IS-4990
UC-37

AMES LABORATORY

The Ames Waveform Digitizer Module

USER GUIDE

Version 1.1

**H. B. Crawley, M. S. Gorbics, J. F. Homer, Jr., R. McKay,
W. T. Meyer, E. I. Rosenberg, and W. D. Thomas**

**Ames Laboratory
Iowa State University
Ames, Iowa 50011**

**Prepared For
The U. S. Department of Energy
Under Contract W-7405-eng -82**

DISCLAIMER

Printed in the United States of America

Available from
National Technical Information Service
U.S. Department of Commerce
5265 Port Royal Road
Springfield, VA 22161

IS--4990

DE90 011718

The Ames Waveform Digitizer Module

USER GUIDE

H. B. Crawley, M. S. Gorbics, J. F. Homer, Jr., R. McKay,
W. T. Meyer, E. I. Rosenberg, and W. D. Thomas

Ames Laboratory* and Department of Physics
Iowa State University
Ames, IA 50011

Date Transmitted: February, 1989

*Operated by Iowa State University for
the U.S. Department of Energy under Contract
No. W-7405-ENG-82.

TABLE OF CONTENTS

ABSTRACT

This document describes a waveform digitizer module developed for the DELPHI experiment at the CERN Laboratory in Geneva, Switzerland. The Ames Waveform Digitizer is a single electronics board conforming to the FASTBUS standard (IEEE–960) which digitizes 32 channels of analog waveforms, removes data values below a settable threshold, and can reformat the data and perform fast analysis using an on-board microprocessor. This guide is intended to help the user install and use the modules in a data acquisition system. The technical details necessary for repair or modification will be available in a separate Technical Manual.

1. INTRODUCTION

The Ames Waveform Digitizer is a single-width FASTBUS slave module which digitizes input waveforms at rates up to 15 megasamples per second (mps). Each board has 32 input channels, zero suppression circuitry for data compaction, and an on-board Motorola 68000 microprocessor operating at 16 MHz. A summary of the technical parameters is given in Appendix A.

This document is intended to aid the user who wants to interface the board to a particular data acquisition system. Technical information needed to repair or modify the board is contained in a separate Technical Manual.

We use a few notational conventions which need to be defined. In general, we try to adhere to standard usage where possible. Hexadecimal numbers are denoted by appending a lower-case h at the end, e.g., 100h. An exception to this is the case of addresses for the 68000 microprocessor. In order to conform to standard usage we use a dollar sign before the number, e.g., $4000. Although the number is still hexadecimal, this form carries the additional information that it is a 68000 address that is being discussed. In referring to FASTBUS CSR registers and their bits we follow the notation of the FASTBUS specification (see bibliography). Thus, for example, bit 11 of CSR 10 is called CSR#10h(11) and the range of bits 5 through 7 in CSR 10h is CSR#10h(05:07).

This module was developed by the High Energy Physics group of the Ames Laboratory, Iowa State University, for use with the High-density Projection Chamber (HPC) in the DELPHI experiment at the CERN Laboratory, Geneva, Switzerland. In this experiment, the digitization clock runs at 14.675 MHz and the zero suppression clock at 4.5 MHz. Both of these clocks are supplied to the module from an external source. When we give a value for timings in this guide we implicitly assume these clock frequencies. It is possible to vary these values somewhat for other experiments and timings will change accordingly.

Patents related to this module are pending.

2. FUNCTIONAL DESCRIPTION

2.1 Overview

The operation of the module is divided into three states, LOAD, DUMP, and READOUT, with control of these states determined by external signals on the FASTBUS TR lines as described in section 2.7. During the LOAD state flash analog-to-digital converters (FADCs) digitize the input waveforms and store the results in the cache memories. During the DUMP state the data are transferred from the cache memory through a zero suppression circuit and into a front-end buffer (FEB). The module is in the READOUT state whenever it is not in the LOAD or DUMP state and it is only in this state that FASTBUS access is permitted and that the microprocessor can operate.

The LOAD state begins when the module receives a WARNING/CLEAR signal on the TR lines and ends when 255 data samples have been taken. Provision has been made for taking eight values of presample data between the WARNING/-CLEAR and an event trigger (usually called the beam cross-over, or BCO, in the technical documentation). This allows the user to record samples of the baseline just before an event occurs, if prior knowledge of the event time exists (for example, in a colliding beam storage ring where the beam cross-over time is precisely known). In other applications, it is possible to use the WARNING/CLEAR for the event trigger and the eight presample values simply become the first eight data values. See section 3.1 for more details.

At the end of the LOAD state, the board automatically enters the DUMP state, which lasts about 450 μs when using a 4.5 MHz DUMP clock. At the end of the DUMP state, the board automatically goes to the READOUT state. Asserting the external QRST signal at any time during the LOAD or DUMP also puts the module in the READOUT state.

As figure 2.1 shows, the input section of each channel first has an analog buffer circuit whose primary function is to provide a low-impedance input to the FADC. The FADC samples the voltage of the waveform 255 times on the downward transitions of the externally supplied clock and stores the results in its local cache memory.

Figure 2.2 shows a block diagram of the entire board. The cache memories are grouped together in four blocks of eight, each block having its own zero suppression

circuit. At the completion of the LOAD state (i.e., 255 samples for each channel are in the cache memory) the board enters the DUMP state. The eight channels in a block are read out sequentially through a zero suppression circuit that applies programmable threshold and width criteria to the digitized waveform. The zero suppressed data are stored in the FEB, which is divided into four areas reserved for data from separate events. Selection of one of the four FEB areas is made via external signals applied to two of the FASTBUS TR lines. Note that these lines affect writing to the FEB only during the DUMP state (see section 2.7). Once the data are in the FEB, the DUMP state ends and the board is ready to accept a new event, writing it into a different area of the FEB. If at any time during the LOAD or DUMP state external logic decides the event is not interesting, a signal applied to the external QRST line will abort the data taking and place the board in the READOUT state, ready to take a new event.

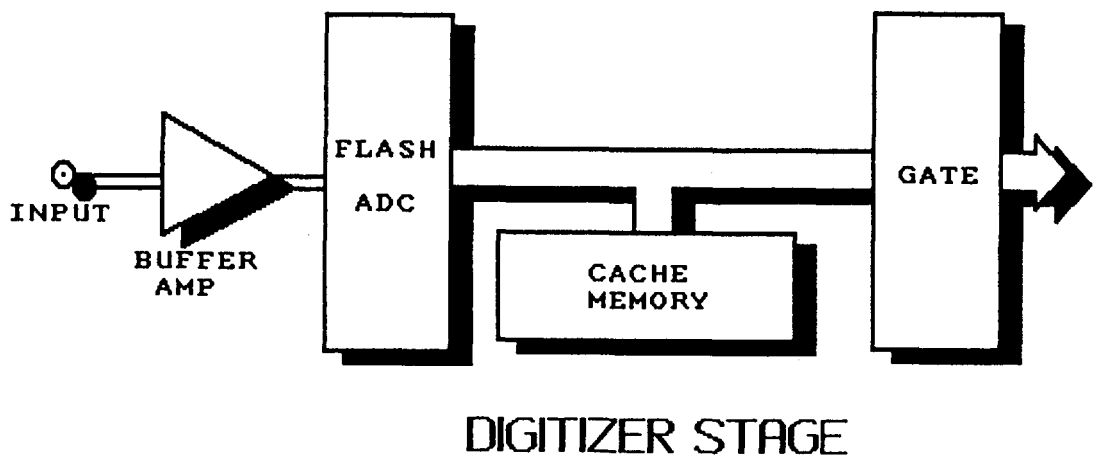

Figure 2.1. The digitizer input stage.

The FEB memory is connected to a common bus which goes both to the FASTBUS coupler and to a 68000 microprocessor with 128 Kbytes of random access memory (RAM). Thus the contents of the FEB are directly available to the user via FASTBUS and to the microprocessor for further analysis.

The readout of the event, either directly from the FEB or from a microproc-

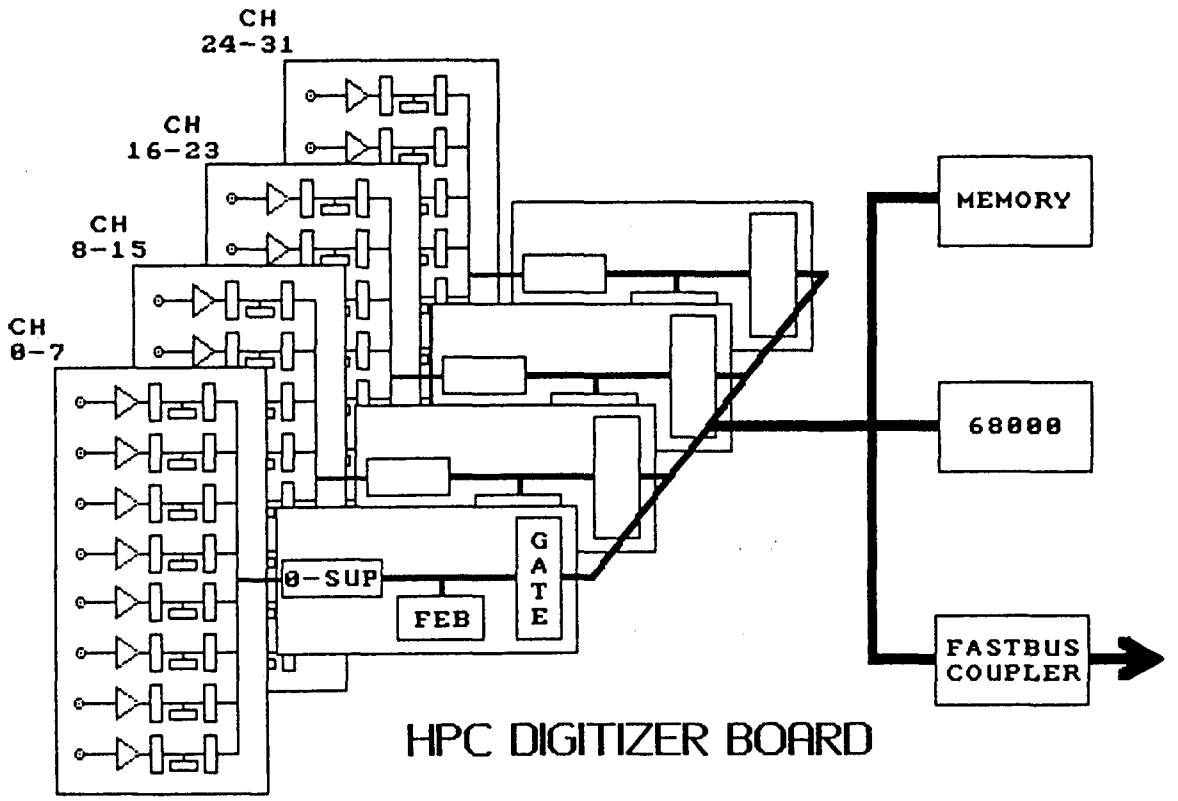

Figure 2.2. A block diagram of the digitizer module.

essor-created buffer in the RAM, is asynchronous with data acquisition. By that we mean that during the time when FASTBUS or the processor is reading an event out of the FEB an event trigger may cause the board to enter the LOAD state. At this point all FASTBUS and 68000 activity is suspended until the event is written into the FEB or is aborted via the QRST signal. When the event is disposed of, the board returns to the activity in progress before the event trigger arrived.

The remainder of this chapter describes each section of the board in more detail.

2.2 Input signals and Analog buffer

The analog waveforms to be digitized are input via two 34-pin connectors (see appendix B). Each signal consist of a differential input from a twisted pair cable. Termination of each line is 50 Ohms to ground. The signals are routed directly to

two interior signal planes of the board. Pairs of traces for each signal are positioned one above the other and where traces are adjacent to each other they alternate in polarity to minimize crosstalk. These two signal planes are sandwiched between the two ground planes to provide isolation from digital noise. The thickness of the board has been chosen to achieve a characteristic impedance close to that of the incoming cables.

The analog buffer converts this differential signal to a unipolar signal with an approximate gain of two. It is capable of slew rates in excess of 200 V/μs and has an input range of zero to two volts to match the zero to four volt range of the FADC (see section 2.3). The pedestal of the analog buffer is adjusted to a small positive value (approximately 35 mV).

The analog buffer is realized in a quad 30-pin hybrid package based on a LM6361 operational amplifier. Linearity measurements of the system indicate that linearity characteristics are dominated by the FADC, rather than the analog buffer.

2.3 Flash ADC and Cache Memory

The heart of the digitizer is a Thomson TS8328 flash ADC. This device samples an input signal up to 20 million times per second and converts it to an 8 bit digital value. The manufacturer provides inputs (tap points) for reference voltages at 1/4, 1/2, and 3/4 of full scale to permit the user to adjust the response curve to fit a particular application. The nominal input range is from zero to 3.5 volts with a minimum step size of 5 mV per count.

We have chosen to operate it in a bi-linear fashion in order to maximize the effective dynamic range available. In this mode, the first 64 ADC counts correspond to 5 mV steps and counts 64 through 255 correspond to about 19 mV each, placing full scale at 4.0 V. Consultations with the manufacturer assured us that the device could perform satisfactorily in this mode. If we define an effective dynamic range as the full scale value divided by the minimum step size we get a value of 800 to 1. Figure 2.3 shows the resulting response curve as measured on one channel. Measurements of linearity show that the absolute maximum conversion error is ± 1 count and that in an average sense the error is typically ± 0.25 counts.

Adjustments to the reference voltages are made on the front panel with the four trim pots. These set the nominal voltage for the entire board. Individual

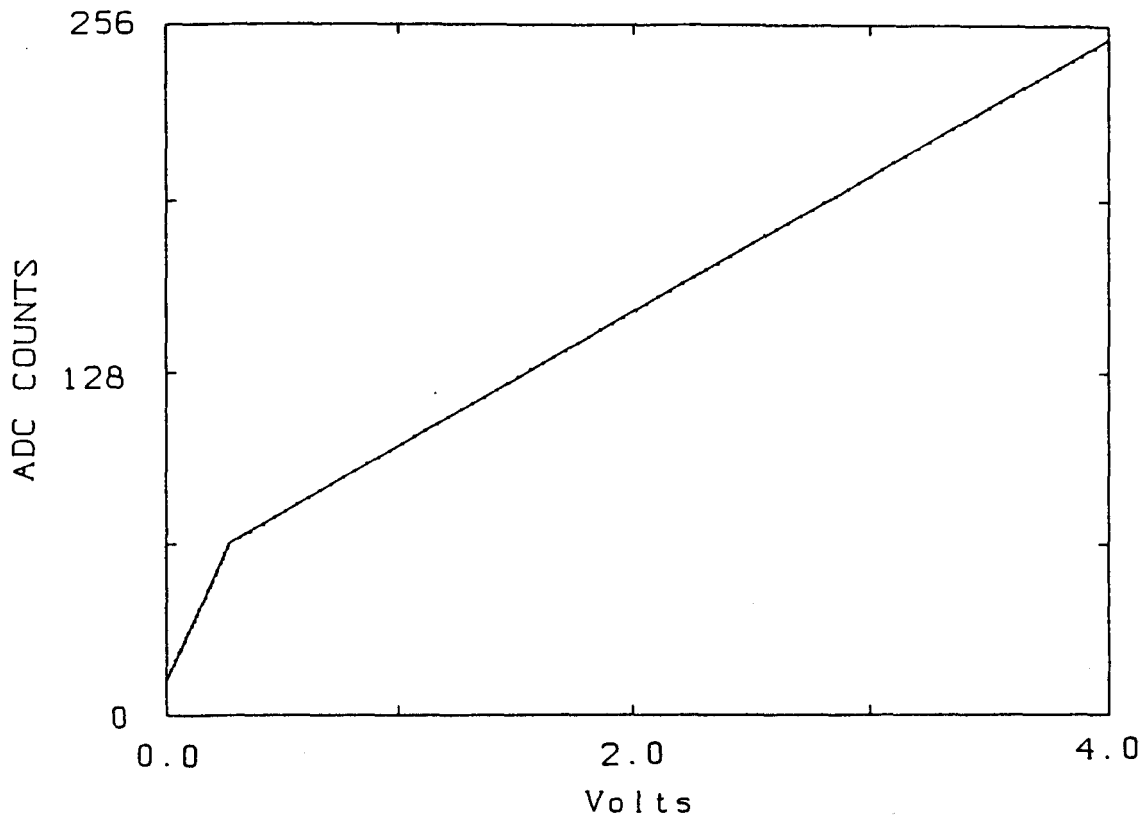

Figure 2.3. A measured bi-linear FADC response.

FADCs may be tuned by the selection of trim resistors. Each FADC has a small resistor (typically 5 Ohms and one third watt) on both its quarter and full scale reference voltages. Because of these trim resistors, one should note, the reference voltages displayed at the test points on the front panel are not exactly the same as that received by the FADCs.

If the user wishes to operate with the FADC in a linear mode over its entire range, only a minor modification to the reference voltage circuit is required. Note that without the sharp bend in the response curve only a small amount of current is drawn by the FADC at the corresponding tap point. This will make the adjustments with the trim resistors ineffective.

The digital output values from the FADC are placed in a fast (35 ns access time) memory chip for later readout by the zero suppression circuit. The data bus between the FADC and the cache memory is isolated from the zero suppression circuit by means of a digital gate. This allows the input channels to be isolated during digitization and yet be read out by a common zero suppression circuit.

The performance of the "flash and cache" system is sensitive to both the symmetry of the FADC clock and the phase of this clock with respect to the timing of the cache memory. Adjustments are provided for both of these parameters and will be required if the load clock frequency is changed. Consult the technical reference manual for more information.

2.4 Zero Suppression and Front-end Buffer

The zero suppression system provides for quick removal of data near the pedestal level and of certain types of noise pulses. This is done during the transfer of data from the cache memory to the FEB (DUMP state). Parameters for each channel are loaded into a threshold memory to control the zero suppression behavior. The FEB provides temporary storage for four complete events and is accessible by both the embedded microprocessor and the FASTBUS coupler.

Zero suppression begins after the digitization is completed. Data from each of the 32 channels are clocked from the cache memory to the front-end buffer via one of the four zero suppression circuits. The zero suppression circuit controls the address to which the data are transferred and suppression of a particular datum is accomplished by overwriting it in the buffer. After the zero suppression of a single channel is complete, a count of the number of surviving data words (including the count itself) is inserted in the low order byte of the first word in the buffer.

The decision to suppress a data value is based on an eight bit value stored in the threshold memory for each channel. This memory is accessible for reading and writing from FASTBUS. The threshold memory begins at 10000h in the FASTBUS data space and consists of 32 4-byte words of which only the lowest order byte is defined. The lower five bits form a threshold value, below which any data is suppressed. The upper three bits form a width. Any consecutive group of data words above threshold, called a cluster, which is shorter than this width is also suppressed. This feature allows high frequency noise to be removed at an early stage. One exception to the criteria described above is presample data. The first eight data values are assumed to be presamples taken before the event time and are always passed to the FEB. These are used to measure accurately the pedestal value just before the data of interest were collected.

The zero suppression system can be disabled by setting CSR#0⟨11⟩. This has the same effect on the behavior of the zero suppression system as setting all width and threshold values to zero, except that the threshold memory is not modified. This is useful for performing a pedestal check during data collection, as the user can quickly change and restore the system without detailed knowledge of the threshold data.

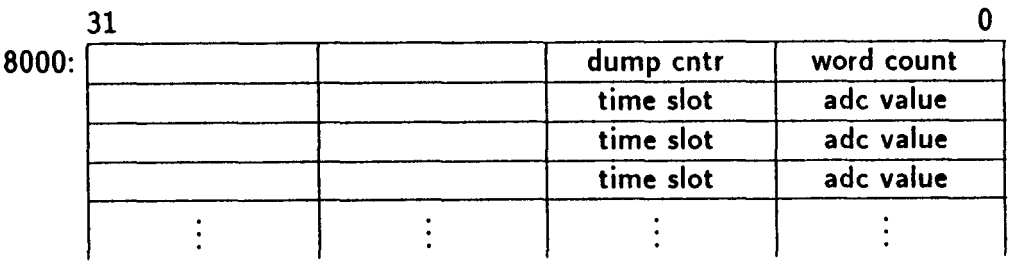

Figure 2.4. Format of data in the FEB

The FEB begins at 8000h in the FASTBUS data space. Figure 2.4 shows the format of the data for one channel for a single event. This buffer is 256 32-bit words long of which only the lower two bytes of each word are defined. The lowest order byte is the output of the ADC. The next higher order byte is the number of the time slot in which the ADC value was collected. This time information is necessary since the zero suppression may have removed some of the data samples. Further, the number of data words is stored at the beginning of the buffer along with a "good dump counter" (see below). Space for four separate events is provided to allow event buffering.

Thus a single event is stored in 32 separate locations in the FEB beginning at the addresses 8000h, 8400h, 8800h, ..., or 8100h, 8500h, 8900h, ..., etc., depending on which of the four event buffers is being used. The event buffer is selected at the time the data enters the FEB by obtaining the address bits 8 and 9 (300h) from the FASTBUS backplane (see section 2.7 regarding TR lines).

The good dump counter is a feature for crosschecking the data. At the end of each complete zero suppression cycle (i.e., not terminated early by QRST) the

counter is incremented. When the power is turned on or the FASTBUS is reset, this counter is set to zero. Thus when the data from a single event is collected by several boards, each channel of every board should display the same value for the good dump counter. If this is not the case, an error has occurred in the readout and data from different events have been incorrectly combined.

2.5 FASTBUS Coupler and CSRs

The FASTBUS Coupler is the interface between devices on the board and the FASTBUS backplane. FASTBUS is defined by the IEEE standard 960-1986, and all of the functions described here conform to this standard. Familiarity with this standard is assumed throughout the text. This module uses a modified version of a CERN–designed coupler (see Bibliography). The coupler assumes the FASTBUS segment uses negative–logic ECL signals.

If a board loses either its -5.2 V or +5.0 V power the ECL–TTL converter chips in the coupler could lock up the backplane for the entire segment. We have partially addressed this problem by placing these power signals to the converter chips on "islands" isolated from the power signals on the rest of the board by fuses. If the -5.2 V is missing on the entire board the +5.0 V fuse to the island will blow, preventing the lock up on the backplane. Unfortunately, it is not possible to have this work the other way; a missing +5.0 V will not cause the -5.2 V fuse to blow. If the +5.0 V power is missing to the entire board, the green "Power OK" front panel LED will go out. If the -5.2 V power is missing anywhere on the board, or the +5.0 V is missing on the island, the red "Power Fail" front panel LED will be lit. Therefore, if the segment backplane seems to be locked up, the user should check the status of the front panel LEDs.

This implementation responds to all forms of addressing (i.e., Geographic, Logical, and Broadcast). Both single word and block transfer modes are supported.

Control and Status Registers (CSRs) are used both to monitor the state of the FASTBUS device and to modify its operation. The FASTBUS standard requires only a small number of CSRs and describes many optional CSRs. Table 2.5 lists the control and status registers implemented in the Ames Waveform Digitizer. Access to most of the CSRs is provided through dataspace to allow the microprocessor to read and write to them.

Table 2.5. Control and Status Registers

CSR#	Data address	Description
0	18040h	Status and Control, Manufacturer's ID
1	18080h	Serial Number
3	(no access)	Logical Address
7	(no access)	Broadcast Class N selection
10h	18200	68000 Control
11h	18400	Trigger Accounting Number
(no access)	18800h	Word count
(no access)	1A000h	Special Functions

Table 2.6. CSR#0 Bit Definitions

Bit	Read Significance	Write Significance
00	Error Flag	Set Error Flag
01	Enabled (logical addressing)	Enable logical addressing
06	Front Panel LED	Set Front Panel LED
07	FEB 0 Has Data	(not used)
08	FEB 1 Has Data	(not used)
09	FEB 2 Has Data	(not used)
10	FEB 3 Has Data	(not used)
11	Zero-suppress override on	Set Zero-suppress override
16	LSB of Device Type	Clear Error Flag
17	Device Type	Disable Logical Addressing
18	Device Type	(not used)
19	MSB of Device Type	(not used)
20	LSB of Manufacturer's ID	(not used)
21	Manufacturer's ID	(not used)
22	Manufacturer's ID	Reset Front Panel LED
23	Manufacturer's ID	Reset FEB 0 Bit
24	Manufacturer's ID	Reset FEB 1 Bit
25	Manufacturer's ID	Reset FEB 2 Bit
26	Manufacturer's ID	Reset FEB 3 Bit
27	Manufacturer's ID	Clear Zero-suppress override
28	Manufacturer's ID	(not used)
29	Manufacturer's ID	(not used)
30	Manufacturer's ID	Reset (Same as Front Panel Button)
31	MSB of Manufacturer's ID	(not used)(Clear Data)

Table 2.6 defines the significance of the bits in CSR#0. Bits in this register follow a special convention, defined in the FASTBUS standard. If during a write operation a '1' is written to a particular bit this bit will be set to '1', but if a '0' is written to this bit, its state will not change. In order to clear a bit in this register the user must write a '1' to the bit location 16 positions to the left, in the upper half of a 32-bit word. With this system the user can, with a single 32-bit write operation, set, clear, or leave unchanged any bit in the 16-bit register. Moreover, a broadcast operation can set or clear a bit without affecting other bits in the register which may vary from module to module. The FASTBUS specification leaves undefined what happens if both the set and reset bits for the same register bit are set. On this module, this will result in the register bit being reset.

CSR#0⟨00⟩ indicates if an error has been detected on the board. If this bit is set then at least one of the following conditions must have occurred: One of the supply voltages has failed (fuse blown, or power supply failure); the 'watch dog' circuit has fired indicating the load or dump cycle has lasted too long; or the bit was set by the FASTBUS or the 68K with a write operation to this bit. This last possibility would be either for debugging purposes or to signal detection of a serious error by the microprocessor. The error bit is cleared by a reset operation or the appropriate write operation to CSR#0⟨16⟩.

CSR#0⟨06⟩ is connected to a front panel light. When this bit is set the light is lit. This is useful for the 68K to signal the operator or indicate the progress of a program. The host system might use it to indicate a particular board to the operator, perhaps for adjustment or replacement or visual feedback during system checkout. This bit is cleared by writing a "1" to CSR#0⟨22⟩.

CSR#0⟨07:10⟩ are called the 'Valid Data' bits and indicate when the zero suppression subsystem has found good data somewhere on the board for the corresponding event. Thus if the bit corresponding to the FEB event is clear after a trigger, there is no data to be read from this board. Clearly this is advantageous to a FASTBUS readout, but the most gain can be achieved when the microprocessor uses this information in responding to broadcast operations. These bits are either set or cleared during the DUMP cycle. Write operations can clear but not set these bits. They are cleared using the bits in CSR#0⟨23:26⟩.

CSR#0⟨11⟩ is the zero suppression override. When this bit is set, the zero suppression system behaves as if all thresholds and width were set zero. This allows

the user to do pedestal measurements without modifying the threshold memory. This bit is cleared by writing a "1" to CSR#0⟨27⟩.

CSR#0⟨30⟩ is used to cause a reset to the board. This function has the same effect on the module as pressing the front panel reset button, issuing a FASTBUS reset bus command, or executing the initial power up sequence. CAUTION: Since the microprocessor can access this register it can cause a reset, one effect of which is to stop the microprocessor itself.

CSR#0⟨16:31⟩ forms the device identifier. Each type of FASTBUS device is assigned a unique identifier. This allows the host computer system to identify the devices inserted in a FASTBUS crate. The device identifier for the Ames Waveform Digitizer is 01C0h.

CSR#1 serves only to display the serial number of the board. CSR#1⟨16:25⟩ forms the serial number. CSR#1⟨26:31⟩ return zero for convenience. The boards used in the DELPHI experiment use the range 0 through 700.

CSR#3 and CSR#7 store the device's logical address and the Broadcast Class N selections. These CSRs conform exactly to the IEEE standard.

CSR#10h is used to control the activities of the microprocessor. Table 2.7 shows the function of the bits in this register. Bits in this register are manipulated in the same fashion as in CSR#0.

Table 2.7. CSR#10h Bit Definitions

Bit	Read Significance	Write Significance
00	Microprocessor status	Enable microprocessor
05	Interrupt 5 pending	Queue interrupt 5
06	Interrupt 6 pending	Queue interrupt 6
07	Interrupt 7 pending	Queue interrupt 7
16	(not used)	Disable microprocessor
21	(not used)	Clear bit 5
22	(not used)	Clear bit 6
23	(not used)	Clear bit 7

CSR#10h⟨05:07⟩ are used to initiate interrupts to the microprocessor. These bits will remain set until the corresponding interrupt is acknowledged by the microprocessor or they are cleared by a write to CSR#10h⟨21:23⟩. CSR#10h⟨00⟩ is used to start the microprocessor. When this bit is set to '1' the microprocessor is

released from the reset state and will begin execution at the location in the reset vector. Any pending interrupts will be executed immediately in order of priority (7 is highest). When CSR#10h⟨00⟩ is cleared the microprocessor is forced into the reset state. For proper operation, the microprocessor must be held in reset for not less that 100 ms.

The location 18800h in data space is the word count register for block transfers. This register is 12 bits wide and counts down during a block transfer. When zero is reached the board will return a slave status of SS=2, indicating the end of the block transfer. The width of this register limits the length of block transfers to 4096 words.

The location 1A000h in data space (DS#1A000h) is the Special Functions Register. Table 2.8 lists the bits defined in this register and a description of their effects. The bits in this register are modified only by read and write operations on the data bus.

Table 2.8. DS#1A000h Special Functions

Bit	Read Significance	Write Significance
00	Data present	Data present
01	Device available	Device available
02	68K Busy	(not used)

DS#1A000h⟨00:01⟩ are used to control the response of the coupler to various T-pin scans. The T-pin scan is a type of broadcast operation where all boards in the crate respond with one bit determined by its physical location on the segment and these bits form a single 32-bit word on the FASTBUS backplane. If and only if DS#1A000h⟨00⟩ is set the coupler will respond to a Sparse Data Scan (Broadcast Case 3) by asserting its T-pin. In a similar fashion DS#1A000h⟨01⟩ will control the response to a Device Available Scan (Broadcast Case 3a).

During normal operation, the FASTBUS has priority over the microprocessor. That is, microprocessor activity is suspended whenever the board is addressed. Setting DS#1A000h⟨02⟩ will prevent the coupler from answering any FASTBUS requests for access, replying with a busy slave status (SS=1). Only the microprocessor or a reset (FASTBUS Bus Reset, Front panel reset, or power up) can clear this bit. This feature allows time critical calculations to proceed without delay, provided the FASTBUS master can accept the busy response.

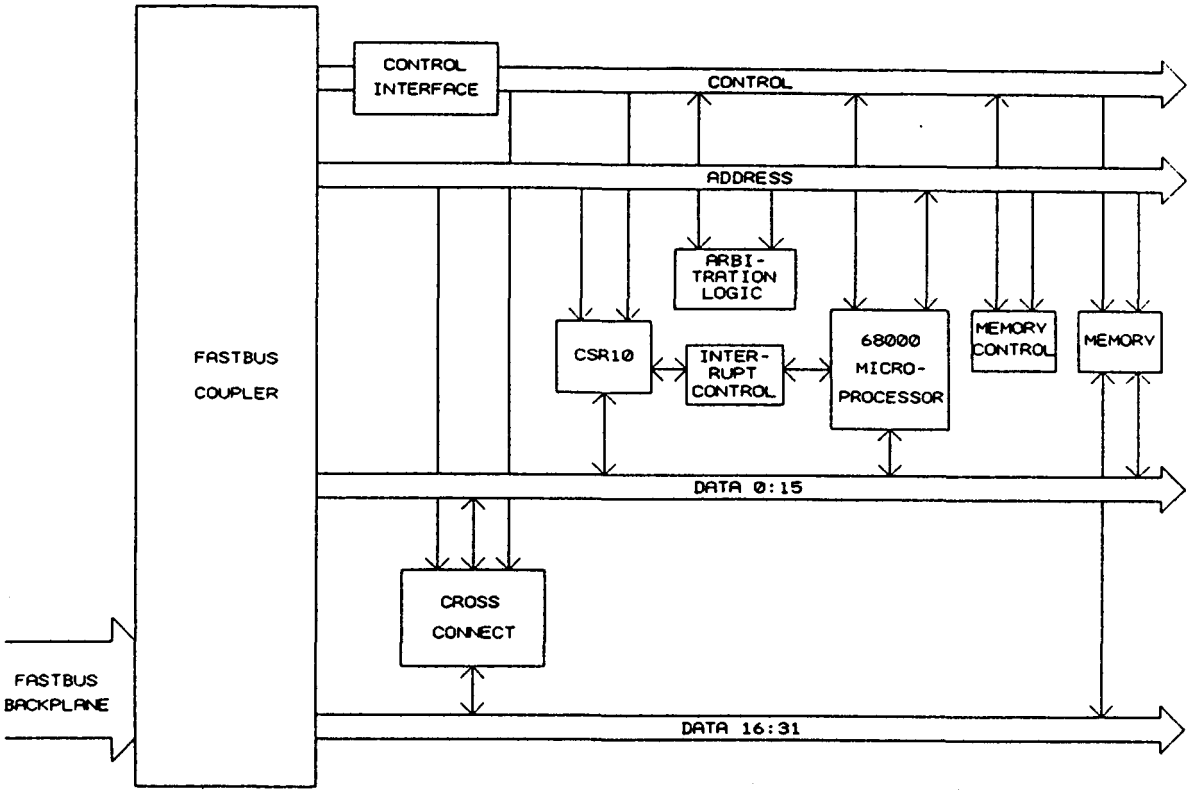

Figure 2.9. A block diagram of the microprocessor circuit.

Triggers have priority over both the FASTBUS and the microprocessor. If a trigger occurs during microprocessor activity the microprocessor is simply halted until the trigger is finished. If a trigger occurs during FASTBUS activity the coupler will reject any primary address cycles and delay any data cycles if they have already begun. Thus the FASTBUS master must be able to accept delays up to 500 μs or avoid access when triggers might occur.

2.6 Microprocessor

The board is equipped with a 16 MHz 68000 microprocessor and 128 Kbytes of RAM. Data and programs are downloaded into the RAM and results are retrieved from RAM via FASTBUS. In addition to RAM access, the processor has access to the threshold memories, FEBs, CSR numbers 0, 1, 10h and 11h and Data Space registers 18800h and 1A000h. Program execution is driven by interrupts set in CSR#10h. A block diagram is provided in figure 2.9.

A typical use of the processor would be to read data values from the FEBs, reformat them, and store them in a RAM buffer where they may be more compact and where data from all 32 channels can be placed in continuous RAM, permitting a single block transfer to read out all the data at once. A more sophisticated system might use the microprocessor to decide if the data in the FEB is consistent with a desired type of event. This result would then be collected with broadcast operations like the Sparse Data Scan (Case 3) from all boards at once.

Three interrupt levels are provided: 5, 6, and 7 (non-maskable), corresponding to bits CSR#10h⟨05:07⟩. By redirecting the interrupt vector before sending the interrupt, the user can use a given interrupt level for many purposes. At the end of the DUMP state, the board logic sends a priority 7 interrupt to the processor automatically. This is to allow the processor to begin event processing immediately. If the user does not want to do anything, an immediate return from the exception takes only a few microseconds (compared to 450 μs for the DUMP stage). If the processor is disabled, this interrupt is queued and has no effect until the processor is started. If the processor is enabled the interrupt at priority 7 will always occur, even if the processor is already at priority level seven.

If this interrupt at the end of DUMP is to be used, the triggering system will be required to hold off additional triggers, not only until the end of DUMP, but until at least some of the interrupt code is executed. This is to prevent the software from confusing data from different events. See chapter 4 regarding the programming environment.

Clearing bit CSR#10h⟨00⟩ disables the processor completely and causes a 68000 reset. Note that this bit must remain in the reset state for at least 100 milliseconds for the microprocessor to reset properly. When the module is first powered up, the processor is disabled because the RAM does not yet contain a program or interrupt vectors. The user must first download the RAM contents and then explicitly set the enable bit in CSR#10h before the processor will respond to interrupts. When the processor is first enabled, it reads an initial stack pointer and program counter from addresses \$0 and \$4 in the RAM and immediately begins executing the instruction pointed to by this program counter.

Note that FASTBUS access and microprocessor program execution are mutually exclusive since they share a common bus. In case of conflict, FASTBUS wins. (But the Special Functions Register provides a way for the processor to lock FAST-

BUS out during critical operations, see section 2.5). For a program to execute in the processor, the interrupt bit must be set and then the AS/AK lock broken. The processor will reset the interrupt bit as part of the interrupt acknowledge. If the processor is executing a program and FASTBUS wants access, the microprocessor support logic performs an arbitration cycle for the local board bus (not the FASTBUS segment) which takes about 500 ns in the worst case, and less than 250 ns in a typical case.

2.7 Clock and Trigger Inputs

In any experiment with more than one digitizer board there must be some method of synchronizing the activities of each board with its partners. The FASTBUS specification provides eight lines on the backplane for the user. These are called the TR lines (Terminated Restricted). Table 2.10 shows the allocation of the TR lines to various time critical functions. As for all signals on the FASTBUS backplane, these lines use negative-logic ECL signals ("1" = −1.4V, "0" = −0.7V). They are converted to TTL signals on the digitizer board.

Table 2.10. TR line Usage

TR0	WARNING/CLEAR	Prepare for data collection
TR1	BCO or TRIGGER	Begin data collection
TR2	EVTLS	LSB of event buffer
TR3	EVTMS	MSB of event buffer
TR4	QRST	Quick Reset
TR5	PSW	Pre-Sample Window
TR6	LOAD CLOCK	Digitization Frequency
TR7	DUMP CLOCK	Zero suppression Frequency

LOAD CLOCK provides the frequency for taking data samples. While the Ames Waveform Digitizer board has been optimized for 15 MHz, frequencies as high as 20 MHz should be feasible. Any significant deviation from 15 MHz may require changing some components and retuning of others.

DUMP CLOCK provides the frequency for the zero suppression (DUMP) stage of processing. 4.5 MHz is recommended.

The WARNING/CLEAR signal serves as the first indication to the board that a trigger will occur. When asserted, it forces the board into the load state. The microprocessor is halted and FASTBUS activity is ignored. WARNING/CLEAR is used in an asynchronous manner and can be as short as 20 ns. Typically 200 ns is used. The leading edge of this signal carries the significant information. There is no protection on the module against a new WARNING/CLEAR signal arriving while a previous event is being processed. It is presumed that the external logic will issue a QRST before sending a new WARNING/CLEAR.

PSW is typically the next signal asserted. If used, it must be asserted only between WARNING/CLEAR and BCO. This signal is used as a gate and provides a time period in which to take data before the actual trigger. Typically PSW is asserted for exactly eight clock cycles as these are the samples which receive special treatment by the zero suppression circuit. There is no requirement that any presample data be taken or that the samples taken must be from consecutive clock pulses. This signal must be synchronized to avoid transitions in the LOAD CLOCK to achieve consistent operation.

BCO also causes the system to begin collecting data except that collection will continue until all 255 samples are taken or the data collection is terminated with QRST. The leading edge of this signal must be synchronized with the LOAD CLOCK. The width is not critical, we use 100 ns.

The signal QRST will abort the data collection at any stage. All pointers will be reset and microprocessor execution and FASTBUS access will be allowed to resume. The assertion of this signal may overlap with WARNING/CLEAR and the last one asserted will define the state of the system. Typically the board is forced into a known state at the start of a data acquisition cycle by asserting both QRST and WARNING/CLEAR simultaneously and releasing QRST before WARNING/CLEAR.

TR lines 2 and 3 form the event pointer. When the zero suppression begins, these two lines are used to select the event buffer in the FEB. Proper control of these lines allows the host system to affect a four event deep buffer in the FEB. Note that these lines affect only the choice of FEB event during DUMP. The event selection in reading from the FEB is done by selecting the correct address.

2.8 Other features

In this section we describe a few additional features not covered in other sections. The user should be aware of these points when planning to use the Ames Waveform Digitizer.

Because the FASTBUS is locked out during the Load and Dump phases of data collection, it is possible that a failure of the hardware or one of the various triggering signals (see section 2.7) could prevent the coupler from answering a FASTBUS request. An example of this would be if the DUMP CLOCK (TR7) were to fail; the DUMP state would never end and the board could never be interrogated by the FASTBUS. To prevent this a 'watch dog' circuit is included which monitors the Load and Dump cycles. A QRST and error will occur if either cycle lasts too long. If the user chooses to change the clock frequencies, the timing in this circuit must be modified. Consult the technical reference manual for specific details.

A device for monitoring the supply voltages has been incorporated. If for any reason a supply voltage falls significantly (typically 20-50%) below its nominal value, this circuit will turn on its front panel LED and set the error flag in CSR#0. Monitoring this flag is important because the effect of some power failures is to corrupt the data without affecting the coupler.

There are two ways to adjust the reference voltages of the FADCs. Adjustments to the output of the reference voltages circuits are made on the front panel with four trim pots. This sets the nominal voltage for the entire board. The second method is by the selection of 'trim' resistors. Each FADC has a small resistor (typically 5 Ohms) on both its quarter and full scale reference voltage. This allows adjustments to individual FADCs. Typically FADCs from the same production batch are similar enough not to require individual adjustments, but this feature can be used if identical replacements are not available.

It must be noted that the voltage which appears on the front panel test points is not exactly that provided to the input to the FADCs. The trim resistors provide a small change in the reference voltage to the FADC (+6 mV/Ohm at quarter reference and -10 mV/Ohm at full scale). The front panel test points measure the output of the four voltage regulations circuits. A current limiting resistor is in series with each test point.

The range of adjustment of the reference voltages is limited to producing a bilinear response in the FADC. Changing a single passive component (specifically,

replacing a 0 ohm resistor by a 50 ohm resistor) will allow adjustment to a fully linear system. Other multi-linear adjustments are possible with changes in a small number of passive components.

The performance of the flash and cache system is sensitive to the symmetry of the clock used by the FADC, and also to its phase relative to the cache address strobe. Adjustments for both are provided and will require tuning if any change in the clock frequency is made.

The Ames Waveform Digitizer requires a non-standard 6.2 supply voltage to extend the dynamic range of the FADC. This supply is received on the FASTBUS pins B40 and B41. These pins are unassigned in the FASTBUS standard and thus the additional voltage will not conflict with standard FASTBUS modules. This voltage is used in two places. It goes through a diode drop to provide the analog supply voltage (ASUP) to the FADC. This is typically 5.4 V, the extra range above 5.0 V being needed to achieve a full scale input of 4.0 Volts. The other place the 6.2 V power is used is to provide well regulated reference voltages for the FADC tap points. In order to have a well regulated 4.0 V full scale reference, we need to provide the regulator with a voltage greater than 5.9 V.

Table 2.11. Front Panel LEDs

LED Name	Color	Meaning
Slave Connect	yellow	FASTBUS Access in progress
Load	red	Load Active
Dump	red	Dump Active
68K Busy	red	FASTBUS Locked out
Programmable	red	CSR#0⟨06⟩ set
Power OK	green	5 Volt Power present
Board Error	red	Error Flag set
Power Fail	red	Power failure detected
ZS Override	red	Zero Suppress disabled
Reset	red	Reset in progress

Table 2.12. Front Panel Testpoints

Signal Name	Significance
15V	Supply Voltage from Backplane
ASUP (5.4 V nom)	Supply Voltage from Backplane
5.0V	Supply Voltage from Backplane
-5.2V	Supply Voltage from Backplane
AGND	Analog Ground
R/4	FADC 1/4 Tap Point Reference Voltage
R/2	FADC 1/2 Tap Point Reference Voltage
3R/4	FADC 3/4 Tap Point Reference Voltage
Vref	FADC Full Scale Reference Voltage

3. Control Sequences

In this chapter we discuss the various ways of controlling the processes on the board. These fall into two categories: data acquisition procedures (i.e., control of the TR lines) and readout procedures (i.e., reading the data from the module via FASTBUS).

3.1 Data Acquisition Sequences

In this section we present examples of how this module can be controlled via the TR lines. Examples are drawn from specific experiments. The trigger and data acquisition needs of the DELPHI experiment imposed a number of constraints on the design of the module, but we have kept as much flexibility as possible. There are several ways in which the control signals on the TR lines can be modified according to the needs of a particular user. Note that the figures show the TR lines as they appear on the segment backplane, i.e., they are negative-logic signals.

3.1.1 Standard DELPHI Sequence

At a colliding beam storage ring, such as LEP, the fact that events can only occur when the beams cross, about every 22 μs, gives the user advanced knowledge of when an event can occur. Accordingly, we send the WARNING/CLEAR signal a few microseconds before the beam crossover and take eight presamples of data in the window between WARNING/CLEAR and BCO. At the beam crossover time, digitization begins and, if the trigger electronics do not detect anything of interest, a QRST (quick reset) aborts the digitizing. Figure 3.1 shows timing diagrams for this sequence. The position of the presample window (PSW) in the interval between WARNING/CLEAR and BCO is variable to allow us to choose a time when there is the least noise from external sources, such as other detectors.

3.1.2 Fast Readout Method 1

Often the user does not know in advance when an event will occur. This is the case, for example, in external particle beams and cosmic ray experiments. In this case it is necessary to begin digitization as soon as possible after the event. One way to do that is by using the WARNING/CLEAR signal for the event trigger and

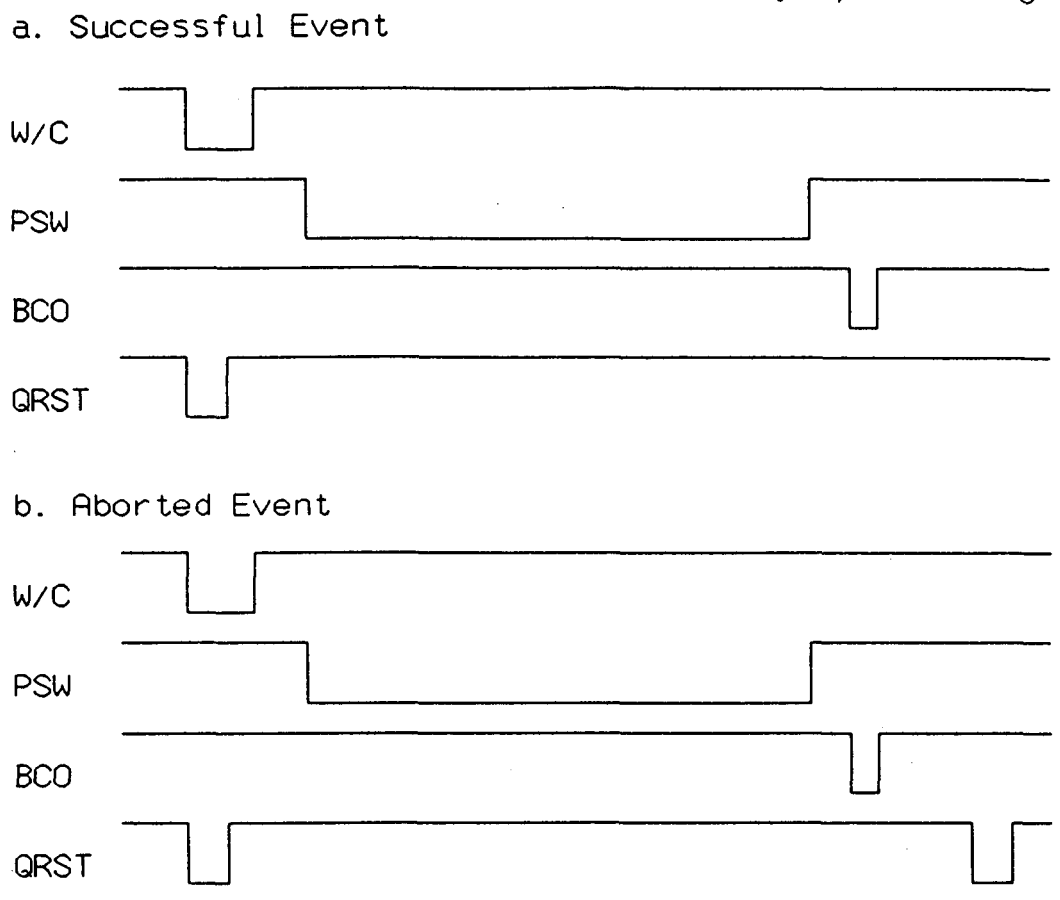

Figure 3.1. Timing signals for the standard sequence.

not using the presample window at all. The first eight samples will then contain valid data rather than a baseline measurement, but the zero suppression circuit will continue to pass the values to the FEB whether they are above threshold or not. The user will have to give these eight samples special treatment, either in the on-board microprocessor or elsewhere, to identify valid data. In this scheme, BCO follows immediately after the W/C signal. The timing diagram is shown in figure 3.2. This scheme preserves the multi-event buffering capability of the module but loses the presample capability. Digitization can begin within about 100 ns of the event.

3.1.3 Fast Readout Method 2

An alternative fast sequence preserves the presample, but at the cost of losing the multi-event buffering. In this scheme the control hardware asserts WARNING/-CLEAR and takes the presample data as soon as it is ready to accept an event. It

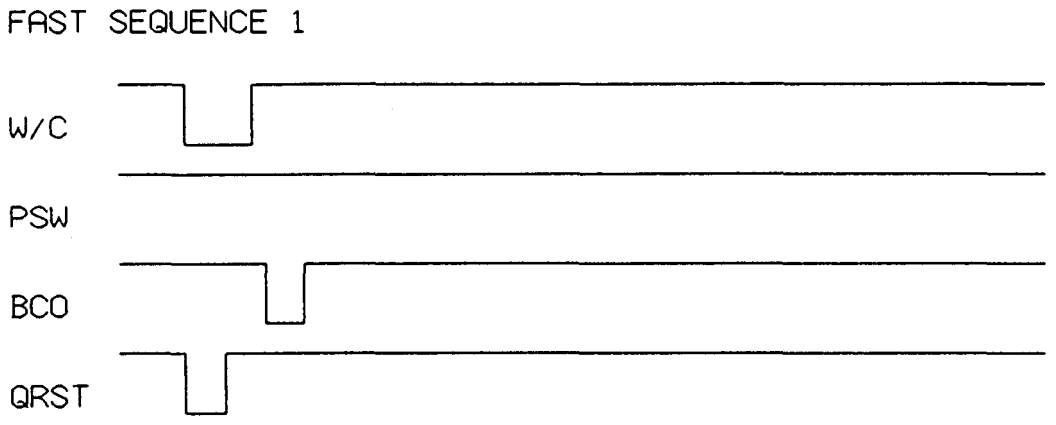

Figure 3.2. The first fast trigger sequence.

waits with the board held in the LOAD state until an event occurs and then triggers data acquisition via the BCO signal. Since the board spends most of its time in the LOAD state, the user must be sure the microprocessor has finished with an event and the data have been read out via FASTBUS before enabling WARNING/CLEAR for the next event. Note that there is no fixed timing between the presample and the data in this scheme. The timing diagram for this method is shown in figure 3.3.

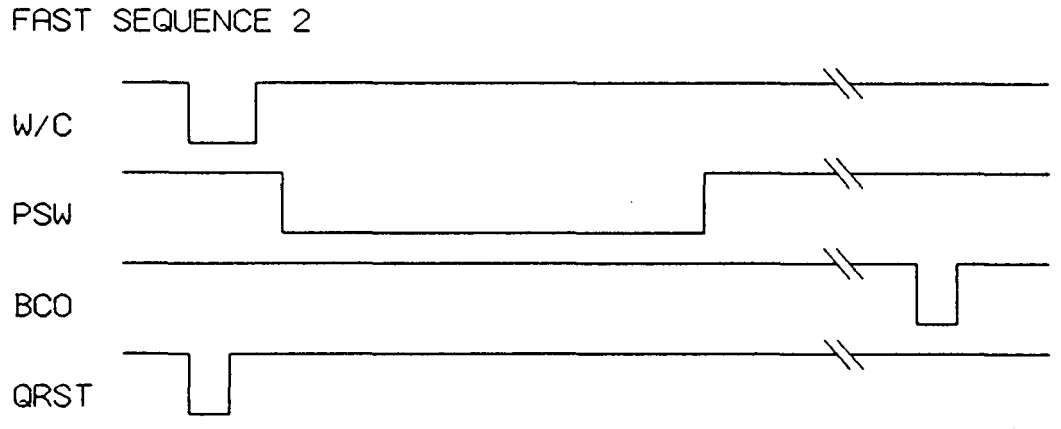

Figure 3.3. The second fast trigger sequence.

3.2 Data Readout Sequences

This section presents two ways of reading the data from the module. The first is to read the FEB directly, and the second is to read a board event buffer (BEB) which has been built in RAM by the microprocessor. BEB readout is the preferred method because it results in a smaller data volume and can be read with a single block transfer instead of the separate block transfers needed for each channel in the FEB readout.

3.2.1 FEB Readout

Normally, only the BEB will be read out, but there may be circumstances under which the user wants to read out the FEB directly.

The FEB has room for four events and the FASTBUS master must specify which of the four it wants. It does this by selecting the proper address in the FASTBUS data space for the board. As described in section 2.4, the FEBs begin at address 8000h with 400h words allocated per channel. The format for the data was also presented in section 2.4. Each channel has four events of up to 100h words each. For FEB event number EVENTNO (= 0, 1, 2, or 3) the sequence is:

```
For CHANNEL = 0 to 31 do
   FASTBUS data read at address IFEB(CHANNEL,EVENTNO)
   Mask to get an 8 bit WORD_COUNT
   If (WORD_COUNT > 9) then
      For POINTER = 0 to WORD_COUNT-1 do
         FASTBUS data read at IFEB(CHANNEL,EVENTNO)+POINTER
         Mask to get low order 16 bits
         Store as next entry in a 16 bit wide buffer
      Enddo
   Endif
Enddo
```

Here `IFEB(CHANNEL,EVENTNO)` is the starting address for event `EVENTNO` in channel `CHANNEL` and can be calculated as:

```
IFEB(CHANNEL,EVENTNO) = 8000h + 400h*CHANNEL + 100h*EVENTNO
```

The test for more than nine data words (eight presamples plus the wordcount) assumes that the presample data is of interest only if there is additional data in

that channel. If valid data can occur in the first eight time slots, this test should be removed.

When all of the data for an event have been read out, the appropriate bit in the range CSR#0(07:10) should be reset.

3.2.2 BEB Readout

In the usual mode of operation, the microprocessor will build an event buffer in RAM containing data from all 32 channels. This is the board event buffer, or BEB. The advantages of this are that it allows the user to put the data in a more compact format, to utilize the full 32 bit width of the FASTBUS data bus (versus only 16 bits for direct readout of the FEB), to read all the data from the board with a single block transfer, and to put the data in a format more convenient for later processing. By having a processor on each digitizer module, this reformatting task proceeds in parallel on many boards, freeing later processors, which see the data serially, from having to spend a lot of time on routine matters. In the same pass through the data, the user can also perform simple calculations (e.g., an energy sum) which may be useful for triggering.

This readout mode is initiated by the FASTBUS master which is in control of the data acquisition telling the digitizer module to format one of the events in the FEB and leave it in the BEB. The master then waits for the reformatting program to finish and then reads out the BEB. In the DELPHI experiment we do this as follows.

1. Write a trigger accounting number to CSR#11h. This is an eight bit "event number", the low order two bits of which specify which of the four event regions in the FEB to read out.
2. Set CSR#10h(06) to start the format program (we use interrupt 6 for this).
3. Break the AS/AK lock and wait until the format program is done (about 7 ms in our case). Test to see if it is done by either
   a. performing a T-pin scan (broadcast class 3a) to see if the "free for use" flag is set, or
   b. reading a word reserved in memory (a "communications flag" register) and testing if bit 0 is set. If it is, formatting has been completed. This communications flag register is simply a location in RAM which has been reserved for this purpose. It is not a special hardware register.

Note that the microprocessor program must explicitly set these two flags as it finishes the formatting.

4. See if data are present. This is done by either
   a. performing a T-pin scan (broadcast class 3) to see if the "data present" flag is set, or
   b. reading the communications flag register in memory and testing if bit 1 is set. If it is set, there are data present. If the word has already been read in step 3, it is unnecessary to read it again.

   Again, the processor must set these two flags as it finishes the processing if it has found valid data on the board.
5. Do a block transfer from the BEB. Good programming practice makes the first word of the BEB a word count. The module supports block transfers of at most 4095 words, so if the BEB contains more than 4095 words, two or more block transfers will be necessary. Users must test for this and handle it explicitly, the module will not do it for them.
6. When the transfer is complete, users should clear all bits in the communications flag register and clear the appropriate bit in CSR#0(07:10).

A sample program to do this sequence using the standard FORTRAN FASTBUS routines is given in Appendix C. A description and sample programs showing how the microprocessor sets up the BEB are given in chapter 4.

4. Programming the Microprocessor

In this chapter we discuss the hardware and suggested software environment for programming the microprocessor. We assume the readers are already familiar with the module description presented in the preceeding chapters and that they are familiar with assembly language programming on the Motorola 68000. The Bibliography contains several books on 68000 assembly language programming. The chapter ends with the presentation and discussion of sample programs which demonstrate the approach we have used.

4.1 Programming Environment

This section presents the hardware and software environments in which the user must program the 68000. We describe the memory map and the allowed addressing modes for each of its areas. We then describe the software architecture we have adopted for the DELPHI experiment.

4.1.1 Hardware

From the programmer's perspective, the microprocessor is quite straightforward. The only peripherals it can access are memory and memory-mapped registers. All user communication is by direct memory access (DMA) using FASTBUS. In effect, the microprocessor can only leave messages in memory which FASTBUS can pick up. Program execution is controlled by three autovectored interrupt levels (5, 6, and 7) in CSR#10h⟨05:07⟩ and an enable bit at CSR#10h⟨00⟩. The full 68000 instruction set is supported but none of the extensions to 68010, 68020, or 68030 processors are available. The complete memory map is shown in table 4.1.

Because FASTBUS uses word addressing and the 68000 uses byte addressing there must be two different addresses for each memory location. FASTBUS accesses data in 32 bit words only, while the 68000 can access data in byte, word, or longword units. The FASTBUS address is simply the longword aligned 68000 address shifted two bits to the right. Throughout this document we use the convention "$000" to indicate a microprocessor address and "000h" to indicate a FASTBUS address, where "000" represents a hexadecimal value. Note that in the addressing convention used by the 68000 the high order bytes come at the low address end of the word.

Table 4.1 Memory Map

68000 Address	FASTBUS Address	Usage
$00000-$1FFFF	0000h-07FFFh	RAM
$20000-$3FFFF	8000h-0FFFFh	Front-end Buffers
$40000-$4007F	10000h-1001Fh	Threshold Memories
$60100	18040h	CSR#0
$60200	18080h	CSR#1
$60800	18200h	CSR#10h
$61000	18400h	CSR#11h
$62000	18800h	Block Transfer Count
$68000	1A000h	Special Functions Register

For example, in the 32 bit word at byte addresses $800 to $803, the high order (most significant) byte is at address $800 and the low order (least significant) is at $803. FASTBUS would access the entire word by going to address 200h. The data from 68000 byte $800 would be in bits 24-31 and from byte $803 would be in bits 0-7. This differs from the byte ordering used on some other machines, most notably VAXs. Table 4.2 illustrates this difference for 32 bit words, there is a corresponding effect with 16 bit words.

Table 4.2 Byte Ordering for 32 Bit Words

	H.O. Byte			L.O. Byte	H.O. Byte			L.O. Byte
68000:	00	01	02	03	04	05	06	07
VAX:	03	02	01	00	07	06	05	04

When the board is powered up or reset, the microprocessor is held in the RESET state to allow the user to load the contents of RAM. Since, there is no ROM or bootstrap program on the module, the user must explicitly load everything, including exception vectors, program code, and data. Of particular importance are the vectors listed in table 4.3, but many of the other exception vectors (e.g., zero divide and illegal instruction) work also. Users not familiar with these should consult one of the manuals on 68000 programming listed in the bibliography.

Because event triggers cause the microprocessor to suspend its activity for as much as 500 μs, the actual elapsed time for program execution depends on the

trigger rate and the ratio of triggers which are not aborted with a QRST. Each trigger which makes it to the end of the dump cycle (500 μs) causes an automatic level seven interrupt which will be processed immediately, even if the processor is currently handling a previous level seven interrupt.

Table 4.3 Exception Vectors

Address (Hex)	Vector
$000	Initial System Stack Pointer (SSP)
$004	Initial Program Counter (PC)
$074	Level 5 Interrupt Autovector
$078	Level 6 Interrupt Autovector
$07C	Level 7 Interrupt Autovector

Microprocessor access to the areas listed in table 4.1 is allowed as follows:

- RAM: Byte, word, or longword access is permitted using any valid 68000 addressing mode.
- Front-end Buffers: Since only the low order (i.e., higher address) 16 bits are valid, word operations are the most natural. Under normal operation, the microprocessor does not write into the FEB, but if it becomes necessary to write to the FEB, word operations should be used. Longword reads and writes will work but are inefficient.
- Threshold Memories: The low order (higher address) byte is all that is significant. It can be accessed via byte, word, or longword operations, but longword access is inefficient.
- CSRs: Access to CSRs 0, 1, 10h, and 11h is via memory mapped addresses as shown in table 4.1. Byte write operations to these addresses should be avoided because of possible corruption to bytes not being written to. Write operations must be word or longword, read operations can be byte, word, or longword.
- Block Transfer Count: The low order twelve bits are all that are significant. Word operations should be used for both reading and writing.
- Special Functions Register: The special functions register is only three bits wide and can be read or written to by byte, word, or longword operations.

4.1.2 Software

Because the microprocessor is in a minimal configuration, the simplest way to program it is in assembly language. We use the Motorola Cross Macro Assembler running on a VAX computer to produce S-code which is downloaded into RAM via FASTBUS. S-code is a method used by Motorola to represent binary data as an ASCII text file for easier handling. A FORTRAN subroutine for loading the RAM with the contents of an S-code file is presented in Appendix D.

The use of absolute addresses gives the programmer the required control over the entire address space, including, for example, downloading event data via FASTBUS into the FEB to be treated like real data. The use of higher level languages is not excluded, but we have no experience to report on using them with this module.

Our use of RAM can be divided into four regions. Addresses $00 to $FF are reserved by the 68000 for exception vectors. These need to be initialized properly as in our program examples. We do not have "user interrupt vectors", so the corresponding address space ($100 to $3FF) can be used for other purposes. We use the space starting at $100 for a "communications vector", which is an area of RAM reserved for passing information between the microprocessor and the host computer. The length and usage of this vector is entirely a matter of programming convention, there is no special hardware involved. Table 4.4 shows how we have defined the communications vector for the DELPHI experiment.

Table 4.4 DELPHI Communications Vector

68000 Address	FASTBUS Address	Contents
$100	40h	BEB Start
$104	41h	First Hardware Channel
$108	42h	Program Version
$10C	43h	Load Time
$110	44h	Load Date
$114	45h	Program Selector
$118	46h	Bit Flags
$11C	47h	Board Serial Number

Just above the communications vector we have the system stack. This is an area reserved for the system to save information during subroutine calls and

exception processing. It grows downward (i.e., toward lower numbered addresses) from an address specified at startup by the Initial Stack Pointer at address $00. We use $800 for the top of the system stack. Because program execution is controlled by interrupts, the processor is always in the Supervisor State. Thus there is no need for a User Stack Pointer.

The RAM from the top of the system stack ($800 for DELPHI) to $1FFFF is available for program code and data.

Throughout this discussion we have assumed the existence of a program running on the FASTBUS host computer which can send the FASTBUS data necessary to control the module. The program we have developed to do this is called ARTEMIS and it has been constructed around the needs of this specific module. This program and documentation are available to interested users.

4.2 Program Development

Because there is no terminal access to the processor, the best way to develop and debug code is by simulating the module's environment on a 68000-based computer with an interactive debugger. One such system that we have used is the debugger, which is commercially available. The program changes needed to run on the GPM are quite straightforward and the ability to set breakpoints, single step, and set and examine registers and memory gives a powerful debugging tool.

If the program bug appears only when the program is running on the digitizer module (usually because it is sensitive to timing), the most effective tool we have found is to reserve an area in RAM as a scratch pad and then to insert MOVE or MOVEM instructions to copy the contents of registers or memory into the scratch pad at key points. Preceeding the MOVE with a MOVE of an easily identified value, such as 'ABC1', 'ABC2', etc., makes it easier to associate data with MOVE statements. This is analogous to the days before symbolic debuggers where the hapless programmer had to pepper the program with print statements to debug. In this case the programmer "prints" to an area of RAM. After the program has terminated, the user can examine the RAM contents via FASTBUS. The user can also insert STOP statements to force execution to end at a certain point. We have also found that a careful examination of the stack contents is often helpful in tracing down bugs.

4.3 Program Examples

In this section we present and discuss some sample programs to illustrate the software environment we have developed.

Listing 4.1 shows the first of these programs, which we call TOY1. This program uses interrupts 5 and 6 to control flashing of the front panel LED. Interrupt 5 starts the flashing and interrupt 6 ends it. The communication vector is a single word, called SELP, which can be either 0 or 1. A value of 0 causes fast (0.5 second) flashes and a value of 1 causes slow (1.0 second) flashes. Even though the task of this program is trivial, it illustrates all the major concepts needed to program the processor. This example should be clearly understood.

The listing begins with a series of EQUates which provide mnemonics to increase program readability and flexibility. Beginning at the line 'VECTOR TABLES', we load the values of the first 64 interrupt vectors. All but five of these point to a general (and primitive) error handler called BN. After these interrupt vectors comes the single word communication vector at address SELP.

When the processor is first enabled, it jumps via the Initial Program Counter to the routine at PROG0. This routine merely stops the processor until an interrupt is received. PROG5 and PROG6 are the autovector interrupt handlers and illustrate an efficient way to use the program selector to switch between alternative programs using the same interrupt. For this example, PROG7 is a trivial 'do nothing' interrupt handler.

When an interrupt 5 is received, program control proceeds via PGMTBL5 to either FSTBLNK or SLOBLNK depending on the value in SELP at the time the interrupt is processed. The light continues to flash until an interrupt 6 clears register D0, which causes a termination of the interrupt 5 by an RTS to PROG5 followed by an RTE.

Finally, listing 4.2 presents an abridged listing of the microprocessor program that we use both for testing the modules and for data acquisition in DELPHI. We have deleted many of the tests which are similar in structure and have kept only the simplest of the tests and our event reformatting routine.

As in TOY1, the program begins with a list of EQUates followed by the table of interrupt vectors. We use the same error trap routine (BN) as TOY1, but the communication vector is now twelve words long. PROG0 jumps to an initialization routine before executing a STOP instruction. PROG5 and PROG6 are identical to

```
$ CREATE TOY1.MAR
* TOY1
*
* Table of Constants and Addresses
*   Note that some Addresses are shown as the Fastbus address
*   with an explicit shift to convert to microprocessor address
ISP             EQU     $800            Initial Stack Pointer
AUTOVECTOR      EQU     $64             Start of AutoVector table
LED_ON          EQU     $40             Bit pattern to turn LED on
LED_OFF         EQU     $400000         Bit pattern to turn LED off
*
CSR0            EQU     $18040<<2       CSR#0  via data space
*
* Program
*

*  VECTOR TABLES
        ORG     0
        DC.L    ISP             Initial Stack Pointer
        DC.L    PROG0           Initial Program Counter
*
*  MISC BAD EVENT VECTORS
        DC.L    BN,BN,BN,BN,BN,BN,BN,BN         BN = "Bad News"
        DC.L    BN,BN,BN,BN,BN,BN,BN,BN
        DC.L    BN,BN,BN,BN,BN,BN,BN
*
*  AUTO VECTOR INTERRUPTS
        ORG     AUTOVECTOR      Fill in autovector table
        DC.L    BN,BN,BN,BN
        DC.L    PROG5
        DC.L    PROG6
        DC.L    PROG7
*
*  TRAPS
        DC.L    BN,BN,BN,BN,BN,BN,BN,BN
        DC.L    BN,BN,BN,BN,BN,BN,BN,BN
*
*  OTHER BAD STUFF
        DC.L    BN,BN,BN,BN,BN,BN,BN,BN
        DC.L    BN,BN,BN,BN,BN,BN,BN,BN
*
*  COMMUNICATIONS VECTOR
*    PROGRAM OUTPUT; DATA TO FASTBUS
    ORG    $40<<2
SELP     DS.L    1        Program Selector for interrupts
*
* Program
        ORG ISP
PROG0   STOP    #$2000          Set interrupt mask to 0
        JMP     PROG0           Tight loop (STOP continues after interrupt)
*
*  AutoVector programs
PROG5   MOVE.L  SELP,D0         Get program selector
        ASL.L   #2,D0           Shift to form address offset
        MOVE.L  #PGMTBL5,A0     Put address of start of jump table in A0
        ADD.L   D0,A0           Add offset from program selector
        MOVE.L  (A0),A0         Put the value in the jump table in A0
        JSR     (A0)            Jump to address pointed to by jump table
        RTE
PROG6   MOVE.L  SELP,D0         See comments for PROG5
        ASL.L   #2,D0
        MOVE.L  #PGMTBL6,A0
        ADD.L   D0,A0
        MOVE.L  (A0),A0
        JSR     (A0)
        RTE
PROG7   RTE                     Do nothing
*
*  SOMETHING BAD HAPPENED (BadNews)
BN      STOP    #$2700
        DS.W    1               Force longword alignment
*
*  GENERAL DATA
*
* Program table for interrupt #5
        ORG     $900
PGMTBL5 DC.L    FSTBLNK,SLOBLNK
*
* Program table for interrupt #6
PGMTBL6 DC.L    NOBLNK,NOBLNK

FSTBLNK MOVE.L  #500,D0         Use 500 ms cycles
        JMP     BLINK

SLOBLNK MOVE.L  #1000,D0        Use 1000 ms cycles
        JMP     BLINK

BLINK   MOVE.L  #LED_ON,CSR0    Turn on LED, then take a nap

        TST.W   D0              If D0 is 0, exit loop
        BEQ     BYEBYE
        MOVE.L  D0,D1           D1 counts the number of 1 ms loops
ZZON    MOVE.L  #1600,D7        Set count for inner (1 ms) loop
LOOP1   DBRA    D7,LOOP1        This loop takes 1 ms at 16 MHz
        DBRA    D1,ZZON

        MOVE.L  #LED_OFF,CSR0   Naptime is over, turn off LED

        TST.W   D0              If D0 is 0, exit loop
        BEQ     BYEBYE
        MOVE.L  D0,D1           Reload D1, then take another nap
ZZOFF   MOVE.L #1600,D7
LOOP2   DBRA    D7,LOOP2
        DBRA    D1,ZZOFF

        JMP     BLINK
BYEBYE  MOVE.L  #LED_OFF,CSR0   Turn out the lights,...
        RTS

NOBLNK  MOVE.L  #0,D0
        RTS

        END
$ EOD
$ R MAIN
N
TOY1.MAR
TOY1.LST
TOY1.SEX
$ PUR TOY1.*
$ DEL TOY1.MAR;0
```

TOY1, but PROG7 now increments a counter and tests a bit to see if it should return immediately or execute PROG6. The program jump tables are called PGMTBLI and PGMTBL and function just like PGMTBL5 and PGMTBL6 in TOY1. We have adopted the convention that interrupt 5 is used for initialization routines and interrupt 6 is for event processing. Starting at address BCN we have a lookup table to rearrange the hardware connections coming from the detector module in DELPHI. This is for the convenience of the offline software and is an artifact of our particular application. We include it as an example of the flexibility possible because of the on-board microprocessor. Following the table there are a few utility routines and the startup code that is part of PROG0.

The program starting at SCAN0 is the first and simplest of many test routines which can be chosen as part of our checkout and calibration procedure. These routines take multiple events and add the FEB contents to existing values in the BEB. The host processor reads the BEB when it has finished sending triggers. There are many other such tests, but we have deleted them from the listing for brevity.

The final program begins at FM_BEB. It is the data reformatter we will use for physics events in DELPHI. It scans the FEBs for data, ignoring those with only presample values, and builds a BEB in a standard DELPHI format. If valid data is found on this module the routine sets the bits in the flags word of the communication vector and the bit in the Special Functions Register that will cause the module to respond to a Sparse Data Scan.

These two examples illustrate the use of the microprocessor in the DELPHI experiment. The general nature of this module and the flexibility given by the microprocessor make it adaptable for a wide range of uses.

```
* ARTEMIS
*
* Table of Constants and Addresses
*   Note that some Addresses are shown as the Fastbus address
*   with an explicit shift to convert to microprocessor address
ISP             EQU     $800            Initial Stack Pointer
AUTOVECTOR      EQU     $64             Start of AutoVector table
*
CSR0            EQU     $18040<<2       CSR#0  via data space
ERRFLG_C0       EQU     $0001           Error Flag W/R
LOGADR_C0       EQU     $0002           Logical addressing W/R
LED_C0          EQU     $0040           Front Panel LED W/R
FEB0_C0         EQU     $0080           Valid data in FEB 0  R/W
FEB1_C0         EQU     $0100           Valid data in FEB 1  R/W
FEB2_C0         EQU     $0200           Valid data in FEB 2  R/W
FEB3_C0         EQU     $0400           Valid data in FEB 3  R/W
*
CSR1            EQU     $18080<<2       CSR#1   via data space
CSR10           EQU     $18200<<2       CSR#10  via data space
CSR11           EQU     $18400<<2       CSR#11  via data space
EOB_REG         EQU     $18800<<2       Word count for Block Transfer
F_REG           EQU     $1A000<<2       Flags reg
FEB             EQU     $08000<<2       Start of FEB Space
EVENT           EQU     $00100<<2       Advance one event in FEB
CHANNEL         EQU     $00400<<2       Advance one channel
THRES           EQU     $10000<<2       Start of Treshold Data
BEB             EQU     $04000<<2       Start of Board Event Buffer
*
*  Data types
PULSE_DT        EQU     $0920           Pulse test data(5)
QPULSE_DT       EQU     $0921           Q Pulse test data(6)
DTEST_DT        EQU     $0922           Delta code test(7)
SAW_DT          EQU     $0923           Sawtest data(4)
SCAN_DT         EQU     $0924           Multi-event scan data(1)
PED_DT          EQU     $0925           Pedestal data(3)
*
* Program
*
*  VECTOR TABLES
        ORG     0
        DC.L    ISP             Initial Stack Pointer
        DC.L    PROG0           Initial Program Counter
*
*  MISC BAD EVENT VECTORS
        DC.L    BN,BN,BN,BN,BN,BN,BN,BN
        DC.L    BN,BN,BN,BN,BN,BN,BN,BN
        DC.L    BN,BN,BN,BN,BN,BN,BN
*
*  AUTO VECTOR INTERRUPTS
        ORG     AUTOVECTOR      Fill in autovector table
        DC.L    BN,BN,BN,BN
        DC.L    PROG5
        DC.L    PROG6
        DC.L    PROG7
*
*  TRAPS
        DC.L    BN,BN,BN,BN,BN,BN,BN,BN
        DC.L    BN,BN,BN,BN,BN,BN,BN,BN
*
*  OTHER BAD STUFF
        DC.L    BN,BN,BN,BN,BN,BN,BN,BN
        DC.L    BN,BN,BN,BN,BN,BN,BN,BN
*
*  COMMUNICATIONS REGISTERS
*    PROGRAM OUTPUT; DATA TO FASTBUS
    ORG     $40<<2
    DC.L    BEB>>2          Location of the BEB in the FASTBUS address space
MODU    DS.L    1           Module number
PAD1    DS.L    1           Starting Pad number
        DS.B    4           Program/BEB Version numbers
        DS.L    1           Load Time (hhmm-decimal)
        DS.L    1           Load Date (yymmdd-decimal)
SELP    DS.L    1           Program Selector for int 5 program
FLAGS   DC.L    0           Free/Data present Flag
BDSER   DS.L    1           Board Serial Number
IMASK   DC.L    0           Mask indicating interrupts received
COUNT   DC.W    0,0           Count of primes from sieve program
DCOUNT  DC.L    0
*
* Program
        ORG ISP
PROG0   JSR     START
PROG0X  STOP    #$2000          Set interrupt mask to 0
        JMP     PROG0X          Tight loop (STOP continues after interrupt)
*
*  AutoVector programs
PROG5   MOVE.L  SELP,D0
        ASL.L   #2,D0
        MOVE.L  #PGMTBLI,A0
        ADD.L   D0,A0
        MOVE.L  (A0),A0
        JSR     (A0)
        RTE
PROG6   MOVE.W  #LED_C0,CSR0+2
        MOVE.L  SELP,D0
        ASL.L   #2,D0
        MOVE.L  #PGMTBL,A0
        ADD.L   D0,A0
        MOVE.L  (A0),A0
        JSR     (A0)
        MOVE.W  #LED_C0,CSR0
        RTE
PROG7   ADD.W   #1,DCOUNT
        BTST.B  #0,IMASK+3
        BNE     PROG6
        RTE
*
*  SOMETHING BAD HAPPENED (BadNews)
BN      STOP    #$2700
        DS.W    1               Force longword alignment
*
*  GENERAL DATA
*
* Program table for interrupt #5
        ORG     $900
PGMTBLI DC.L    LINTEST0,SCAN0,NULL,PED0,SAW0,PTEST0,QTEST0,DTEST0
*
* Program table for interrupt #6
PGMTBL  DC.L    LINTEST,SCAN,FM_BEB,PED,SAW1,PTEST,QTEST,DTEST
*
*  Table to remap channels because of scrambled connections
*
BCN     DC.B    17,19,21,23,16,18,20,22
        DC.B    01,03,05,07,00,02,04,06
        DC.B    09,11,13,15,08,10,12,14
        DC.B    24,26,28,30,25,27,29,31
*
*  Offsets to each event.  These offsets are added to the values in
*  the BCA table.
EVOFF   DC.L    0,EVENT,2*EVENT,3*EVENT
*
*  Table of channel addresses.  These address are computed from the
```


Listing 4.2

```
*  table above in the startup routines.  These addresses
*  indicate the location and order of data in the FEB according to
*  the pre-amp channels
BCA      DS.L     32
NBUCK    DC.W     0,0
OFFSET   DC.L     0
*****************************************************************
*
*  Utility routines
*
*  Compute number of buckets to search
PROCNT   MOVE.L   D0,-(SP)
         TST.W    COUNT
         BNE      PRO0
         MOVE.L   #$00FA000A,COUNT
PRO0     MOVE.W   COUNT,D0
         SUB.W    COUNT+2,D0
         MOVE.W   D0,NBUCK
*
*  Compute offset to first bucket
         CLR.L    D0
         MOVE.W   COUNT+2,D0
         ASL.L    #2,D0
         ADD.W    #3,D0                    Offset to data byte
         MOVE.L   D0,OFFSET
         MOVE.L   (SP)+,D0
         RTS
*****************************************************************
*
*  Start Up
START    MOVE.W   CSR1,D0          Get board serial number
         AND.L    #$03FF,D0        Only 10 bits are valid
         MOVE.L   D0,BDSER         Save it in communications area
*
*  If serial number less than 695 then the channels are
*  rearranged from the table shown
         CMP.W    #695,D0
         BGE      SS0
         MOVE.L   #BCN+32,A0
         MOVE.W   #31,D1
SS1      MOVE.B   D1,-(A0)
         DBRA     D1,SS1
*
*  Compute table of addresses to channels
SS0      MOVE.L   #BCN,A0          Board channel numbers
         MOVE.W   #31,D0           Number of channels
         MOVE.L   #BCA,A1          Board channel addresses
X_PTR    CLR.L    D1               Our accumulator
         MOVE.B   (A0)+,D1         Board channel number
         MULU     #CHANNEL,D1      Compute offset from start of FEB
         ADD.L    #FEB,D1          Compute actual address
         MOVE.L   D1,(A1)+         Save in table
         DBRA     D0,X_PTR         Repeat 31 more times.
         RTS
*****************************************************************
*  SCAN0     -- prepare for taking data for mulitple scans
*  SCAN      -- add FEB data to BEB for another scan
*
*  BEB: [--------Blocklet WC-----------]
*       [---Err Flgs---][--Data Type---]
*       [---Bd Ser #---][--# Calls-----]
*
*       [--------Blocklet WC-----------]
*
SCAN0    MOVE.L   #BEB,A0                  Pointer to BEB
         MOVE.L   #128*32+4,(A0)+          Word count
         MOVE.L   #SCAN_DT,(A0)+           Data type for SCAN
         MOVE.W   BDSER+2,(A0)+            Board Serial Number
         MOVE.W   #0,(A0)+                 No calls yet
         MOVE.W   #128*32-1,D0             Total number of data words
S0       CLR.L    (A0)+
         DBRA     D0,S0
         MOVE.L   #256*32+4,(A0)+          Trailing Word Count
         RTS
*****************************************************************
*
*  A1    Pointer to table of channel addresses
*  A2    Pointer to BEB destination
*  A3    Pointer into FEB
*
*  D2    Count channels
*  D3    Count Buckets
*
SCAN     MOVE.L   #BEB,A2                  Start of BEB
         ADD.W    #1,10(A2)                INC number of Calls
         ADD.L    #3*4,A2                  Compute start of data
*
*  Loop over channels
         MOVE.L   #BCA,A1                  Pointer to channel addresses
         MOVE.W   #31,D1                   Counter of channels
*
*  Loop over buckets
S1       CLR.L    D0
         MOVE.L   #255,D2                  Bucket counter
         MOVE.L   (A1)+,A3                 Start location in FEB for this channel
         ADD.L    #3,A3                    Point to start location
S2       MOVE.B   (A3),D0                  Get byte from FEB
         ADD.W    D0,(A2)+                 Add to output (no linearzation)
         ADD.L    #4,A3                    Advance pointer in FEB
         DBRA     D2,S2
         DBRA     D1,S1
         RTS
*****************************************************************

*****************************************************************
*  The following routines are similar in structure to SCAN and have
*  been deleted from the listing for brevity:
*
*  LINTEST0 -- prepare for takeing data for lintest.
*  LINTEST  -- take data for lintest.
*  PTEST0   -- prepare for taking data for pulsetest
*  PTEST    -- take data for pulsetest
*  QTEST0   -- prepare for taking data for pulsetest
*  QTEST    -- take data for pulsetest
*  PED0     -- prepare for taking data for pedestal check
*  PED      -- take data for pedestal check
*  SAW0     -- prepare for taking data for sawtooth test
*  SAW1
*  DELTA TEST
*  DTEST0
*  DTEST
*****************************************************************

*****************************************************************
*  Data Reformat
*  Data Format in the BEB
*
*  BEB:    [-------------Blocklet WC(32)--------------]
*          [----Err Flgs(12)----][----Data Type(20)---]
*          [-Mod#(8)-][-Pad1(8)-][-Dsize(8)-][-Acc(8)-]
*          [-----Pad #(16)------][-WdCnt(8)-][NClst(8)]
*          [-----1st TS(16)-----][-----# Amps(16)-----]
```


Listing 4.2 (continued)

```
*           [---A0----][---A1----][---A2----][---A3----]
*           [---A4----][---A5----][---A6----][---A7----]
*           [-----1st TS(16)-----][-----# Amps(16)-----]
*           [---B2----][---B3----][---B4----] ...
*
*           [---En----][----0----][----0----][----0----]
*           [-----Pad #(16)------][-WdCnt(8)-][NClst(8)]
*           [-----1st TS(16)-----][-----# Amps(16)-----]
*           [---A0----][---A1----][---A2----][---A3----]
*           [---A4----][---A5----][---A6----][---A7----]
*           [-----1st TS(16)-----][-----# Amps(16)-----]
*           [---B2----][---B3----][---B4----] ...
*
*           [---En----][----0----][----0----][----0----]
*
*           [-------------Blocklet WC(32)--------------]
*
*  Channels with only presample data are dropped.  Longword alignment
*  is forced at the end of each cluster.
*
*  Register Usage
*    A0         Pointer to Address of Current Channel in FEB
*    A1         Pointer to Current Data Word in FEB
*    A2         Pointer to Output position in BEB
*    A3         Pointer to Start of Current Cluster in BEB
*    A4         Pointer to End of Current Channel
*
*    D0         Scratch Register
*    D1         Scratch Register
*    D2         Channel Counter
*    D3         Word Count for Current Channel
*    D4         Last channel for this board
*    D5         Number of clusters in this channel
*
*
NULL    RTS
FM_BEB    MOVEM.L    A0-A5/D0-D5,-(SP)    Save Some Registors for local use
*
*  Locate start of indicated event/channel (FEB)
    MOVE.W    #0,F_REG        Clear flags register
    MOVE.L    #0,FLAGS        Clear FLAGS communication register
    LEA       IFEB,A0         Address of IFEB array
    MOVE.L    PAD1,D2         Load 1st pad number
    MOVE.L    D2,D4           Copy into D4
    LSL.L     #1,D2           Two byte offset per channel
    ADD.L     D2,A0           Add offset for 1st pad
    LSR.L     #1,D2           Shift D2 back again
    ADD.L     #31,D4          Calculate last channel #
*
*  Setup Destination (BEB)
    MOVE.L    #BEB,A2             Start of BEB
*
*  Generate Header for Blocklet
    MOVE.L    #0,(A2)+          Future site of Blocklet Word Count
    MOVE.W    #0,(A2)+          No Errors so far
    MOVE.W    #$910,(A2)+        HPC pad data
    MOVE.B    MODU+3,(A2)+          Module number
    MOVE.B    PAD1+3,(A2)+          First Pad Number
    MOVE.B    #1,(A2)+          Signifies 1 byte per time slot
    MOVE.B    CSR11,(A2)+
*
*  Begin a new channel
X_CHAN      SUBA.L    A1,A1     Clear contents of A1
    CLR.L     D0                Be sure upper bits are zero
    MOVE.W    (A0),D0           Load start of FEB for this channel
*  Note: in the future, we will need to pick which of the four events
*  in the FEB we want to read out.  Do it here by masking LS two bits
*  of trig acc no., shifting left 8 bits, and adding to D0.
    LSL.L     #2,D0             Convert FB address into 68K address
    MOVE.L    D0,A1             Store it in A1
    ADD.L     #3,A1             Upper word empty
    CLR.L     D3
    MOVE.B    (A1)+,D3          FEB word count for this channel
*
*  Check for more than presample
    CMP.W     #9,D3
    BLE       MT_CHAN
*
*  Compute End of Data in FEB for this channel
    MOVE.L    D0,A4             Copy start of FEB to A4
    MOVE.L    D3,D0             Temp
    ASL.L     #2,D0             Times 4 for bytes
    ADD.L     D0,A4             End of data
*
*  First Cluster is the presample
    MOVE.L    A2,A5             Save start of channel in BEB
    MOVE.L    #1,D5             Initialize cluster counter
    MOVE.W    D2,(A2)+          Channel Number
    ADD.L     #2,A2             Save space to store word count and no of clust
    MOVE.W    #0,(A2)+          1st time slice
    MOVE.W    #8,(A2)+          Number of amplitudes
    MOVE.W    #7,D1
L0    ADD.L     #3,A1
    MOVE.B    (A1)+,(A2)+         Transfer Presample data
    DBRA      D1,L0
*
*  Begin a new cluster
X_CLU     MOVEA.L    A2,A3             Save start of cluster in BEB
*
*  Cluster data
    CLR.L     D0
    ADD.L     #2,A1
    MOVE.B    (A1)+,D0          First TS
    MOVE.W    D0,(A2)+          Save TS in Cluster Header
    ADD.L     #2,A2             No. of amplitudes (fill later)
    ADD.L     #1,D5             Increment number of clusters in this channel
X_AMP     CMP.L  A1,A4          End of FEB? (Old data could fake it)
    BLE       BACK_UP
    MOVE.B    (A1)+,(A2)+         Transfer one amplitude to BEB
    ADD.L     #2,A1
    ADD.B     #1,D0             Next TS
    CMP.B     (A1)+,D0
    BEQ       X_AMP
BACK_UP   SUB.L     #3,A1             Back Up
*
*  Compute Number of Amplitudes in this cluster in BEB
    MOVE.L    A2,D0             End of current channel
    SUB.L     A3,D0             Start of this Cluster
    SUB.L     #4,D0             Number of bytes in header
    ADD.L     #2,A3
    MOVE.W    D0,(A3)             Save it
WORDALIGN:  MOVE.W     A2,D0     Pad with 0s to next longword boundary
    AND.W     #0003,D0
    BEQ       ALIGNDONE
    CLR.B     (A2)+
    JMP       WORDALIGN
ALIGNDONE   CMP.L      A1,A4
*
*  End of channel ?
    BGT       X_CLU             No, Do more clusters
*
*  Advance to next channel
```


Listing 4.2 (continued)

```
E_CHAN    MOVE.L   A5,A3       Restore start of channel in A3
    MOVE.L   A2,D0            Copy current pointer in BEB
    SUB.L    A3,D0            Subtract start of channel address
    LSR.L    #2,D0            Convert to longwords
    ADD.L    #2,A3            Point to no. of lwords for this pad
    MOVE.B   D0,(A3)+         Put no. of longwords in channel header
    MOVE.B   D5,(A3)+         Save number of clusters in this channel
MT_CHAN   ADD.L    #2,A0            Advance channel pointer
    ADD.W    #1,D2            Advance channel number
*
*  Last Channel?
    CMP      D4,D2
    BLE      X_CHAN
*
*  Finish Blocklet
    MOVE.L     A2,D0            Current end of BEB
    SUB.L    #BEB,D0            Compute length in bytes
    ASR.L    #2,D0            Now longwords
    ADD.L    #1,D0            Extra WC at end of Blocklet
    MOVE.L     D0,(A2)+         Trailing WC
    MOVE.L     D0,BEB             Leading WC
*
*  Finished
    MOVE.W   #1,F_REG         Set 'done' bit
    MOVE.L   #1,FLAGS         Ditto
    CMP      #5,D0            Is there data on this board?
    BLE      DONE_FMT
    MOVE.W   #3,F_REG         Set 'data present' bit also
    MOVE.L   #3,FLAGS         Ditto
DONE_FMT    MOVEM.L    (SP)+,A0-A5/D0-D5
    RTS
IFEB:    DC.W     $A800,$BC00,$D400,$EC00,$9800,$A000,$B800,$C400
         DC.W     $D800,$9000,$9400,$B400,$C000,$D000,$E800,$AC00
         DC.W     $C800,$E400,$9C00,$B000,$DC00,$F000,$8C00,$A400
         DC.W     $CC00,$F400,$8800,$E000,$B400,$F800,$8000,$FC00
         DC.W     $BC00,$D800,$E800,$9000,$9800,$AC00,$C000,$CC00
         DC.W     $E000,$8C00,$A800,$B800,$C800,$DC00,$8400,$A400
         DC.W     $D000,$EC00,$9400,$C400,$E400,$F000,$8800,$B000
         DC.W     $D400,$F400,$8000,$B400,$A000,$F800,$9C00,$FC00
         DC.W     $C400,$D800,$E800,$9000,$9C00,$BC00,$C800,$DC00
         DC.W     $8C00,$9400,$A000,$C000,$D400,$E000,$8800,$A400
         DC.W     $CC00,$EC00,$9800,$B400,$E400,$F400,$8400,$B000
         DC.W     $D000,$F000,$8000,$B800,$AC00,$F800,$A800,$FC00
         DC.W     $AC00,$C000,$D800,$DC00,$8C00,$9800,$B000,$C400
         DC.W     $C800,$E000,$8800,$9C00,$B800,$CC00,$EC00,$F400
         DC.W     $9000,$BC00,$E400,$9400,$D000,$E800,$8400,$A000
         DC.W     $D400,$F000,$8000,$B400,$A800,$F800,$A400,$FC00
         DC.W     $8000,$8400,$8800,$8C00,$9000,$9400,$9800,$9C00
         DC.W     $A000,$A400,$A800,$AC00,$B000,$B400,$B800,$BC00
         DC.W     $C000,$C400,$C800,$CC00,$D000,$D400,$D800,$DC00
         DC.W     $E000,$E400,$E800,$EC00,$F000,$F400,$F800,$FC00
         END
```

Listing 4.2 (continued)

Appendix A – Technical Parameters

Power Needs

+15.0 V	0.8 A
+6.2 V	3.6 A
+5.0 V	7.2 A
−5.2 V	2.3 A

Analog Inputs

Number of Channels	32
Cables	2 Twisted Pair (32 or more conductors)
Connectors	2 34-pin 3M (H34-202-TL)

Digitizer Stage

Digitization Frequency	15 MHz
Input Range	
at input* (at FADC)	0–2.0 V (0-4.0 V)
Step Size (lower quadrant)	
at input* (at FADC)	2.5 mV (5.0 mV)
Step Size (upper three quadrants)	
at input* (at FADC)	9.5 mV (19.0 mV)
Effective Dynamic Range	800:1
Number of Samples per Channel	255

Zero Suppression Stage

Clock Frequency	4.5 MHz
Maximum time to suppress	500 μs

Front End Buffer

Number of Events	4

Microprocessor

Processor	Motorola 68000
Clock Frequency	16 MHz
Memory (SRAM)	128 KBytes (no wait states)
Interrupt Levels	5,6, and 7

FASTBUS

Addressing Modes	Geographical, Logical, Broadcast

*assumes an analog input buffer with a nominal gain of 2.0

Appendix B – Input Connectors

Upper Connector

Channel 0–	Pin 1	Channel 8–	Pin 17
Channel 0+	Pin 2	Channel 8+	Pin 18
Channel 1–	Pin 3	Channel 9–	Pin 19
Channel 1+	Pin 4	Channel 9+	Pin 20
Channel 2–	Pin 5	Channel 10–	Pin 21
Channel 2+	Pin 6	Channel 10+	Pin 22
Channel 3–	Pin 7	Channel 11–	Pin 23
Channel 3+	Pin 8	Channel 11+	Pin 24
Channel 4–	Pin 9	Channel 12–	Pin 25
Channel 4+	Pin 10	Channel 12+	Pin 26
Channel 5–	Pin 11	Channel 13–	Pin 27
Channel 5+	Pin 12	Channel 13+	Pin 28
Channel 6–	Pin 13	Channel 14–	Pin 29
Channel 6+	Pin 14	Channel 14+	Pin 30
Channel 7–	Pin 15	Channel 15–	Pin 31
Channel 7+	Pin 16	Channel 15+	Pin 32

Lower Connector

Channel 16–	Pin 1	Channel 24–	Pin 17
Channel 16+	Pin 2	Channel 24+	Pin 18
Channel 17–	Pin 3	Channel 25–	Pin 19
Channel 17+	Pin 4	Channel 25+	Pin 20
Channel 18–	Pin 5	Channel 26–	Pin 21
Channel 18+	Pin 6	Channel 26+	Pin 22
Channel 19–	Pin 7	Channel 27–	Pin 23
Channel 19+	Pin 8	Channel 27+	Pin 24
Channel 20–	Pin 9	Channel 28–	Pin 25
Channel 20+	Pin 10	Channel 28+	Pin 26
Channel 21–	Pin 11	Channel 29–	Pin 27
Channel 21+	Pin 12	Channel 29+	Pin 28
Channel 22–	Pin 13	Channel 30–	Pin 29
Channel 22+	Pin 14	Channel 30+	Pin 30
Channel 23–	Pin 15	Channel 31–	Pin 31
Channel 23+	Pin 16	Channel 31+	Pin 32

Note: Pins 33 and 34 of each connector are unused.

Appendix C - Sample Readout Routines

```
$!*************************************************
$!*****                                           ********
$!*****  Copyright 1988 by Ames Laboratory        ********
$!*****                                           ********
$!*****  High Energy Physics                      ********
$!*****  12 Physics                               ********
$!*****  Iowa State University                    ********
$!*****  Ames IA 50011                            ********
$!*****                                           ********
$!*************************************************
$!
$!C*****************************************************************************
C*****************************************************************************
      SUBROUTINE READ_ONE_BEB(ACCNO, PRIADD, IBEB)
*****************************************************************************72
*
*  A sample routine to be called after the event trigger.
*  It assumes the user has already done the following:
*             1.  Loaded the microprocessor program.
*             2.  Set the interrupt vector for interrupt 6.
*             3.  Enabled the microprocessor.
*  It reads the BEB from a single waveform digitizer located at FASTBUS
*  address PRIADD, using reads to the flag bits in memory to determine
*  if the microprocessor has finished and if it has found data.
*  ACCNO is a trigger accounting number, the least two significant
*  bits of which specify which of the four events to read from the FEB.
*  IBEB is an array to receive the data.
*
*  Note that this FORTRAN version is only a sample routine to demonstrate
*  the algorithm.  It executes slowly, has no error checking, and contains
*  a potentially infinite loop.
*
*****************************************************************************72

      INCLUDE 'COMMON:FBADRCOM.FOR'
      INTEGER*4 ACCNO, PRIADD, IBEB(*), WDCOUNT
      INTEGER*4 IFLAG, KOUNT, KOUNT1, POINTER, FEB_EVENT

      INTEGER*4 COMM$BEB, COMM$FLAG_REG, EOB_REG, CSR0, CSR10, CSR11
      INTEGER*4 CSR0$M_FEB_CLR, CSR10$M_INT6

      PARAMETER (COMM$BEB       = '40'X)          ! Start of the BEB
      PARAMETER (COMM$PROG_SELECT = '46'X)        ! Program selector
      PARAMETER (COMM$FLAG_REG  = '47'X)          ! Flag register location
      PARAMETER (EOB_REG        = '18800'X)       ! Block Xfer word count reg
      PARAMETER (CSR0$M_FEB_CLR = '00800000'X)    ! Clears CSR#0<7>
      PARAMETER (CSR10$M_ENABLE = '00000001'X)    ! Enables 68K
      PARAMETER (CSR10$M_INT6   = '00000040'X)    ! Sets interrupt level 6
      PARAMETER (CSR0           = '00'X)
      PARAMETER (CSR10          = '10'X)
      PARAMETER (CSR11          = '11'X)
*
*  Initialize variables
*
      IBEB(1) = 0
      POINTER = 0
*
*  Write trigger accounting number to CSR#11h
*
      CALL FWC(STATUS, CNTRL, PRIADD, CSR11, ACCNO)
*
*  Send interrupt 6 by setting CSR#10h<06>.  We assume the interrupt vector
*  has already been set correctly and that the program has been loaded into
*  RAM.
*
      CALL FWC(STATUS, CNTRL, PRIADD, CSR10, CSR10$M_INT6)
```

```
*
*  Now wait .007 seconds while program executes on microprocessor.
*  WAIT is assumed to be a system- or user-supplied wait routine.
*
   10 CALL WAIT(.007)
*
*  Read the flag register and test to see if the reformatting is done.
*  If it is not done go back to the wait instruction.  If it is done
*  see if there is data to be read out.  (Note that there is no
*  protection against an infinite loop here, a real routine should
*  set a maximum number of iterations).
*
      CALL FRD(STATUS, CNTRL, PRIADD, COMM$FLAG_REG, IFLAG)
      IF (IAND(IFLAG,'1'X) .EQ. 0) GOTO 10
      IF (IAND(IFLAG,'2'X) .EQ. 0) GOTO 999
      CALL FETCH_BEB( PRIADD, COMM$BEB, IBEB, WDCOUNT)
*
*  Readout done
*
  999 CONTINUE
      FEB_EVENT = ISHFT(CSR0$M_FEB_CLR,IAND(ACCNO,'3'X))
*
*  FREE_FEB is a routine that can be used to tell the trigger system that
*  we are done with this event.  It can also be used to reset the event bit in
*  CSR#0, if desired.  Depending on the application, this routine may not
*  be necessary.
*
      CALL FREE_FEB(PRIADD, FEB_EVENT)
      END



      SUBROUTINE FETCH_BEB( PRIADD, BEB_POINT, IBEB,
     >                      WDCOUNT)
*************************************************************************72
*
*  Read the entire contents of one module's BEB into the array IBEB.
*  BEB_START is the data space address of the start of the BEB.  PRIADD
*  is the module's primary address.  WDCOUNT returns the number of words
*  read into IBEB.
*
*************************************************************************72
      INCLUDE 'COMMON:FBADRCOM.FOR'
      INTEGER*4 PRIADD, BEB_START, BEB_POINT, IBEB(*), WDCOUNT
      INTEGER*4 KOUNT, KOUNT1, POINTER
      INTEGER*4 EOB_REG

      PARAMETER (EOB_REG = '18800'X)        !Block Xfer word count register
*
*  Get address of start of BEB from communications vector area
*
      CALL FRD(STATUS, CNTRL, PRIADD, BEB_POINT, BEB_START)
*
*  Get the word count from the first word in the BEB.
*
      CALL FRD(STATUS, CNTRL, PRIADD, BEB_START, KOUNT)
      WDCOUNT = KOUNT
*
*  Now do a block transfer of the data.  We must check if there is more than
*  4095 words (32 bit).  If so, we must use several block transfers to read
*  them all
*
   10 KOUNT1 = KOUNT
      POINTER = 0
      IF (KOUNT .GT. 4095) KOUNT1 = 4095
      KOUNT  = KOUNT - KOUNT1
```

```
*
*  Write word count to EOB register so slave will send SS=2 to end transfer.
*  If a master terminated block transfer is used,  the EOB register must still
*  be set to a value equal to or larger than the number of words to be
*  transferred to avoid a premature SS=2.
*
      CALL FWD(STATUS, CNTRL, PRIADD, EOB_REG, KOUNT1)
      CALL FRDB(STATUS, CNTRL, PRIADD, BEB_START + POINTER,
     >          IBEB, 4*KOUNT1)
      IF (KOUNT .LE. 0) GOTO 20
      POINTER = POINTER + KOUNT1
      GOTO 10
   20 CONTINUE
      END


      SUBROUTINE FREE_FEB(PRIADD, FEB_EVENT)
*****************************************************************72
*
*  A sample routine.  All this version does is to reset the event bit
*  in CSR#0.
*
*****************************************************************72

      INCLUDE 'COMMON:FBADRCOM.FOR'
      INTEGER*4 PRIADD, FEB_EVENT
      CALL FWC(STATUS, CNTRL, PRIADD, 0, FEB_EVENT)
      END


      SUBROUTINE READ_MULTI_BEB(ACCNO, BR_CLASS, TPATTERN, IBEB)
*****************************************************************72
*
*  A sample routine to be called after the event trigger.
*  It assumes the user has already done the following:
*             1.  Loaded the microprocessor program.
*             2.  Set the interrupt vector for interrupt 6.
*             3.  Enabled the microprocessor.
*             4.  Initialized all waveform digitizers to broadcast class
*                 BR_CLASS.
*             5.  TPATTERN has been correctly initialized.
*  It reads the BEB from all waveform digitizers located in the root
*  segment that have been set to respond to broadcast class BR_CLASS.
*  It uses T-pin scans to determine if the microprocessors have finished
*  and if they have found data.
*
*  ACCNO is a trigger accounting number, the least two significant
*  bits of which specify which of the four events to read from the FEB.
*  IBEB is an array to receive the data.  TPATTERN is a bit pattern
*  corresponding to the expected T-pin scan if all waveform digitizers
*  respond.  (I.e. it is a 32 bit word with a 1 in each bit position
*  corresponding to a digitizer module.
*
*  Note that this FORTRAN version is only a sample routine to demonstrate
*  the algorithm.  It executes slowly, has no error checking, and contains
*  a potentially infinite loop.
*
*****************************************************************72

      INCLUDE 'COMMON:FBADRCOM.FOR'
      INTEGER*4 ACCNO, BR_CLASS, IBEB(*)

      INTEGER*4 IFLAG, KOUNT, KOUNT1, POINTER, FEB_EVENT, TSCAN,
     >          TPATTERN, I, PRIADD, NOW, WDCOUNT
```

```
      INTEGER*4 COMM$BEB, COMM$FLAG_REG, EOB_REG, CSR0, CSR10, CSR11,
     >          CSR0$M_FEB_CLR, CSR10$M_INT6, TPIN_3, TPIN_3A

      PARAMETER (COMM$BEB       = '40'X)          ! Start of the BEB
      PARAMETER (COMM$PROG_SELECT = '46')         ! Program selector
      PARAMETER (COMM$FLAG_REG  = '47'X)          ! Flag register location
      PARAMETER (EOB_REG        = '18800'X)       ! Block Xfer word count reg
      PARAMETER (CSR0$M_FEB_CLR = '00800000'X)    ! Clears CSR#0<7>
      PARAMETER (CSC10$M_ENABLE = '00000001'X)    ! Enable 68K
      PARAMETER (CSR10$M_INT6   = '00000040'X)    ! Sets interrupt level 6
      PARAMETER (CSR0           = '00'X)
      PARAMETER (CSR10          = '10'X)
      PARAMETER (CSR11          = '11'X)
      PARAMETER (TPIN_3         = '00000009'X)    ! T-Pin class 3 (data scan)
      PARAMETER (TPIN_3A        = '00000019'X)    ! T-Pin class 3a (dev avail)
*
*  Initialize variables
*
      NOW = 1
      IBEB(NOW) = 0
*
*  Write trigger accounting number to CSR#11h
*
      CALL FWCM(STATUS, CNTRL, BR_CLASS, CSR11, ACCNO)
*
*  Send interrupt 6 by setting CSR#10h<06>.  We assume the interrupt vector
*  has already been set correctly and that the program has been loaded into
*  RAM.
*
      CALL FWCM(STATUS, CNTRL, BR_CLASS, CSR10, CSR10$M_INT6)
*
*  Now wait .007 seconds while programs execute on microprocessors.
*  WAIT is assumed to be a system- or user-supplied wait routine.
*
   10 CALL WAIT(.007)
*
*  Do a class 3a T-pin scan to see if all the boards have finished
*  executing the reformatting routine.  If they have not, wait some more
*  and try again.  (Note that there is no protection against an infinte
*  loop here, a real routine should set a maximum number of iterations).
*
      CALL TPIN(TPIN_3A, TSCAN)
      IF (TSCAN .NE. TPATTERN) GOTO 10
*
*  Do a class 3 T-pin scan to see if any of the modules contain data.
*
      CALL TPIN(TPIN_3, TSCAN)
      TSCAN = IAND(TSCAN,TPATTERN)
      IF (TSCAN .EQ. 0) GOTO 999
*
*  Loop over all boards with data
*
      DO I = 0, 31
        PRIADD = I
        IF (BTEST(TSCAN,I)) THEN
        CALL FETCH_BEB(PRIADD, COMM$BEB, IBEB(NOW),
     >                 WDCOUNT)
        NOW = NOW + WDCOUNT
        IBEB(NOW) = 0
        ENDIF
      ENDDO
*
*  Readout done
*
  999 CONTINUE
```

```
      FEB_EVENT = ISHFT(CSR0$M_FEB_CLR,IAND(ACCNO,'3'X))
*
*   FREE_FEBM is a routine that can be used to tell the trigger system that
*   we are done with this event.  It cal also be used to reset the event bit in
*   CSR#0, if desired.  Depending on the application, this routine may not
*   even be necessary.
*
      CALL FREE_FEBM(BR_CLASS, FEB_EVENT)
      END


      SUBROUTINE FREE_FEBM(BR_CLASS, FEB_EVENT)
*******************************************************************************72
*
*   A sample routine.  All this version does is to reset the event bit
*   in CSR#0.
*
*******************************************************************************72

      INCLUDE 'COMMON:FBADRCOM.FOR'
      INTEGER*4 BR_CLASS, FEB_EVENT
      CALL FWCM(STATUS, CNTRL, BR_CLASS, 0, FEB_EVENT)
      END


      SUBROUTINE TPIN(PRIADD, TSCAN)
*******************************************************************************72
*
*   Do a T-pin scan using PRIADD as the primary address, return the
*   result as TPIN.
*
*******************************************************************************72

      INCLUDE 'COMMON:FOPARA.FOR'
      INCLUDE 'COMMON:FBADRCOM.FOR'
      INTEGER*4 PRIADD, TSCAN
*
      CNTRL(2) = FONOSA
      CALL FRDM(STATUS, CNTRL, PRIADD, 0, TSCAN)
      CNTRL(2) = FOEG
      END


*******************************************************************************
*
*    COMMON block FOPARA inserted by INCLUDE statements.
*
*******************************************************************************
        INTEGER FOGERR, FONOAR, FONOPA, FOEG,  FOGKUP,
        1       FONOSA, FONODC, FOASUP, FOBLKE, FOBUFE
        PARAMETER (
        1  FOGERR = '0001'X, FONOAR = '0002'X, FONOPA = '0004'X,
        2  FOEG   = '0008'X, FOGKUP = '0010'X, FONOSA = '0020'X,
        3  FONODC = '0040'X, FOASUP = '0080'X, FOBLKE = '0100'X,
        4  FOBUFE = '0200'X)


*******************************************************************************
*
*    COMMON block FBADRCOM inserted by INCLUDE statements.
*
*******************************************************************************
        COMMON /FBADR/
```

```
1    PRIMAD, GEOAD, BRODAD, SECAD, IDATA,
2    IBN, IBSN, CNTRL(2), STATUS(4), ICSR10, ICSR11,
3    IAMODE, ISPACE, NBSN(25), TIMESTRING
CHARACTER TIMESTRING*25
INTEGER PRIMAD, GEOAD, BRODAD, SECAD, IDATA
INTEGER CNTRL, STATUS
```

Appendix D – Subroutine to Load S-Code Files

```
C*************************************************************
C  LOAD THE SPECIFIED 68K PROGRAM IN S-CODE
C*************************************************************
      SUBROUTINE LD68K(FILENAME)
C
C  WARNING ** THE S-CODE DATA MUST BEGIN ON LONGWORD BOUNDARYS
C  FOR PROPER OPERATION
C
      INCLUDE 'COMMON:FBADRCOM.FOR'
      INCLUDE 'COMMON:M68KCOM.FOR'
      PARAMETER COMM_REG                =  '00000040'X
      PARAMETER COMM$BEB                =   0 + COMM_REG
      PARAMETER COMM$MODULE             =   1 + COMM_REG
      PARAMETER COMM$FIRST_PAD          =   2 + COMM_REG
      PARAMETER COMM$PROGRAM_VERSION    =   3 + COMM_REG
      PARAMETER COMM$LOAD_TIME          =   4 + COMM_REG
      PARAMETER COMM$LOAD_DATE          =   5 + COMM_REG
      PARAMETER COMM$PROGRAM_SELECT     =   6 + COMM_REG
      PARAMETER COMM$FLAG_REGISTER      =   7 + COMM_REG
      PARAMETER COMM$SERIAL_NUMBER      =   8 + COMM_REG
      PARAMETER COMM$BIT_MASK           =   9 + COMM_REG
      PARAMETER COMM$COUNTER            =  10 + COMM_REG
      CHARACTER FILENAME*20, LINE*60, SX*2, NUMB*8
      INTEGER IRLEN, IADDR, IIDATA(4)
      INTEGER INA, INW, IBDATA(256)
C
      COMMON/STACKC/ INW, INA, IBDATA, IADDR, IIDATA, N
C
C  INITIALIZATION
      CALL RUN68K(0)                              !Halt 68k, CSR#0<0>=0
      IS = SMG$PUT_CHARS(ID_68S,FILENAME(1:10),2,18)
C
C  GET S-CODE
      OPEN( UNIT=11, FILE=FILENAME, DEFAULTFILE='M68K:.SEX',
     1   STATUS='OLD', ERR=98)
      INW = 0
      DO WHILE( .TRUE.)
      READ(11,'(A,A)') SX,LINE
C
C  HEADER RECORDS
      IF(SX .EQ. 'S0') THEN
C
C  DATA RECORDS
      ELSE IF (SX .EQ. 'S1') THEN
        READ(LINE, '(Z2,Z4,4Z8)') IRLEN, IADDR, IIDATA
        N = (IRLEN-1-2+3)/4
        CALL STACK
      ELSE IF (SX .EQ. 'S2') THEN
        READ(LINE, '(Z2,Z6,4Z8)') IRLEN, IADDR, IIDATA
        N = (IRLEN-1-3+3)/4
        CALL STACK
      ELSE IF (SX .EQ. 'S9') THEN
        IADDR=-1
        CALL STACK
        CLOSE(UNIT=11)
        GOTO 98
      ENDIF
      ENDDO
C
C  LOAD COMPLETE
98    CONTINUE
C
      END
```

```
C*************************************************************
C  COMBINE DATA INTO LARGEST BLOCKS POSSIBLE FOR LOADING
C*************************************************************
        SUBROUTINE STACK
        INCLUDE 'COMMON:FBADRCOM.FOR'
        INTEGER IBDATA(256),IIDATA(4)
        COMMON/STACKC/ INW, INA, IBDATA, IADDR, IIDATA, N
        IADDR = IADDR/4
C
        IF( INW .EQ. 0 ) INA = IADDR
C
        IF(IADDR-INA .NE. INW .OR. INW .GE. 253) THEN
          CALL FWD(STATUS,CNTRL,PRIMAD,'00018800'X,INW)
          CALL FWDB(STATUS,CNTRL,PRIMAD,INA,IBDATA,INW*4)
          INW = 0
          INA = IADDR
        ENDIF
          DO I = 1,N
            INW = INW + 1
            IBDATA(INW) = IIDATA(I)
          ENDDO
        END

        COMMON /FBADR/
        1    PRIMAD, GEOAD, BRODAD, SECAD, IDATA,
        2    IBN, IBSN, CNTRL(2), STATUS(4), ICSR10, ICSR11,
        3    IAMODE, ISPACE, NBSN(25), TIMESTRING, IGEOGPM
        CHARACTER TIMESTRING*25
        INTEGER PRIMAD, GEOAD, BRODAD, SECAD, IDATA
        INTEGER CNTRL, STATUS

        COMMON/M68KCOM/ ID_68M, ID_68S, ID_68P, ID_68C,
        1 ID_68SD
```

Bibliography

FASTBUS

1. Louis Costrell and W. K. Dawson, FASTBUS for Data Acquisition and Control, IEEE Trans. Nucl. Sci., **NS-30**, 2147 (1983).
2. Institute of Electrical and Electronic Engineers, IEEE Standard FASTBUS Modular High-Speed Data Acquisition and Control System, ANSI/IEEE Std 960-1986, Distributed by Wiley-Interscience, New York, NY (1985).
3. G. Frémont, A FASTBUS Prototyping Card Incorporating a Comprehensive TTL Slave Coupler, CERN-EP F685E, 1985.

M68000 Processor

4. Motorola Semiconductor Products Inc., MC68000 16-/32-Bit Microprocessor, 3501 Ed Bluestein Blvd., Austin, TX 78721, October 1985.
5. William Cramer and Gerry Kane, 68000 Microprocessor Handbook, Second Edition, Osborne McGraw-Hill, 1986.
6. Leo J. Scanlon, The 68000: Principles and Programming, Howard W. Sams and Co., Inc, Indianapolis, IN 46268, 1983.
7. H. Müller, The GPM General Purpose FASTBUS Master/Slave – User Manual, CERN-EP F6808, 1985.
8. Horst von Eicken, Software Support for Motorola 68000 Microprocessor at CERN – MoniCa Symbolic Debugger, CERN-DD, 1986.

Ames Waveform Digitizer

9. H. B. Crawley et al., IEEE Trans. Nucl. Sci. **NS–34**, 261 (1987).
10. W. T. Meyer and M. S. Gorbics, A Simple Microprocessor for FASTBUS Slave Modules, to be published in Computers in Physics.