A FASTBUS LOGIC STATE ANALYZER

BY

SERGIO ZIMMERMANN

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering
in the School of Advanced Studies of
Illinois Intitute of Technology

Approved George D. Krafft
Adviser

Chicago, Illinois
December, 1985

ACKNOWLEDGMENT

To make this thesis possible many persons and institutions from U.S.A. and Brazil contributed both, directly and indirectly.

From U.S.A. the Author wishes to thank Dr. George D. Kraft, his adviser from IIT, for his support and help in this thesis. He wishes to thank Edward Barsotti, leader of the Data System Group of CDF from Fermilab, for chosing him to develop this work and supporting its development. He wishes to thank Willian Graves for helping him to correct the manuscripts, Rudy Vabalaitis for doing the figures and Joel Zautner for helping him to test the hardware and software of the Snoop Modules. All these gentlemen are from the Data System Group of CDF of Fermilab. He wishes to thank Jeffrey Marsh from the Software Group of CDF of Fermilab for explaining him features of the VAX/VMS Operating System. He wishes to thank Dr. Leon Lederman and Dr. Roy Rubinstein from Fermilab, for proposing and organizing this joint colaboration Fermilab-Brazil. He wishes to thank the designers of the Snoop Module, Helmut V. Waltz and David B. Gustavson, from the Stanford Linear Accelerator Center, for providing the two Snoop Modules and associated valuable information.

From Brazil, the Author wishes to thank his colleagues of the Electronic Division of the Instituto de Fisica da UFRGS, Miguel Fachin Jr., Mauro Wyrsykoski, Mauro Fin, Arthur Boss Jr., Agostinho Bulla, Celso Muller and Paulo Pedro Petry for handling the extra load of work while he was absent. He wishes to thank Dr. Fernando Zawislak, Dr. Bernardo Liberman, Dr. Edemundo do Rocha Vieira and Dr. James Viccaro from the Instituto de Fisica da UFRGS for giving him valuable support and conditions for his absence for two years from his duties at that institution. He wishes to thank Miguel Fachin Jr., Aderbal Fernandes Lima and Marco Bittencourt from Equipamentos Cientificos e Industriais Ltda. for handling the extra load of work. He wishes to thank the Conselho Nacional de Desenvolvimento Cientifico e Tecnologico for the financial support.

The Author wishes to thank his wife, Marise, his parents, Bruno and Emma, and his brother and sister-in-law Paulo and Cristina, and a friend, Carlos Gary Faria for all the help and support.

S. Z.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

FASTBUS (1)* is a standardized modular data-bus system for data aquisition, data processing and control. A FASTBUS system consists of multiple bus segments which can operate independently or be linked together with Segment Interconnects (see Appendix A) for passing data and other information.

At Fermi National Accelerator Laboratory (Fermilab), a data acquisition system for the Collider Detector Facility (CDF) based on FASTBUS is being developed. It will use about 60 FASTBUS Segments interconnected. Some form of diagnostic system, a kind of FASTBUS Logic State Analyzer, will be necessary to help to find, in real time, software and hardware faults occurring on the bus of this system.

The faults that are expected to happen on FASTBUS will range from very simple faults, like a constant absence of a signal on one FASTBUS line, to very complex faults, such as an error created by a faulty interaction among different FASTBUS Masters.

What tool(s) should be developed to expose these problems? There are currently no adequate tools (software or hardware) for examining improper behavior on FASTBUS. At this moment it is possible to foresee that a system that could record FASTBUS transactions into a silo memory (see

*Numbers in parentheses refer to numbered references in the bibliography.

Appendix B) for later analysis, would be very helpful. The silo memory should be able to record activity on, at least, the most important FASTBUS lines.

The electric connections among FASTBUS modules are made by a set of signal lines called a Segment. The usual situation is that the required functionality at a given location is attained by a number of Modules being grouped together in a Crate in order to share a common backplane bus (see Fig. 1).

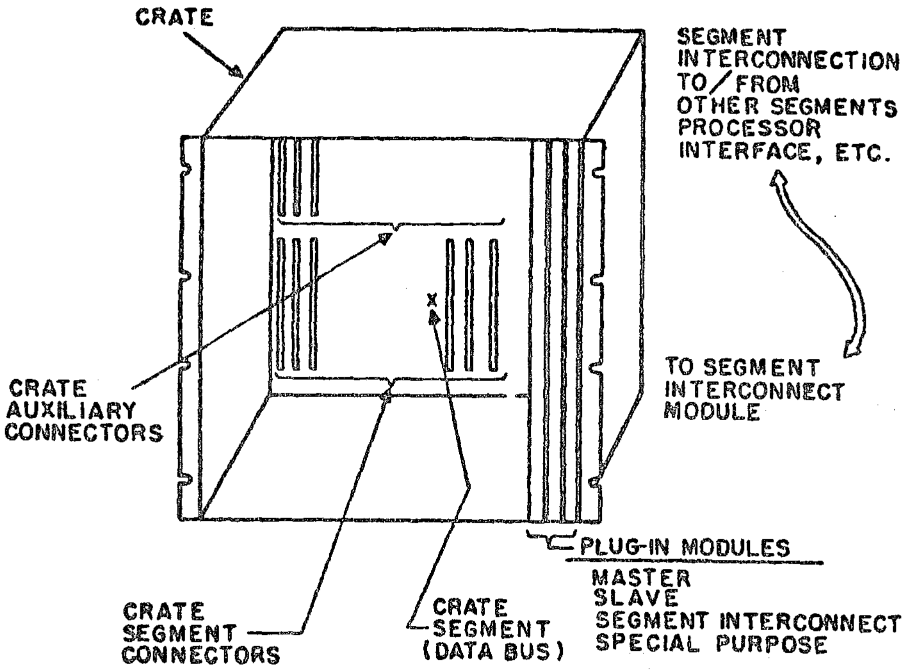

CRATE
SEGMENT INTERCONNECTION TO/FROM OTHER SEGMENTS PROCESSOR INTERFACE, ETC.
TO SEGMENT INTERCONNECT MODULE
CRATE AUXILIARY CONNECTORS
X
PLUG-IN MODULES
MASTER
SLAVE
SEGMENT INTERCONNECT
SPECIAL PURPOSE
CRATE SEGMENT CONNECTORS
CRATE SEGMENT (DATA BUS)

Figure 1. Basic FASTBUS Elements

Multiple Masters on a single Segment share a common bus. Contention for use of a common bus may reduce throughput, as seen by a given Master, because of the time the Master spends waiting to gain Mastership. Since Segments operate independently, distributing the Masters among several Segments can reduce the contention problem. Even so, a Master on one Segment must be able to quickly communicate with a slave on another Segment. This ability is provided by Segment Interconnects (SIs) which temporarily link independent segments (see Fig. 2).

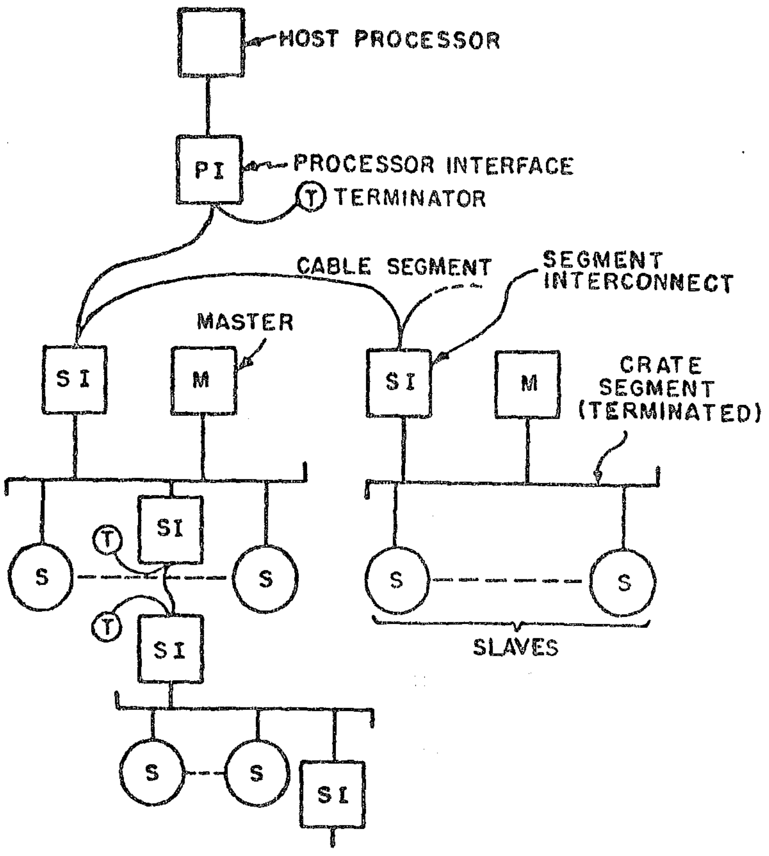


Figure 2. Example of FASTBUS System Topology

A system with this characteristic should have the following facilities:

1. The recording of FASTBUS cycles should be done in one segment or in different segments at the same time, depending upon the needs of the system.

2. It should have trigger capabilities to select only the FASTBUS cycles of interest.
3. It should be able to present to the user the cycles recorded in a comprehensive manner.
4. It should be easy to operate.
5. It should be able to analyze FASTBUS transactions and minimally interfere with the FASTBUS operations.

The Snoop Module (see Appendix B) was developed at the Stanford Linear Accelerator Center. It was used to develop this FASTBUS diagnostic system. The Snoop Module consists of a high-speed emitter-coupled logic (ECL) front-end, with monitoring and test capabilities, controlled by a microprocessor. Some modifications in the hardware of the Snoop Module were done in order to implement the characteristics required by this diagnostic system.

The following steps were done to implement the FASTBUS Analysis System using Snoop Modules:

1. Fully test and debug the Snoop Module.
2. Implement a set of FASTBUS monitoring tests that take full advantage of the flexibility of the Snoop module and are useful to trace FASTBUS problems.
3. Implement specific modifications to the basic circuitry of the Snoop Module.
4. Develop software tools that enable the user, in an easy way, to set the FASTBUS monitoring tests.

5. Network the Snoop Modules in a multi-segment configuration and test their ability to trace FASTBUS problems.
6. Test the ability of this diagnostic system to monitor FASTBUS transactions in different configurations.

CHAPTER II

MONITORING FASTBUS IN THE DATA ACQUISITION SYSTEM

FASTBUS is a very new standard which has never been used in an application of this size before. Consequently, there is no clear understanding of the types of problems that will occur during operation of the system.

To identify and trace such problems, several different forms of monitoring the bus have been implemented. In the next sections, a description of the FASTBUS monitoring capabilities is given and in what specific situations these capabilities are used.

Error Detection and Reporting Specified by the FASTBUS Standard

The FASTBUS standard specifies mandatory and optional error detection and reporting techniques. The FASTBUS Slave Status Responses are one example of mandatory error detection and reporting. The Slave Status Responses are encoded on the FASTBUS SS lines and are sent to a Master by a Slave or Segment Interconnect, along with the appropriate FASTBUS acknowledgement. These responses are used by the Masters to determine their next course of action. Two sets of Slave Status Responses are defined for FASTBUS: one for Address Cycles and another for Data Cycles. The Slave Status Responses for a Data Cycle are shown in Table 1. These responses can be used by the Master to report the type of error found and can help the service person in tracing the fault.

Table 1. Slave Data Time SS Responses

SS<2:0>	Interpretation
0	Valid Action
1	Busy
2	End of Block
3	User Defined
4	Reserved
5	Reserved
6	Data Error - Data Reject
7	Data Error - Data Accept

The FASTBUS standard also specifies some optional error detection mechanisms which report the occurrence of errors using bits in a Control and Status Register (CSR). For example, Table 2 shows bits 8 through 13 of CSR-2 and their significance when a FASTBUS Master reads these bits. A user of a FASTBUS system can read these bits as an aid in determining the cause of the errors which may occur.

Finally, the FASTBUS standard provides a set of user defined error detection and reporting techniques for FASTBUS module designers. See, for example, response SS=3 on Table 1.

Table 2. Partial CSR 2 Bit Assignments

Bit	Significance for a read performed by a FASTBUS Master.
08	Non-existing Address
09	Device Data Overflow
10	Word Count Overflow
11	Device Full
12	Device Not Empty
13	Inputs Not Present

<u>Monitoring Using Display Modules and Test Point Boards</u>

Display Modules (2) are FASTBUS modules which show the logical level of each FASTBUS line. This is accomplished by means of buffers which drive light emitting diodes (LEDs) on the front panel of the module (see Fig. 3). They also have internal circuitry which allows the user to assert the FASTBUS Wait line each time a transition occurs on the AR, AG, AS, AK, DS or DK lines. This provides a means to perform single step operations on FASTBUS.

Test Point boards are boards with the same dimensions as a FASTBUS module. Through a connector, each FASTBUS line is accessible and is interfaced by a buffer to the front panel of the board. The state of all FASTBUS lines can be examined using this board and an oscilloscope.

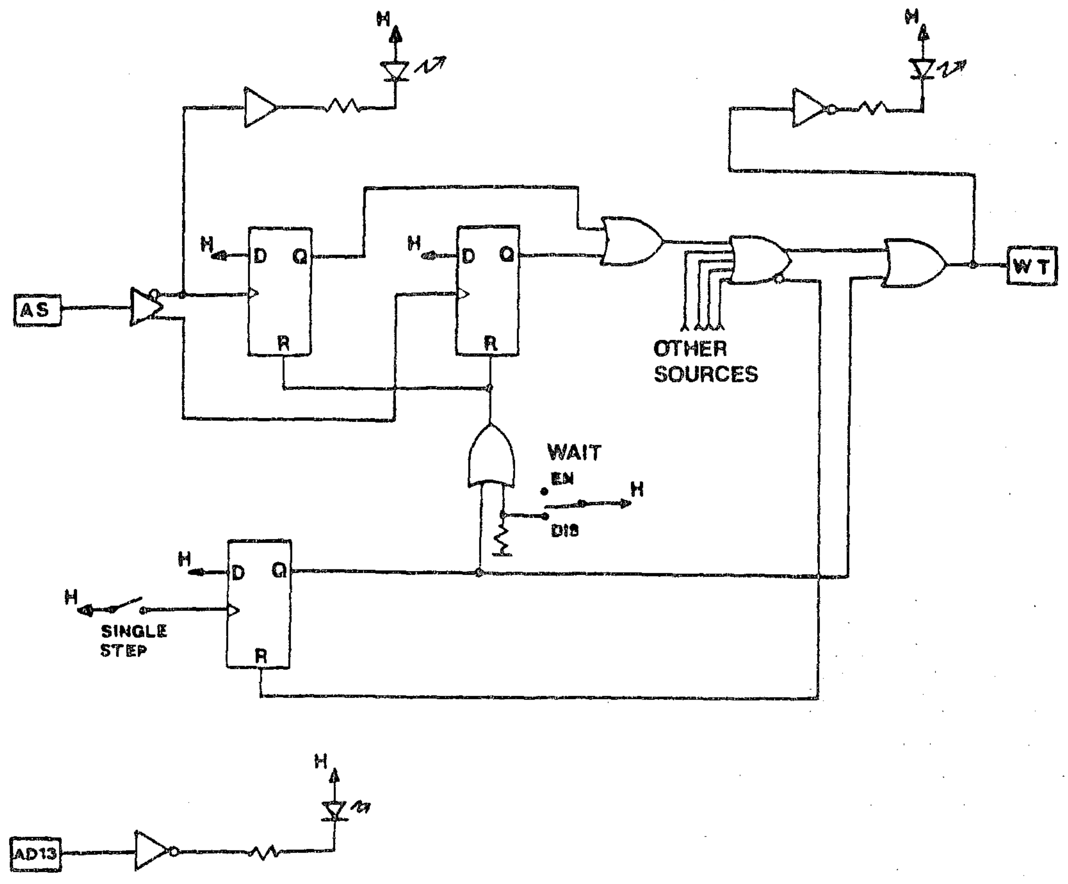


Figure 3. Basic Circuit of FASTBUS Display Modules

The Display Modules and the Test Point boards are useful for the development of FASTBUS modules. However, they proved to be of little value for tracing faults on a huge FASTBUS system. For example, a FASTBUS multi-master system cannot normally be single stepped.

The Need for Another Form of Monitoring: The FASTBUS Logic Analyzer

There are faults which occur on FASTBUS that can only be produced by the data acquisition system itself, or that may be simulated on test systems external to the data acquisition. A logic state analyzer is the main tool used to locate these faults.

However, logic state analyzers are not designed for monitoring FASTBUS. They have the following disadvantages when used to trace problems on FASTBUS:

1. It is difficult to connect them to a FASTBUS Segment.
2. It is difficult to configure them to record FASTBUS transactions in the most efficient way.
3. It is difficult to use them to trace data and control information going from Segment to Segment.
4. Logic state analyzers of good quality are expensive (approximately $25,000).

Because of these disadvantages, the usage of Snoop Modules was implemented to monitor FASTBUS. The Snoop implementation is described in this thesis.

CHAPTER III

HARDWARE CHANGES AND TESTS IMPLEMENTED WITH THE SNOOP MODULES

Some hardware changes had to be implemented on the Snoop Module board in order to provide certain modes of silo memory recording which were not available on the original board. More hardware changes were made to implement additional self test capabilities. Finally, the Snoop Module, itself, was subject to a variety of tests to establish how reliably it functioned.

Hardware Modifications Regarding the Arbitration Cycle

The original circuitry of the Snoop Module did not provide a means to detect specific Arbitration Cycles on the bus. This is an important condition to select which FASTBUS transactions must be recorded into the silo memory.

The Arbitration Logic circuit of the Snoop Module was modified in order to implement this feature. This circuit enables the Snoop Module to request mastership of the bus. Figure 4 shows the original circuit; and Figure 5 shows the circuity with the changes implemented.

The (XOR-NOT) gates are comparators which have their outputs set to one when both the Internal Arbitration Level and the Arbitration Level of the Master which granted bus mastership are equal.

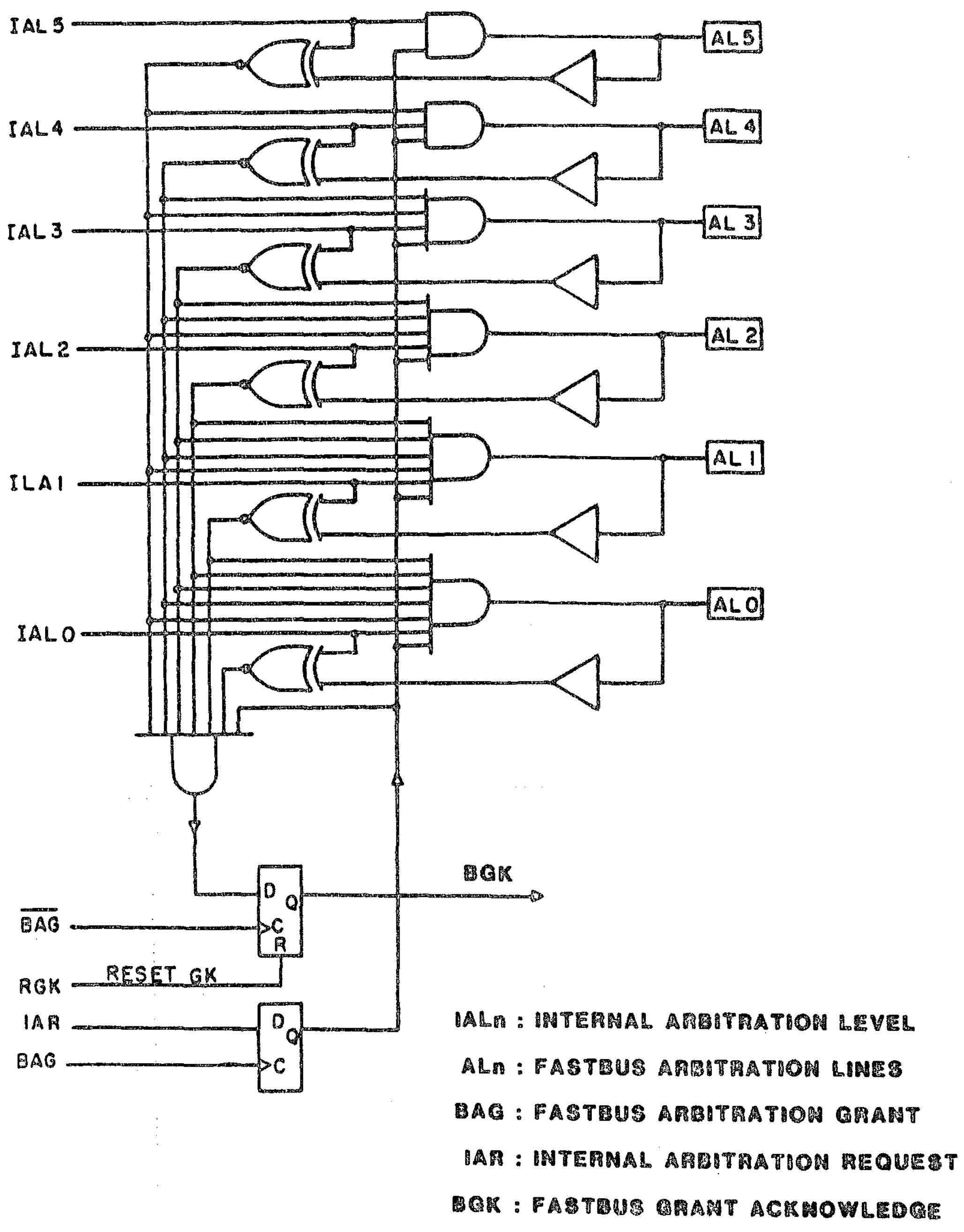
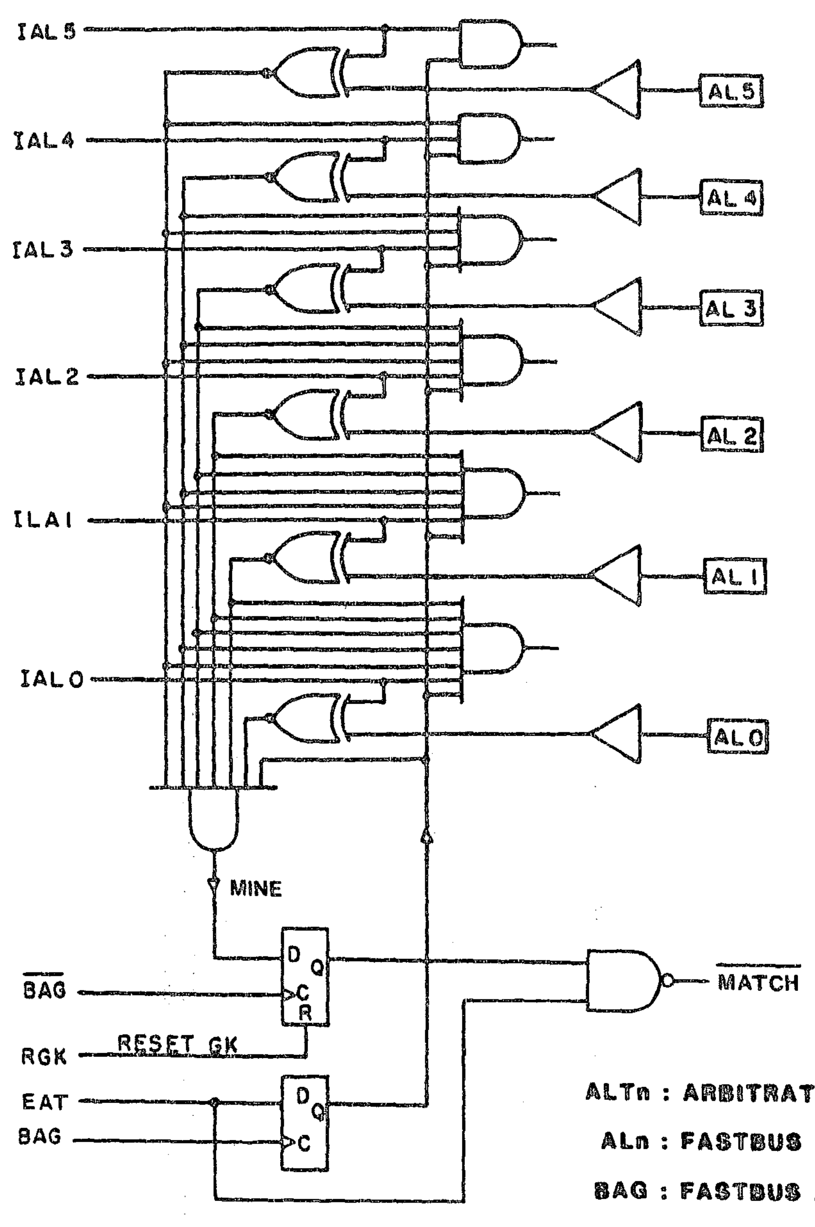

IAL5
IAL4
IAL3
IAL2
ILA1
IAL0
AL5
AL4
AL3
AL2
AL1
AL0
BGK
D
Q
C
R
$\overline{\text{BAG}}$
RGK
RESET GK
IAR
BAG
IALn : INTERNAL ARBITRATION LEVEL
ALn : FASTBUS ARBITRATION LINES
BAG : FASTBUS ARBITRATION GRANT
IAR : INTERNAL ARBITRATION REQUEST
BGK : FASTBUS GRANT ACKNOWLEDGE

Figure 4. Original Arbitration Circuitry in the Snoop Module



ALTn : ARBITRATION LEVEL TRAP
ALn : FASTBUS ARBITRATION LINES
BAG : FASTBUS ARBITRATION GRANT
RAT : RESET ARBITRATION TRAP
EAT : ENABLE ARBITRATION TRAP

Figure 5. Changes Implemented in the Arbitration Circuitry

Hardware Modifications to Record FASTBUS AS, AK, DS and DK lines

When timing monitoring is being done on FASTBUS, it is imperative to record the FASTBUS timing lines (AS, AK, DS and DK) into the silo memory. This feature is necessary to enable the user to determine when the Address and Data Cycles have occurred. This facility was implemented by adding one more 1 K x 4 ECL randon access memory (RAM) to the silo memory.

Hardware Modifications to Increase Self Test Capabilities

Hardware changes were implemented to enable the user to test part of the front-end logic of the Snoop Module, independent of FASTBUS transactions.

The output buffers which enable the Snoop Module to drive the FASTBUS lines (see Fig. 6) were modified as shown in Figure 7. With this new scheme, known test data may be recorded into the silo memory for later checking independent of the FASTBUS transactions.

Test Recording FASTBUS Transactions Into The Silo Memory

Many tests were done with the Snoop Module in order to debug and test the board. One of the tests implemented was done to determine if the trigger and the recording into the silo memory were reliable operations.

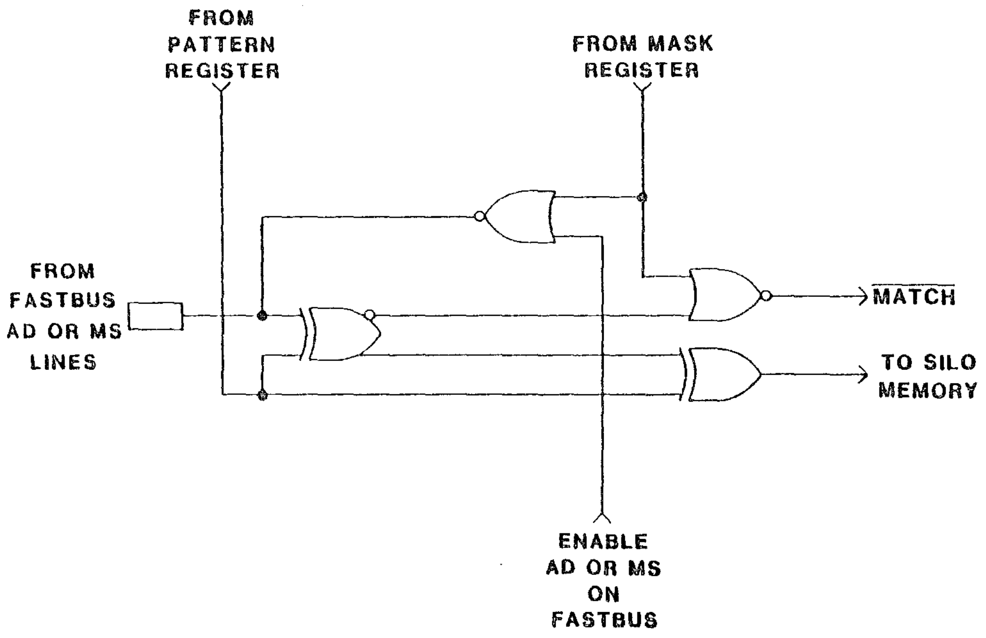


Figure 6. Original Circuit of the Input/Output Buffers Connected to FASTBUS AD and MS Lines

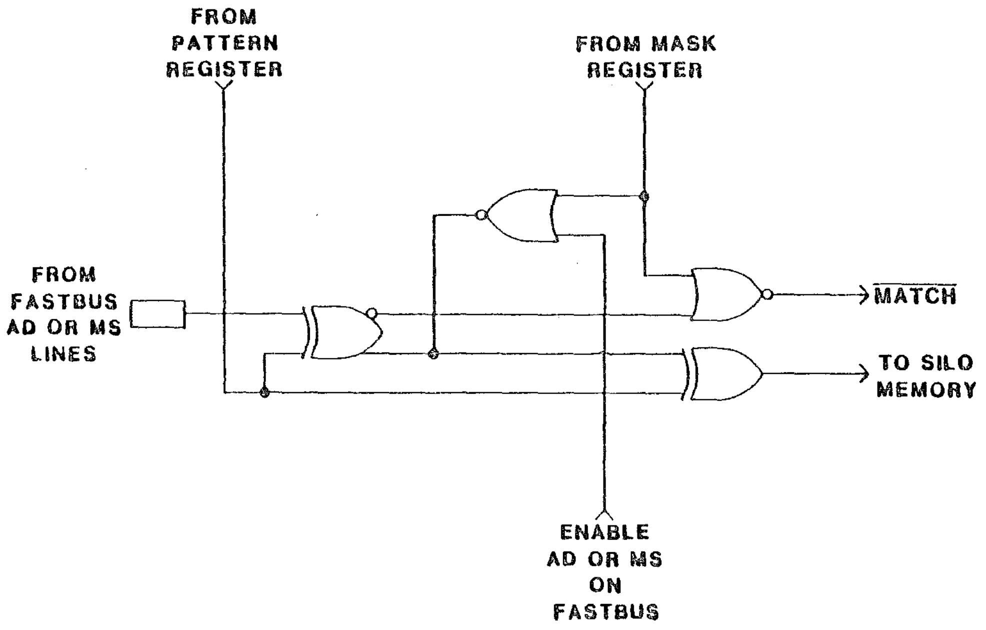


Figure 7. Changes Implemented to Enable Tests with the Silo Memory

A Snoop Module was connected to a FASTBUS Segment where a Master was performing known FASTBUS cycles involving arbitration, address and data with two different Slaves. Two different triggers (see Appendix B) were used to start recording FASTBUS transactions into the silo memory. These two triggers were alternated and used to record different patterns of data in the silo memory. The silo memory was filled using one trigger and then checked for errors; afterwhich, the other trigger was used to refill the silo memory for further error checking. This error checking, which was performed by the control microprocessor of the Snoop Module, was repeated 24 hours a day for two weeks without a single error being detected.

CHAPTER IV

SOFTWARE FOR THE FASTBUS LOGIC STATE ANALYZER

Two different operation modes have been developed for using the Snoop Modules: one consists of a terminal interfaced to a single Snoop Module residing in one Segment, and the other consists of a host computer connected to one or more Snoop Modules residimg on different Segments. The following sections describe these two configurations and their associated software.

The Snoop Module Interfaced to a Terminal

Figure 8 shows a typical configuration of a Snoop Module monitoring a FASTBUS segment when interfaced to a terminal. This mode of operation is well suited for developing FASTBUS modules in a test system or for monitoring single segments of a data acquisition system. Its setup time is minimal and no connection to a host computer is needed.

An on-board microprocessor controls the front-end logic of the Snoop Module through a set of control and status registers (see Appendix B). The application software developed provides the user with a set of commands which enable him to control these registers and read the silo memory. This software was written in two different languages: Forth (3 and 4) and Assembler. Assembler was used in all speed critical points.

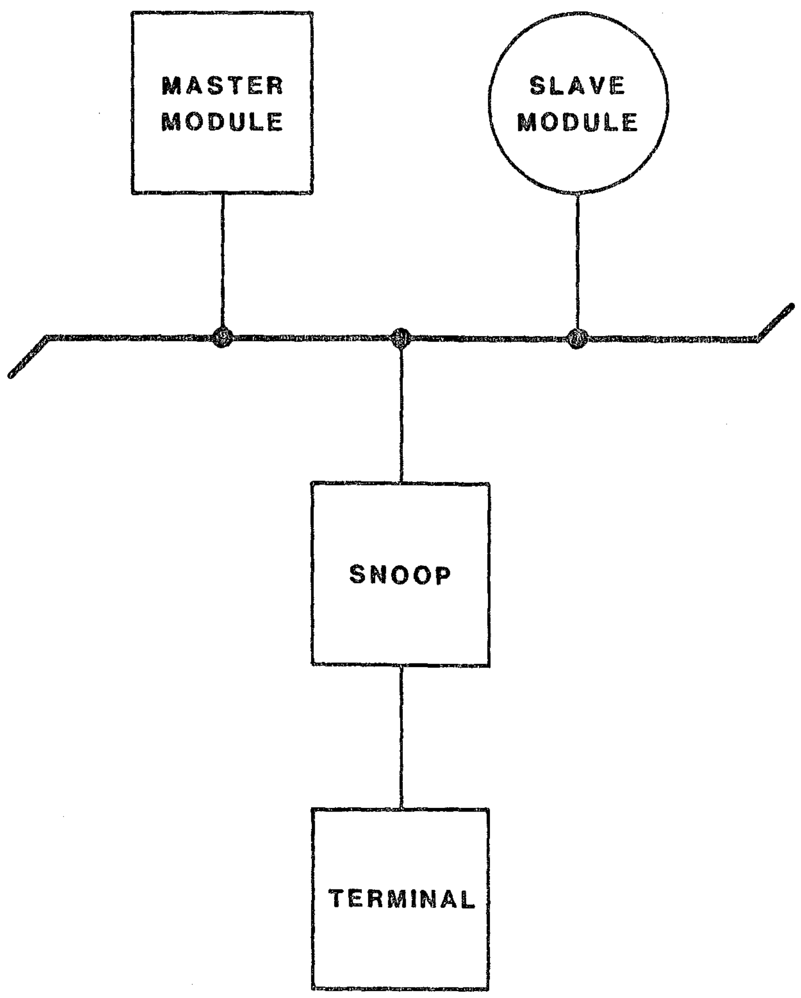


Figure 8. The Snoop Module Interfaced to a Terminal and Connected to One FASTBUS Segment

The commands, normally called words by Forth, allow the user to configure the Snoop Module to record FASTBUS transactions, to observe the different steps in the recording process, and to examine the silo memory. These words give the user complete control of the various monitoring capabilities of the Snoop Module (see Appendix C).

These Forth words are listed below:

1. Words to set the Arbitration, Address and Data Traps:
   a. <pattern (byte)> TP-L!: Set the AL trap for the Arbitration Trap.
   b. <pattern (double word)> <mask (double word)> TP-A-AD!: Set the AD trap for the Address Trap.
   c. <pattern (double word)> <mask (double word)> TP-D-AD!: Set the AD trap for the Data or the Second Address Trap.
   d. <mask (double word)> TP-A-AD-M!: Set the mask of the AD trap for the Address Trap.
   e. <pattern (double word)> TP-A-AD-P!: Set the pattern of the AD trap for the Address Trap.
   f. <mask (double word)> TP-D-AD-M!: Set the mask of the AD trap for the Data or Second Address Trap.
   g. <pattern (double word)> TP-D-AD-P!: Set the pattern of the AD trap for the Data or Second Address Trap.
   h. <pattern (byte)> <mask (byte)> TP-A-MS!: Set the MS trap for the Address Trap.
   i. <pattern (byte)> <mask (byte)> TP-D-MS!: Set the MS trap for the Data or the Second Address Trap.

j. <mask (byte)> TP-A-MS-M!: Set the mask for the MS trap for the Address Trap.

k. <pattern (byte)> TP-A-MS-P!: Set the pattern for the MS trap for the Address Trap.

l. <mask (byte)> TP-D-MS-M!: Set the mask for the MS trap for the Data or Second Address Trap.

m. <pattern (byte)> TP-D-MS-P!: Set the pattern for the MS trap for the Data or Second Address Trap.

n. ?TP: Show all traps set to the trigger sequence chosen.

o. ?TP-L: Get the setup of the AL trap for the Arbitration Trap.

p. ?TP-A: Get the setup for the AD and MS trap for the Address Trap.

q. ?TP-D: Get the setup for the AD and MS trap for the Data or the Second Address Trap.

2. Words to set different Trigger Sequences:

   a. TR-SEQ-LAD: Set arbitration, address and data cycle as trigger sequence.

   b. TR-SEQ-LAA: Set arbitration, address and another address cycle as trigger sequence.

   c. TR-SEQ-LA: Set arbitration and an address cycle as trigger sequence.

   d. TR-SEQ-AD: Set address and data cycle as trigger sequence.

   e. TR-SEQ-AA: Set address and a second address cycle as trigger sequence.
   f. TR-SEQ-A: Set address cycle as trigger sequence.
3. Words to set the Trigger Source:
   a. TR-SRC-I: Set the use of the internal trigger source.
   b. TR-SRC-E: Set the use of the external trigger source.
   c. TR-SRC-M: Set the use of the microprocessor to trigger. The trigger is asynchronous with FASTBUS transactions.
4. Words to set the Position of the Trigger among other transactions recorded:
   a. TR-POS-B: Set the trigger position to the beginning of cycles recorded.
   b. TR-POS-E: Set the trigger position to the end of cycles recorded.
5. Words to select the type of FASTBUS cycles to record:
   a. TY-REC-ALL: Set to record all types of cycles.
   b. TY-REC-NOD: Set to record all type except data cycles.
6. Words to select the type of sampling strobe:
   a. SS-SRC-FB: Set FASTBUS transactions as source for generation of sampling strobes.

   b. SS-SRC-IC: Set the internal 50 MHz clock as source for generation of sampling strobes.
   c. SS-SRC-EC: Set an external clock as source for generation of sampling strobes.

7. Words to read the setup of the fast front-end logic:
   a. ?SET-REC: Show all the setup of the fast front-end logic for the recording of FASTBUS transactions into the silo memory.
   b. ?TR-SRC: Show the trigger source setup.
   c. ?TR-POS: Show the trigger position setup.
   d. ?TR-SEQ: Show the trigger sequency setup.
   e. ?TY-REC: Show the type of cycle to record setup.
   f. ?SS-SRC: Show the sampling strobe source setup.

8. Words to control the silo memory address counter:
   a. <silo address (word)> SIA!: Set the silo address counter.
   b. ?SIA: Read the silo address counter.
   c. SIA1+: Increment the silo address counter by 1.

9. Words to read the silo memory:
   a. <1st address (word)> <no. of cycles to list (word)> SILO: List all FASTBUS lines recorded in the silo memory, starting at 1st address, until the number of cycles to list is

completed.

b. SILOC: Read all FASTBUS lines recorded in the silo memory in the address previous set in the silo memory address counter. After listing, decrement the address counter by one.

c. LID: List the identification of each FASTBUS line shown by SILOC.

10. Words to control the execution of the recording process:

a. START-REC: Record FASTBUS cycles.

b. ?STATUS-REC: Read the status of the recording into the silo memory of the Snoop Module. It stays showing the status until <ESC> is pressed or the recording operation finishes.

c. STOP-REC: Stop the recording into silo memory.

d. REC: Start recording and show the status of the recording. Stop recording when <ESC> is pressed or when the recording process finishes.

Figure 9 gives a typical example of the interaction between the user and the Snoop Module in this mode of operation. Note that comments were added just for illustration and that "OK" is the prompt sent by Forth. Figure 10 shows the Forth words written to control the trigger position when the user is working with this mode of operation.

```
TR-SEQ-AD  OK                 ;Set the trigger sequence:
                              ;address and data cycles.

TR-POS-B  OK                  ;Set trigger position.

?TR-POS                       ;Check trigger position set.
TRIGGER POSITION: Start of FASTBUS cycles recorded   OK

?TP-A                         ;Read the address trap
                              ;previously set.
ADDRESS TRAP: AD pattern: #00000018   AD mask: #00000000
              MS pattern: #1          MS mask: #0
OK
REC                           ;Start the recording operation.
     TRIGGER WAS NOT FOUND    ;Stop this process by typing the
OK                            ;key <ESC>.

#17. TR-A-AD!  OK             ;Change the AD lines trap for
                              ;address trap.
REC                           ;Start again the recording
                              ;operation.
     TRIGGER WAS FOUND        ;Snoop front-end logic found
                              ;the trigger sequence.
     CYCLES BEING RECORDED    ;Record FASTBUS cycles into
                              ;silo memory.
     IT IS NOT RECORDING      ;Finishes automatically the
                              ;recording.
OK

0 5 SILO                      ;Examine some cycles recorded
SIAD AIagGKar AL EGrd    AD     PApe SRwt MSss TOdt
000   0 0 1 0 11  0 0  00000017  1 1  0 0  1 0  0 0
001   0 0 1 0 11  0 0  00000000  1 1  0 0  2 0  0 1
002   0 0 1 0 11  0 1  00A20000  0 1  0 0  0 0  0 1
003   0 0 1 0 11  0 0  00000018  1 1  0 0  0 0  0 0
004   0 0 1 0 11  0 0  00004000  0 1  0 0  2 0  0 1
005   0 0 1 0 11  0 0  12AC3280  1 1  0 0  0 0  0 1
OK
```

Figure 9. Operation of the Snoop Module Connected to a Terminal

```
: TR-POS-E                       ( Set trigger position in the)
                                 ( end of cycles recorded.)
   CB-TR-BG/END*  CCR4! ;        ( Do this resetting bit in)
                                 ( control register 4.)

: TR-POS-B                       ( Set trigger position in the)
                                 ( begin of cycles recorded.)
     CB-TR-BG/END*  SCR4! ;      ( Do this setting bit in)
                                 ( control register 4.)

: ?TR-POS                           ( Show the trigger position)
     ." TRIGGER POSITION: "         ( set.)
     CW4@  CB-TR-BG/END*   AND      ( Do this by reading bit)
                                    ( in control register 4.)
     IF    ." Start of FASTBUS cycles recorded"
     ELSE  ." End of FASTBUS cycles recorded"
     THEN  SPACE ;
```

Figure 10. Example of the Forth Words Written to Control the Trigger Position

The Snoop Module Interfaced to a Host Computer

The employment of a host computer to control the Snoop Module creates numerous interesting possibilities:

a. The use of software utility packages resident in the host to develop the software of the Analysis System, like the menu oriented command interface, software graphics packages, etc.

b. The use of the devices connected to the host like disks, graphic printers, etc.

c. The use of the host computer to analyze the data recorded by the Snoop Modules. For example, if an error happens a few times every hour, a program can be developed for the host to automatically setup the Snoop Modules and check the transactions recorded.

d. The application software developed for the host computer is relatively independent of the Snoop Module hardware, decreasing the number of changes necessary in the software in case of hardware upgrading.

A VAX from Digital Equipment Corporation is used for the host computer in this system. These computers are also used to control the data acquisition system.

Figure 11 shows a possible configuration in which two Snoop Modules controlled by the host computer are monitoring a FASTBUS system. Note that the Unibus Processor Interface (5) is a FASTBUS module which interfaces the VAX and FASTBUS. One typical capability which this configuration provides is the recording of data and control information going from a Master located in one Segment to a Slave located in another. This can be accomplished in the following way: the Snoop Module, closer to the Master, is programmed to detect the trigger sequence of interest, and the other Snoop Module programmed to receive the trigger from the first. When the trigger sequence is detected by the Snoop Module closer to the Master, both modules start recording the bus activity at the same time. Once the recording process is finished, the user can call the host facilities to analyze the FASTBUS transactions.

The application software developed for this type of configuration can be divided into two groups: software resident in the host computer and software resident in the

Snoop Module. This software is described in the following sections. The communication between the host and the Snoop Modules is described at the end of this chapter.

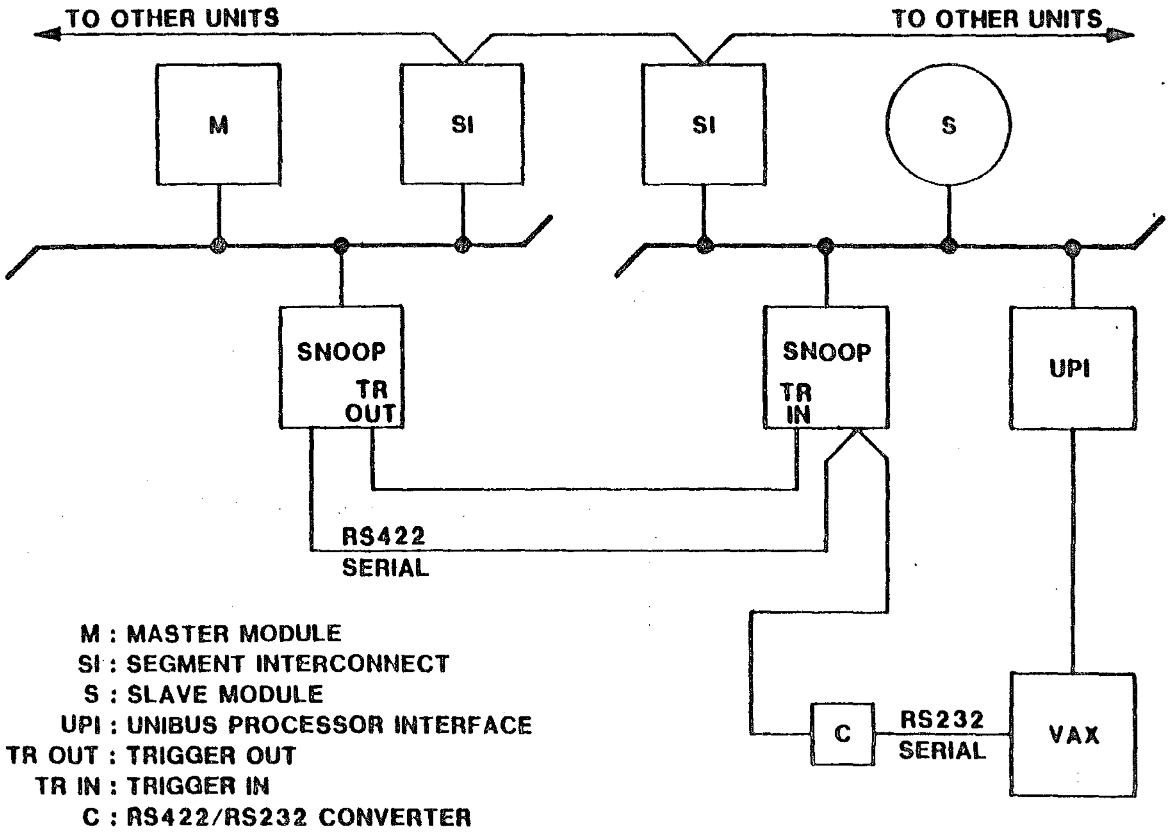


Figure 11. The Snoop Module Interfaced to a Host and Connected to a FASTBUS System.

<u>The Software Resident in the Host Computer:</u> the application software developed for the host computer was written in FORTRAN (6).

This software has three different groups of routines:

1. Routines which enable the user to control the system and analyze the transactions recorded.

2. Routines to control the Snoop Module and read the silo memory.
3. Routines to perform the communication with the Snoop Modules.

The interaction among these groups is shown on Figure 12.

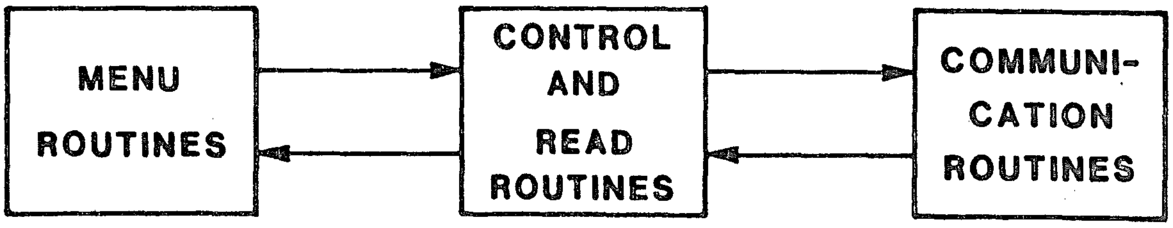


Figure 12. Interactions Among the Groups of Routines

A software package was developed at Fermi National Accelerator Laboratory which enables programs to obtain both the commands and the data for the FASTBUS logic analyzer by means of menus (7). This software package was used to implement the menus for the Analysis System. In total there are five different types of menus:

1. The main menu.
2. The Snoop Module setup menu.
3. The trap words setup menu.
4. The analysis of the recorded FASTBUS cycles menu.
5. The stop recording menu.

Figure 13 shows the main menu. This menu is activated when the FASTBUS logic analyzer software begins running. This menu allows the user to choose the operation which he whishes to execute. Note that just two of the options presented have been developed: options A and B. The other options will be part of future developments and are explained in Chapter V.

```
          ================M e n u================
                FASTBUS LOGIC STATE ANALYZER
                     M A I N   M E N U

A - Setup for the Recording of FASTBUS Transactions

B - Analysis of the FASTBUS Cycles Recorded

C - Setup for the assertions of the Wait Line.

D - Save Information on Disk

E - Call Information from Disk

F - Selftest of the Analysis Modules

==============================================================
```

Figure 13. Main Menu

Figures 14 and 15 show the menus used to setup the Snoop Modules before recording FASTBUS transactions. This menu provides a means of setting-up the front-end logic of the Snoop Modules for different forms of recording. Option D of Figure 14 is of particular interest. In this mode of operation, when the Snoop Modules are interfaced to a host computer, the recording process is completed only when all modules have stopped recording into their silo memory. In the case in which a Snoop Module is located in a FASTBUS segment and no transactions are occurring, this option allows the user to disconnect the Snoop Module from the Analysis System without having to change its internal setup. Also note that this menu consists of two switchable pages.

```
                ================M e n u================
          SETUP FOR THE RECORDING OF FASTBUS TRANSACTIONS
A                    ANALYSIS MODULE NUMBER [n]
B   RECORD FASTBUS TRANSACTIONS
C   EXECUTE THE MENU OF ANALYSIS OF FASTBUS CYCLES RECORDED
D   CONNECTED TO THE ANALYSIS SYSTEM    Option: [Yes/No]
E   TRIGGER SOURCE                      Option: [n]
      1 - Internal trap sequence
      2 - External
      3 - Record Command
F   TRIGGER POSITION                    Option: [n]
      1 - Start of cycles recorded
      2 - End of cycles recorded
G   SET TRAP WORDS
H  NEXT PAGE

Also 1=Previous Menu
============================================================
```

Figure 14. First Page of the Setup for Recording Menu

```
A   TRIGGER SEQUENCE                    Option: [n]
      1 - Arbitration, address and data cycle
      2 - Arbitration, address and a second address cycle
      3 - Arbitration and a second address cycle
      4 - Address and data cycle
      5 - Address and second address cycle
      6 - Address cycle
B   SAMPLING STROBE SOURCE              Option: [n]
      1 - FASTBUS transactions
      2 - Internal clock (50 MHz)
      3 - External clock
C   TYPE OF CYCLE TO RECORD             Option: [n]
      1 - All FASTBUS cycles
      2 - Arbitration and address cycles
D  LAST PAGE

Also 1=Previous Menu
============================================================
```

Figure 15. Second Page of the Setup for Recording Menu

Figure 16 shows one example of the Trap Words Menu. This is a menu with different formats, depending upon the Trigger Sequence chosen. For example, if the trigger sequence chosen was just an Address Cycle, this menu will allow the user to set only an address trap.

```
                ================M e n u================
                           T R A P   W O R D S
Sequence: Arbitration, address and data cycle

A AL pattern for Bus Arbitration Trap  Pattern = [00]

B AD pattern for the Address Trap      Pattern = [00000000]
C AD mask for the Address Trap            Mask = [FFFFFFFF]
D MS pattern for the Address Trap      Pattern = [00]
E MS mask for the Address Trap            Mask = [FF]

F AD pattern for Data Trap             Pattern = [00000000]
G AD mask for Data Trap                   Mask = [FFFFFFFF]
H MS pattern for the Data Trap         Pattern = [00]
I MS mask for the Data Trap               Mask = [FF]

Also 1=Previous Menu
=============================================================
```

Figure 16. Trap Words Menu

Figure 17 shows the menu for the analysis of the FASTBUS transactions recorded into the silo memory. Options A, B and C are presently developed. Option C is useful when the user wants to compare the FASTBUS transactions recorded by one Snoop Module with the transactions recorded by another.

```
                ================M e n u================
            ANALYSIS OF THE FASTBUS TRANSACTIONS RECORDED

A  Record FASTBUS Transactions

B  List of transactions from one Module

C  List of transactions from two Modules

D  Timing Diagram

E  Disassemble list of transactions

Also 1=Previous Menu
=============================================================
```

Figure 17. Analysis of the FASTBUS Transactions Recorded Menu

The last menu developed is the menu to stop the recording process. It is shown in Figure 18. This menu is shown when the user activates the recording process (option B of Figure 14 and option A of Figure 17).

```
               ================M e n u================
                          STOP RECORDING MENU

A  Stop Recording and Return to the Analysis Menu

B  Stop Recording and Return to the Setup Menu

Also 1=Previous Menu
==============================================================
```

Figure 18. Stop Recording Menu

The menus interact with the Snoop Module by means of a set of function subprograms. These function subprograms perform the following:

1. Setup the Snoop Modules.
2. Read the silo memory of the Snoop Modules.
3. Control the recording process.
4. Build different forms of tables to enable the user to analyze the transactions recorded.

A list of these function subprograms and their respective use is given below:

1. Function subprograms to set and read the arbitration trap:

   STATUS=AN_F_set_trap_arb_AL (mod_no,patt)

   STATUS=AN_F_get_trap_arb_AL (mod_no,patt)

2. Function subprograms to set and read the address trap:

   ```
   STATUS=AN_F_set_trap_add_AD_mask (mod_no,mask)
   STATUS=AN_F_get_trap_add_AD_mask (mod_no,mask)
   STATUS=AN_F_set_trap_add_AD_patt (mod_no,patt)
   STATUS=AN_F_get_trap_add_AD_patt (mod_no,patt)
   STATUS=AN_F_set_trap_add_MS_mask (mod_no,mask)
   STATUS=AN_F_get_trap_add_MS_mask (mod_no,mask)
   STATUS=AN_F_set_trap_add_MS_patt (mod_no,patt)
   STATUS=AN_F_get_trap_add_MS_patt (mod_no,patt)
   ```

3. Function subprograms to set and read the data or another address trap:

   ```
   STATUS=AN_F_set_trap_dat_add_AD_mask (mod_no,mask)
   STATUS=AN_F_get_trap_dat_add_AD_mask (mod_no,mask)
   STATUS=AN_F_set_trap_dat_add_AD_patt (mod_no,patt)
   STATUS=AN_F_get_trap_dat_add_AD_patt (mod_no,patt)
   STATUS=AN_F_set_trap_dat_add_MS_mask (mod_no,mask)
   STATUS=AN_F_get_trap_dat_add_MS_mask (mod_no,mask)
   STATUS=AN_F_set_trap_dat_add_MS_patt (mod_no,patt)
   STATUS=AN_F_get_trap_dat_add_MS_patt (mod_no,patt)
   ```

4. Function subprograms to set and read the trigger sequence:

   ```
   STATUS=AN_F_set_trigger_sequence (mod_no,tr_seq)
   STATUS=AN_F_get_trigger_sequence (mod_no,tr_seq)
   ```

5. Function subprograms to set and read the trigger source:

   ```
   STATUS=AN_F_set_trigger_src (mod_no,trigger_src)
   ```

STATUS=AN_F_get_trigger_src (mod_no,trigger_src)

6. Function subprograms to set and read the sampling strobe source:

STATUS=AN_F_set_strobe_src (mod_no,strobe_src)

STATUS=AN_F_get_strobe_src (mod_no,strobe_src)

7. Function subprograms to set and read the position of the trigger:

STATUS=AN_F_set_trigger_pos (mod_no,trigger_pos)

STATUS=AN_F_get_trigger_pos (mod_no,trigger_pos)

8. Function subprograms to set and read the type of cycle recorded:

STATUS=AN_F_set_type_cycles_rec (mod_no,type_rec)

STATUS=AN_F_get_type_cycles_rec (mod_no,type_rec)

9. Function subprograms to control the execution of the recording process:

STATUS=AN_F_set_start_rec

STATUS=AN_F_set_stop_rec

STATUS=AN_F_get_status_rec (mod_no)

10. Function subprograms to build lists to analyze the transactions recorded in the silo memory:

STATUS=AN_F_analyze_list_one_mod (mod_no)

STATUS=AN_F_analyze_list_two_mod (modA_no,modB_no)

11. Function subprograms to connect or disconnect the Snoop Module from the Analysis System:

STATUS=AN_F_set_mod_hiber (mod_no)

STATUS=AN_F_set_mod_wake (mod_no)

STATUS=AN_F_get_mod_dreams (mod_no)

Each of these function subprograms returns to the caller routine a number (STATUS) indicating if the operation was successfully completed.

Figure 19 shows an example of one of the functions subprograms written. In this subprogram the command "Encode" translates a hexadecimal number to ASCII.

Software Executed by the Snoop Modules When Connected with the Host. When the Snoop Module is connected to the host computer, it executes software similar to that which it executes when it is connected to a terminal. The basic difference between the two modes of operation is in the form of communication, which is described in the next section.

The host computer sends Forth words and numbers to the Snoop Modules to control and read the Snoop Module. These words are similar to the words used when the Snoop is connected to a terminal, with the difference that numbers set the different options. When the host reads the setup of the Snoop Module, again, it reads numbers instead of messages. These numbers are the numbers specified in the menus (for example, see Fig. 14 and 15). There are some words which are used in both modes of operation - the first mode is when the Snoop Module is interfaced to a terminal, and the second mode when it is interfaced to a host. Figure 20 shows some examples of these words.

```
      Integer*4 Function AN_F_set_trigger_src
     .                               (mod_no,trigger_src)
c
c     Description
c     ===========
c     Set the trigger source of the Snoop Module
c
c     Call Parameters
c     ===============
      Integer*4        mod_no
      Integer*4        trigger_src
c
c     External Routine: Transmit data to the module.
c     ==============================================
      External         AN_N_output_data
c
c     Parameter Statements
c     ====================
      Include          'ANALYZER_PAR.FOR/nolist'
      Parameter        out_char_no = 9
c
c     Local Declarations
c     ==================
      Integer*4        AN_N_out_data_buffer_size
      Character*(out_char_no) AN_N_out_data_buffer
      Integer*4        ok
c
c     Data Statements
c     ===============
      Data    AN_N_out_data_buffer      / '  TR-SRC!'/
      Data    AN_N_out_data_buffer_size / out_char_no /
c
c     Format Statement
c     ================
100   Format(I1)
c
c     Executable Code
c     ====================================================
      Encode (1,100,AN_N_out_data_buffer) trigger_src
      ok = AN_N_output_data (mod_no,
     .                       AN_N_output_data_buffer,
     .                       AN_N_output_data_buffer_size)
      if ( ok .ne. AN_N_success ) then
         AN_F_set_trigger_src = AN_F_network_error
      else
         AN_F_set_trigger_src = AN_F_success
      endif
      return
      end
```

Figure 19. Example of a Host Computer Function Subprogram to Setup the Snoop Module

n TR-SRC!	-	Set the trigger source
		n = 1: Internal trigger source
		n = 2: External trigger source
		n = 3: Asynchronous trigger

n TR-POS!	-	Set the trigger position among the cycles recorded n = 1: Start of cycles recorded n = 2: End of cycles recorded

?SS-SRC.	-	Read the sampling strobe source set
		n = 1: FASTBUS transactions
		n = 2: Internal clock (50 MHz)
		n = 3: External clock

n TP-L!	-	Set the arbitration trap n = pattern of the trap

Figure 20. Examples of Forth Words Used by the Host to Control and Read the Snoop Module

The Communication Among the Host and the Snoop Modules. The amount of data exchanged between the host computer and the Snoop Modules is small and can be done adequately at low data rates (for example, 9600 bits per second). Therefore, a simple form of communication was developed using serial lines. Another feature provided by this simple communication technique is the ability to send broadcast messages. This is important to synchronize the operations of different Snoop Modules.

The basis of this serial communication technique is the Binary Synchronous Communication (BISYNC) protocol from International Business Machines, which "provides a set of rules for synchronous transmission of binary coded data" (8). To develop this serial communication protocol, the control characters of BISYNC, as well as, a subset of its data transfer rules were used. Figure 21 shows the interconnection of the host computer with the Snoop Modules.

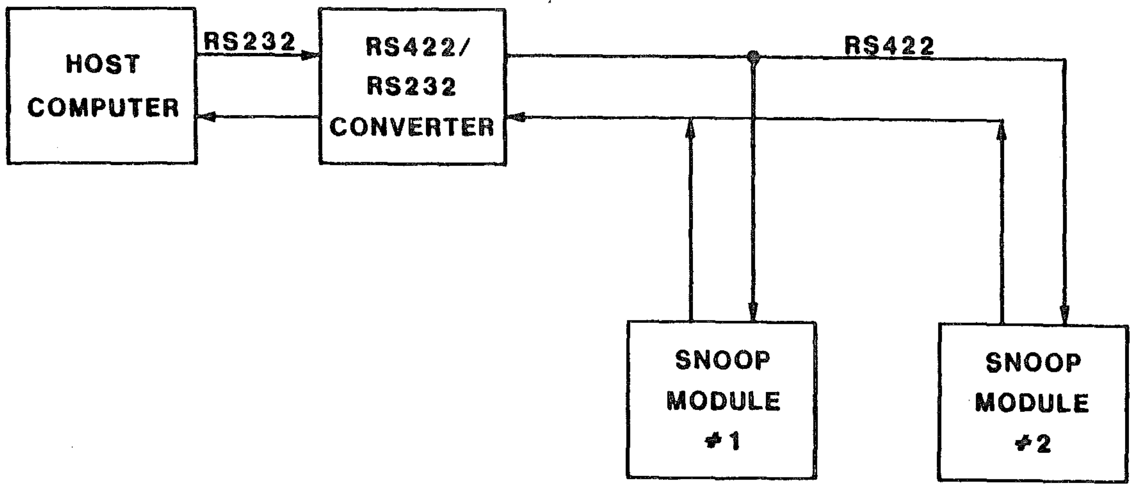


Figure 21. The Serial Interconnection of the Host and the Snoop Modules

The protocol consists of the following:

1. The communication is asynchronous and uses only ASCII characters.
2. A centralized multipoint environment is used, with the host being the control station and the Snoop Modules the tributary stations.
3. The control station regulates all transmission by means of polling and selection. By sequentially polling each tributary station, the control station directs the incoming message traffic. The outgoing traffic is regulated by the control station selecting the desired tributary station to receive the message. All transactions are between the

control station and the selected tributary station.

4. The operation to start any communication is accomplished by the control station transmitting the following message:

   [EOT Polling or selection ENQ PAD]

5. The polling or selection field is made up of 3 characters which contain the necessary information to perform polling or selection with different modules. The first two characters are the address of the tributary station in ASCII and the third is a P for polling and a S for selection. If the module number is smaller the 10, it is sent a space and the decimal module address.

6. The control or the tributary station can be in the receiving or transmitting mode, depending on whether data or control must be received or transmitted by the station.

7. The control station can send only one block of text per selection. If it has to send more characters than can be supported by the block size, it has to select the tributary again for additional transmission of another message block.

8. The meaning of the control characters for the FASTBUS logic analyzer serial communication protocol are as following:

a. STX - START OF TEXT: This character precedes a block of text characters. Text is that portion of a message treated as an entity to be transmitted through to the ultimate destination without change.

b. ETX - END OF TEXT: The ETX character terminates the text. ETX requires a reply indicating the receiving station status, with the exception of the broadcast message.

c. EOT - END OF TRANSMISSION: This character indicates the end of a message transmission, which may contain one or more text messages. It causes a reset of all stations on the line. It is also used to respond to a poll when the polled station has nothing to transmit.

d. ENQ - ENQUIRY: The ENQ character is used to obtain a repeat transmission of the response to a message block if the original response was garbled or was not received when expected. It also indicates the end of a poll or selection sequence.

e. ACK - AFIRMATIVE ACKNOWLEDGEMENT: This reply indicates that the previous block was accepted without error and the receiver is ready to accept the next block of transmission.

f. NAK - NEGATIVE ACKNOWLEDGEMENT: NAK indicates that the previous message was received in error and the receiver is ready to accept a retransmission of the erroneous block.

g. PAD - PAD CHARACTER: A PAD character is added following each transmission to ensure that the last significant character is sent before the data set transmitter turns off. BELL character (hex '07') is used as PAD.

9. The parity bit is use to validate the characters received by the tributary and control station.

10. Address 0 is the broadcast address. All tributary stations recognize this address and accept the text if it is correct. There is no reply from the tributary stations.

11. A timeout is used to limit the waiting time tolerated by the control station to receive a reply. When a timeout occurs, the control station will again send its previous message and wait for the reply. If a specific number of timeouts occurs for the same message, the control station assumes that the tributary is damaged or disconnected from the network, and aborts the transmission.

12. The control station checks for errors in the transmission, or replies, indicating that errors are being detected by the tributary stations. If these errors happen repeatedly, it aborts the

operation and reports the fault. Some examples of this error detection are the control stations repeatedly receiving NAK as replies when a text is being transmitted or repeatedly receiving ENQ as an answer for an ACK after a text was successfully received.

Based on these rules, the process for reading some data from the Snoop Modules is the following: first the host has to select the Snoop Module and indicate which data it needs to read; after this it has to poll the Snoop Module and read the data. Figures 22, 23 and 24 show examples of this protocol.

```
CTRL    [EOT  2 S ENQ PAD]        [STX Text ETX PAD] ..
TRIB_1                                               ..
TIRB_2                   [ACK PAD]                   ..

CTRL    ..         [ENQ  1 P ENQ PAD]                   ..
TRIB_1  ..                                              ..
TRIB_2  .. [ACK PAD]                [STX Text ETB PAD] ..

CTRL    .. [ACK PAD]                [ACK PAD]
TRIB_1  ..
TRIB_2  ..         [STX Text ETX PAD]      [EOT PAD]
```

Figure 22. Typical Data Link Message Traffic

```
CTRL    [EOT  2 S ENQ PAD]         [STX Text ETX PAD]          ..
TRIB_2                    [ACK PAD]                 [NAK PAD]..


CTRL    ..[STX Text ETX PAD]        [EOT PAD]
TRIB_2 ..                   [ACK PAD]
```

Figure 23. Data Link with a NAK

```
CTRL    [EOT  1 S ENQ PAD]         [STX Text ETX PAD]          ..
                                                     (Timeout)
TRIB_1                    [ACK PAD]                             ..


CTRL    ..[STX Text ETX PAD]        [EOT PAD]
TRIB_1 ..                   [ACK PAD]
```

Figure 24. Data Link with the Occurrence of Timeout

Two function subprograms were developed for the host which enable it to communicate with the Snoop Modules: a function to send data to the modules and another to read data from the modules. These functions are invoked by the following:

```
STATUS = AN_N_input_data(mod_no,in_buffer,in_buffer_size,
                          no_char_received)
STATUS = AN_N_output_data(mod_no,out_buffer,out_buffer_size)
```

Each of these function subprograms returns a number (STATUS) to the caller routine indicating if the operation was successfully completed. If some error was detected in the operation, this number corresponds to the error detected.

Figure 25 shows part of the software that performs the operation which starts the communication with a tributary. Note that the System Service Routines of the VAX/VMS computer (9) are called to perform the following tasks:

1. Assigning a serial port to this process (sys$assign).
2. Transmiting and receiving of the characters exchanged with the Snoop Module (sys$qio and sys$qiow).
3. Checking for timeout (sys$schdwk).
4. Causing the process to go into hibernation (sys$hiber).
5. Stoping the process in case of error in one of the previous operations (call lib$stop).

As it has already been explained, the Snoop Module has to be able to interface to a terminal or to the host. In order to accomplish this goal, the communication protocol was made switchable between one mode which performs communication with a terminal and another mode which permits communication with the host (see Appendix C).

The protocol necessary to perform the communication directly with a terminal already exits in the Forth software package resident on the Snoop Module - it is the standard way that Forth communicates with the user.

```
c       Assign the serial port SNOOP$PORT to this
c       process if it was not previously assigned.
c
        if ( .not. assign_done ) then
           ok = sys$assign ('SNOOP$PORT',serial_port, ,)
           if ( .not. ok ) call lib$stop ( ok )
           assign_done = .true.
        endif
c
c       Queue a read to the serial port to receive the
c       answer from the tributary. When the answer is
c       received, an interruption is generated and the
c       subroutine "AN_N_sel_int" or "AN_N_poll_int" is
c       executed, depending if it is selection or polling.
c
        ok=sys$qio (,%val(serial_port) ,
       .                %val(io$_ttyreadall .or. io$m_purge) ,
       .                %ref(io_status_block) ,
       .                AN_N_sel_int , ,       !or AN_N_poll_int
       .                %ref(in_buffer),%val(in_buffer_size) , ,
       .                %ref(terminator_mask) , , )
c
c       Send the inicialization message selecting or polling
c       the tributary station: [EOT mod_no S or P ENQ PAD].
c
        byte_count = 1
        out_buffer (byte_count) = EOT
        byte_count = byte_count + 1
        out_buffer (byte_count) = mod_no_ascii (1:1)
        byte_count = byte_count + 1
        out_buffer (byte_count) = mod_no_ascii (2:2)
        byte_count = byte_count + 1
        out_buffer (byte_count) = 'S'        !or 'P'
        byte_count = byte_count + 1
        out_buffer (byte_count) = ENQ
        byte_count = byte_count + 1
        out_buffer (byte_count) = PAD
        ok=sys$qiow (,%val(serial_port) ,
       .                 %val(io$_writevblk.or.io$m_noformat) ,
       .                 %ref(io_status_block) , , ,
       .                 %ref(out_buffer) , %val(byte_count)
        if ( .not. ok ) call lib$stop ( ok )
c
c       Wait for the answer by turning the process into
c       hibernation. Schedule a wake up to test timeout.
c       The timeout flag is turn false in the int. routine.
c
        timeout_flag = .true.
        ok = sys$schdwk ( , , timeout_period , )
        if ( .not. ok ) call lib$stop ( ok )
        ok = sys$hiber ( )
```

Figure 25. Part of the Communication Functions for the VAX

CHAPTER V
FUTURE DEVELOPMENTS

There are more developments involving the FASTBUS Logic Analyzer which Fermilab plans to pursue in the next year. It is anticipated that the use of this FASTBUS logic analyzer facility will point towards other implementations which test and diagnose errors on FASTBUS and are not presently identified.

Part of these new developments were pointed out in the discussion about the menus in Chapter IV. These future developments are:

1. Routines to allow the Snoop Module to control the assertion of the FASTBUS Wait line.
2. Routines to save and recall information from the disk of the host computer.
3. Routines to provide selftest capability to the Snoop Module.
4. Routines to show the FASTBUS transaction recorded in the silo memory in timing diagram form and in a FASTBUS disassembled form. As an example of a FASTBUS transaction expressed in disassembled form see Figure 26.

```
SIAD AIagGKar  AL  EGrd      AD     PApe  SRwt  MSss   OPERATION
000    0 0 1 0 11   0 0  00000017   1 1   0 0   1 0 Primary Add
001    0 0 1 0 11   0 0  00000000   1 1   0 0   2 0 Secondary Add
002    0 0 1 0 11   0 1  00A20000   0 1   0 0   0 0 Randon Read
003    0 0 1 0 11   0 0  00000018   1 1   0 0   0 0 Primary Add
004    0 0 1 0 11   0 0  00004000   1 1   0 0   2 2 ERROR=Parity
005    0 0 1 0 11   0 0  12AC3280   1 1   0 0   0 0 ERROR=Timeout
```

Figure 26. Example of FASTBUS Transaction Expressed in Disassembled Form

CHAPTER VI
CONCLUSIONS

The FASTBUS Analysis System described in this thesis was developed at Fermilab. It was incorporated in systems similar to the configurations shown in Figures 8 and 11. They were used to test the ability of this Analysis System to record FASTBUS messages being exchanged between Masters and Slaves. The Analysis System worked properly.

The data acquisition system being built for the Collider Detector at Fermilab is already in operation in a simple configuration and a detailed implementation is presently being completed. On October 13, 1985, it was used for detecting a proton-antiproton collision for the first time.

From the experience gained in tracing software and hardware faults in the FASTBUS of the present configuration of this data acquisition system, it has been established that this Analysis System will be very useful in diagnosing various classes of FASTBUS faults.

The complexity of Electronic Systems is increasing very rapidly due to the need for increased processing speed and the low cost of electronic hardware. Other distributed processing systems which use techniques similar to this data acquisition system are being developed around the world. The work developed in this thesis is an approach that can be used by these other systems to trace faults on their system buses.

APPENDIX A

FASTBUS DESCRIPTION

This Appendix is a copy of parts of the FASTBUS specification manual (1).

Introduction

FASTBUS is a standardized modular multi-master data-bus system for data acquisition, data processing and control applications. A typical FASTBUS system consists of multiple Crate Segments which operate independently but connected together for passing data and other information. FASTBUS can operate asynchronously using a handshake protocol to reliably accomodate different speed devices without prior knowledge of their speed. It can also operate synchronously without handsake for transfer of data blocks at maximum speed. It also has a multiplexed bus of data and address of 32 lines. These basic characteristics give to FASTBUS Systems a high throughput and speed of operation.

Most FASTBUS design features stem from a consideration of the requeriments of contemporary data acquisition systems. The need for high speed is met by providing for parallel operation of many processors which can communicate with each other as well as with data acquisition and control devices. The communication protocol used by processors and devices has a large data and address field and is defined in an implementation-independent manner so as to be able to take advantage of advances in technology. The need for flexibility is met by a modular design which readily permits many options in system configuration.

Modular instrumentation systems are distinguished by the method used to interconnect the devices that form the system. Mechanical, electrical, and logical aspects of the connection have to be specified. The electrical connections are made by a set of signal lines called a SEGMENT. While FASTBUS DEVICES can be simply connected by CABLE SEGMENTS, such an arrangement may incur speed penalties. The more usual situation is that the required functionality at a given location is attained by a number of MODULES grouped together in a CRATE in order to share a common backplane bus (Fig. 1). This bus, called a CRATE SEGMENT or SEGMENT, like the CABLE SEGMENT forms a logical element of a FASTBUS system.

Using the FASTBUS protocol, a SEGMENT functions as an autonomous bus interconnecting one or more MASTER DEVICES with a number of SLAVE DEVICES. All bus operations involve a MASTER-SLAVE relationship between the initiator, which must be a MASTER, and the responder, which must be a SLAVE. A MASTER is capable of requesting and obtaining control of the SEGMENT to which it is connected in order to communicate with the SLAVE. If the communication is with another MASTER then, for the duration of the operation, the responding MASTER acts as a SLAVE. A SLAVE cannot gain bus mastership but can make a Service Request that a MASTER on the same SEGMENT can use to initiate a procedure to service the request. MASTERS have a more versatile interrupt mechanism in that they can gain bus mastership and write an interrupt message to an interrupt service device. With multiple MASTERS on a SEGMENT, techniques must be provided to resolve concurrent requests for use of the bus. Each MASTER is assigned an Arbitration Level to use during Arbitration Cycles. In response to timing signals from the SEGMENT Arbitration Timing Controller, circuitry in each MASTER determines which of the contending MASTERS will next be granted bus mastership. No time penalty is usually associated with this arbitration procedure since the next MASTER can be selected before the current MASTER completes its operation.

Multiple MASTERS on a single SEGMENT share a common bus. Contention for use of this bus may reduce throughput as seen by a given MASTER because of the time its spends waiting to gain Mastership of a busy bus. Since SEGMENTS operate independently, distributing the MASTERS among several SEGMENTS can reduce the contention problem and increase throughput to the extend that the information needed by each MASTER can be localized on its SEGMENT.

A MASTER on one SEGMENT must also be able to quickly communicate with a SLAVE on another SEGMENT. This ability is provided by SEGMENT INTERCONNECTS (SIs) which temporarily link independent SEGMENTS (Fig. 2). All SEGMENTS through which the operation passes must be available at the same time in order to complete an intersegment operation. The arbitration mechanism, along with circuitry in each SI, extends the resolution of bus contention problems to off- as well as on-SEGMENT MASTERS. Since one SEGMENT can be linked to any of a number of different SEGMENTS, system configurations can be implemented which optimize time-critical data paths.

While most, if not all, MASTERS will have some processing ability, the FASTBUS system design also envisages the connection of large and small computers to the system. Such a connection is made by a PROCESSOR INTERFACE which gains entry to the FASTBUS system through either a CABLE or a CRATE SEGMENT (Fig. 2). System requirements dictate that each system contain one processor which has complete knowledge of the structure of the system. In particular, it must be able to access every SEGMENT of the system and know how the SEGMENTS are to be interconnected. This processor, called the HOST, initializes the system by telling each side of each SI what operations it is to pass on to its other SEGMENT. By using GEOGRAPHICAL ADDRESSING, the HOST can ascertain the physical location and type of each DEVICE in the system and, as needed, assign LOGICAL ADDRESSES to the DEVICES. LOGICAL ADDRESSES allow a DEVICE to use an INTERNAL ADDRESS Field matched to its needs which is independent of position within a SEGMENT.

The principal characteristics and capabilities of FASTBUS can be summarized as follows:

1. Speed limited only by propagation and logic delays (typically better than 10 MHz for ECL)
2. Large Address and Data Fields (32 bits)
3. Segmented Bus to allow parallel processing
4. System-wide commun
5. Block transfers with or without handshake
6. Uniform system-wide protocol
7. Interrupt and arbitration features.

The FASTBUS operations make use of a multiline bus whose signal assignments are as indicated in Table 3. A CABLE SEGMENT consists of the group of 60 lines at the top of the list while a CRATE SEGMENT includes in addition the other listed lines as well as power lines.

FASTBUS Operations

Most FASTBUS operations begin with a MASTER requesting and being granted bus mastership. The MASTER then selects a SLAVE by a primary address cycle and follows this by any number of data transfer cycles after which the bus is released.

The primary address cycle is started by the MASTER asserting the SLAVE'S address on the 32 Address Data (AD) lines followed by Address Sync (AS) (see Fig. 27). This assertion of the address word sets up a path, through SEGMENT INTERCONNECTS if necessary, between MASTER and SLAVE. When the SLAVE recognizes its address, it responds with the Address Acknowledge signal (AK).

Table 3. FASTBUS Signals

Mnemonic	Signal Name	Use	Number
AS	Address Synchronism	T	1
AK	Address Acknowledge	T	1
EG	Enable Geographical	CT	1
MS	Mode Select	C	3
RD	Read	C	1
AD	Address/Data	I	32
PA	Parity	I	1
PE	Parity Enable	I	1
SS	Slave Status	I	3
DS	Data Synchronism	T	1
DK	Data Acknowledge	T	1
WT	Wait	A	1
SR	Service Request	A	1
RB	Reset Bus	A	1
BH	Bus Halted	C	1
AG	Arbitration Grant	TA	1
AL	Arbitration Level	IA	6
AR	Arbitration Request	A	1
AI	Arbitration Request Inhibit	CA	1
GK	Grant Acknowledge	TA	1
TX	Serial Line Transmit	S	1
RX	Serial Line Receive	S	1
GA	Geographical Address Pins (position encoded, not bussed)	F	5
TP	T Pin (not bussed)	X	1
DL	Daisy Chain Left	X	3
DR	Daisy Chain Right	X	3
TR	Terminated Restricted Use	X	8
UR	Unterminated Restricted Use	X	2
FP	F Pins (Free use, not bussed)		4
R	Reserved		5

Table 3.(continued)

Description of Symbols

T	= Timing for address and data cycles
C	= Control for address and data cycles
I	= Information for address and data cycles
CT	= Control and timing
A	= Asynchronous - timing not directly related to data transfers
TA	= Timing for Arbritation bus
IA	= Information for Arbitration bus
CA	= Control for Arbitration bus
S	= Serial data, timing independent of parallel bus
F	= Fixed information - constant
X	= Special Purpose

The protocol requires that AS and AK remain asserted until the operation is completed.

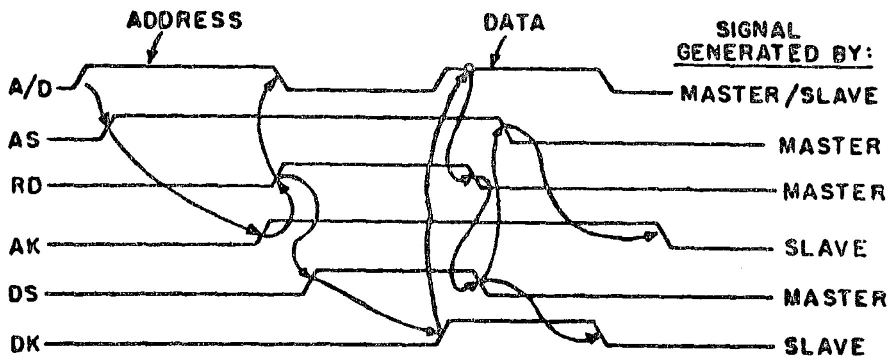


Figure 27. Basic Handshake Read Operation (As Seen by Master)

On receipt of the AK response from the SLAVE, the MASTER removes the address information from the AD lines and uses these lines for data during the ensuing data transfer cycles. After the AS/AK lock between MASTER and SLAVE has thus been established, a Read operation can be initiated by the MASTER asserting the Read (RD) and Data Sync (DS) lines as in Fig 27. The SLAVE responds by placing data on the AD lines and issuing DK which is used by the MASTER to latch the data. For a Write operation, the MASTER asserts data on the AD lines and follows this assertion by the Data Sync (DS). The SLAVE responds by issuing a Data Acknowledge (DK). The operation is terminated by the MASTER removing all its signals, including AS, from the bus. The SLAVE, sensing the removal of AS, removes all its signals including AK.

Since Address and Data Cycles are easily distinguishable, the three Mode Select lines MS<2:0>, are used by the MASTER to modify the meaning of the address information and to independently specify the type of data transfer. In a primary address cycle, control or data space can be specified as well as single or multiple listener (Broadcast) mode. In a data cycle, random data, secondary address, or handshake or pipelined (non-handshake) block transfer can be specified.

Similarly the three Slave Status information lines, SS<2:0>, are used to indicate the success or reason for failure of an Address or a Data Cycle. Addressing difficulties can occur at SEGMENT INTERCONNECTIONS because the SI does not respond (Network Failure) or cannot gain access to its Far-side SEGMENT (Network Busy) or gets preempted by a higher priority operation (Network Abort). Bus lockup caused by unused addresses on the destination SEGMENT are avoided by timers in the MASTER and in the SI which places the address on the destination SEGMENT.

During a Data Cycle, in addition to being able to indicate that it can either accept no more data or has no more data to send, a SLAVE can also signal that it is currently busy or that it has detected one of several classes of error.

Arbitration for Bus Mastership

One of the most important requirements of a multi-processor system is a method for allocating control of the SEGMENTS to the various MASTERS which may be contending for bus Mastership simultaneously. Part of the circuitry to accomplish this, which resides on each independent SEGMENT, is called the Arbitration Timing Controller (ATC).

Ten bus lines are dedicated to SEGMENT priority arbitration. Each master is assigned a six-bit Arbitration Level. Masters wishing to gain bus mastership assert the Arbitration Request line (AR).If the Grant Acknowledge, GK, line is not asserted, the ATC starts an Arbitration Cycle by asserting the Arbitration Grant, AG, signal. Requesting Master respond by asserting their arbitration levels on to the six Arbitration Level Lines AL<5:0>. On each Arbitration Level line an asserted bit will override any non-asserted bits. Each requester continually compares its Arbitration Level with the code on the AL lines bit-by-bit from most to least significant positions. If a requester detects a bit on the bus which it is not asserting, it removes from the bus all of its own bits of lesser significance. After a time determined by the ATC, only the highest Arbitration Level remains asserted on the AL lines and each competitor knows if it has won or lost. When the Arbitration Timing Controller has determined that the bus is completely free (AS=AK=WT=GK=0) it stops asserting AG and the winning Master responds by asserting GK and assuming bus Mastership. The MASTER continues to assert GK until it is willing to allow another Arbitration Cycle. This is usually after the last Address Cycle of its sequence of operarions thus allowing the next MASTER to be selected before the current MASTER has finished its Data Cycles.

Of the 64 possible priority codes, zero is not used because it is easily confused with an idle bus. Codes 1 through 31 are available for use within the SEGMENT, and 32 through 63 are available for use as "system" priorities for unique assignment within communicating parts of a system. The Local priorities 1-31 must be assigned uniquely to DEVICES within a given SEGMENT, but can be reused on every SEGMENT. When a SEGMENT INTERCONNECT connects a MASTER to another SEGMENT, the Level used for arbitration on the second SEGMENT will normally be that of the SI rather than that of the originating MASTER. However, if one of the system priorities was used by originating MASTERS, the SI will propagate that priority onto the second SEGMENT, which it is free to do since the system priorities are unique along a route. The system priorities can be useful in preventing undue delay for important Broadcasts, and can help expedite important messages which might otherwise suffer from fluctuating priorities as they form paths through the system.

No interruption or preemption of the current operation is possible. A MASTER is free to keep the bus as long as it wishes. If it sees AR=1 while AS=AK=1, it knows that other MASTERS in the system are being blocked by the current operation. The controlling MASTER should normally release the bus within a reasonable time in order to allow other MASTERS to acquire bus Mastership. It should allow either one Arbitration Cycle or a random Retry Delay to occur before again requesting bus Mastership. The general solution to the problem of contention and deadlock in FASTBUS is to give up and retry after a random delay.

Control and Status Registers

Certain registers and functions in DEVICES need to be separated in address space from the normal data registers in a way which provides some protection from accidental access and which does not interfere with the allocation of addresses to the normal data portions of the DEVICES. For example, two memory DEVICES should be able to have their addresses set so that the memories are adjacent in address space, allowing them to be used as one larger memory. However, they may contain control registers and status registers associated with memory protection or error detection and correction, and these registers must also be accessible. Furthermore, it is desirable that DEVICES have basic status and information registers in standard locations so that they can be readily accessed by standard shared programs.

The method chosen to accomplish this is to select control/status register (CSR) space in a primary address cycle by suitable coding of the MS lines. This is followed by a secondary address cycle to select a register in CSR space, and a data cycle to transfer to or from the registers. Secondary addressing provides a full 32-bit address for use within a DEVICE, which is enough address space so that it can easily be allocated in standard ways without fear of a shortage. Standard locations in dedicated CSR registers are specified for all the usual control and status bits. DEVICES are required to contain an identifier unique to the DEVICE type which is used during system initialization. This identifier is located in status register 0 so that even simple DEVICES with no address decoders can respond correctly with little added cost.

Segment Interconnects

A SEGMENT INTERCONNECT monitors the activity on the two SEGMENTS it connects, waiting for an address to appear which is in the set of addresses it has been programmed to recognize. It responds to a recognized address asserted on one of the SEGMENTS (Near side) by requesting use of the other SEGMENT (Far side) and asserting the given address on that SEGMENT when it gains control. The two SEGMENTS remain locked together until the operation is complete. The address asserted on the Far side may, in turn, be recognized by another SEGMENT INTERCONNECT and may be passed to yet another SEGMENT. An arbitrary number of SEGMENTS can be linked as needed for a given operation. The address contains all the information needed to direct the appropriate SIs to form the correct connections.

When a MASTER initiates FASTBUS operation it always starts an internal Response Timer set to timeout at a time appropriate for the SEGMENT on which it resides. If the operation has to pass through one or more SIs, the MASTER must be made aware that additional delays will be encountered before a response is received. Any SI passing an operation asserts WAIT (WT) on the SEGMENT from which the operation arrived and starts a timer suitable for the SEGMENT to which the operation is passed. The WT signal causes a MASTER (and the SIs acts as a MASTER on the SEGMENT to which it passes an operation) to stop its timer. The timer is reinitialized when the WT signal is removed. In this way an operation can work its way through a system without causing timeouts to occur unless, of course, a SEGMENT is reached which neither gives a normal response nor asserts the WT signal.

Interrupts

An interrupt is a request from a DEVICE to a processor for service or attention. Since Interrupts may have to cross SEGMENT boundaries, and since they must carry information, they are handled by normal FASTBUS operations.

The interrupting DEVICE addresses an interrupt-sensing control register region in a processor interface and writes its own address and possibly other information into the registers. The processor then has all the information needed to access the interrupting device and service it at some later time.

Ancillary Logic on a Segment

The implementation of a Segment requires circuitry which is common to all Devices on the Segment. This Ancillary Logic controls the execution of Arbitration Cycles, monitors Address cycles and flags Geographical Addresses on the Segment with EG, generates the System Handshake for Broadcast operations, issues signals to halt activity on the Segment when the Run/Halt switch is set, provides logic ones and zeroes for encoding tha GA pins and provides terminators at both ends of the bus for most signal lines. Like a Master, the Ancillary Logic has to be aware of the timing characteristics of the bus to which it is attached.

APPENDIX B

SNOOP MODULE DESCRIPTION

This appendix was written based on articles published (10, 11 and 12) and on interviews with Hemut V. Walz and David B. Gustavson, the designers of the Snoop Module.

Introduction

The FASTBUS Snoop Module has been devised for diagnosing problems inside crate segments, and for monitoring communications from segment to segment in FASTBUS systems. Since all bus-segment signal lines are accessible at each crate module location, such a diagnostic module may be used to monitor and record in a silo memory FASTBUS transactions within a crater segment. The FASTBUS wait line (WT) may be used to single-step bus cycles and implement programmable trap functions.

Snoop Module Organization

The basic hardware organization of the Snoop Module is show in Figure 28. A fast front-end section connects the module to the crate segment bus. This section handles diagnostic recording and control of the crate segment, with response capability to match the fastest device on the bus. It also provides interface and control for master-slave operation and connection to the serial bus lines. Hardware realization is on emitter coupled logic (ECL) in general, with all speed-critical parts implemented with 100 K ECL circuits and a high degree of parallelism. Control and supervision of the fast front-end section is handled by a compact microprocessor section, which includes a second, general-purpose, UART-type serial port. A powerful 16-bit CPU (MC68000) has been selected to optimize handling of serial communications, interrupt driven control of the front-end diagnostic functions and master-slave operations, and replacement of random logic by firmware-based processor control throughout the module.

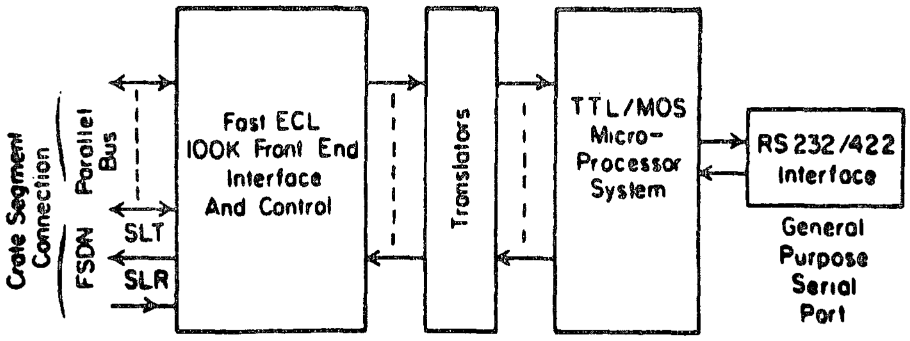

Crate Segment Connection
Parallel Bus
FSDN
SLT
SLR
Fast ECL 100K Front End Interface And Control
Translators
TTL/MOS Micro-Processor System
RS232/422 Interface
General Purpose Serial Port

Figure 28. Snoop Module Organization

The module organization combines the low functional complexity and extensive hardware parallelism of the fast ECL front-end with the high level of integration of the processor section, achieving a single-width module implementation. A detailed block diagram is show in Figure 29. Based on this block diagram, a description of important design details is offered in the following two sections.

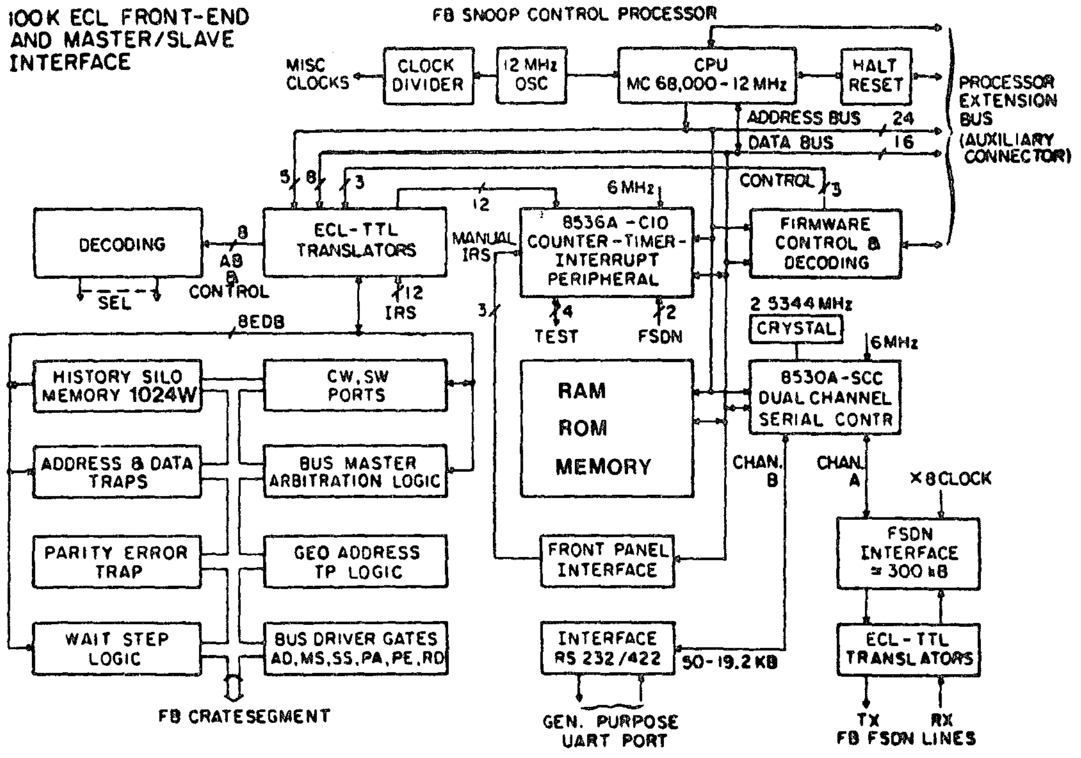


Figure 29. Snoop Module Block Diagram

Fast ECL Front-End

The fast ECL front-end contains the basic diagnostic funtions of the Snoop Module: programmable wait-step logic, traps for address, address-data, address-address, and parity-error detection, activity history silo memory, and master-slave interface logic. To allow processor control of this section, control and status word registers, address decoding, and ECL-TTL level conversion are also provided. This section is implemented with a mixture of 100 K and 10 K ECL integrated circuits.

Address-Data and Parity-Error Traps. The address-data rap is illustrated in Figure 30. The 32 AD lines, and the S<1:0> lines are compared with a pair of 34 bit registers ıt address and data sync times. The contents of these registers allows each bit to be specified as 0, 1 or "don't care". The trap may be used as an address trap, an address-data or an address-address combination trap. The combination trap can be utilized to detect extended address cycles and trigger the recording in the silo memory or assert the FASTBUS wait line.

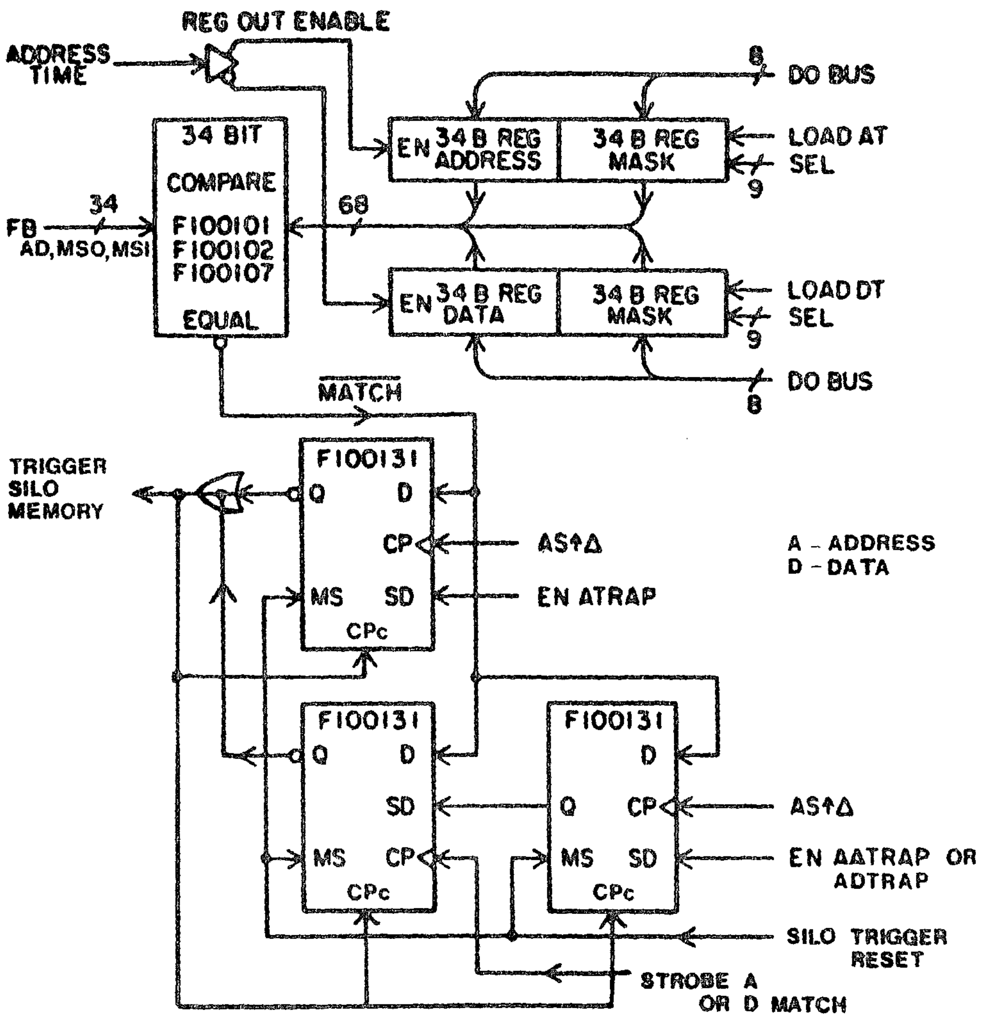


Figure 30. Address and Data Trap Logic

The response time for address or data detection is 8 ns. The parity-error trap consists of a 33 bit parity checker (F100160) driving an array of five parallel flip-flops. These flip-flops are clocked by appropriately delayed timing signals and their outputs are used as wait sources driving the WAIT output line. The response time for parity-error detection is 15 ns.

Activity History Silo Memory. The silo memory is able to record 1 K FASTBUS cycles with a speed in excess of 50 MHz (Fig. 31).

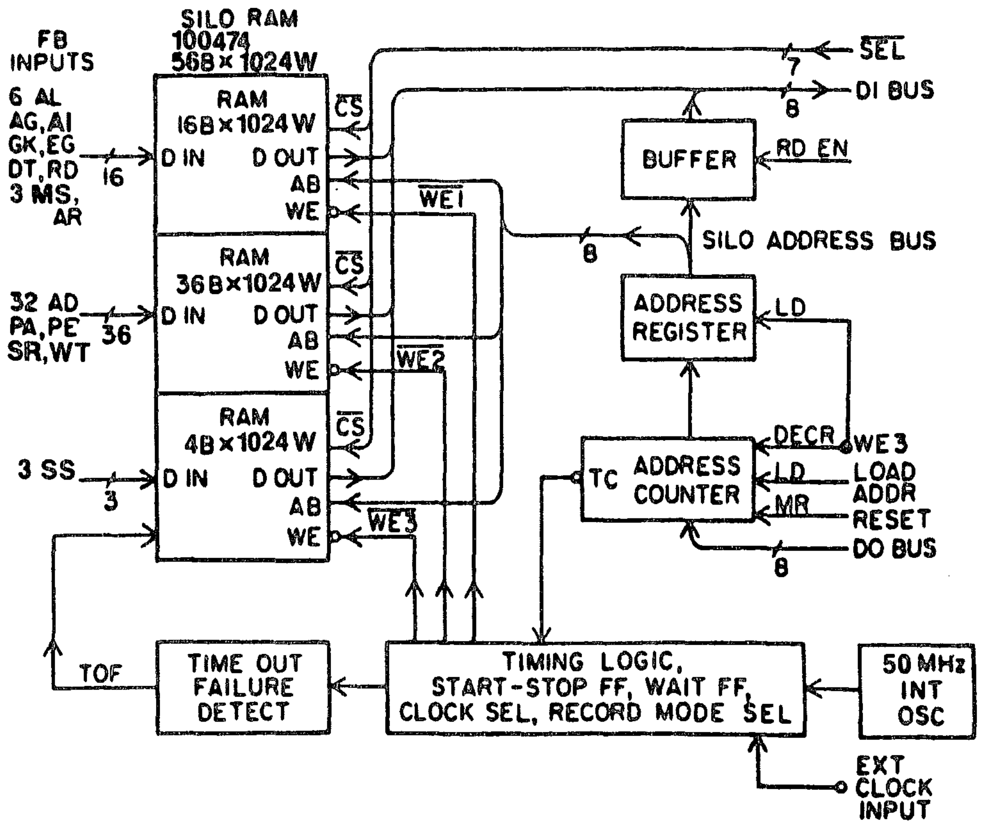


Figure 31. Activity History Silo Memory

For each cycle, 55 bus signals and a time-out failure bit (TOF) are recorded. Several programmable modes for start and stop of silo recording are available. For example, the address-data trap, described previously, may be used to start recording with automatic stop and wait generation when the memory is filled. Choices of FASTBUS-synchronized or real-time clock recording modes are available. For read-out the silo is addressed from the processor and each word is multiplexed onto the 8-bit data bus to the processor in 7 bytes. The data path from the crate segment bus lines through the silo memories onto the data input bus to the processor is also used during WAIT = 1 to read the bus status. Table 4 shows the FASTBUS lines recorded.

Table 4. FASTBUS Signals Recorded in the Silo Memory

AL<5:0>	- Arbitration Level
AG	- Arbitration Grant
AI	- Arbitration Request Inibit
GK	- Grant Acknowledge
AR	- Arbitration Request
RD	- Read
MS<2:0>	- Mode Select
PA	- Parity Enable
PE	- Parity Enable
SS<2:0>	- Slave Status
SR	- Service Request
WT	- Wait
EG	- Enable Geographic Address
AD<32:0>	- Address/Data

Programmable Wait-Step Logic. The wait-step logic asserts the WAIT line in response to bus timing and control signal transitions. The selection of signals used as trigger inputs is made by setting enable bits in two control word registers. Available trigger sources are AS, AK, DS, DK, AG, GK, address-data trap, and parity-error trap.

A typical wait circuit is shown in Figure 32. The wait response delay is 5 ns.

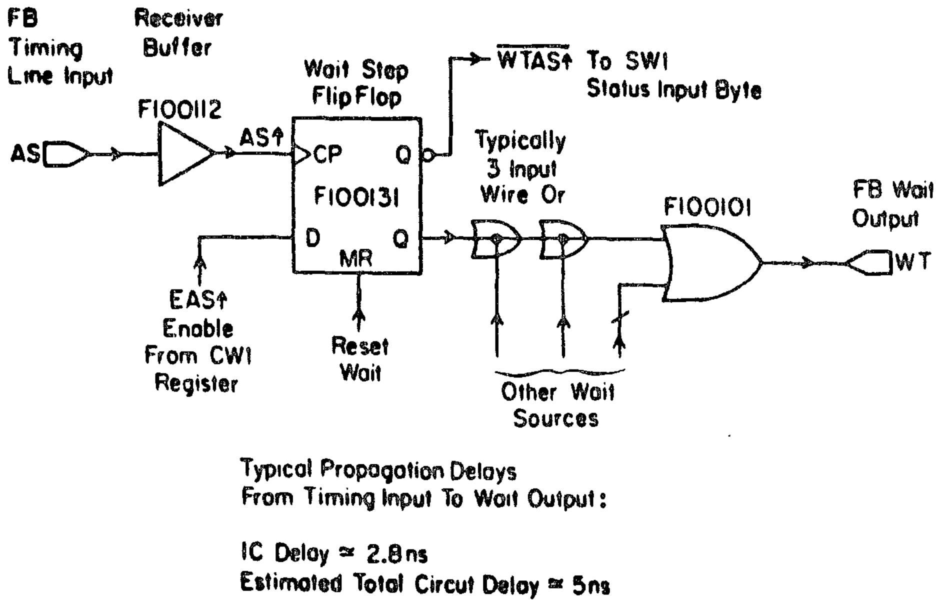

FB Timing Line Input
Receiver Buffer
F100112
AS
AS↑
Wait Step Flip Flop
CP
Q
F100131
D
MR
Q
WTAS↑ To SW1 Status Input Byte
Typically 3 Input Wire Or
F100101
FB Wait Output
WT
EAS↑ Enable From CW1 Register
Reset Wait
Other Wait Sources
Typical Propogation Delays From Timing Input To Wait Output:
IC Delay ≈ 2.8ns
Estimated Total Circut Delay ≈ 5ns

Figure 32. Typical Wait-Step Logic

Master-Slave Interface Logic. To support master-slave capability of the Snoop Module, hardware is provided for bus priority arbitration with a control word register for the module arbitration level. For slave mode, geographic address recognition is implemented. FASTBUS protocol response and generation is handled with processor interrupts, status word input buffers and control word output registers. Master-slave operation is described in more detail in some of the next sections.

Control Processor

Processor design is the result of the following Snoop Module requirements:

1. Highly compact implementation in order to accomodate all module hardware on one PC board.
2. Multi-channel, programmable, high-speed interrupt handling.
3. Maximum CPU execution speed to optimize throughput and minimize memory requirements.
4. CPU with 32 bit registers and instructions to handle 32-bit wide FASTBUS data.

The implementation uses approximately 25 integrated circuit packages. For the CPU the MC68000 (12 MHz) microprocessor was chosen.

Two interrupt handling schemes are combined. The 15 interrupt inputs from the fast front-end section and the module front panel are processed by a Zilog 8536A CIO. Interrupt sources from the dual serial IO port (Zilog 8530A SCC) and CIO are connected into a Zilog-type daisy chain configuration. Both interrupt handlers are then connected to the interrupt control inputs of the CPU.

The serial ports are available from the SCC controller unit. By means of jumpers the serial interface standard is selected from RS232 and RS422 formats. Standard modem control signals are also available. Receiver, transmitter and status interrupts are generated to the CPU.

Software

The control microprocessor of the Snoop Module executes an EPROM based Forth. This Forth, besides its standard words, has an MC68000 Assembler compiler defined. This Assembler was written in Forth and also uses the reverse polish notation.

APPENDIX C

SNOOP MODULE SOFTWARE

This appendix lists the software developed for the control microprocessor of the Snoop Module. This software was written in Forth and Assembler. Assembler was used in all speed critical points.

The words developed for this application were written in the VAX computer, using the the VAX Editor (13) to create the disk file. After this they were downloaded to the Snoop Module through a Fortran written routine. This Fortran routine read a line of this file and sent it to the Snoop Module through a serial port. If the Snoop Module answered with a message that was different than the prompt OK, the entire message was displayed on the terminal. This allowed the program grammar to be debugged.

After downloading these words and debuging them, the Forth Vocabulary was increased permanently by transfering the entire program to an EPROM Programmer and writing a new set of EPROMs.

```
0. #18000. #4000 LCMOVE     ( Memory map EPROM Forth into RAM)
1 #FEFF95. LC!
: NECHO #7CA #D22 ! ;       ( Turn of the Echo)
NECHO
FORGET NECHO                ( Forget the NECHO)
#1EF8 DP !                  ( Change deposit pointer)
: ECHO #BEC #D22 ! ;
: DOCOL #962 , ; IMMEDIATE ( Enables definition be executed)

( ********************************************************)

(       Useful Words and Constants)
(       ========================)

(       Description)
(       ===========)
(       Useful Forth Words for this application)

1   CONSTANT .TRUE.
0   CONSTANT .FALSE.
1   CONSTANT .YES.
0   CONSTANT .NO.

: BASE@       BASE @ ;
: BASE!       BASE ! ;

: DCONSTANT   CREATE , , DOES> DUP @ SWAP 2+ @ SWAP   ;
: DVARIABLE   CREATE 4 ALLOT DOES> ;

: L2!         2SWAP 2OVER L! ROT ROT 2. D+ L! ;
: L2@         2DUP 2. D+ L@ ROT ROT L@ ;

: JUST-CR     #0D EMIT ;

: ARRAY       ( DEF WORD ARRAY: <SIZE> ARRAY <NAME> => )
              2 cra DOES> 2 cal ;

( ********************************************************)

(       Description)
(       ===========)
(       The next words are the words done for the control)
(       of the Fast Fornt-End logic of the Snoop Module.)

( =========================================================)

(       Simbolic addressing for the Snoop Module)
(       ====================================)

( Labeling Description:)
( A=addr, SI=silo, B=byte, CW=control word, SW=status word)

#FEFFE9.  DCONSTANT ASIB1
#FEFFED.  DCONSTANT ASIB2
```

```
#FEFFE1.  DCONSTANT ASIB3
#FEFFE3.  DCONSTANT ASIB4
#FEFFE5.  DCONSTANT ASIB5
#FEFFE7.  DCONSTANT ASIB6
#FEFFEB.  DCONSTANT ASIB7
#FEFFC3.  DCONSTANT ACW1
#FEFFC5.  DCONSTANT ACW2
#FEFFC7.  DCONSTANT ACW3
#FEFFC9.  DCONSTANT ACW4
#FEFFCB.  DCONSTANT ACW5
#FEFFCF.  DCONSTANT ACW7
#FEFFD1.  DCONSTANT ACW8
#FEFFC5.  DCONSTANT ASW1
#FEFFC7.  DCONSTANT ASW2
#FEFFC9.  DCONSTANT ASW3

#FEFFCD.  DCONSTANT ATPL    ( AL pattern for the L trap)
#FEFFDD.  DCONSTANT ATPDMS  ( MS mask and pattern for D trap)
#FEFFDF.  DCONSTANT ATPAMS  ( MS mask and pattern for A trap)
#FEFFF1.  DCONSTANT ATPDAD  ( AD mask pattern for D trap)
#FEFFE1.  DCONSTANT ATPAAD  ( AD mask pattern for A trap)

#FEFFC1.  DCONSTANT ASIALO  ( A0 to A7 of Silo Address)
#FEFFC3.  DCONSTANT ASIAHI  ( A8 to A9 of Silo Address)

#FEFF89.  DCONSTANT ACIO-PA  ( Port A of CIO)
#FEFF8B.  DCONSTANT ACIO-PB  ( Port B of CIO)
#FEFF8D.  DCONSTANT ACIO-PC  ( Port C of CIO)
#FEFF8F.  DCONSTANT ACIO-C   ( Control of CIO)


(       Control and Status Word Bits)
(       ============================)

(       Control Word 2)

#1   CONSTANT CB-TR-RES-L
#40  CONSTANT CB-SEL-AA/AD*
#80  CONSTANT CB-SEL-SEQ/SIN*

(       Control Word 3)

#4   CONSTANT CB-SILO-AD-RES
#8   CONSTANT CB-MP-SILO-AD-DEC
#10  CONSTANT CB-MP-START-REC
#20  CONSTANT CB-MP-STOP-REC

(       Control Word 4)

#1   CONSTANT CB-FB/RT*-SEL
#2   CONSTANT CB-EXT/INT*-CP
#4   CONSTANT CB-REC-DAT
#8   CONSTANT CB-TR-ST/SP*
```

```
(       Control Word 5)

#2  CONSTANT CB-SILO-EN*
#40 CONSTANT CB-ITREN
#80 CONSTANT CB-EXTREN

(       Control Word 7)

#4  CONSTANT CB-L-TP

(       Status Word 1)

#1  CONSTANT SB-BAR
#2  CONSTANT SB-BAS
#4  CONSTANT SB-BDS
#8  CONSTANT SB-BDK

(       Status Word 3)

#1  CONSTANT SB-REC*

(       Zilog CIO)

#1  CONSTANT CB-SILO-TR*
#2  CONSTANT CB-0422-EN

(       Status message numbers)
(       ======================)

#10  CONSTANT MES#-NO-REC-PROC-EN
#11  CONSTANT MES#-NO-TR-INPUT-EN

(       Other Constants)
(       ===============)

#7FF CONSTANT TIME

( ==========================================================)

(       Variables)
(       =========)

VARIABLE  REC-PROC-EN   ( .TRUE., => START_REC was typed)
VARIABLE  EXTREN        ( .TRUE., => external trigger)
VARIABLE  DREAMS        ( .TRUE., => Module in hibernation)

VARIABLE  CW1                ( Control words)
VARIABLE  CW2
VARIABLE  CW4
VARIABLE  CW5
VARIABLE  CW7
VARIABLE  CW8

VARIABLE   TPL               ( AL pattern for L trap)
```

```
DVARIABLE  TPDAD-PATTERN    ( AD pattern for D trap)
DVARIABLE  TPDAD-MASK       ( AD mask ffor D trap)
DVARIABLE  TPAAD-PATTERN    ( AD pattern for A trap)
DVARIABLE  TPAAD-MASK       ( AD mask for A trap)
VARIABLE   TPDMS-PATTERN    ( MS pattern for D trap)
VARIABLE   TPDMS-MASK       ( MS mask for D trap)
VARIABLE   TPAMS-PATTERN    ( MS pattern for A trap)
VARIABLE   TPAMS-MASK       ( MS mask for A trap)

( ==========================================================)

(        Useful words to Display Numbers)
(        =============================)

: HEX->ASCII     ( Converts one hex number <0..F> to ASCII)
      DUP 9 > IF #37 + ELSE #30 + THEN ;

: NHU.           ( List a unsigend hexadecimal nibble number)
     #F AND HEX->ASCII EMIT ;

: BHU.           ( List a unsigned hexadecimal byte number)
       BASE@ HEX SWAP 0 <# # # # # #>
       SWAP 4 + SWAP 4 -
       TYPE BASE! ;
: DHU.           ( List unsigned hexadecimal double number)
     BASE@ HEX ROT ROT <# # # # # # # # # #> TYPE BASE! ;

(        Useful words for this application)
(        ================================)

(        Load the respective control word form the memory)
(        to the top of Satck.)

: CW1@   CW1 @ ;
: CW2@   CW2 @ ;
: CW4@   CW4 @ ;
: CW5@   CW5 @ ;
: CW7@   CW7 @ ;
: CW8@   CW8 @ ;

(        Set the respective control word and save in)
(        memory)

: CW1!   DUP CW1 ! ACW1 LC! ;
: CW2!   DUP CW2 ! ACW2 LC! ;
: CW3!   ACW3 LC! ;
: CW4!   DUP CW4 ! ACW4 LC! ;
: CW5!   DUP CW5 ! ACW5 LC! ;
: CW7!   DUP CW7 ! ACW7 LC! ;
: CW8!   DUP CW8 ! ACW8 LC! ;

(        Set or clear one bit in a control word)

: SCW1!  CW1@ OR CW1! ;
: CCW1!  #FF XOR CW1@ AND CW1! ;
: SCW2!  CW2@ OR CW2! ;
: CCW2!  #FF XOR CW2@ AND CW2! ;
: SCW4!  CW4@ OR CW4! ;
: CCW4!  #FF XOR CW4@ AND CW4! ;
: SCW5!  CW5@ OR CW5! ;
: CCW5!  #FF XOR CW5@ AND CW5! ;
: SCW7!  CW7@ OR CW7! ;
: CCW7!  #FF XOR CW7@ AND CW7! ;
: SCW8!  CW8@ OR CW8! ;
: CCW8!  #FF XOR CW8@ AND CW8! ;

(        Toggle one bit in a control word)

: TCW1!  CW1@ XOR CW1! ;
: TCW2!  CW2@ XOR CW2! ;
: TCW4!  CW4@ XOR CW4! ;
: TCW5!  CW5@ XOR CW5! ;
: TCW7!  CW7@ XOR CW7! ;

(        Read to the top of stack a status word)

: SW1@   ASW1 LC@ ;
: SW2@   ASW2 LC@ ;
: SW3@   ASW3 LC@ ;

(        Read the silo memory)

: SI1@  ASIB1 LC@ ;
: SI2@  ASIB2 LC@ ;
: SI3@  ASIB3 LC@ ;
: SI4@  ASIB4 LC@ ;
: SI5@  ASIB5 LC@ ;
: SI6@  ASIB6 LC@ ;
: SI7@  ASIB7 LC@ ;

( ===========================================================)

(       TEST: If the is recording, EXIT form word)
(       ====================================)

: ?REC-PROC-EN-EXIT
     REC-PROC-EN @ .TRUE.  =
     IF
       CR ." It can not execute this word: It is recording "
       BEGIN
         DEPTH
       WHILE
         DROP
       REPEAT
       R> DROP EXIT
     THEN ;

( ===========================================================)
(       Words to Handle the Pattern and Mask for the Traps)
(       ==============================================)

: TP-L!                         ( Set AL lines trap for L Trap)
                                ( <AL TRAP>   -- )
     ?REC-PROC-EN-EXIT
     TPL @   #C0   AND          ( Mask bits 6 and 7 of byte)
     OR DUP TPL !               ( Restore new L Trap)
     #3F XOR                    ( Complement trap)
     ATPL LC! ;

: SHUFF3  ASSEMBLER             ( Word used in the 2 next words)
     3      D2      MOVEQ,
     BEGIN,
       D1     A0 ()    .BYTE MOVE,
       2      A0       .LONG ADDQ,
       D1     8        BY-COUNT .LONG LSR,
       D0     A0 ()    .BYTE MOVE,
       2      A0       .LONG ADDQ,
       D0     8        BY-COUNT .LONG LSR,
     D2 F -UNTIL,
FORTH ;

CODE TP-A-AD-NT!             ( Set the AD lines trap for A trap)
         ( <32 BITS TO MATCH> <32 BIT MASK, 1=IGNORE> -- )
     ATPAAD #L    A0          .LONG MOVE,
     SP )+        D0          .LONG MOVE, ( D0 = mask)
     SP )+        D1          .LONG MOVE, ( D1 = pattern)
     D0       TPAAD-MASK .W .LONG MOVE,
     D1    TPAAD-PATTERN .W .LONG MOVE,
     SHUFF3
NEXT;
: TP-A-AD!
     ?REC-PROC-EN-EXIT
     TP-A-AD-NT! ;

CODE TP-D-AD-NT!             ( Set the AD lines trap for D trap)
         ( <32 BITS TO MATCH> <32 BIT MASK, 1=IGNORE> -- )
     ATPDAD #L    A0          .LONG MOVE,
     SP )+        D0          .LONG MOVE, ( D0 = mask)
     SP )+        D1          .LONG MOVE, ( D1 = pattern)
     D0       TPDAD-MASK .W .LONG MOVE,
     D1    TPDAD-PATTERN .W .LONG MOVE,
     SHUFF3
NEXT;
: TP-D-AD!
     ?REC-PROC-EN-EXIT
     TP-D-AD-NT!  ;

: TP-A-AD-P!                        ( Set AD trap pattern A trap)
     TPAAD-MASK 2@ TP-A-AD! ;
: TP-A-AD-M!                        ( Set AD trap mask A trap)
     TPAAD-PATTERN 2@ 2SWAP TP-A-AD! ;
```

```
: TP-D-AD-P!                       ( Set AD trap pattern D trap)
     TPDAD-MASK 2@ TP-D-AD! ;
: TP-D-AD-M!                       ( Set AD trap mask D trap)
     TPDAD-PATTERN 2@ 2SWAP TP-D-AD! ;

CODE TP-A-MS-NT!             ( Set the MS lines trap for A trap)
          ( <16 BITS TO MATCH> <16 BIT MASK, 1=IGNORE> -- )
     SP )+      D0            MOVE,
     SP )+      D1            MOVE,
     D0    TPAMS-MASK .W      MOVE,
     D1   TPAMS-PATTERN .W    MOVE,
     D1          1            BY-COUNT LSR,
     D2          1            BY-COUNT .BYTE ROXR,
     D0          1            BY-COUNT LSR,
     D2          1            BY-COUNT .BYTE ROXR,
     D1          1            BY-COUNT LSR,
     D2          1            BY-COUNT .BYTE ROXR,
     D0          1            BY-COUNT LSR,
     D2          1            BY-COUNT .BYTE ROXR,
     D2       ATPAMS .L       .BYTE MOVE,
NEXT;
: TP-A-MS!
     ?REC-PROC-EN-EXIT
     TP-A-MS-NT! ;

CODE TP-D-MS-NT!        ( Set the MS lines trap for the D trap)
          ( <16 BITS TO MATCH> <16 BIT MASK, 1=IGNORE> -- )
     SP )+      D0            MOVE,
     SP )+      D1            MOVE,
     D0     TPDMS-MASK .W     MOVE,
     D1  TPDMS-PATTERN .W     MOVE,
     D1          1            BY-COUNT LSR,
     D2          1            BY-COUNT .BYTE ROXR,
     D0          1            BY-COUNT LSR,
     D2          1            BY-COUNT .BYTE ROXR,
     D1          1            BY-COUNT LSR,
     D2          1            BY-COUNT .BYTE ROXR,
     D0          1            BY-COUNT LSR,
     D2          1            BY-COUNT .BYTE ROXR,
     D2          4            BY-COUNT LSR,
     D2      ATPDMS .L       .BYTE MOVE,
NEXT;
: TP-D-MS!
     ?REC-PROC-EN-EXIT
     TP-D-MS-NT! ;

          ( <16 BITS TO MATCH> <16 BIT MASK, 1=IGNORE> -- )
: TP-A-MS-M!                      ( Set MS trap mask, A trap)
     TPAMS-MASK @ TP-A-MS! ;
: TP-A-MS-P!                      ( Set MS trap pattern, A trap)
     TPAMS-PATTERN @ SWAP TP-A-MS! ;
: TP-D-MS-M!                      ( Set MS trap mask, D trap)
     TPDMS-MASK @ TP-D-MS! ;
: TP-D-MS-P!                      ( Set MS trap pattern, D trap)
```

```
    TPDMS-PATTERN @ SWAP TP-D-MS! ;

: ?TP-L.                          ( List the L trap)
        TPL @ #3F  AND BASE@ HEX SWAP . BASE! ;

: ?TP-L                           ( List the L trap with label)
        ." ARBITRATION TRAP"
        CR ."     AL lines trap : Pattern = #"
        ?TP-L. CR ;

: ?TP-A-AD-P.                     ( Trap A: List the AD pattern)
    TPAAD-PATTERN 2@ DHU. ;
: ?TP-A-AD-M.                     ( Trap A: List the AD mask)
    TPAAD-MASK 2@ DHU. ;
: ?TP-A-MS-P.                     ( Trap A: List the MS pattern)
    TPAMS-PATTERN @ . ;
: ?TP-A-MS-M.                     ( Trap A: List the MS mask)
    TPAMS-MASK @ . ;

: ?TP-A                           ( List all A trap with labels)
      ." ADDRESS TRAP"
      CR ."     AD lines trap : Pattern = #" ?TP-A-AD-P.
      ."     Mask = #" ?TP-A-AD-M.
      CR ."     MS lines trap : Pattern = #" ?TP-A-MS-P.
      7 SPACES
      ."     Mask = #" ?TP-A-MS-M. CR ;

: ?TP-D-AD-P.                     ( Trap D: List the AD pattern)
    TPDAD-PATTERN 2@ DHU. ;
: ?TP-D-AD-M.                     ( Trap D: List the AD mask)
    TPDAD-MASK 2@ DHU. ;
: ?TP-D-MS-P.                     ( Trap D: List the MS pattern)
    TPDMS-PATTERN @ . ;
: ?TP-D-MS-M.                     ( Trap D: List the MS mask)
    TPDMS-MASK @ . ;

: ?TP-D                           ( List all A trap with labels)
      ." ADDRESS TRAP"
      CR ." AD lines trap : Pattern = #" ?TP-D-AD-P.
      ."     Mask = #" ?TP-D-AD-M.
      CR ." MS lines trap : Pattern = #" ?TP-D-MS-P.
      7 SPACES
      ."     Mask = #" ?TP-D-MS-M. CR ;

: ?TP
      ?TP-L ?TP-A ?TP-D ;

( ===========================================================)

(        Description)
(        ===========)
(        Words to handle the Silo Address Counter.)
(        Note that the silo address counter is actually)
(        decremented, but this is transparent to the user.)
(       All numbers that are read or stored are)
(       complemented.)

: SIA1+                    ( Increment by one the Silo Address)
      ?REC-PROC-EN-EXIT
      CB-MP-SILO-AD-DEC  CW3! ;

: SIA@                     ( Read Silo Address into the stack)
      ASIAHI LC@ 3 AND SWAB
      ASIALO LC@ OR
      #3FF XOR ;           ( Complement the address)

: SIA!                     ( Load Silo Address into counter)
      ?REC-PROC-EN-EXIT
      #3FF XOR             ( Complement the address)
      CB-SILO-AD-RES  CW3!
      DUP SWAB 3 AND CW1@ #FC AND
      OR CW1!  ASIALO LC!
      SIA1+ ;              ( Load into the address latch)

: SIA.                     ( List the Silo Address)
      BASE@ HEX
      SIA@ 0 <# # # # #> TYPE SPACE
      BASE! ;

: SIA                      ( List the Silo Address with label)
      ." Silo Address Counter = #" SIA. SPACE ;

( ==========================================================)

(       Description)
(       ===========)
(       Words to read the Silo Memory)

: LID         ( List the name of each line from Silo Memory)
      3 SPACES
      ." SIAD AIagGKar AL EGrd     AD    "
      ." PApe SRwt MSss TOdt" SPACE ;

: SILOC       ( Reads one sampling of FASTBUS cycle recorded)
              ( into Silo Memory. The Silo Address must be)
              ( set previously.)
      ?REC-PROC-EN-EXIT
      BASE@ HEX
      SPACE SIA.  2 SPACES              ( List Silo Address)
      SI1@
        DUP 1 AND NHU. SPACE            ( AI)
        2 / 1 AND NHU. SPACE            ( AG)
      SI2@
        DUP #80 / NHU. SPACE            ( GK)
        1 AND NHU. SPACE                ( AR)
      SI1@
        DUP #C0 AND #40 / NHU.          ( AL5-4)
        4 / #F AND NHU. 2 SPACES        ( AL3-0)
      SI2@
        DUP #40 AND #40 / NHU. SPACE    ( EG)
        #10 AND #10 / NHU. 2 SPACES     ( RD)
      SI3@
        DUP #10 / NHU.                  ( AD31-28)
        #F AND NHU.                     ( AD27-24)
      SI4@
        DUP #10 / NHU.                  ( AD23-20)
        #F AND NHU.                     ( AD19-16)
      SI5@
        DUP #10 / NHU.                  ( AD15-12)
        #F AND NHU.                     ( AD11-8)
      SI6@
        DUP #10 / NHU.                  ( AD7-4)
        #F AND NHU. 2 SPACES            ( AD3-0)
      SI7@
        DUP #80 AND #80 / NHU. SPACE     ( PA)
        DUP #40 AND #40 / NHU. 2 SPACES ( PE)
        DUP #20 AND #20 / NHU. SPACE     ( SR)
        #10 AND #10 / NHU. 2 SPACES      ( WT)
      SI2@
        2 / 7 AND NHU. SPACE            ( MS)
      SI7@
        DUP 2 / 7 AND NHU. 2 SPACES     ( SS)
        1 AND NHU. SPACE                ( TO)
      SI2@
        #20 AND #20 / NHU. SPACE        ( DT)
      SIA1+ BASE! ;

: SILO                    ( List silo memory using Lines ID.)
              ( Stack: <FIRST ADDRESS> <NO.OF CYCLES> --)
      ?REC-PROC-EN-EXIT
      CR LID CR           ( List the 1st line ID.)
      OVER  SIA!
      OVER  SWAP -
      1                   ( Counter=#10 -> list lines ID.)
      ROT ROT
      DO
        1+ DUP #10 AND    ( Counter is at #10?)
        IF
          DROP 1          ( Yes. Counter=1)
          LID CR          ( List the lines ID again)
        THEN
        2 SPACES  SILOC CR
      LOOP DROP ;

( ==========================================================)

(         Description)
(         ===========)
(         Words to set the Fast Front-End Logic of the)
(         Snoop Module to record FASTBUS cycles.)

(         Set  Words)
(         ==========)

: SS-SRC-FB           ( Set FASTBUS cycle to generate strobes)
    CB-FB/RT*-SEL     SCW4! ;
: SS-SRC-IC           ( Set internal clock to generate strobes)
    CB-EXT/INT*-CP    CCW4!
    CB-FB/RT*-SEL     CCW4! ;
: SS-SRC-EC           ( Set external clock to generate strobes)
    CB-EXT/INT*-CP    SCW4!
    CB-FB/RT*-SEL     CCW4! ;
: SS-SRC!             ( Set the generator of strobes.)
                      ( Read the number form stack)
                      ( 1=FASTBUS, 2=Int.Clock, 3=Ext.Clock)
    DUP 1 =
    IF DROP SS-SRC-FB
    ELSE
      2 =
      IF    SS-SRC-IC
      ELSE  SS-SRC-EC
      THEN
    THEN ;
: ?SS-SRC@            ( Which is the sampling strobe source?)
                      ( Answer into stack)
    CW4@  CB-FB/RT*-SEL  AND
    IF  1
    ELSE  CW4@  CB-EXT/INT*-CP  AND
      IF  3
      ELSE  2
      THEN
    THEN ;
: ?SS-SRC.            ( Which is the sampling strobe source?)
    ?SS-SRC@  . ;     ( List the correspondent number)

: ?SS-SRC             ( Which is the sampling strobe source?)
                      ( List a message)
    ." SAMPLING STROBE SOURCE: "
    ?SS-SRC@  DUP  1 =
    IF DROP  ." FASTBUS Transactions"
    ELSE  2 =
      IF     ." Internal Clock (50 MHz)"
      ELSE   ." Extrenal Clock"
      THEN
    THEN SPACE ;

: TR-SRC-I            ( Set trigger source = internal)
    ?REC-PROC-EN-EXIT
    CB-ITREN   SCW5!
    CB-EXTREN  CCW5!
    .FALSE.  EXTREN ! ;

: TR-SRC-E            ( Set trigger source = external)
    ?REC-PROC-EN-EXIT
    .TRUE. EXTREN ! ( It is not possible to load directly)
                    ( into the CW5. Store into flag.)
```

```
                        ( Will be store at CW5 at START-REC.)
    CB-ITREN   CCW5!
    CB-EXTREN  CCW5! ;

: TR-SRC-M              ( No trigger source needed)
    ?REC-PROC-EN-EXIT
    CB-EXTREN  CCW5!
    CB-ITREN   CCW5!
    .FALSE.  EXTREN ! ;

: TR-SRC!               ( Set trigger source 1=I,2=E,3=none)
    ?REC-PROC-EN-EXIT
    DUP 1 =
    IF
      DROP
      TR-SRC-I
    ELSE
      2 =
      IF
        TR-SRC-E
      ELSE
        TR-SRC-M
      THEN
    THEN ;

: ?TR-SRC@              ( Which is the trigger source?)
                        ( Answer into the stack)
    CW5@  CB-ITREN  AND
    IF  1
    ELSE
      EXTREN @ .TRUE.  =
      IF   2
      ELSE 3
      THEN
    THEN ;
: ?TR-SRC.              ( Trigger source? <answer is a number>)
    ?TR-SRC@   . ;
: ?TR-SRC               ( Trigger source? <answer is message>)
     ." TRIGGER SOURCE: "
     ?TR-SRC@
     DUP 1 =
     IF  DROP  ." Internal trigger sequence"
     ELSE  2 =
       IF     ." External trigger"
       ELSE   ." No trigger used"
       THEN
     THEN SPACE  ;

: TR-SEQ-LAD            ( Set trap seq.= arb., add, data)
   ?REC-PROC-EN-EXIT
   CB-SEL-SEQ/SIN*  SCW2!
   CB-SEL-AA/AD*    CCW2!
   CB-L-TP          SCW7! ;
```

```
: TR-SEQ-LAA          ( Set trap seq.= arb., add, add)
   ?REC-PROC-EN-EXIT
   CB-SEL-SEQ/SIN*  SCW2!
   CB-SEL-AA/AD*    SCW2!
   CB-L-TP          SCW7! ;

: TR-SEQ-LA           ( Set trap seq.= arb., add)
   ?REC-PROC-EN-EXIT
   CB-SEL-SEQ/SIN*  CCW2!
   CB-L-TP          SCW7! ;

: TR-SEQ-AD           ( Set trap seq.= add, data)
   ?REC-PROC-EN-EXIT
   CB-SEL-SEQ/SIN*  SCW2!
   CB-SEL-AA/AD*    CCW2!
   CB-L-TP          CCW7! ;

: TR-SEQ-AA           ( Set trap seq.= add, add)
   ?REC-PROC-EN-EXIT
   CB-SEL-SEQ/SIN*  SCW2!
   CB-SEL-AA/AD*    SCW2!
   CB-L-TP          CCW7! ;

: TR-SEQ-A            ( Set trap seq.= add)
   ?REC-PROC-EN-EXIT
   CB-SEL-SEQ/SIN*  CCW2!
   CB-L-TP          CCW7! ;

: TR-SEQ!             ( Number set the trigger sequence)
   DUP 1 =
   IF DROP TR-SEQ-LAD
   ELSE DUP 2 =
     IF DROP TR-SEQ-LAA
     ELSE DUP 3 =
       IF DROP TR-SEQ-LA
       ELSE DUP 4 =
         IF DROP TR-SEQ-AD
         ELSE 5 =
           IF TR-SEQ-AA
           ELSE TR-SEQ-A
           THEN
         THEN
       THEN
     THEN
   THEN ;

: ?TR-SEQ@            ( Which is the trigger seq. set?)
                      ( Leave the answer into the stack)
                      ( 6 = A ,  5 = AA  , 4 = AD  )
                      ( 3 = LA , 2 = LAA , 1 = LAD )
   CW7@  CB-L-TP  AND
   IF
     0
   ELSE
```

```
      3
    THEN
    CW2@  CB-SEL-SEQ/SIN*  AND
    IF
      1+
      CW2@  CB-SEL-AA/AD*  AND
      IF
        1+
      THEN
    ELSE 3 +
    THEN ;

: ?TR-SEQ.           ( List the number of trap sequence)
    ?TR-SEQ@   . ;

: ?TR-SEQ             ( List the type of trap sequence)
    ." TRIGGER SEQUENCE: "
    ?TR-SEQ@
    DUP 1 =
    IF  DROP  ." Single address cycle"
    ELSE
      DUP 2 =
      IF  DROP  ." Address cycle followed by a data cycle"
      ELSE
        DUP 3 =
        IF DROP  ." Two different address cycles"
        ELSE
          DUP 4 =
          IF DROP  ." Arbitration cycle followed by "
                   ." address cycle"
          ELSE
            5 =
            IF ." Arbitration, address and data cycle"
            ELSE
            ." Arbitration followed by two different address
            ." cycles"
            THEN
          THEN
        THEN
      THEN
    THEN SPACE  ;

: TY-REC-ALL          ( Set to record all types of cycles)
    ?REC-PROC-EN-EXIT
    CB-REC-DAT  SCW4! ;

: TY-REC-NOD          ( Set to record no data cycles)
    ?REC-PROC-EN-EXIT
    CB-REC-DAT  CCW4! ;

: TY-REC!             ( Number set the type of cycle record)
    ?REC-PROC-EN-EXIT
    1 =
    IF
```

```
      TY-REC-ALL
    ELSE
      TY-REC-NOD
    THEN ;

: ?TY-REC@              ( Which type of cycle is to record?)
                        ( Leave the answer into stack.)
                        ( 1 = All, 2 = No data cycle)
    CW4@  CB-REC-DAT  AND
    IF  1  ELSE  2  THEN ;

: ?TY-REC.              ( Number of the type of cycle to rec)
    ?TY-REC@   .  ;

: ?TY-REC               ( List message about ty. cycle to rec.)
    ." TYPE OF CYCLE TO RECORD: "
    ?TY-REC@  1 =
    IF ." All FASTBUS cycles"
    ELSE ." Only arbitration and address cycles"
    THEN SPACE ;

: TR-POS-E              ( Trigger position: end cycles rec.)
    ?REC-PROC-EN-EXIT
    CB-TR-ST/SP*  CCW4! ;

: TR-POS-B              ( Trigger position: start cycles rec.)
    ?REC-PROC-EN-EXIT
    CB-TR-ST/SP*  SCW4! ;

: TR-POS!               ( Set trigger position: 1-Start, 2- End)
    ?REC-PROC-EN-EXIT
    1 =
    IF   TR-POS-B
    ELSE TR-POS-E
    THEN ;

: ?TR-POS@              ( Which is the trigger position?)
                        ( Number into stack: 1-Start, 2-End)
    CW4@  CB-TR-ST/SP*  AND
    IF    1
    ELSE  2
    THEN SPACE  ;

: ?TR-POS.              ( List number of trigger position)
    ?TR-POS@  . ;
: ?TR-POS               ( List message about trigger pos. set)
    ." TRIGGER POSITION: "
    ?TR-POS@  1 =
    IF    ." Begin of FASTBUS cycles recorded"
    ELSE  ." End of FASTBUS cycles recorded"
    THEN SPACE  ;

: ?SET-REC
    CR  10 SPACES
    ." SET OF THE ECL LOGIC TO RECORD FASTBUS CYCLES" CR CR
    ?TR-SRC CR
    ?TR-SEQ CR
    ?TR-POS CR
    ?TY-REC CR
    ?SS-SRC CR
    ?TP-L
    ?TP-A
    ?TP-D  ;

( ==========================================================)

(        Description)
(        ===========)
(        Set of words to perform recording operations into)
(        silo memory of the Snoop Module)

(         Useful  Words)
(         =============)

: ?TERM-EXIT       ( If <ESC> pressed, leave the word)
                   ( that called)
    ?TERMINAL IF R> DROP EXIT THEN ;

: TIMER            ( Wait the time given in the stack)
    0 DO
      #20 #20 * DROP
    LOOP ;

: MES-NO-REC-PROC-EN
    ." THE RECORD PROCESS IS NOT ACTIVATED " ;

(         Set Words)
(         =========)

: ?TR-INPUT-EN@     ( Is some source of trigger enabled?)
                    ( If no, print message)
    CW5@ CB-ITREN   AND
    CW5@ CB-EXTREN  AND
    OR
    IF
      .YES.
    ELSE
      MES#-NO-TR-INPUT-EN
    THEN ;

: ?TR@              ( Is the Snoop Triggered?)
                    ( Answer into stack)
    REC-PROC-EN @  .TRUE. =
    IF
      ?TR-INPUT-EN@   .YES.   =
      IF
        ACIO-PB LC@   CB-SILO-TR*   AND
        IF
```

```
          .YES.
        ELSE
          .NO.
        THEN
      THEN
    ELSE
      MES#-NO-REC-PROC-EN
    THEN ;
: ?TR.                 ( Is the Snoop Triggered?)
                       ( Answer as # listed)
    ?TR@   .  ;
: ?TR                  ( Is the Snoop Triggered?)
                       ( Message as answer)
    5 SPACES
    ?TR@ DUP
    .YES.  =
    IF DROP ." TRIGGER FOUND"
    ELSE
      DUP .NO. =
      IF DROP  ." WAITING FOR TRIGGER"
      ELSE
        MES#-NO-REC-PROC-EN  =
        IF  MES-NO-REC-PROC-EN
        ELSE  ." NO SOURCE OF TRIGGER ENABLED"
        THEN
      THEN
    THEN  20 SPACES JUST-CR ;

: ?BUS-ACT@            ( There are activity into the bus?)
    SB-BAR SB-BAS OR  SB-BDS OR  SB-BDK OR
    DUP SW1@ AND  .NO.
    #7FF 0 DO
      DROP
      OVER SW1@ AND
      SWAP OVER XOR
      IF
        .YES.
        LEAVE
      ELSE
        .NO.
      THEN
    LOOP
    SWAP DROP SWAP DROP ;
: ?BUS-ACT.  ?BUS-ACT@   . ;
: ?BUS-ACT   5 SPACES ?BUS-ACT@
    .YES.  =
    IF    ." BUS IS ACTIVED"
    ELSE  ." NO BUS ACTIVITY DETECTED" THEN
    20 SPACES JUST-CR ;

: ?REC@                 ( Is recording cycles into silo memory?)
   REC-PROC-EN @  .TRUE. =
   IF
     SW3@  SB-REC*  AND
```

```
      IF    .NO.
      ELSE  .YES.
      THEN
    ELSE
      MES#-NO-REC-PROC-EN
    THEN ;
: ?REC.  ?REC@   . ;
: ?REC
    5 SPACES
    ?REC@  DUP  .YES. =
    IF
      DROP ." CYCLES BEING RECORDED"
    ELSE
      .NO. =
      IF   ." IT IS NOT RECORDING"
      ELSE MES-NO-REC-PROC-EN
      THEN
   THEN 20 SPACES JUST-CR ;

: ?STATUS-REC        ( Status of the recording procedure)
              ( <ESC> or the recording done leave this loop)
    CR
    REC-PROC-EN @ .TRUE. =
    IF
      ?TR-INPUT-EN@  .YES. = ( Is the trigger input enabled?)
      IF
        BEGIN
          ?TERM-EXIT
          ?BUS-ACT@  .YES. = ( Yes. Is the bus active?)
          IF
                             ( Yes. Is the Silo Triggered?)
            ?TR@  .YES.  =  DUP  NOT
            IF  TIME TIMER  THEN
            ?TR
          ELSE
            ?BUS-ACT
            .FALSE.
            TIME TIMER
          THEN
        UNTIL  CR
      THEN
      BEGIN
        ?BUS-ACT@   .YES. =  ( There is bus activity?)
        ?TERM-EXIT
        IF
          ?REC@  .NO. =  DUP  ( Is still recording?)
          IF    CR    THEN
          ?REC
        ELSE
          ?BUS-ACT
          .FALSE.
        THEN
        TIME  TIMER
      UNTIL
   ELSE
     MES-NO-REC-PROC-EN
   THEN CR ;

: STOP-REC                         ( Stop recording)
     CB-TR-RES-L     SCW2!         ( Reset trap sequence)
     CB-EXTREN       CCW5!         ( Reset external trigger input)
     CB-MP-STOP-REC  CW3!          ( Stop recording)
     CB-SILO-EN*     SCW5!         ( Disable recording into silo)
     .FALSE. REC-PROC-EN ! ;

: START-REC
   DREAMS @ .FALSE. =              ( If hiber, does not record)
   IF
     0   SIA!                      ( Inicialize the Silo Address)
     .TRUE. REC-PROC-EN !
     CB-SILO-EN*     CCW5!         ( Enable recording into silo)
     EXTREN @                      ( There is trigger in. enable?)
     CW5@   CB-ITREN   AND OR
     IF
       CW4@  CB-TR-ST/SP*  AND NOT ( Yes.Position trigger?)
       IF
         CB-MP-START-REC CW3!        ( Micro starts the rec.)
       THEN
       EXTREN @ .TRUE. =           ( Is the trigger int or ext?)
       IF
         CB-EXTREN       SCW5!
       ELSE
         CB-TR-RES-L     CCW2!       ( Enable trap sequence)
       THEN
     ELSE
       CB-MP-START-REC CW3!          ( Micro. start the rec.)
     THEN
   THEN ;

: REC      START-REC  ?STATUS-REC  STOP-REC ;

( ========================================================== )

(       Description)
(       ===========)
(       Control if the module executes the action asked)
(       by the host.)

: HIBER                             ( Does not execute)
     .TRUE. DREAMS !  ;

: WAKE                              ( Execute)
     .FALSE. DREAMS ! ;

: ?DREAMS.                          ( Which is its state?)
     DREAMS @ . ;

( ========================================================== )
(        Initialization procedure)
(        ========================)

: INIT
      .FALSE. REC-PROC-EN !
      .FALSE. EXTREN      !
      .FALSE. DREAMS      !

      0    CW1!                   ( Control words)
      1    CW2!
      #27  CW3!
      #2D  CW4!
      #7A  CW5!
      #3   CW7!
      #FF  CW8!

      0. #FFFFFFFF. TP-A-AD!     ( Trap words)
      0. #FFFFFFFF. TP-D-AD!
      0  7          TP-A-MS!
      0  7          TP-D-MS!
      0             TP-L!

      ACIO-C    LC@  DROP          ( Zilog CIO)
      1   ACIO-C  LC!              ( Reset the CIO)
      0   ACIO-C  LC!
      #2B ACIO-C  LC!
      #FD ACIO-C  LC! ;
INIT

( ==========================================================)

(        Word Name)
(        =========)
(        EPROM)

(        Description)
(        ===========)
(        Reads the program from RAM to the DATA IO in the)
(        Intel Format to create a EPROM with the)
(        new program.)

: EPROM                         ( Read program in Intel format)
                    ( <last address> <first address> EPROM)
        DO
          CR #3A EMIT           ( :)
          #10                   ( Initialize the check sum)
          #10 BHU.
          I #100 / DUP BHU.  +    ( Emit address)
          I #FF AND DUP BHU. +
          0 BHU.                ( Emit 00)
          I #10 + I             ( Indexes to read each byte)
          DO
            I C@ + #FF AND      ( Find check sum)
          I C@ BHU.          ( Emit byte)
        LOOP
        1- #FF XOR           ( Complement of 2 to check sum)
        BHU.                 ( Emit check sum)
      #10 +LOOP
      CR                     ( End of transfering)
      #3A EMIT               ( :)
      0 BHU.                 ( 00)
      0 BHU.  0   BHU.       ( 0000)
      1 BHU.                 ( 01)
      CR ;

( ***********************************************************)

(       Description)
(       ===========)
(       This modules control the communication with the host)
(       of the FASTBUS LOGIC STATE ANALYZER.)
(       To the FORTH in the Snoop Module be able to perform)
(       this communication protocol, the EXPECT, KEY and)
(       EMIT routines have to be changed. The EXPECT had to)
(       be changed, basically, in the following way:)
(        - Detect when the module is addressed and if the)
(          operation is polling or selection.)
(        - If it was selection, receive the text sent)
(          checking for erros. When all text were received,)
(          pass it to the Forth INTERPRETER to execut it.)
(        - If it was polling and there is text to send to)
(          the host, send it, otherwise send EOT.)
(       A basic difference in this communication regarding)
(       the normal use of the EMIT and EXPECT routines is)
(       that text will be sent just when the module is)
(       polled. The KEY routine had to be changed to delete)
(       CTRL_S and CTRL_Q which are used by the host to)
(       synchronize the communication.)

(       Constant Declarations)
(       =====================)

#04 CONSTANT EOT        ( control characters of the protocol)
#03 CONSTANT ETX
#02 CONSTANT STX
#05 CONSTANT ENQ
#06 CONSTANT ACK
#07 CONSTANT PAT
#15 CONSTANT NAK
#17 CONSTANT ETB
#53 CONSTANT  S
#50 CONSTANT  P
#14000. DCONSTANT OUTBUF_BEGIN
#17FFB. DCONSTANT OUTBUF_END
#17FFC. DCONSTANT OUTPUT_POINT_ADDRESS

(       Varible Declarations)
(       ====================)

VARIABLE  CHAR_TRANS_BLOCK  ( # of characters transmited)
VARIABLE  INTEXT_ADD        ( Input Text address)
VARIABLE  POINTER           ( Pointer of general use)
VARIABLE  INBUF #28 ALLOT #28 ALLOT  ( Input buffer = 82)
VARIABLE  INBUF_END         ( Input buffer end address)
VARIABLE  MOD_ADD_FLAG     ( If true, the module addressed)
VARIABLE  LAST_MES         ( Save last message in case of ENQ)
VARIABLE  MOD_NO
#2031 MOD_NO !

( ==========================================================)

(       Module Name)
(       ===========)
(       NEW_KEY)

(       Description)
(       ===========)
(       This module substitue the original KEY of the FORTH.)
(       Its function is to delete CTRL_S or CTRL_Q that are)
(       sent by the VAX to synchronize the communication of)
(       both units. When any of these 2 characters are read,)
(       the routine delete then and wait for the next)
(       character. It is written in Assembler.)

CODE NEW_KEY
   #FEFF81. #L    A0     .LONG MOVE, ( A0=Address of ACIA)
   BEGIN,
     BEGIN,
       BEGIN,
         A0 ()        D1     .BYTE MOVE, ( There is character)
         1            D1     .BYTE ANDI, ( to read?)
       NE
       UNTIL,
       4 A0 D)      D0     .BYTE MOVE,    ( Yes. Read)
       #007F        D0        ANDI,       ( Mask the 8th bit)
       #11          D0     .BYTE CMPI,    ( Is CTRL_Q?)
     NE
     UNTIL,
     #13          D0     .BYTE CMPI,    ( No. Is CTRL_Q?)
   NE
   UNTIL,
   D0          SP -)        MOVE,  ( No. Save into the stack)
NEXT;

( ==========================================================)

(       Module Name)
(       ===========)
(       NEW_EMIT)

(       Description)
(       ===========)
(       This module substitue the original EMIT routine of)
(       the FORTH. When it is called, it moves the character)
(       that is into the top of the stack to an internal)
(       buffer located in the RAM memory between the)
(       addresses OUTBUF_BEGIN to OUTBUF_END. The pointer)
(       that address this memory is a long word and)
(       is always reset to OUTBUF_BEGIN when the module is)
(       selected. Therefore, if the host wants to read some)
(       data, it has to select the module asking for the)
(       data to read and then poll the module and read this)
(       data. If the procedure is not this, the pointer)
(       will empty the buffer. Written in assembler)

CODE NEW_EMIT
   OUTBUF_POINT_ADDRESS #L
             A0          .LONG MOVE,   ( Address pointer -> A0)
   A0 ()     A1          .LONG MOVE,   ( Move to A1 the pointer)
   SP )+     D0          .WORD MOVE,   ( Stack -> D0)
   D0        A1 ()       .BYTE MOVE,   ( D0 -> buffer)
   #1        A1          .LONG ADDQ,   ( Add one to the pointer)
   A1        A0 ()       .LONG MOVE,   ( Save the new pointer)
NEXT;

( ========================================================)

(        Module Name)
(        ===========)
(        WRITE_BYTE)

(        Description)
(        ===========)
(        This module writes characters to the serial port.)
(        It has the capability to detect CTRL_S and CTRL_Q)
(        that the VAX sends in order to synchronize it)
(        receiver with the speed that the Snoop sends)
(        characters. When the Snoop receives CTRL_S it stop)
(        sending characters and when receives CTRL_Q it)
(        start again.)

CODE WRITE_BYTE
   #FEFF81. #L  A0      .LONG MOVE, ( Address of ACIA -> A0)
   #10 #W       D0      MOVE,       ( # times wait CTRL_S)
   BEGIN,
     BEGIN,
       A0 ()       D1       .BYTE MOVE,   ( Character to read?)
       1           D1       .BYTE ANDI,
       NE
       IF,
         4 A0 D)     D1       .BYTE MOVE,      ( Yes. Read. )
         #13         D1       .BYTE CMPI,      ( Is CTRL_S? )
         EQ
         IF,
           BEGIN,
          BEGIN,
            A0 ()    D1   .BYTE MOVE, ( Character to read?)
            1        D1   .BYTE ANDI,
          NE
          UNTIL,
          4 A0 D)     D1       .BYTE MOVE,    ( Yes. Read )
          #11         D1       .BYTE CMPI,    ( Is CTRL_Q? )
          EQ
        UNTIL,
      THEN,
    THEN,
    1 #W    A0 ()   .BYTE MOVE,  ( Is possible to output?)
    A0 ()    D1     .BYTE MOVE,
    1        D1     .BYTE ANDI,
  NE
  UNTIL,
 D0 EQ                           ( Yes. Check again?)
 -UNTIL,
 SP )+    D0       MOVE,         ( Yes. Output.)
 D0       4 A0 D)  .BYTE MOVE,
 #22. #L  A0       .LONG MOVE,   ( Up to date the cursor)
 D7       A0       .LONG ADD,    ( position.)
 1        A0 ()    ADDQ,
NEXT;

( ==========================================================)

(      Module Name)
(      ===========)
(      READ_BYTES)

(      Description)
(      ===========)
(      This routine reads the bytes being sent by the Host.)
(      Anytime that EOT is received, it disconnect the)
(      transmitter from the Network and empty the input)
(      buffer. It will read the bytes until the PAT, which)
(      signal the end of that message.)

(      Executable Code)
(      ===============)

: READ_BYTES
    INBUF               ( Create a pointer in the stack)
    BEGIN KEY           ( Read one byte)
    DUP EOT =           ( Is the byte read EOT)
    IF
                              ( Disconnect from network)
      .FALSE. MOD_ADD_FLAG ! ( Signal: is not addressed)
      SWAP DROP  INBUF  SWAP ( Reset pointer)
      OVER  C!               ( Store EOT in input buffer)
      1+                     ( Increment by 1 the pointer)
      .FALSE.           ( Signal must continue reading)
    ELSE
```

```
      DUP PAT =          ( Is the byte read PAT?)
      IF
        DROP .TRUE.      ( Signal to stop reading bytes)
      ELSE
        SWAP DUP INBUF_END @  >  ( Is input buffer full?)
        IF
          SWAP DROP        ( Do not save the byte)
        ELSE
          SWAP OVER  C!    ( Save the byte)
          1+               ( Increment by 1 the pointer)
        THEN
        .FALSE.          ( Signal to continue reading)
      THEN
    THEN
    UNTIL
    0 OVER C!
    0 SWAP 1+ C! ;       ( Store two 0 in the end of buffer)

( ===========================================================)

(      Module Name)
(      ===========)
(      INPUT_TEXT)

(      Description)
(      ===========)
(      This routine inputs text from the host after the)
(      Snoop was addressed. It is called by NEW_EXPECT. It)
(      is called with no data stored into the stack, but)
(      with INBUF_ADD and INBUF_END initialized. As soon)
(      as it start executing this routine, it sends an ACK,)
(      to ACK the address. There is just 1 way to leave)
(      this routine: when the address connection with the)
(      host is broken. It leaves the routine with a flag)
(      .true. or .false.. If .true. there is data into the)
(      the area point by INTEXT_ADD. This text is passed)
(      to the Forth INTERPRETER.)

(      Utility Routines)
(      ================)

: SEND_ACK   ACK WRITE_BYTE         ( Send acknowledge)
             PAT WRITE_BYTE
             ACK LAST_MES ! ;

: SEND_NAK   NAK WRITE_BYTE         ( Send No Acknowledge)
             PAT WRITE_BYTE
             NAK LAST_MES ! ;

: SEND_LAST_MES         ( Repeat the previous ACK or NAK sent)
             LAST_MES @  WRITE_BYTE
             PAT WRITE_BYTE ;

(      Main Routine)
(       ============)

: INPUT_TEXT
    SEND_ACK

(                 Read the text)

    .FALSE.               ( This flag is simply deleted)
    BEGIN
      DROP                ( Delete the flag)
      READ_BYTES          ( Read the bytes sent by the host)
      MOD_ADD_FLAG @      ( Is the module still addressed?)
      IF
        1                 ( Was the parity correct?)
        IF
          INBUF           ( Inicialize the buffer pointer)
          DUP C@  STX   = ( There is STX begin of buffer?)
          IF
            .FALSE. SWAP
            INTEXT_ADD @  POINTER  ! ( pointer to INTEXT)
            #20 POINTER @ C!  ( Store a space in the first)
                              ( character of INTEXT. It is)
                              ( necessary.)
            POINTER @ 1+ POINTER !    ( Pointer + 1)
            1+  INBUF_END @  SWAP     ( Start saving buffer)
            DO
              I C@  0  = NOT  ( Is NOT end of buffer?)
              IF
                I C@  ETX  =  ( Is end of text block?)
                IF
                  0  POINTER @  C!    ( Mark end of buffer)
                  0  POINTER @ 1+  C!
                  DROP .TRUE. ( Leave BEGIN..UNTIL loop)
                  LEAVE
                ELSE
                  I  POINTER @  1 CMOVE  ( Save input text)
                  POINTER @  1+  POINTER ! ( Pointer + 1)
                THEN
              ELSE
                LEAVE
              THEN
            LOOP            ( If flag .true. data in inbuf,)
                            ( otherwise flag .false.)
            DUP
            IF
              SEND_ACK
            ELSE
              SEND_NAK
            THEN
          ELSE
            C@  ENQ  =  ( There is ENQ in begin of buffer?)
            IF
              SEND_LAST_MES
              .FALSE.   ( Doesn't leave BEGIN..UNTIL loop)
          ELSE
            SEND_NAK
            .FALSE.     ( Doesn't leave BEGIN..UNTIL loop)
          THEN
        THEN            ( If flag .true. data in inbuf,)
                        (  otherwise flag .false.)
      ELSE
        SEND_NAK
        .FALSE.
      THEN              ( If flag .true. data in inbuf,)
                        ( otherwise flag .false.)
      DUP
    ELSE
      .FALSE.           ( No data in inbuf)
      .TRUE.            ( Leave this BEGIN ... UNTIL loop)
    THEN
  UNTIL

(               Read the End of Transmission)

  DUP                   ( Is the text correct?)
  IF
    BEGIN               ( Must read now just EOT)
      READ_BYTES
      INBUF  DUP        ( Initialize a buffer pointer)
      C@  EOT  =        ( Is EOT?)
      IF
        4 +  C@  ENQ  =   ( Was it a new polling?)
        IF
          DROP  .FALSE.   ( Invalid text. Drop old flag)
          .TRUE.          ( Leave BEGIN...UNTIL loop)
        ELSE
          .TRUE.          ( Valid text. Leave loop)
        THEN
      ELSE
        C@  ENQ  =        ( Is asking for last message?)
        IF
          MOD_ADD_FLAG @ ( There is address connection?)
          IF
            SEND_LAST_MES ( Send last message)
            .FALSE.       ( Do not leave loop)
          ELSE
            DROP .FALSE.  ( Error in the text)
            .TRUE.        ( Leave the BEGIN...UNTIL loop)
          THEN
        ELSE
          MOD_ADD_FLAG @  ( There is address connection?)
          IF
            SEND_NAK
            .FALSE.       ( Do not leave loop)
          ELSE
            DROP .FALSE.  ( Error in the text)
            .TRUE.        ( Leave the BEGIN...UNTIL loop)
          THEN
          THEN
        THEN
      UNTIL
    THEN

(              Reset the OUTPUT TEXT POINTER)

    DUP                        ( Was the text read correct?)
    IF
      OUTBUF_BEGIN
      OUTBUF_POINT_ADDRESS L2! ( Yes. Reinicialize pointer)
    THEN ;

( ==========================================================)

(       Module Name)
(       ===========)
(       OUTPUT_TEXT)

(       Description)
(       ===========)
(       This routine outputs the text stored into the output)
(       text buffer. The output text buffer is located into)
(       RAM from address OUTBUF_BEGIN to OUTBUF_END and)
(       the pointer that signal its end is stored
(       at OUTBUF_POINT_ADDRESS. This routine)
(       is called by NEW_EXPECT when the module is polled)
(       by the host. The output text buffer is filled by)
(       NEW_EMIT.)

(       Utility Routines)
(       ================)

: ?OUTTEXT_EMPTY  ( Test if the output text buffer is empty)
   2DUP OUTBUF_POINT_ADDRESS
   L2@  D< NOT ; ( .TRUE.=Empty, .FALSE.= Still with data)

(       Main Routine)
(       ============)

: OUTPUT_TEXT
   OUTBUF_BEGIN      ( Start pointer to output text buffer)
   BEGIN
     ?OUTTEXT_EMPTY ( Output text buffer empty?)
     NOT
   WHILE

(              Transmit the text)

     STX WRITE_BYTE   ( Transmit STX)
     DECIMAL 78 0     ( Maximun # of characters per block)
     DO
       2DUP LC@ WRITE_BYTE    ( Send a character)
       1. D+                  ( Increment the pointer)
    I 1+  CHAR_TRANS_BLOCK !  ( Save # char. this block)
    ?OUTTEXT_EMPTY            ( Is the end of text?)
    IF
      LEAVE            ( Yes. Leave the loop)
    THEN
  LOOP
  ?OUTTEXT_EMPTY       ( It was end of text?)
  IF
    ETX WRITE_BYTE     ( Yes. Transmit the ETX)
  ELSE
    ETB WRITE_BYTE     ( No. Transmit the ETB)
  THEN
  PAT  WRITE_BYTE      ( Send PAT)

(              Read the answer of the host)

  BEGIN
    READ_BYTES
    MOD_ADD_FLAG @     ( Is the module still addressed?)
    IF
      1                ( Parity error?)
      IF
        INBUF C@  ACK  = NOT  ( Yes. Host NOT send ACK?)
        IF
          INBUF C@  NAK  =    ( Yes. Host send NAK?
          IF
                           ( Yes. Retransmit block again)
            CHAR_TRANS_BLOCK @  ( Load # of character)
            0 D-                ( Calculate old pointer)
            .TRUE.              ( Leave this loop)
          ELSE
            ENQ WRITE_BYTE      ( Ask for the answer again)
            PAT WRITE_BYTE
            .FALSE.             ( Do NOT leave this loop)
          THEN
        ELSE
          .TRUE.                ( Leave this loop)
        THEN
      ELSE
        ENQ WRITE_BYTE          ( Ask for the answer again)
        PAT WRITE_BYTE
        .FALSE.                 ( Do NOT leave this loop)
      THEN
    ELSE
      2DROP
      OUTBUF_POINT_ADDRESS
       L2@           ( No. Overwrite the pointer to stop)
      .TRUE.         ( Leave this loop)
    THEN
  UNTIL
REPEAT
DROP                      ( Drop pointer)
MOD_ADD_FLAG @            ( Is the module still addressed?)
IF
```

```
      EOT WRITE_BYTE         ( Yes. Send EOT)
      PAT WRITE_BYTE
      40 0 DO I DROP LOOP    ( Wait a short time)
      EOT WRITE_BYTE         ( Send EOT again)
      PAT WRITE_BYTE
    THEN ;

( ============================================================)

(       Module Name)
(       ===========)
(       NEW_EXPECT)

(       Description)
(       ===========)
(       This routine substitue the original EXPECT routine)
(       of the FORTH of the Snoop Module. It has the)
(       possibility to perform the necessary network)
(       procedure for the FASTBUS Logical Analyzer.)
(       This routine call, principally, 3 other routines:)
(       READ_BYTES, INPUT_TEXT and OUTPUT_TEXT, which were)
(       already described earlier. Basically this routine)
(       detects the address of the Snoop Module being)
(       transmitted by the VAX and connects the Module to)
(       the VAX. After this, it checks to see if the module)
(       is being polled or selected. If it is being polled)
(       it executes the routine OUTPUT_TEXT and, if is being)
(       selected, it executes INPUT_TEXT. The only time that)
(       the loop of this routine is left is after a )
(       selection and the text was received successfully.)

: NEW_EXPECT DOCOL
     INBUF  +    INBUF_END  ! ( Initialize input buffer end)
     INTEXT_ADD  !            ( Initialize input text add.)
     #FF #FEFF92. L!          ( Turn off the LED = not add.)
     #0  ACIO-PB LC!          ( Disable RS422 output)
     .FALSE. MOD_ADD_FLAG !   ( Deaddress the module)
     READ_BYTES
     BEGIN               ( Leaves this loop when received text)
       BEGIN
         1               ( Was the parity correct?)
         IF
           INBUF         ( Create a pointer to input buffer)
           DUP C@  EOT  =    ( 1-Does inbuf start with EOT?)
           IF
             1+ DUP C@
             MOD_NO  C@  =  ( 2-Is the address the same?)
             SWAP  1+ SWAP OVER C@  MOD_NO 1+ C@  =
             AND
             IF
               1+
               DUP C@  DUP  S =
               SWAP  P  =   ( 3-Is selection or polling? )
               OR
```

```
          IF
            1+
            DUP C@  ENQ  =  ( 4-Does it finishes ENQ?)
            IF
              DROP .TRUE.   ( The module is addressed)
              .TRUE. MOD_ADD_FLAG  !
              #7F  #FEFF92.  L!  ( Turn on LED=add.)
              CB-0422-EN         ( Enable output RS422)
              ACIO-PB LC!
            ELSE
              DROP .FALSE.       ( It wasn't addressed)
            THEN
          ELSE
            DROP .FALSE.
          THEN
        ELSE
          2 - DUP C@  0  =    ( Is broadcast address?)
          SWAP 1+ SWAP OVER C@ 0 =
          AND
          IF
            1+ DUP C@ S =     ( Is selection?)
            IF
              1+ DUP C@ ENQ = ( Does it finish with ENQ?)
              IF
                DROP .TRUE.   ( The module is addressed)
                .TRUE. MOD_ADD_FLAG !
                #7F #FEFF92. L!
              ELSE
                DROP .FALSE.
              THEN
            ELSE
              DROP .FALSE.
            THEN
          ELSE
            DROP .FALSE.
          THEN
          ELSE
            DROP .FALSE.
          THEN
        THEN
      ELSE
        DROP  .FALSE.
      THEN
    ELSE
      .FALSE.
    THEN
  NOT                   ( If module addressed, leave loop)
  WHILE
    READ_BYTES
  REPEAT
  INBUF 3 + C@  S =  ( Is selection?)
  IF
    INPUT_TEXT  ( Input text of data. Return)
                ( TRUE if text was received correctly.)
```

```
                    ( It returns with the module deadd.)
      ELSE
        OUTPUT_TEXT ( Output text of data)
        .FALSE.     ( Do no go to the INTERPRETER)
        .FALSE. MOD_ADD_FLAG  !    ( Deaddress the module)
      THEN
      #FF  #FEFF92.  L!         ( Turn off the LED)
      #0  ACIO-PB LC!           ( Disable RS422 output)
  UNTIL ;

( ===========================================================)

(       Module Name)
(       ===========)
(       HST)

(       Description)
(       ===========)
(       When this routine is executed, the Snoop module)
(       start using the Network Protocol to communicate)
(       with the Host. To return to the protocol to)
(       communicate with a terminal is necessary to reset)
(       or turn the Snoop on.)

: HST OUTBUF_BEGIN
    OUTBUF_POINT_ADDRESS L2! ( Init. OUTBUF_POINT_ADDRESS)
    #6C0 #C50 !              ( Disable the CR word)
    ' NEW_EMIT 2- UEMIT !    ( Change EMIT by NEW_EMIT)
    ' NEW_EXPECT UEXPECT !   ( Change EXPECT by NEW_EXPECT)
    ' NEW_KEY 2-  UKEY  !    ( Change KEY by NEW_KEY)
    #20 #17CF C!             ( Change OKNN by 2 spaces)
    #20 #17D0 C!
    #7CA #17DA ! ;

( ===========================================================)

VOC-LINK @ #3AA !      ( Link this new vocabulary to the old)
#80F4 2@ #38A 2!

.end                   ( Signal VAX program to stop download)

( ***********************************************************)
```

BIBLIOGRAPHY

(1) U.S. NIM Committee, 1983. FASTBUS Modular High Speed Data Acquisition and Control System for High Energy Physics and Other Applications. U.S. Department of Energy, Washington, DC.

(2) Treptow, K. 1983. FASTBUS Display Module. Data System Group of CDF Note no. DS$LIB00029, Fermi National Accelerator Laboratory, Illinois.

(3) Brodie, L. 1981. Starting Forth. 1st ed., Prentice-Hall, Inc., New Jersey.

(4) Brodie, L. 1984. Thinking Forth. 1st ed., Prentice-Hall, Inc., New Jersey.

(5) Barsotti, E. J., Larwill, M. H., Ingen, C., 1983. Overview of the new UNIBUS Processor Interface to FASTBUS. CDF Note no. 230, Fermi National Accelerator Laboratory, Illinois.

(6) Digital Equipment Corporation. 1982. VAX-11 FORTRAN Language Reference Manual. Order no. AAD034C-TE. U.S.A.

(7) White, V., 1985. Mencom, A Menu Oriented Command Interface Package, CDF Note no. 298.0., Fermi National Accelerator Laboratory, Illinois.

(8) International Business Machines Corporation, 1970. General Information - Binary Synchronous Communications, (GA27-3004 File TP-09). 3rd ed., IBM Corporation, North Carolina.

(9) Digital Equipment Corporation. 1982. VAX/VMS System Services Reference Manual. Order no. AA-D018C-TE. U.S.A.

(10) H. V. Walz and R. Downing, 1981. FASTBUS Snoop Diagnostic Module. IEEE Trans. on Nuclear Science, NS-28, No.1, pp.380-384. U.S.A.

(11) H. V. Walz, D. B. Gustavson and R. Downing, 1983. Progress on the SLAC Snoop Diagnostic Module for FASTBUS, IEEE Trans. on Nuclear Science, NS-30, No. 1, pp.220-222. U.S.A.

(12) H. V. Walz and D. B. Gustavson, 1983. Status of the SLAC Snoop Diagnostic Module for FASTBUS, IEEE Trans. on Nuclear Science, NS-30, No. 4, pp.2276-2278. U.S.A.

(13) Digital Equipment Corporation, 1984. VAX EDT Reference Manual. Order No. AA-J726A-TC. U.S.A.