is currently 1) and accumulated in the MR register. The resulting beam sample is written into the FIFO. If the beam sample belongs to the beam that is requested at the analog port, it is then written to the *beamdac* port (the D/A converter). Before returning from the interrupt, the slave clears the SLINT flag. Then the program returns to the *wait* loop to wait for the next SLINT interrupt.

The slave continues its cyclic, single-interrupt-driven operation until the assertion of RESET or a system power-down.

```
.MODULE/ROM/ABS=0    slave_code;

{The following are declarations for data memory mapped ports. One
is used to clear a flag, while others are used to write data to
the DAC, FIFO and read the hardware switches on the slave board}

.PORT             beamout;       {FIFO}
.PORT             beamdac;       {DAC}
.PORT             clrslint;      {Clear the slave interrupt}
.PORT             slave_id;      {Slave identity dipswitch}
.PORT             dac_beam_sel;  {Analog output selection switch}

{The following are variables that contain the system
configuration info}

.VAR/DM/ABS=8160     sensor_num;
.VAR/DM/ABS=8161     beam_num;
.VAR/DM/ABS=8162     index_num;
.VAR/DM/ABS=8163     slave_num;
.VAR/DM/ABS=8164     beams_per_slave;
.VAR/DM/ABS=8165     last_slave_beam_num;

{The following is the main body of the slave program}

                     JUMP beam_form;   {Vectored addr. for IRQ0}
                     RTI;
                     RTI;
                     RTI;
```

```
                    IMASK=b#0000;     {Disable interrupts}
                    ICNTL=b#00000;
                    DM(clrslint)=MX0;
                    TRAP;             {TRAP until pc_comm ends}

pc_comm_end:        I1=0;       {This initial routine is used}
                    M1=1;       {to transfer the indexes from}
                    L1=DM(index_num); {sample_mem into the}
                    I4=2048;                {index_mem in PM}
                    M4=1;
                    L4=L1;
                    CNTR=L1;
                    DO index_store UNTIL CE;
                       MX0=DM(I1,M1);
index_store:           PM(I4,M4)=MX0;

                    MX0=DM(sensor_num);
                    MY0=255;
                    MR=MX0*MY0(UU);
                    SI=MR0;
                    SR=LSHIFT SI BY -1 (HI);
                                      {SR1 contains the length}
                    I1=MX0;           {of the circular sample}
                    M1=MX0;           {buffer}
                    L1=SR1;
                    SI=DM(slave_id);  {Determine this slave's}
                    SR=LSHIFT SI BY -12 (HI);
                                      {ID# and the starting}
                    AX0=DM(slave_num); {location of the first}
                    AY0=SR1;          {index.Also determine}
                    AF=AX0-AY0;       {the # of indexes}
                    AF=AF-1;          {for this slave}
                    IF EQ JUMP last_slave;
                    SE=DM(beams_per_slave);
                    JUMP all_slave;
last_slave:         SE=DM(last_slave_beam_num);

all_slave:          MX0=DM(sensor_num);
                                      {This routine calculates}
                    MY0=DM(beams_per_slave);
                                      {the starting address of}
                    MR=MX0*MY0(UU);   {this slave's indexes}
                    SI=MR0;
                    SR=LSHIFT SI BY -1 (HI);
                    MX0=SR1;
                    AR=AX0-AY0;
```

(listing continues on next page)

```
                AY1=AR;
                AR=AY1-1;
                MY0=AR;
                MR=MX0*MY0(UU);
                SI=MR0;
                SR=LSHIFT SI BY -1 (HI);
                AX1=2048;
                AY1=SR1;
                AR=AX1+AY1;
                I5=AR;            {Starting address of the}
                M5=1;             {indexes for this slave}
                M7=-1;
                MX0=SE;
                MY0=DM(sensor_num); {# of indexes per beam}
                MR=MX0*MY0(UU);
                SI=MR0;
                SR=LSHIFT SI BY -1 (HI);
                L5=SR1;           {Total # of indexes to}
                M3=0;             {be used by this slave}
                L3=L1;
                I6=8192;
                M6=0;
                L6=1;
                MY1=DM(I6,M6); {Shading coefficient; there}
                AY1=DM(sensor_num);
                           {is only one now since it is}
                AR=AY1-1;   {a rectangular window}
                AX1=AR;
                ICNTL=b#00001;
                IMASK=b#0001; {Enable slave interrupt IRQ0}

wait:           SI=DM(dac_beam_sel); {Setup down counter to}
                SR=LSHIFT SI BY -12(HI);
                                  {be used in deciding}
                AY1=SR1;          {which beam to send out}
                AF=AY1+1;         {to the DAC}
                JUMP wait;  {Wait for sample buffer update}

beam_form:      CNTR=SE;
                DO beam_end UNTIL CE;
                   MR=0; AY0=PM(I5,M5); {Read index}
                   M3=I1;
                   I3=AY0;
                   MODIFY(I3,M3); {Modify index}
                                       MX0=DM(I3,M3); {Get first sample}
                                       AY0=PM(I5,M5);
```

```
                        CNTR=AX1;
                        DO single_beam_sample UNTIL CE;{Beamform}
                           I3=AY0;
                           MODIFY (I3,M3);
single_beam_sample:        MR=MR+MX0*MY1(SS),
MX0=DM(I3,M3),AY0=PM(I5,M5);
                        MR=MR+MX0*MY1(RND), AY0=PM(I5,M7);
                        AF=AF-1;              {Check which beam
to}
                        IF EQ JUMP dac_write; {send to the DAC}
beam_end:               DM(beamout)=MR1;      {Write result to
FIFO}
                     MODIFY(I1,M1);            {Advance circular
sample}
                     DM(clrslint)=MX0;         {buffer pointer}
                     RTI;

dac_write:           DM(beamdac)=MR1;        {Write result to DAC}
                     JUMP beam_end;

.ENDMOD;
```

Listing 15.4 Slave Firmware

15.7 SYSTEM SOFTWARE

The system software consists of the PC program that is responsible for the user interface and the downloading of the system configuration data. The code is written in the C language and compiled on the Microsoft C Compiler.

The declaration section at the beginning of the program includes certain useful libraries, defines a number of variables and declares the 8-bit parallel I/O port addresses. These ports are located on an Analog Devices RTI-817 parallel I/O card which is plugged into the PC's backplane. Following this section there are a series of function definitions and the execution loop of the program.

The program interactively takes in the system variables from the user. Some questions are displayed on the screen which are answered by the user via the keyboard. The values that have to be entered by the user are: the number of sensors, the number of beams to be formed, the number of slaves in the system, the cartesian coordinates for the sensor locations and the spherical coordinates for the desired beams. The program assigns these values to variables and arrays in order to calculate the necessary tap delays for beamforming.

The program converts the spherical coordinate beam directions to cartesian coordinates. Then it calculates the necessary delays using the equation (2). The propagation speed of sound in water is assumed to be 1470 m/s, which is typical for the ocean water. (This value can be changed easily in the code to conform to the application environment.) The program identifies the beams that the system is unable to produce with the given array configuration. This task is accomplished by checking whether any one of the required delays falls outside of the sample buffer length.

Next, the program determines the maximum number of beams that can be formed with the given system configuration. It also calculates the number of beams to be assigned to each slave and the number of beams to be assigned to the last slave. These values are stored as variables to be downloaded to the master.

The program downloads the system configuration information to the master, beginning with six pieces of system set-up data, as explained earlier in the master firmware section. Then the program sends all the calculated indexes to the master, followed by a checksum, which, as explained earlier, corresponds to the number of pieces of data (number of indexes + 6) just downloaded. If the master acknowledges that the download was successful, the downloading operation is completed by sending a confirmation message to the PC screen. If the master indicates an unsuccessful download, the downloading operation is terminated by sending a failure message to the the screen. In this case, the user is given the choice of aborting or retrying the download.

The download is easily executable by the user. Once the system parameters are entered, it takes at most a few seconds for the PC to download all the information to the master.

15.8 ENHANCEMENTS

There are several ways to improve the performance, functionality and the user interface of the example beamformer. Possible additional features as well as some architectural and circuit level enhancements are briefly discussed in this section.

15.8.1 Additional Features

A large number of features can be added to this system without great difficulty. One feature is the choice of frequencies for the input sampling

clock. It is possible to route an external clock to the A/D boards by incorporating additional hardware on the master board. The external sampling clock option would an additional piece of data collected by comm.*c* during the system parameter configuration. The program would download the information to the master, which would activate the necessary signals for clock selection. The program would also have to modify the beam assignments, because the system cycle may be shorter or longer depending on the choice of sampling frequency.

A software enhancement is the ability to save the current system parameters into a file. This allows the system to be restarted by instructing *comm.c* to boot up the system using the saved parameters instead of getting them from the user. A user could edit this file to restart the system with a new set of configuration parameters in a very short time. This feature would make *comm.c* even more user-friendly.

Another feature is the addition of a filtering and smoothing circuit for the output of the D/A converter. Smoothing the output of the D/A converters would make it possible to feed the analog output beams into a spectrum analyzer or a general purpose data acquisition system for further signal analysis.

The availability of various shading windows for the inputs is another useful enhancement. The modifications would have to be done in *comm.c* and also the system firmware programs. In *comm.c*, the shading window option would be gotten from to the user and the shading coefficients would be calculated on the fly and downloaded to the master. The master would send these coefficients to the slaves instead of the unit rectangular window.

The shading factors would reside in the program memory space of each slave and could ultimately be used by the slaves during beamforming. The downloading overhead would be minimal. The additional program memory accesses during the beamforming loop should not result in a performance degradation since they can be performed in parallel with the data accesses. Careful calculations are needed to determine the exact performance consequences of such a system modification.

15.8.2 Performance Improvements

There are several ways to improve the performance of the beamformer described in this chapter by making relatively minor modifications to the hardware and software. Some modifications, with ascending levels of complexity, are discussed in this section.

15 Sonar Beamforming

An important performance issue for a real-time beamformer is the beam throughput. The main goal of such a system is to form as many simultaneous beams as possible using the existing technology. There are several ways to improve the beam throughput within the existing distributed processing architecture. The most obvious and relatively easy way to achieve this goal is to replace the ADSP-2100s with ADSP-2100As. The ADSP-2100A has an 80ns instruction cycle time as opposed to 125ns cycle time of the ADSP-2100. This upgrade would result in ≈50% increase in system beam throughput.

The ADSP-2100A is pin and source code compatible with the ADSP-2100. This allows the easy upgrade of the system with no firmware changes. The modifications that are needed are mostly in hardware. The timing requirements during the ADSP-2100A's data and program memory access operations must be carefully analyzed and faster devices should be placed on the critical data paths. It is possible to upgrade only the existing memory components to compensate for the new shorter data and program memory access cycles.

A modification to *comm.c* would also be necessary because it would have to be able to assign a larger number of beams per slave. The input bandwidth of the external signal processor would have to be carefully evaluated, because it is likely that the output bandwidth of the upgraded system, in maximum configuration, would be higher than the input capacity of the external processor. If such an incompatibility resulted, you could use fewer slaves to match the output bandwidth requirements. The overall consequence would be a cost reduction for the less demanding users and higher performance for the more demanding users.

Another improvement is increasing the maximum number of sensor inputs. It is possible to add more input channels to the beamformer. However, each added A/D converter would have to be placed in a unique location in the master CPU's data memory address space, requiring some additional address decoding circuitry on the A/D boards. There is enough room for more digital components on these boards and the changes in the wiring would be relatively simple.

One important effect of adding channels is a reduction of the maximum number of beams that can be formed simultaneously, because the master will have to read more inputs within one system cycle and consequently will have less time to read the results from the slave FIFOs. It is possible to keep the beam throughput at the current level by upgrading the

processors while increasing the number of sensor inputs. These performance tradeoffs should be considered carefully before expanding the A/D capabilities of the system.

A major improvement is to redesign the A/D boards with multi-channel A/D converter hybrids replacing the single-channel AD1332s. Analog Devices' AD1334 would be the optimal choice. The system redesign effort that is necessary to implement the substitution of AD1334s is of moderate difficulty. The savings in the number of A/D cards would prove this redesign effort to be very valuable, especially if the need for input channels is expected to rise.

The AD1334 contains four sample and hold circuits, a 4-to-1 analog multiplexer, an AD7672 12-bit, 5μs A/D converter and an output FIFO. The sample and hold circuit (AD585) and the A/D converter are the same as the ones used in the AD1332. The output FIFO is 12 bits wide and 64 locations deep. It is possible to use this hybrid in a mode where all of the sample and hold circuits sample the inputs simultaneously.

Some overhead analog circuitry must be added externally because of the lack of on board low-pass filters in the AD1334. The AD1334 has the same package as the AD1332, so it would be possible to fit as many as three A/D hybrid packages on the same board even with the additional digital and analog overhead circuitry that is needed. Such a construction strategy would allow each A/D board to handle up to 12 sensor inputs.

Some minor modifications in the master firmware would also be needed in order to properly address the AD1334s.

15.9 REFERENCES

Baggeroer, A. B. 1978. "Sonar Signal Processing." In *Applications of Digital Signal Processing*, A. V. Oppenheim, Ed. Englewood Cliffs, NJ: Prentice-Hall, Inc.

Curtis, T. E. and R. J. Ward. 1980. "Digital Beam Forming for Sonar Systems." *IEEE Proceedings*, Vol. 127, Pt. F, No. 4.

Gray, D. A. 1985. "Effect of Time-Delay Errors on the Beam Pattern of a Linear Array." *IEEE Jourmal of Oceanic Engineering*, Vol. OE-10, No. 3.

Hodgkiss, W. S. and V. C. Anderson. 1981. "Hardware Dynamic Beamforming." *Jour. Acoust. Soc. Am.* 69(4).

Janssen, R. J. 1987. "Sonar Beamforming and Signal Processing." *Electronic Progress*, Vol. 28, No. 1, Raytheon Co.

Karagozyan, K. 1988. "A Multi-Processor Based Digital Beamforming

15 Sonar Beamforming