A Method of Fast Data Transfer

from FASTBUS

by

Baya Oussena

Presented as a Thesis for the Degree of Master of Science

Department of Physics and Astronomy,

University of Glasgow,

December 1991.

Abstract

One major factor which affects the counting efficiency of a nuclear physics experiment is the dead time of the detectors and the data acquisition system. Experiments performed by Glasgow University photonuclear group typically involve the readout of $\sim$ 1000 ADC's and $\sim$ 1000 scalers which contain information on the products of a photo-disintegration event. These require fast readout to minimise dead time and to this end a method of programming the model 1821 FASTBUS Segment Management Interface (SMI) to increase data throughput coming from FASTBUS has been developed.

The electronic hardware used is comprised of VME-bus, CAMAC, and FASTBUS systems. The VME-based CPU is the heart of the data acquisition system. FASTBUS is mainly used for ADC's and TDC's while CAMAC is mostly used to control the experimental parameters such as detector thresholds, trigger logic, high voltage etc. Each FASTBUS crate is controlled by a LeCroy 1821 Segment Manager Interface (SMI), and the interfacing to the VME CPU is accomplished either by using the VME fast memory module type HSM8170 or the slower CAMAC interface type LeCroy 2891A. The HSM8170 is connected to the SMI using the 32-bit LeCroy ECL bus.

The VME CPU runs the OS9 operating system, and the data acquisition software has been written almost entirely in C. Software for the sequencer in the 1821 SMI is written in machine code, although it is hoped in the future to

develop a simple assembler.

Two different SMI codes have been developed. These are called CODE1 and CODE2. The first attempt, CODE1, uses the slow, CAMAC connection at the front panel of the 1821 SMI for module initialisation and data readout. To improve the data throughput, it was decided to develop CODE2 which uses the rear panel ECL bus connection to a fast VME memory, and require no intervention from the VME host CPU to initiate data readout. Associated C routines written for the VME CPU handle downloading of the code to the SMI and create FASTBUS module addressing SMI instruction words.

Finally, the performance of the two FASTBUS readout methods has been compared on a test setup where more than 100 ADC channels are read for each event. Under these conditions, the dead time for a CODE2 readout was found to be approximately a factor of 8 less the dead time for CODE1.

DECLARATION

The original work in this thesis comprises the bulk of that described in chapter 4. This involved the development of CODE1, the creation and developement of CODE2 and its associated C written subroutines and the test measurements made to compare the speed of the two codes. This thesis has been composed by myself.

Baya Oussena

Acknowlegements

My special thanks go to my supervisor Dr J.R.M.Annand for his endless guidance, advice and encouragement during this work and for his critical comments and discussions during the composition of this thesis.

I am grateful to Professor R.O.Owens, the director of the Kelvin Laboratory, for affording me the use of the Kelvin Laboratory facilities and for providing me with financial assistance without which this work would not have been possible.

I would like to thank Dr I.Anthony, Dr G.Miller and Dr P.D.Harty for their comments concerning the writing up of the thesis.

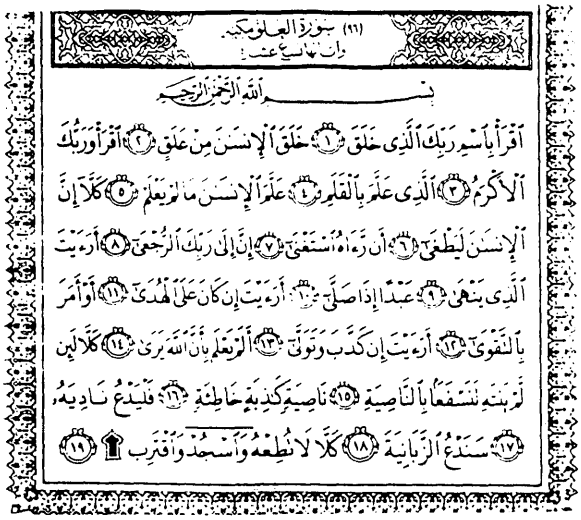
I woulk like to express my thanks to Dr J.C.McGeorge, Mrs Eileen Taylor, Mrs Gwen Miller and the students R.Crawford, G.Cross and S. Doran for their general help.

All of the Kelvin Laboratory Staff deserve thanks for their enthusiasm and humour all of which have provided a most enjoyable working environment.

I should not forget to thank my friend Fatima and her husband for their kindness and the endless support they gave me to continue with this work.

Even far away from Algeria, my whole family did not stop sending me their help and especially the moral support I needed. I am very grateful to them

and through this acknowledgment I would like to express the best thoughts and the best thanks I have for each of them.



To my parents who, with their love,

have helped my studies reach a successful conclusion.

Contents

Chapter 1

INTRODUCTION

In general, a nuclear physics experiment aims to shed light on an aspect or aspects of nuclear structure, typically by bombarding the nucleus of interest with a chosen probe (photon, proton etc.) and measuring the energies and momenta of the final state products of the reaction between the probe and the nucleus. Through the measurement of the interaction of the probe and nucleus, details of the nuclear structure may be inferred. For this purpose, particle detectors and associated electronic apparatus are required. Detector signals are processed by analogue and digital circuitry, with the latter often making logical decisions to determine if a particular event in the detection system is potentially interesting. If so, a logic signal is sent to trigger the data acquisition system, which in modern facilities is invariably built around one or more microcomputers.

A simple but not untypical experimental layout is sketched in figure 1.1. A beam of "probe" particles bombards a target containing the nuclei of interest. The reaction products are detected by counters "Det_1" and "Det_2" which generate pulses having amplitudes proportional to the kinetic energy of the particles and may be capable of particle identification. If the analogue signals are digitised by ADC's to give the energies and particle types and the directions of travel are known from the geometry of the set up then it is easy to calculate the momenta.

Time pick-off of detector signals by voltage discriminators produces logic sig-

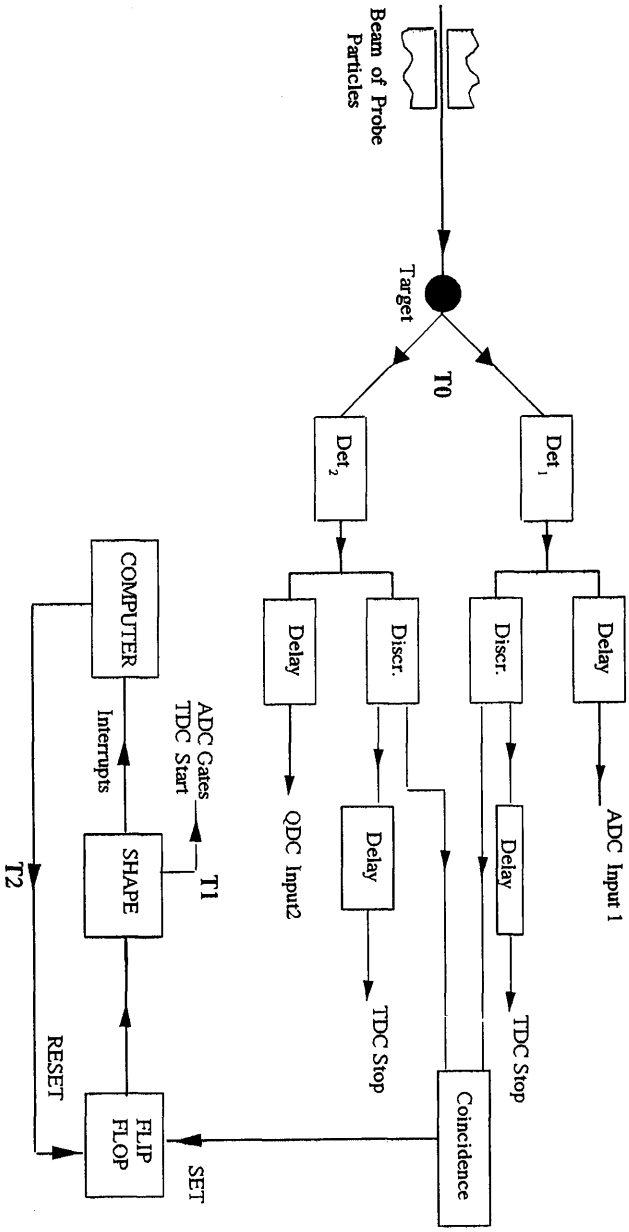


Figure 1.1: **Hypothetical Nuclear Physics Experiment**

nals and temporal coincidences between these can show that multiple particles are produced in the one probe, nucleus collision. The coincidence output is used to trigger charge and time to digital convertors and to provide an interrupt signal to cause the computer to initiate readout of data.

As their name implies, Analogue to Digital Convertors (ADC's) measure quantities such as charge, voltage or time and output a number which is proportional to the analogue quantity. Usually the ADC's are housed in a standard bussed hardware system (CAMAC or FASTBUS) from which the numbers produced by the ADC's are transferred via suitable interfaces into the main Random Access Memory (RAM) of the computer which controls the experiment. The computer would initialise and monitor the ADC's, oversee the transfer of data and operate on the data once it is in RAM. Operations might involve storage in some standard format on disk or magnetic tape, analysis and sorting into spectra and possibly transfer to another computer. This could take over storage, analysis and display tasks, reducing system overheads on the front-end experimental control computer. Storage on tape allows the data from the experiment to be replayed offline, when more complicated and sophisticated analysis than is possible online, can be performed.

One major factor which affects the efficiency of an experiment is the dead time of the detectors and the data acquisition system. This is the finite time required to process an event. Suppose m is the true counting rate and the

detector registers k counts in a time T. Since each detected count engenders a dead time τ, a total dead time $k\tau$ is accumulated during the counting period T. During the dead period, a total of $mk\tau$ counts is lost [1, 2]. Thus the ratio of observed counts to true counts registered in any time can be given by $R = 1 - m'\tau$, where $m' = k/T$ is the observed counting rate. This ratio approaches zero as the observed counting rate approaches the reciprocal dead time.

The total dead time τ can be broken into two components, T_1 and T_2. T_1, depending only on the detectors and the electronics used by the experiment, could be quite short ($\sim$ 100 ns) so that except at exceptionally high counting rates it would not affect the counting efficiency. However T_2, the time for the computer to read out and process the event's data, would generally be much larger, perhaps around 1 ms, so that it would have a non-negligible effect ($R = 0.9$) even at a modest counting rate of 100Hz. Thus to maximise the counting efficiency, T_2 requires to be minimised. This might be performed by increasing the CPU speed, reducing system overheads, improving bus interface hardware and making the data readout software more efficient.

At the Kelvin Laboratory, the data acquisition system ACQU is based on three linked bus sytems, VME-bus, CAMAC, and FASTBUS. Most of the signal digitising and data readout is performed through FASTBUS. Each FASTBUS crate is controlled by a LeCroy 1821 Segment Manager Interface (SMI) and the

goal of this project has been to produce new software to run on the FASTBUS SMI and VME-bus CPU which makes efficient use of new SMI to VME-bus interface hardware.

A general description of the hardware and software of the Kelvin Laboratory data acquisition system is presented in chapters 2 and 3, while details of the SMI and the new software are given in chapter 4. Chapter 5 presents test comparisons of the old and new SMI interface systems and assesses the success of the project.

Chapter 2

HARDWARE DESCRIPTION

The electronic hardware used in an experimental set up at the Institut für Kernphysik, the University of Mainz, shown schematically in figure 2.1, illustrates the type of system which may be handled by the Kelvin Laboratory data acquisition system. It is comprised of VME-bus, CAMAC, and FASTBUS systems. The VME-based CPU is the heart of the data acquisition system. FASTBUS is mainly used for ADC's and TDC's while CAMAC is mostly used to control the experimental parameters such as detector thresholds, trigger logic, high voltage etc. The test system used for the present work is shown in the photograph of figure 2.2. Although not as extensive as the Mainz system it includes all of the main elements, VME-bus, CAMAC and FASTBUS.

2.1 VME system

The VME crate used has twelve free slots (double height) and space for mounting peripherals. The present system includes an Eltec E6/68030 microcomputer, mass storage peripherals and more specialist modules such as the CBD8210 CAMAC Branch Driver, the VIC8250 VME to VME inter-crate communications module and the HSM8170 high speed ECL ported memory.

Figure 2.1: **General Configuration**

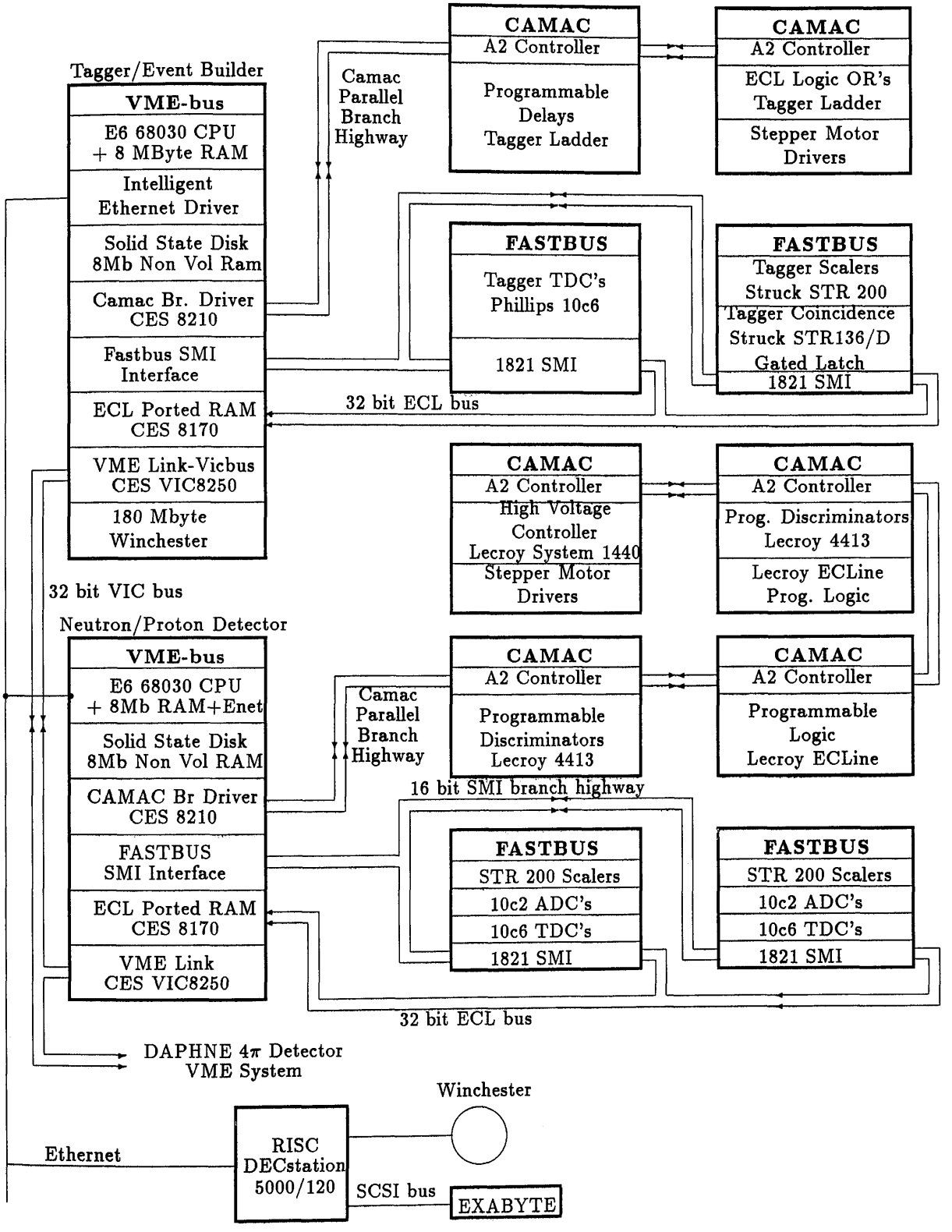


Figure 2.2: **Kelvin Laboratory Test System**

2.1.1 Standard Hardware *(Eltec E6/E5)*

The heart of the VME bus system is the MC68030 based Eltec E6 single board computer [3]. The less powerful MC68020/E5 [4]. can also be used with identical source code.

These machines have 32-bit address registers, 32-bit data registers and features such as :

1) 7Mip (E6) or 3Mip (E5) integer performance,

2) 1 to 16 Mbytes RAM, 128 kbytes EPROM

3) Hardware floating point coprocessor,

5) Interface circuitry for the VME bus,

the auxiliary VSB bus and SCSI bus,

6) Interface circuitry for ethernet communications.

The EPROM contains boot programs for various operating systems and simple debugging facilities. The peripherals used with the present Eltec are :

1) 150 Mbyte hard disk

2) 150 Mbyte streaming tape

3) floppy disks $5\frac{1}{4}$ inch or $3\frac{1}{2}$ inch.

Figure 2.1 shows the connections between the various buses on the acquisition system which are implemented by more specialist VME hardware.

2.1.2 VME-VME connection *(VIC8250)*

The VIC8250 [5] is a transceiver for the so called VIC bus, or vertical bus, which has been developed by the company CES to connect VME backplanes, and has been adopted by CERN as a standard VME to VME connection. Up to 15 backplanes may be connected by twisted pair cable of maximum length 100m, and one VIC8250 must be programmed as the bus master with the others as slaves. VIC bus allows a VME CPU to access any address in any connected VME system, but this can result in VME-bus arbitration problems requiring specialist software solutions. Instead the ACQU system uses the internal buffer memory of the VIC8250 for inter-VME communications. Slave VME systems write data to the buffer in their local VIC8250 and the master reads this buffer over the VIC bus. Special mailbox locations in the VIC8250 buffer are used to synchronise read/write operations.

2.1.3 VME-CAMAC connection *(CBD8210)*

The VME-bus is interfaced to CAMAC through the CES CBD8210 CAMAC Branch Driver [6]. This module, based on a Saclay design, is a double height VME card which maps a 24 bit VME address to a CAMAC CNAF and drives a parallel branch of up to 7 CAMAC crates. The CBD8210 can drive one CAMAC

branch with the number of the branch to be driven selected by a front panel switch.

The CBD8210 provides four internal registers to handle communications. The CSR (Control Status Register) addressed by *C0 N29 A0 F0*, contains most of the status information necessary for correct functioning of the CBD8210.

The IFR (Interrupt Flag Register) addressed by *C0 N29 A0 F4* is write only and provides the facilities to set or clear external interrupt flags by software. This is potentially useful for test purposes.

The CAR (Crate Address Register) addressed by *C0 N29 A0 F8* is used for multiple addressing of crates on a CAMAC branch.

The BTB Register is addressed by *C0 N29 A0 F9*. When it is read, we get the information regarding which crates in the branch are on line, and when written to, a CAMAC branch initialisation is generated (BZ signal).

The CBD8210 maps any standard **B,C,N,A,F** CAMAC address/command to a unique 24 bit VME address as follows :

bits [23:22] = 1:0
bits [21:19] : B = Branch Address (0 to 7 Front Panel Switch)
bits [18:16] : C = Crate Number (1 to 7 Standard addressing)
bits [15:11] : N = CAMAC Station Number
bits [10:07] : A = CAMAC Subaddress
bits [06:02] : F = CAMAC Function
bits [1] : CAMAC Word Length : 0=24 bits, 1=16 bits
bits [0] = 0.

In the C language the VME address for a BCNAF CAMAC command is generated as follows :

$$bcnaf = b + (c \ll 16) + (n \ll 11) + (a \ll 7) + (f \ll 2) + (l \ll 1)$$

where "$\ll$" means left shift. The three classes of CAMAC functions (read, write, test) are implemented as follows:

$$\begin{aligned} read\text{-}value &= *\ bcnaf \\ *\ bcnaf &= write\text{-}value \\ test &= *\ bcnaf \end{aligned}$$

The CBD8210 can generate external VME-bus interrupts (IRQ) to the CPU at priority 2 or 4 when it receives an external logic signal at the front panel. The interrupt vector number can be jumper set from 1 to 255 so that conflicts with any other interrupting peripherals can be avoided.
The CPU response to interrupt requests is quite fast. The E6 hardware acknowledges the IRQ within approximately one microsecond and generally the IRQ service routine is initiated within 10 to 15 μs.

2.1.4 VME-FASTBUS connection

The interfacing to FASTBUS is more complicated than CAMAC [7]. Each FASTBUS crate is controlled by a LeCroy 1821 Segment Manager Interface (SMI) and the interfacing is accomplished either by using the VME fast memory module HSM8170 or the slower CAMAC interface type LeCroy 2891A.

The CAMAC based LeCroy 2891A [8] provides a bi-directional link be-

tween FASTBUS and the VME-bus. Due to its indirect nature, it is relatively slow, but it is reliable nonetheless. The 8 main control registers in the SMI are mapped to equivalent registers in the 2891A, via a ribbon cable connection, so that the SMI can be programmed by issuing appropriate CNAF's. For example the CAMAC command F(0) A(0-7) will read the contents of the 1821 registers 0 to 7 and the command F(16)A(0-7) will write data into the 1821 registers 0 to 7. The model 2891A has the capability to address multiple 1821's. To select any 1821, the module select register is programmed. The module select register is loaded with the desired SMI address by the command F(17)A(1) and read by the command F(1)A(1).

To increase the speed of data transfer another module has been added to connect the VME-bus to FASTBUS. This is the CES ECL ported memory type HSM8170 [9], which allows fast data transfer from FASTBUS to VME buffer memory at a maximum speed of 10 MHz. A FIFO (First In First Out) buffer of 64 words of 32 bits allows the maximum data transfer rate into the main memory without handshake between the 1821 SMI and the HSM8170.
Control of the HSM8170 is performed through 4 registers : The control register, the interrupt and status/ID register, the address pointer register and the word counter register. As in the present application the HSM8170 interrupts are not used, the interrupt and control registers are programmed to disable the interrupts. The usable HSM8170 memory size is fixed through the control register.

The address pointer register allows the selection of the starting address in memory where the data will be transferred and the word counter register is initiated with the maximum number of words to be transferred into the memory.

The HSM8170 is connected to the SMI using the 32-bit LeCroy ECL bus. A small interface board, 1821/ECL [10, 11], connected to the SMI via the auxiliary backplane, converts internal SMI logic to differential ECL logic. The 1821 SMI cannot receive data through the 1821 ECL data ports, which is why the slow connection via CAMAC is necessary.

2.2 CAMAC system

CAMAC [12]-[14] in the Kelvin Laboratory data acquisition project is primarily used for programmable circuitry which allows the remote control of experimental parameters such as signal thresholds, trigger logic conditions and detector high voltages. However the readout of CAMAC ADC's is also supported.

The main piece of CAMAC hardware is the crate, which has 25 stations. Stations 24 and 25, the rightmost stations, are reserved for the controller, while stations 1 to 23 are normal stations used for CAMAC slave modules. Each module connects to the CAMAC bus, known as the dataway , which constitutes a series of bussed and individual lines to perform data read, data write, strobing and addressing.

The crate controller is the heart of the CAMAC crate. The type A crate controllers, used in our system, interface between the parallel branch and the CAMAC dataway and have no particular dependence on the type of computer involved.

The crate controller only responds to branch commands which correspond to its own crate number (*C*), which is selected by a front panel switch. In response to the NAF command, it sets the appropriate dataway lines and issues a strobe signal to the slave module. In general a module will not support all possible NAF permutations, but those which it does support must be part of the CAMAC standard. In response to a valid command which it supports, the module will generate a valid command accepted (X response) and act on the command. If the command requires data transfer, the read or write lines will be used.

2.3 FASTBUS system

2.3.1 Introduction

FASTBUS [15]-[19], was originally conceived in the middle 1970's in response to the needs of high energy physicists for more powerful and sophisticated data acquisition hardware. It was developed to provide high speed data acquisition for large detector systems, as encountered in particle physics experiments. However the increased size and complexity of medium energy nuclear physics

experiments have made it increasingly useful in this field.

A typical system might consist of the bus itself (also known as the segment), modules and a host computer. General categories of module include processor interfaces, segment interconnects, ADC's, memories, logic signal processors and diagnostic modules. The segment is a 32-bit bus with multiplexed address and data lines. It supports asynchronous transfers with handshake protocol, several addressing and data transfer modes, arbitration with priority levels and autonomous operation of individual segments.

At the Kelvin Laboratory we use the Struck type STR104F FASTBUS crate which has an easily demountable CERN specification power supply of 3.5 kW DC capability. The FASTBUS crate is 19 inches wide and has 26 slots of which none are privileged. The board dimensions are 366.7 mm high by 400 mm deep, about 4 times the size of a CAMAC board.

2.3.2 FASTBUS modules

There are two basic categories of FASTBUS modules, masters and slaves. The slave modules, which are mainly ADCs, TDCs, scalers etc., cannot gain mastership of the segment but can only assert information on the segment in response to a specific request by a master.

Compared to CAMAC modules, FASTBUS modules are more sophisticated and

more complicated to program. They provide 32-bit subaddress capabilities and would normally support several addressing modes. The registers of FASTBUS modules are divided into two distinct regions, Data Space (DSR) and Control Status Space (CSR), which are separately accessible. The purpose and size of the data space is defined by the designer, whereas some CSR registers have standard functions. Each module contains in the 16 Read Only MSB (Most Significant Byte) of its standard register CSR0, a module specific identifier code. The full 32 bits (Write Only) of CSR0 are used to control the functions of the slave module.

A brief description is now given of some FASTBUS modules used at the Kelvin Laboratory.

1- *Phillips 10c2/10c6 ADCs*

The Analogue to Digital Converter modules, Phillips 10c6 Time to Digital [20] and 10c2 Charge to Digital, have 32 channels [21]. Each channel can be individually programmed with a pedestal correction and a lower and upper level threshold. The data which satisfy the threshold conditions are transferred from the ADC to a LIFO (Last In First Out) buffer, where they are stored two ADC channels per 32-bit word, with a header word per event. Data can be read a minimum of 8.5 μs after receipt of a trigger signal and block readout can typically occur at 10 MHz rate. For increased throughput when reading out many modules, MULTIBlock readout is used. MULTIBlock mode potentially

allows a whole crate of 10c modules to look like one contiguous buffer to the master, enabling readout of multiple modules as if they were one giant module.

2- *Struck STR136 Gated Latch*

This is an edge-triggered 64-input gated latch [22]. While a gate signal is applied, any input will be latched and the latched inputs may be read over FASTBUS as well as being available as outputs. The 64 latched bits are read through a block transfer on DSR0 and DSR1.

3- *Struck STR200 Scaler*

This contains 32, 32-bit 100 MHz scalers [23]. They may be read through registers DSR0 to DSR31 and block transfer is supported.

4- *LeCroy 1821 SMI*

The practical use of all these slave modules depends on having a suitable FASTBUS master to read them out. The LeCroy 1821 SMI (Segment Manager Interface) is a programmable FASTBUS module which can act as a slave, a master, a snoop, or a processor interface. In the present application the SMI is always the segment manager, issuing the commands both to initialise slave modules and where applicable to read data from them. It also provides interfaces between FASTBUS and the VME-bus. It is futher described in chapter 4.

2.3.3 Addressing modes

The basic mode of FASTBUS addressing is geographical, ie a module is accessed by its physical slot number in the FASTBUS crate. This is known as the primary address cycle. A secondary address cycle, which involves writing a 32-bit "register offset" to the module, gives access to the internal registers of the module. An alternative to geographical addressing is logical addressing, where the device is assigned a logical address of 32 bits consisting of the device address and an internal address [24]. Each device capable of being logically addressed contains a device address register, which is fully accessible by standard FASTBUS operations and which must be initialised by the system startup procedure.

Where it is desired to program several slaves simultaneously, broadcast addressing may be used. Unlike logical or broadcast addressing, all FASTBUS slaves must support geographical addressing and this is the mode used in the present acquisition system.

2.3.4 FASTBUS operations

There are basically four phases in a FASTBUS operation [25]. These are the arbitration, addressing, data read/write and bus release cycles.

In the present case the 1821 SMI is always configured to be the master, and arbitration is unnecessary. During the primary address cycle, the geograph-

ical address of the desired module is placed in the Address Data (AD) lines. Once the slave recognises its address on the AD lines, it responds by asserting the address acknowledge line. The address cycle results in the establishment of a link between the master and the slave. After receiving the slave's address acknowledge, the master can clear the address from the AD lines and thus use them for data transmission, such as the transmission of a secondary address which is accomplished through a write data cycle.

After writing a secondary address a master will normally proceed to transfer data. In the present application block mode is used for data transfer.

Chapter 3

SOFTWARE DESCRIPTION

3.1 Overview of Data Acquisition

Kelvin Laboratory experiments typically involve the readout of $\sim$ 1000 ADC's and $\sim$ 1000 scalers which contain information on the species and momenta of particles associated with a photo-disintegration event. These require fast readout to minimise deadtime and the acquisition software should have the flexibility to allow easy changing of the experimental hardware configuration.

A general aspect of a data acquisition system is shown in figure 3.1. The main functions of the data acquisition software are :

1- *Control of Data Transfer from ADC's*

2- *Data Storage*

3- *Data Analysis - Sorting into spectra - Display of spectra*

To carry out these processes efficiently, the functions named above should be independent tasks, hence the need for a multitasking system. The OS9 operating system written originally for MC68000 microcomputers offers multitasking, good real time response and reasonable source level debugging facilities [26], which become indispensable when the complexity of the code increases.

For maximum efficiency, data readout and transfer from ADC's should be interrupt driven, and ideally the time slicing priorities of the various tasks would be "tuned" to make best use of the CPU. However, care should be

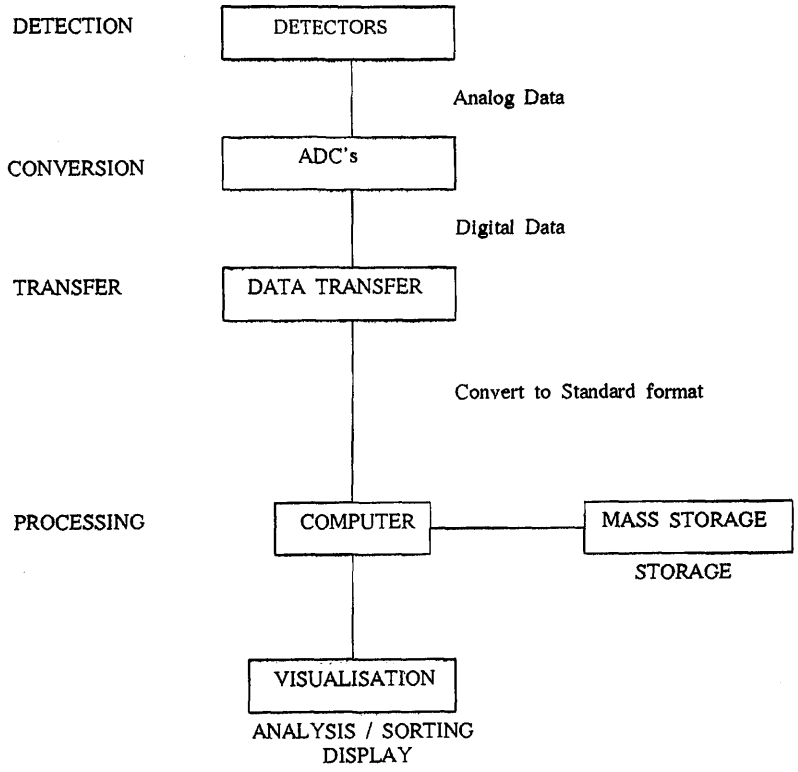


Figure 3.1: **General Aspect of the Acquisition**

exercised in performing the latter, especially where the progress of one task depends on the progress of another.

3.2 OS9 Operating System

OS9 [27, 28] is a multitasking, real time operating system for the 68000 family of microprocessors, which is widely used in nuclear or high energy physics data acquisition systems as well as a variety of scientific or industrial applications. OS9 has two distinct states in which object code can be executed. These are "user" state, where processes are time sliced with some restriction on access to hardware addresses and "system" state, where processes are not time sliced and have unlimited access to any address. OS9 system calls and interrupt service routines run in system state. System state routines often deal with physical hardware present on the system.

3.2.1 OS9 Input/Output Structure

OS9 input/output operations are handled by three programs usually written in assembly language. They are respectively :

1. File Manager

This includes general purpose code to service a particular class of device eg. a disk or tape. It handles the file structuring of a device and has very little device

dependence. It is not of any use for CAMAC or FASTBUS operations, but we have to use it to comply with the OS9 way. Four file managers are included in our system. The one used in the present data acquisition is the SBF *(Sequential Block File Manager)* which is normally used with sequential block structured devices such as tape drives.

2. Device Driver

This module, in conjunction with the file manager, handles the actual operation of a device, and in practice will be somewhat device dependent. We use it to initialise VME slave modules, install interrupts etc. We could use it for data readout, but this is not necessary. Its function is to contain the device's interrupt service routine and provide the means of loading it into the OS9 operating system.

3. Device Descriptor

This is a data module read in by the device driver to specify addresses, interrupt vectors etc. for a specific device. Each physical device has an associated descriptor and one device driver can handle several descriptors and hence devices.

3.2.2 OS9 Interrupts

The OS9 operating system provides the user with 192 vectored interrupts *(vectors 64 to 255)*, allowing the system to handle many interrupting devices. Vec-

tors 1 to 64 are reserved for the system. Interrupt service routines are executed in system state at priorities ranging from 1 to 7, where 7 is the highest. Low priority interrupts give way immediately to those of higher priority and only resume after the higher priority interrupt has completed.

3.2.3 Multitasking and Intertask Communications

When the multiple tasks of the acquisition system are loaded and executed, interprocess communication is necessary to synchronise processes and to pass data between them. Synchronisation is handled by the use of signals and events, while data are passed via shared memories.

1. Signals

The process expecting a signal must contain a signal intercept routine to catch this signal, otherwise it will be killed by the first signal it receives.
Signals are not queued, so they may be lost if they are not serviced by the intercept routine. The present application uses signals only at the end of the data acquisition to cause an orderly shutdown of the system.

2. Events

Unlike signals, events are queued so that no event can be lost. A process "waits" for an event to occur or "sends" an event to another process. Events are named and can be assigned values. Thus checks can be made by a potential receiver

in systems where several different events are used.

Events are used to handle the communication between four subprocess tasks "acqu", "hist", "store", "slave". The three tasks "hist", "store", "slave" wait for events from the task "acqu" which show it has accumulated a full buffer of event mode data.

3. Shared Memories

Shared memories are created to pass data between the different subprocess tasks "acqu", "hist", "store", "slave". Each subprocess must be linked to the shared memory before it can perform any access to it. In the present application two shared memories are created. One is used as a shared device ID memory, containing tables of information on the system hardware, and the other is used to define two swinging buffers used for transfer of data.

3.3 General Developments

In addition to the native assembly language [29], OS9 offers the high level programming language C [30, 31] which, with its ability to manipulate real hardware addresses, is highly suited to data acquisition programming.

Apart from a few lines of assembler, the data acquisition software "ACQU" which initialises, monitors and performs data readout of hardware in the VME-bus, FASTBUS and CAMAC standards has been written in the C

language. A block diagram of the software modules and their interconnections is given in figure 3.2. ACQU consists of six principal tasks :

1- Supervisor task ("vme-supervise"),

2- Data readout task ("acqu"),

3- Data storage task ("store"),

4- Histogramming task ("hist"),

5- Slave system control Start/Stop task ("slave"),

6- Master system control task ("control").

3.3.1 Supervisor Task : vme-supervise

The program supervisor, "vme-supervise" performs four essential initialisation functions before it goes to *"sleep"*. These are :

1- Hardware initialisation,

2- Shared memories initialisation,

3- Interprocess communications initialisation,

4- Start up of subprocesses (acqu, hist, store, slave).

The hardware initialisation is based on information read in from parameter files which are created using the standard editor. All files are liberally commented (lines beginning with "*") to improve readability. Three main parameter files are used :

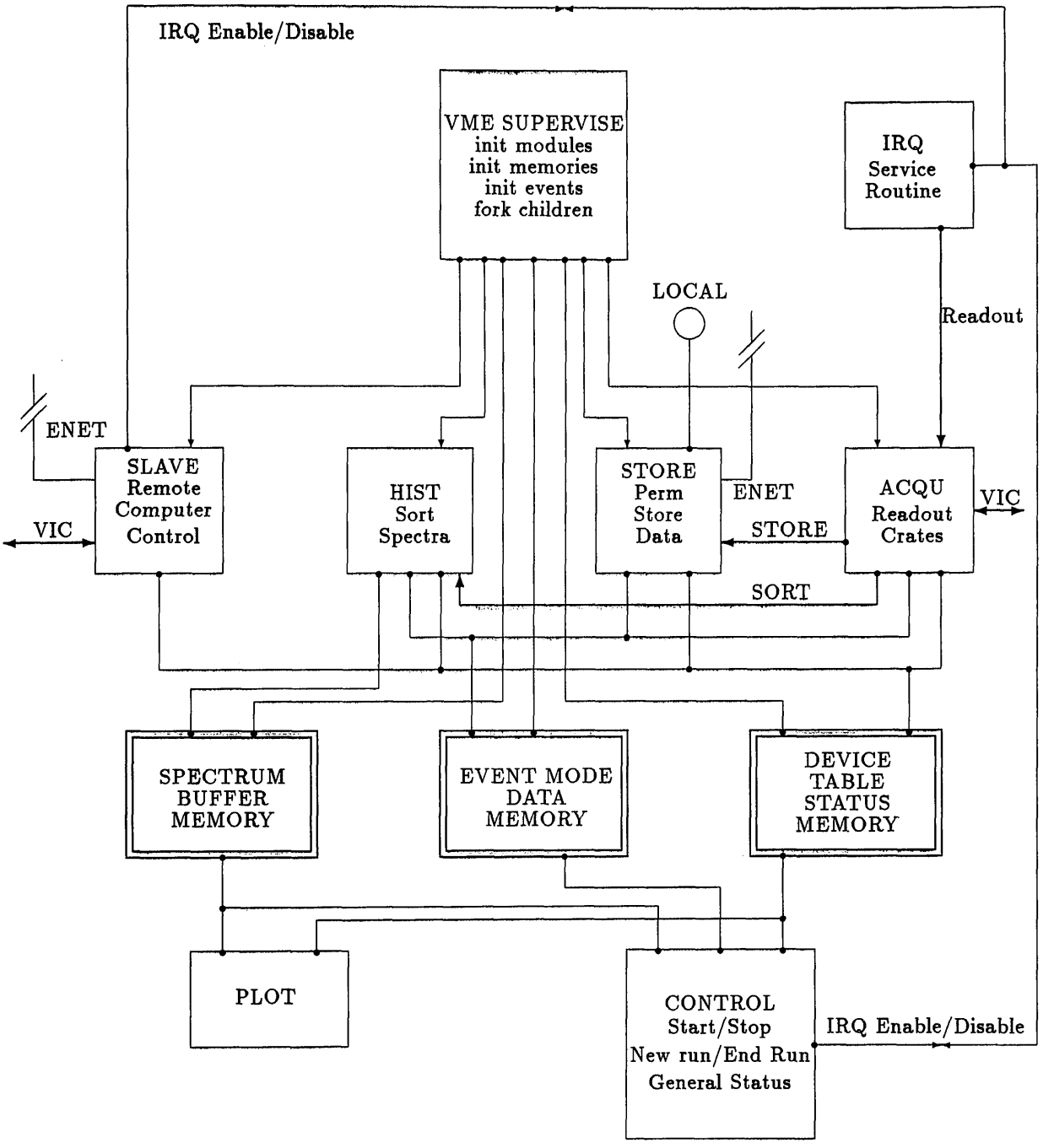


Figure 3.2: **Block Diagram of Data Acquisition Software**

1. *Master Parameter File*

This defines the shared memories such as the hardware module table, data buffers and spectrum buffers. Also defined are OS9 event names used for the communication between semi-independent tasks, and output paths such as ethernet or local mass storage.

2. *CAMAC Parameter File*

The CAMAC parameter file is rather simpler in structure than the master parameter file. Each non-comment line relates to a single CAMAC module. There are 4 parameters to a line, which are: crate number, station number, module name, module specialist initialisation file. Table 3.1 contains the names of the CAMAC modules currently recognised by the ACQU system along with their function.

3. *FASTBUS Parameter File*

The FASTBUS parameter file is also relatively straightforward to understand. An example is given in appendix C. The FASTBUS master is assumed to be a LeCroy 1821 SMI (Segment Manager Interface). Each FASTBUS crate has a section consisting of one line pertaining to the SMI setup, followed by "n" lines, where "n" is the number of slave modules (ADC's, logic modules etc.) which reside in that particular FASTBUS crate, i.e. one line per module. The SMI line has 4 parameters which are : SMI number, RAM number, number of

slaves and readout mode. A list of supported FASTBUS modules is given in table 3.1.

After initialisation is complete, the supervisor may be re-awakened by a signal from the control process to make an orderly shutdown of the acquisition system. The supervisor and its child tasks execute in the background state, so that the terminal is available to control the acquisition system by running the control module "control".

3.3.2 Subprocesse Tasks : acqu, hist, store, slave

The "acqu" subprocess must be run to give data readout, but other subprocesses are optional depending on what is required of the acquisition system.

The four subprocess tasks "acqu", "store", "hist", "slave", run simultaneously. On receipt of an event from the interrupt routine, the "acqu" task performs the readout of CAMAC and FASTBUS modules into a data buffer in shared memory. Data transfer from CAMAC is accomplished by a simple read address operation. However data transfer from FASTBUS is more complicated, since this bus is inherently more complex than CAMAC, and the usual FASTBUS master, the 1821 SMI, has itself to be programmed. Details of FASTBUS readout programming are given in Chapter 4.

When a data buffer is full, "acqu" can optionally send an event signal

CAMAC modules used

Name	Module	Function
A2CONTROL	A2 Controller	Parallel Branch
LRS2249A	LeCroy 2249A	Q ADC 10bit
LRS2249W	LeCroy 2249W	Q ADC 11bit
LRS2249SG	LeCroy 2249SG	Q ADC 10bit
LRS2259	LeCroy 2259	V ADC 11bit
LRS2228A	LeCroy 2228A	TDC 11bit
LRS4413	LeCroy 4413	16chan. LED
LRS4418	LeCroy 4418	16chan.delay
LRS4508	LeCroy 4508	Dual PLU
LRS2551	LeCroy 2551	12ch.Scaler
LRS2891A	LeCroy 2891A	SMI interface
HYTEC310S	Hytec 310s	4 chan scaler
SEN2PA2049	SEN 2049	Dual attenuator
SEN2PA2048	SEN 2048	Dual cable delay

FASTBUS modules used

Name	Module	Function
PHIL_10c6	Phillips 10c6	TDC 10bit
PHIL_10c2	Phillips 10c2	Q ADC 10bit
STRUCK_200	Struck 200	100 MHz Scaler
STRUCK_136D	Struck 136/Diff	64bit Latch
STRUCK_136	Struck 136	64bit Latch

Table 3.1: **Supported CAMAC and FASTBUS modules**

to the storage task "**store**" to write the data to mass storage or ethernet and/or to the histogramming task "**hist**" to sort the data into spectra. Data transfer is performed through two swinging data buffers. A flow chart depicting the operation of "**acqu**" is given in figure 3.3.

Data storage may be on a local peripheral or on a remote device via ethernet. The TCP protocol is used for ethernet communications between dissimilar computers and operating systems. This high level protocol has been tested between an Eltec E6 running OS9 and a variety of VAX's running VMS, where it has proved to be adequately fast and extremely reliable. When "**store**" receives the signal from "**acqu**" it copies the data buffer to ethernet or local device and makes it available to "**acqu**" for futher data. A flow chart of the subprocess "**store**" is shown in figure 3.4.

The subprocess "**hist**" copies a data buffer over to a special histogram buffer from which it does the sorting. When it has finished it flags that it is ready to receive another. Apart from generally adding to system overheads, "**hist**" does not hinder data buffer storage. A general flow chart of the subprocess "**hist**", is shown in figure 3.5.

The subprocess "**slave**" causes the VME system to run in slave mode. The assumption is that there are several coupled VME systems and that one of the remote systems is the master which controls start/stop etc.

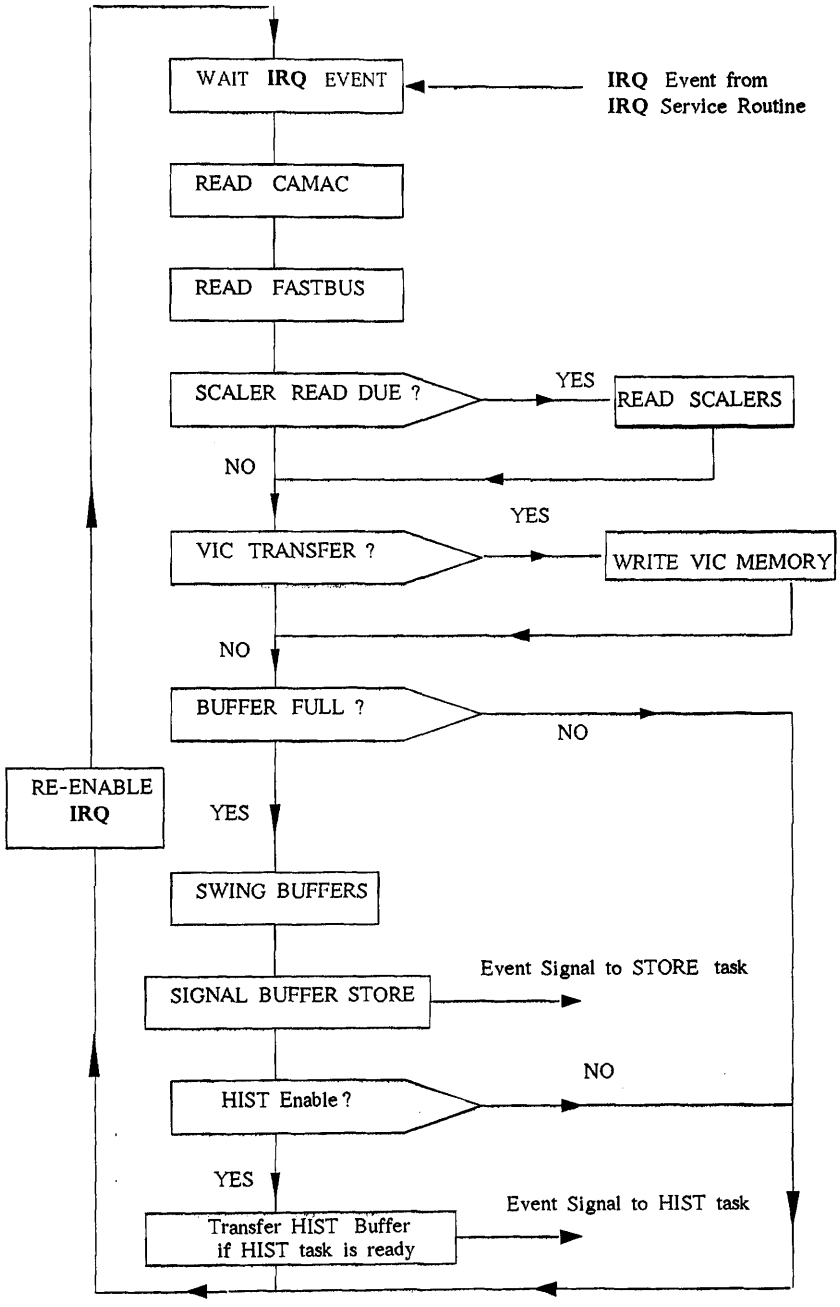


Figure 3.3: **Flow Chart of Subprocess "acqu"**

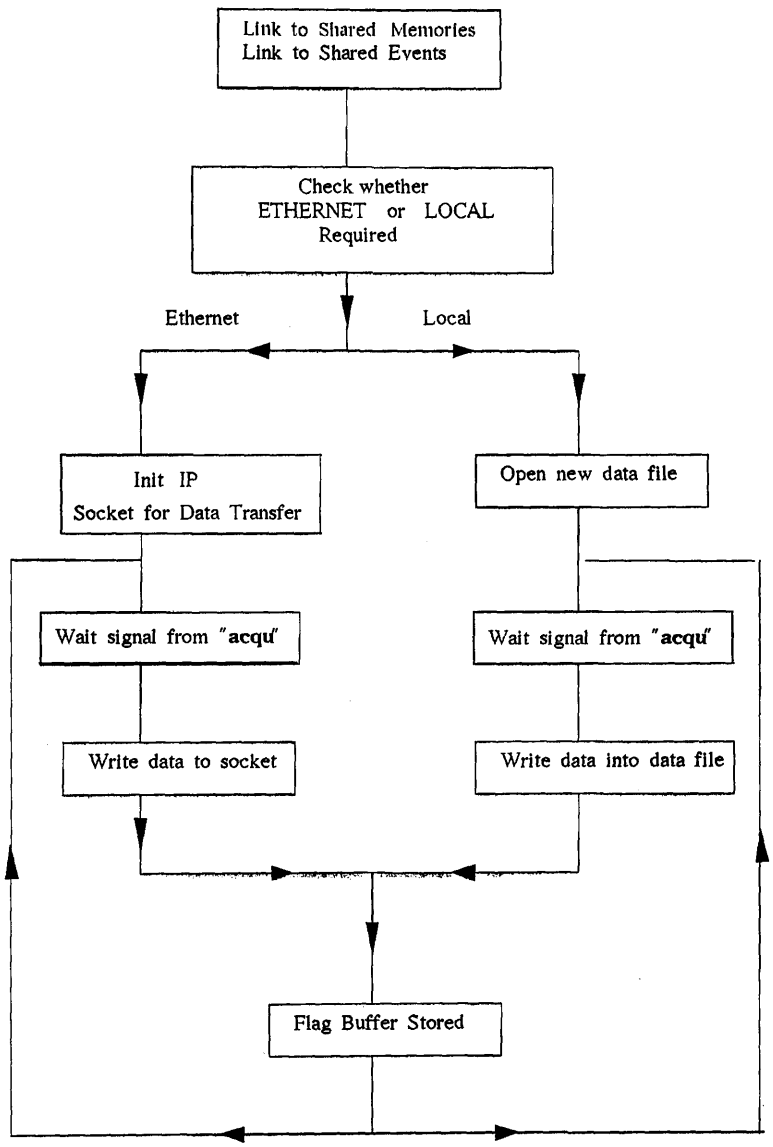


Figure 3.4: **Flow Chart of Subprocess "store"**

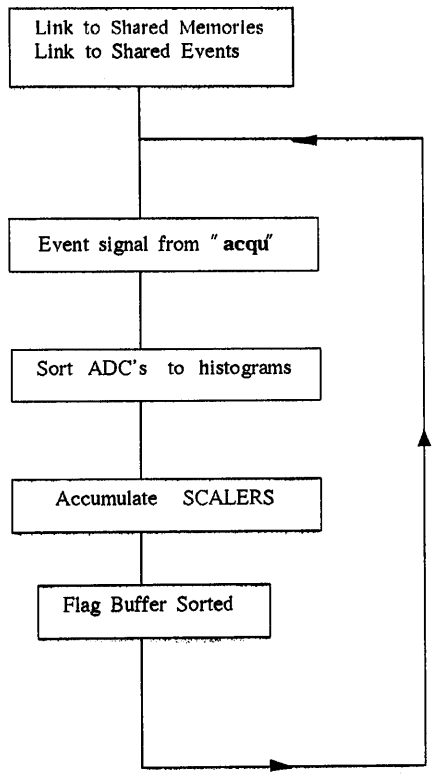


Figure 3.5: **Flow Chart of Subprocess "hist"**

3.3.3 Acquisition System Controlling Task "control"

To provide the interaction between the user and the data acquisition system, a module, named "control" has been written. It allows the user to manipulate the acquisition and also to retrieve status information. To start any acquisition, the user should run "control" and issue the command which enables the interrupts. It is only when the interrupts are enabled, that readout operations are started. If the optional task "hist" has been started, the user can ask at any time for a histogram or for a plot using the appropriate command.

3.3.4 Interrupt routine : CBD-IRQ

For fastest response to any trigger signal generated in an experiment, interrupt driven readout of FASTBUS and CAMAC is used. Normal processes are time sliced by OS9 but interrupt service routines override system time slicing and run with minimal delay.

The interrupt routine CBD-IRQ is part of the Kelvin Laboratory written device driver. The driver can potentially handle a variety of VME-bus modules, but at present only interrupts from the CAMAC branch driver CBD8210 are implemented.

The assembler written interrupt service routine is kept very short to avoid upsetting the OS9 time slicing algorithm, and merely serves to trigger

the otherwise dormant task "acqu" using an OS9 **event**. This triggering takes place within $\sim 150\mu s$ of receipt of the external interrupt signal.

Chapter 4

SMI PROGRAMMING

The model 1821 FASTBUS Segment Manager Interface (SMI) is a programmable FASTBUS master [32, 33]. It was originally designed to readout and test the LeCroy 1800 series of data acquisition modules [34]. As more FASTBUS experience was gained, the 1821 SMI's programmability provided users with some flexibility in designing and implementing FASTBUS data acquisition systems, and it has subsequently been used to control a variety of modules.

The most important application of the SMI is as a segment master. Once programs have been downloaded from a host computer, the SMI can handle bus protocols, and is also capable of such tasks as the writing to or reading from slave modules. It can also perform data compression and pedestal subtraction.

4.1 1821 SMI Hardware

The following description of the SMI is based on the contents of the LeCroy manual [32]. The 1821 SMI is a double width FASTBUS module consisting of two boards, the 1821-1 and the 1821-2. The 1821-1 provides the FASTBUS interface and control. It consists of a high speed ECL sequencer capable of fetching and executing approximately 32 million instructions per second. The sequencer instruction word is 64 bits wide and its memory is 256 words deep. Currently only 48 bits of the instruction word are used. These are divided into 7 fields, each specifying particular operations which can be executed simultaneously.

The different fields are listed in table 4.1. The sequencer instruction set consists of 11 instructions, which are listed in table 4.2. Of the 11 instructions only 6 have been used in our SMI program development. These are : STRT, RETN, NOP, JUMP and CJMP. The use of the instructions NCAL and NRET would have simplified the programming of the SMI, but their operation in practice did not comply with the specification. Because of its high speed and the ability to execute different operations simultaneously, the sequencer can potentially execute over 100 million operations per second. A diagram of the sequencer is given in figure 4.1.

The second board, the 1821-2, provides the host interface system. It consists of 8 I/O registers, sequencer program memories (EPROM and RAM), 4K of 32-bit data memory, 8K of 10-bit pedestal memory and the pedestal subtraction hardware.
Using the 8 I/O registers, the host communicates with all the subsystems of the 1821-2 interface card.

4.1.1 Host I/O registers

Eight 16-bit registers numbered R0 to R7 are employed to latch data passed between the host and the SMI. They are shown in figure 4.2 along with their interconnections. R0 and R3 are configuration registers; R1, R2, R4, R5, and R6 are input/output registers and register R7 is used to generate strobes and

Field	Definition
OP-CODE	Defines the instruction to be executed. There are 11 instructions currently defined
CONDITION CODE MULTIPLEXER	Defines the Condition Code to be tested
BUS DEFINITION	Defines HSDATA and IAD Bus sources
HSDATA	8-bit data field that can be loaded onto the HSDATA Bus
STROBES	Defines the strobes that latch or set different conditions within the sequencer.
DATA CONTROL	Defines the mode of the 32-bit register (either BYTE or WORD), whether data is piped to other subsystems.
FASTBUS PROTOCOL	Defines the FASTBUS lines to be SET/RESET, and the mode (SLAVE or MASTER)

Table 4.1: **Instruction Field Definition**

Instructions	Code	Use
STRT	0h	Fetch address on Initial Word Address lines (IWA)
RETN	4h	Fetch address in Return Address Register (RAR)
NEXT (NOP)	8h	Fetch address in Next Sequential Address Register (NSAR)
JUMP	Ch	Fetch address on HSDATA Bus
CJMP	Ah	Fetch address on HSDATA Bus if CC bit is TRUE, else fetch address in NSAR
CALL	Dh	Fetch address on HSDATA Bus and latch NSAR address into RAR
CCAL	Bh	Fetch address on HSDATA Bus if CC bit is TRUE, else fetch address in NSAR
NCAL	9h	Fetch address in NSAR, and latch it into RAR
NRET	5h	Fetch address in RAR, and latch NSAR into RAR
LSTR	1h	Fetch IWA address, and latch NSAR into RAR
CRET	2h	Fetch address in RAR if CC bit is TRUE, else fetch IWA address

Table 4.2: **Instruction Set Definition**

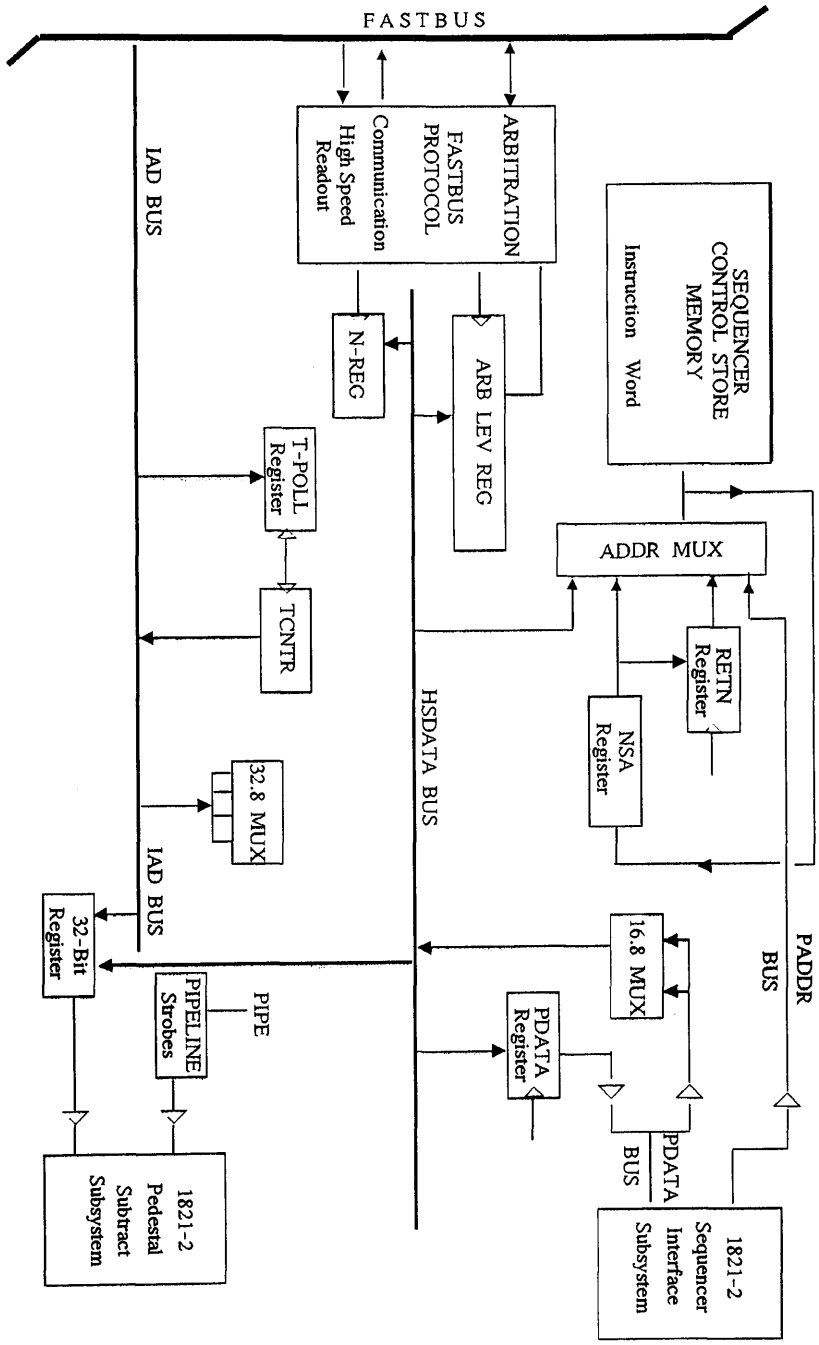


Figure 4.1: **1821 Block Diagram of the Sequencer**

monitors status. A brief description of the registers is given in the following :

Register R0 :

This register is used to control data paths for program download, upload, menu memories and sequencer program memory. Table 4.3 gives the function of each bit.

Register R1 :

This is used to load a start address either for program transfer or for sequencer subroutine execution. Readback of this register gives the address plus two status bits, data available and sequencer (active/wait) status. Operation depends on settings in register R0.

Register R2 :

This is used to download sequencer code or dynamically supply subroutine arguments. It can also be used to read back sequencer code in 8-bit sections and an 8-bit status word containing FASTBUS SS or MS codes. Operation depends on the settings in register R0.

Register R3 :

This is used to control the flow of data from the sequencer and to control the pedestal subtractor and the null data suppressor. The definition of each bit is given in table 4.4.

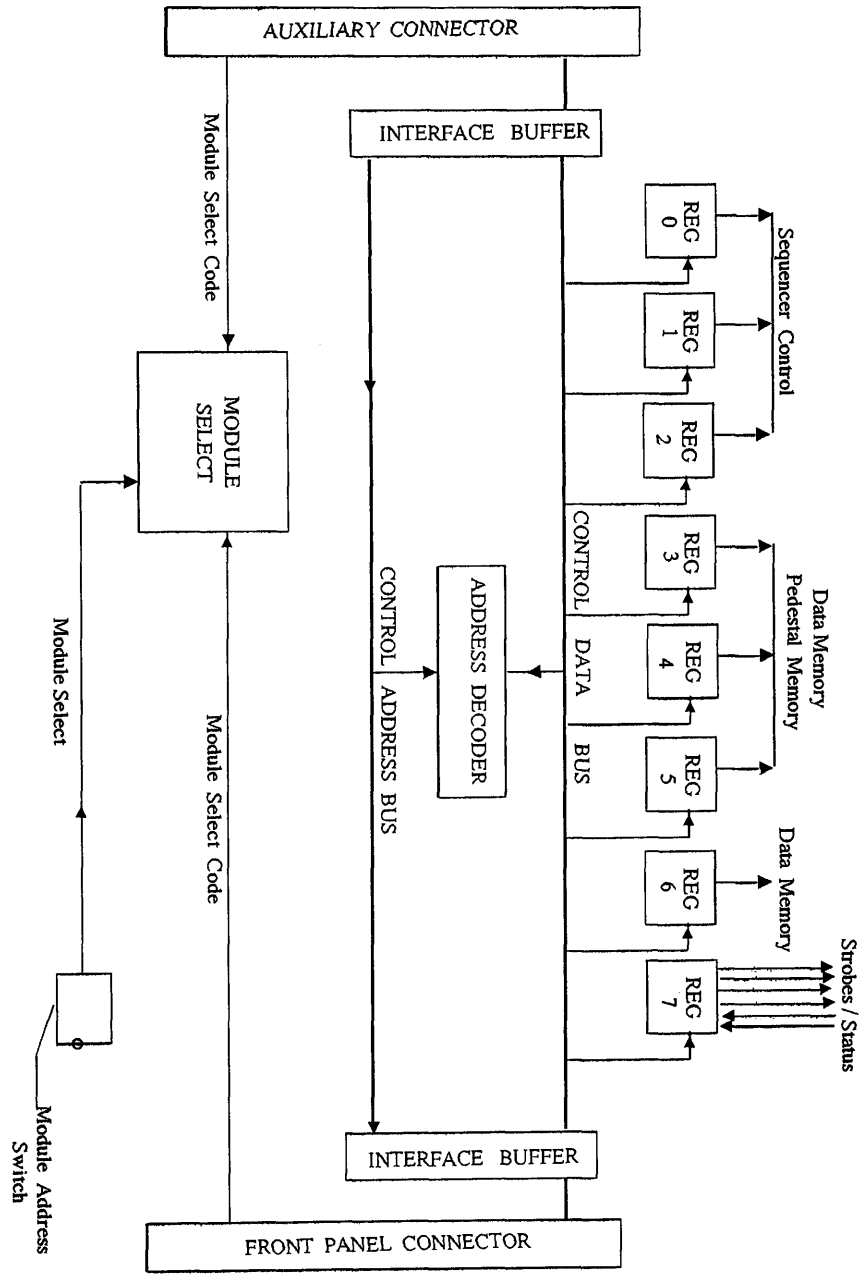


Figure 4.2: **Host I/O Registers**

Bits	Function	
15	Source PADDR bit 0	Src/Dest codes
14	Source PADDR bit 1	0 0 - PMAR (Src); Menu(Dest) 0 1 - Sequencer
13	Destination PADDR bit 0	1 0 - Host I/O register R1 1 1 - Not defined.
12	Destination PADDR bit 1	
11	Source PDATA bit 0	Src/Dest codes
10	Source PDATA bit 1	0 0 - Menu Memory 0 1 - Sequencer
09	Destination PDATA bit 0	1 0 - Host I/O register R2 1 1 - Not defined.
08	Destination PDATA bit 1	
07	Sequencer Program Load Bit	
06	Request Sequencer Attention	
05	User Spare Bit	
04	X	
03	X	
02	MMS2 Menu Memory Select Bits	
01	MMS1	
00	MMS0	

PADDR = Program Address Bus
PDATA = Program Data Bus

Table 4.3: **Register R0 Definition Bits**

The RESET signal is the most important bit. At power-up, register R3 is cleared, immobilising the SMI until a 1 is written to the RESET bit.

Register R4 :

This is mapped to the lower 16 bits of the internal address and data bus. It is used to read from the FASTBUS A/D lines or from the data memory and also to download the data memory for test purposes. Its operation depends on settings in register R3.

Register R5 :

This is mapped to the upper 16 bits of the internal address and data bus and otherwise operates as register R4 .

Register R6 :

The write register stores the starting address and operational mode of the Data Memory Address Register (DMAR), which points to the current location in SMI data memory. The DMAR has auto-increment and auto-decrement modes of operation.
The read register operation provides access to the current value of the DMAR and some data memory status bits.

Register R7 :

This register is used to generate strobes, which are listed in table 4.5.

Bits	Function	
15	RESET	
14	X	
13	Pedestal data	(1=9bit signed, 0=10 bit unsigned)
12	Suppress Zero Numbers	
11	Suppress Negative Numbers	
10	Enable Memory Write Strobe	
09	Select Ped. Mem. as DMB Src/Dest.	
08	Negate Data from Ped. Mem.	
07	X	
06	General Purpose Flags	
05	Aux Connector Control	
04	Internal Control	
03	Source DMB Bit 0	Src/Dest Codes
02	Source DMB Bit 1	0 0 - Data Memory 0 1 - AUX Connector
01	Destination DMB Bit 0	1 0 - Host I/O register R4, R5 1 1 - Pedestal Memory Select.
00	Destination DMB Bit 1	

DMB = Data Memory Bus

Table 4.4: **I/O Register R3 Definition Bits**

The write register operation issues a strobe for each bit set, and multiple strobes are possible.

The read register operation provides status and maintenance bits such as the condition of the DC power.

Up to 16 SMI's may be connected to the 2891A SMI interface. The module select register (table 4.6) specifies which one is addressed and also which port (front or rear panel) of the SMI is used.

4.1.2 ECL Sequencer Control

This subsytem enables the host to program the sequencer and communicate data to and from an executing program. It includes the program data bus, program address bus, menu memories and the program memory address register. There are eight menu memories used to contain program data, which may be downloaded into the sequencer control store memory.

The memory 0 is an EPROM which contains the standard LeCroy SMI code used to initialise the FASTBUS sytem at startup. Memories 1 to 7 are RAM and used to store user written code downloaded from the host.

Bits	Strobes
15	Ped. Data Mem. Write/ Host generate Abort
14	Ped. Data. MemAdr. Latch/ Host Generate RDOC
13	X
12	Ped. Data Comparator Write
11	Read-Out Word Count
10	Pgm. Mem. Write
09	PMAR increment
08	Pgm. Mem. Write
07	X
06	Data mem. Write
05	DMAR count
04	DMAR load
03	ROWC load
02	Initiate auto-download to pgm Mem.
01	zero download address register
00	Sequencer GO

Table 4.5: **Output Register R7 Strobes**

CAMAC Write — Lines —	Operations
W1-W4	Address of peripheral (1821) with which to communicate
W5	0 = Enable Front Panel of 1821 1 = Enable Rear Panel of 1821
W6	0 = Bypass 1 = Normal addressing

Table 4.6: **Module Select Register Bit**

4.1.3 Pedestal Subtractor

The 1821-2 board comes equipped with pedestal subtraction and zero suppression hardware. Both communicate with the data memory through the sequencer's data path. The subsystem was designed to operate with LeCroy ADC's which have no data compression capability. However the Phillips ADC's and TDC's used at the Kelvin Laboratory perform zero suppression and pedestal subtraction operations, so these facilities are not used, although the data still pass through the compression pipeline.

4.1.4 Data Memory

The data memory is used by the host to store and retrieve 32-bit data words read from the FASTBUS crate segment. The data compression pipeline can supply data at rates up to 10 MWord/sec over the 32-bit wide data memory bus, and data may be stored in the data memory or passed directly to the auxiliary connector at these very high rates. Figure 4.3 shows the connections to and from the data memory.

The data memory is 32 bits wide and 4096 words deep (16kbytes). Data are passed to and from the data memory over the data memory bus (DMB) and addressing is supplied by the data memory address register (DMAR).

The DMB is a 32-bit bi-directional bus connecting the data memory to the data

compressor, registers R4 and R5 and the auxiliary connector.

The (DMAR) is a 32-bit preloaded up/down counter used to address the data memory over an address range of 0-4095. Host input register R6 is used to read the current DMAR value and operating mode. Host output register R6 provides the DMAR with its initial value.

4.2 The 1821 SMI Instruction Word

Figure 4.4 shows in detail the seven fields of the instruction word, and the definition of all 48 bits used.

4.2.1 Op-code

The op-code field (bit 0 to bit 3), is loaded with one of the instructions listed in table 4.2 and specifies the basic operation.

4.2.2 Condition Code Multiplexer

Bits 4 to 11 define the Condition Code Multiplixer, through which the state of over 100 hardware lines may be tested. These include FASTBUS master, slave, bus management signals, internal timers, host interface lines and many others. The appropriate test condition must be selected with the condition code

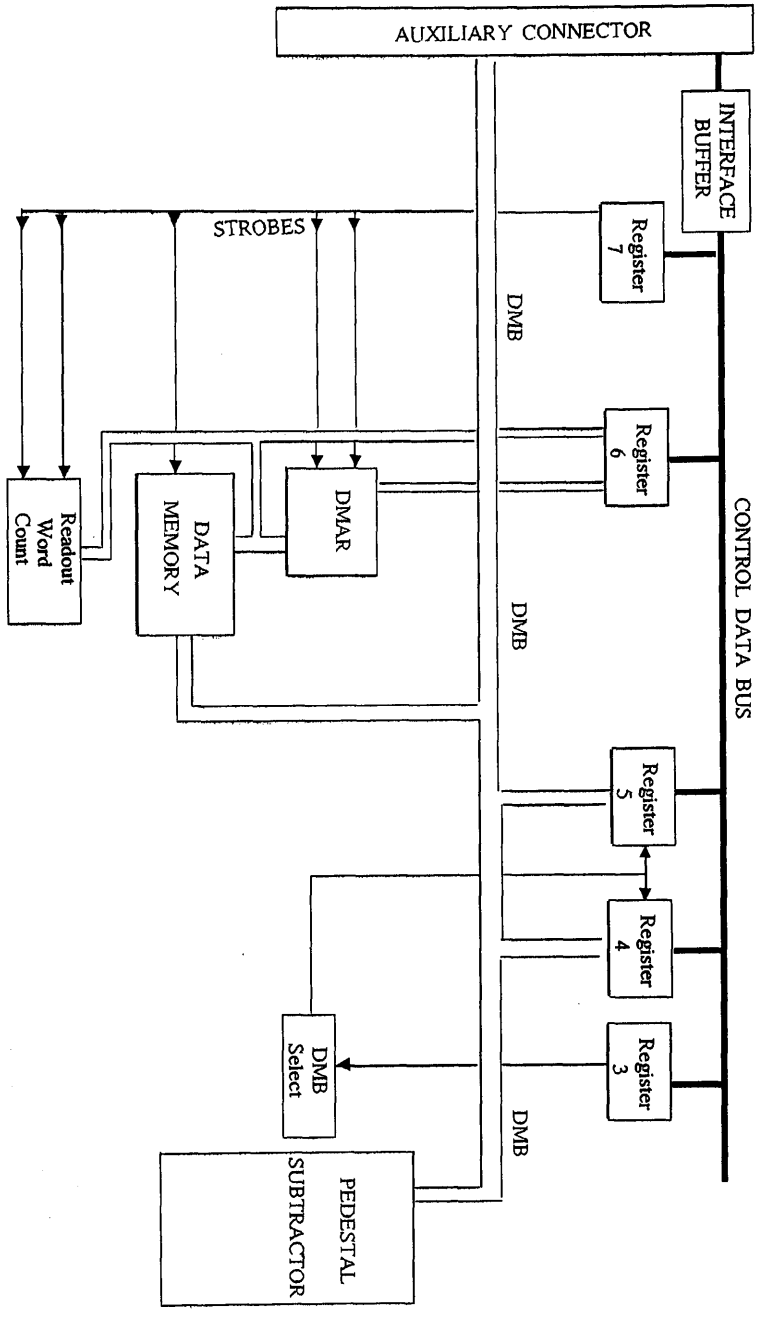


Figure 4.3: **SMI Data Memory**

OP-CODE	CONDITION CODES	BUS DEFINITION	HS DATA	STROBES	DATA CONTROL	FASTBUS PROTOCOL
0 1 2 3	4 5 6 7 8 9 10 11	12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31	32 33 34 35	36 37 38 39 40 41 42 43 44 45 46 47
STRT = 0	bit 4 = CCB7	bit 12 & bit 13 define HSDATA Source	bit 16 = HS 7	bit 24 = search	bit 32 = 0 = byte 1 = word	bit 36 = FBOUT
RETN = 4	bit 5 = Not CC		bit 17 = HS 6	bit 25 = RCT/SHIFT		bit 37 = 0 = RESET = 1 = SET
NOP = 8	bit 6 = CCB5		bit 18 = HS 5	bit 26 = TFCT 1 ┐ ┌ NREG		
JMP = c	bit 7 = CCB4		bit 19 = HS 4	bit 27 = TFCT 0 ┘ └	bit 33 = pipe	bit 38 = 0 = SLAVE = 1 = MASTER
CJMP = a	bit 8 = CCB3	bit 14 & bit 15 define IAD Source	bit 20 = HS 3	bit 28 = TCNT	bit 34 & bit 35 = byte number 0 = byte 0 1 = byte 1 2 = byte 2 3 = byte 3	bit 39 = 0 = COMMUNIUCATION = 1 = BUS
CALL = d	bit 9 = CCB2		bit 19 = HS 2	bit 29 = TIMER RESET		
CCAL = b	bit 10 = CCB1		bit 20 = HS 1	bit 30 = EDREG		bit 40 = fbp 7
NRET = 5	bit 11 = CCB0		bit 21 = HS 0	bit 31 = 32B REG		bit 41 = fbp 6
LSTR = 1						bit 42 = fbp 5 = RB, GK, AK, AS
CRET = 2	CCB 7 = multiplex					bit 43 = fbp 4 = BH, AGK, DK, DS
						bit 44 = fbp 3 = WT, RDOC, TP, RD
						bit 45 = fbp 2 = SR, EDEnb, SS2, MS2
						bit 46 = fbp 1 = AI, EAI, SS1, MS1
						bit 47 = fbp 0 = AG, EG, SS0, MS0

Figure 4.4: **Instruction Word**

multiplexer on the preceding instruction. For example, the user would use the CJMP instruction to branch to an address specified on the HSDATA bus if the condition code (CC) specified in the preceding instruction was true.

4.2.3 Bus Definition

Bits 12 to 15 constitute the Bus Definition field, which defines the IAD bus source (ISRC) and the HSDATA bus source (HSRC), as shown in table 4.7.

4.2.4 HSDATA Bus

The High Speed Data bus (HSDATA) (bit 16 to bit 23) can be loaded from internal 1821 registers, from the instruction word, or from FASTBUS depending on the state of the bus definition field. The Internal Address bus (IAD) can be driven by internal 1821 registers or FASTBUS. When the HSDATA bus is driven by the instruction word, the data are derived from the HSDATA field immediately following the bus definition field.

4.2.5 Strobes

The 8-bit strobe field, (bit 24 to bit 31), allows the user to control the function of the TCNT and TPOLL registers, reset internal timers, load the PDREG and load the 32-bit register.

HSDATA Src.		
	ROT/SHF=0	
		0 = 16:8 data mux
		4 = inst. word HSDATA
		8 = 32:8 IAD mux
		c = TCNT register
	ROT/SHF=1	
		c = NREG register
IAD Src.		
	ROT/SHF=0	
		0 = FASTBUS A/D
		1 = 32-bit register
		2 = 5 bit TCNT register
		3 = 8 bit TCNT register
	ROT/SHF=1	
		0 = IAD bus
		bits 12-15 = bus definition
		bits 25 = strobe (ROT/SHF)

Table 4.7: **Bus Definition Bits**

4.2.6 Data Control

The data control field, (bit 32 to bit 35), allows selection of the operational mode of the 32-bit register (either byte or word). Bits 34-35 define the byte number (0,1,2,3).

4.2.7 FASTBUS Protocol

The FASTBUS protocol field, (bits 36 - 47), allows the user to set or clear various FASTBUS control lines. Different lines are set or cleared depending on the mode (master or slave), which is selected by bits 38-39. Figure 4.5 shows all the different combinations.

4.3 1821 SMI code Developments

Ideally one would use an assembler to generate SMI op-code, and a LeCroy product which runs on IBM PC's [35] was examined with a view to conversion for the present purposes. While in the long term this is a desirable goal, in the short term it was quicker to program the SMI op-code by hand, a delicate task requiring careful attention to detail.

The SMI code files were created using a text editor. Each line corre-

	Bus Definition Bits		Protocol Definition Bits					
bits / mode	38	39	42	43	44	45	46	47
slave communication	0	1	AK	DK	TP	SS2	SS1	SS0
slave bus	0	1	RB	BH	WT	SR	AI	AG
master communication	1	0	AS	DS	RD	MS2	MS1	MS0
master slave	1	1	GK	AGK	RDOC	RDEnb	EAI	EG

Figure 4.5: **Protocol Bits Definition**

sponds to an instruction word. The format used for a line follows the structure of the instruction word. Comment lines can be included, which aid the understanding of the code file. These start with "comm:" and the instruction lines start with "line:". The instruction line read from the code file includes the instruction number and spaces to separate the different fields. A load function separates out the code and sends it to the sequencer control store memory.

Here is an example of a code file. Note that all numbers used within the SMI code are given in hexadecimal.

```
comm:          Sequencer Idle Loop          comments
line: 00  8  42  4  00  00  0  2  00  Master Mode
line: 01  a  42  4  01  00  0  2  00  wait for host ignition
line: 02  0  00  4  00  00  0  0  00
```

The instruction words sent byte by byte to the sequencer control store memory would be as follows :

```
842400000200
a42401000200
000400000000
```

Once the SMI code is loaded into the sequencer control store memory, the sequencer automatically enters into an idle loop, located at address zero and shown in the previous example. To perform any useful task, the host must pass the start address of the relevant subroutine to the sequencer and request execution.

Two different SMI codes have been developed at the Kelvin Laboratory. These are called CODE1 and CODE2. In the first attempt, we have developed

CODE1 which uses the slow connection at the front panel of the 1821 SMI for module initialisation and data readout. CODE1 has been developed to be as simple as possible and is actually being used in the first experiments performed at Mainz.

To improve the data throughput, it was decided to develop CODE2 which uses the rear panel auxiliary connection to a fast VME memory. CODE2 has improved considerably the speed of data transfer to the host by virtually eliminating host intervention in data readout and by using a considerably faster hardware link. The following sub-sections, give details of both codes, CODE1 and CODE2.

4.3.1 Load/Exec function

After power-up the user must download a program to the sequencer control store memory to enable the sequencer to perform useful functions at the request of the host. Access to the sequencer control store memory is through the host registers R0, R1, R2, and R7. Sequencer programs can be loaded either directly from the host or from one of the eight local menu memories. The LOAD and EXEC functions are described in appendices A and B.

4.3.2 Front-panel code: CODE1

CODE1 is structured as a main routine which performs calls to separate subroutines to perform specific tasks.

The 1821 manual includes some basic SMI routines which have been adapted to develop a customised FASTBUS readout code. Readout of a FASTBUS module involves 3 sequences :

1 - The Primary Address Sequence,

2 - The Secondary Address Sequence,

3 - The Block Read Sequence.

1- Primary Address Code

The addressing of the slave with which the master will communicate is performed during the primary address cycle. Figure 4.6 gives the flow chart of the primary address routine. Since primary addressing involves writing to the addressed module, the RD lines are maintained at zero. The EG line is asserted and the desired module address is placed on the AD lines.
When calling the primary address routine, the calling routine should have previously loaded the primary address into the TCNT register and set the appropriate MS codes. Here MS=0 (see MS code table 4.8).

2- Secondary Address Routine

The secondary address operation, shown by the flow chart of figure 4.7,

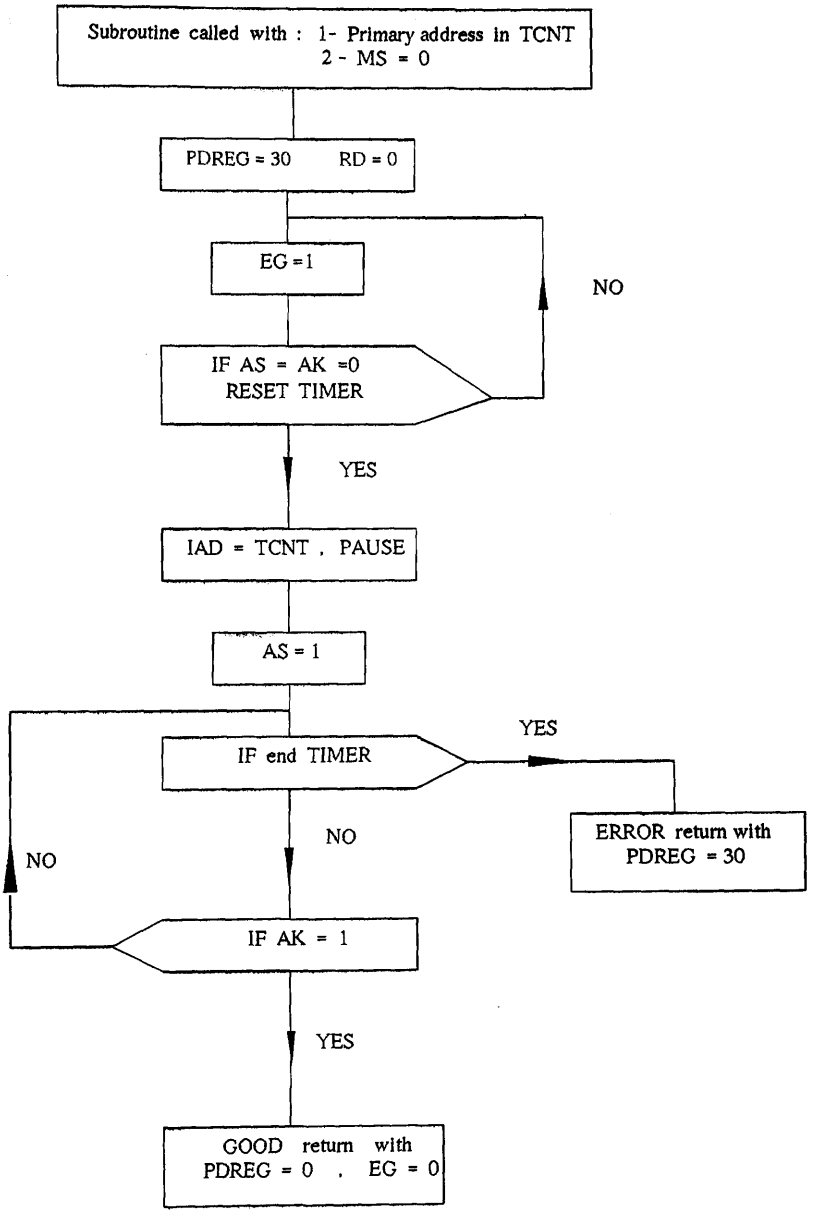


Figure 4.6: **Primary Address Routine**

MS code for Address Cycles		
MS		**Address Type**
0	—-	Specific Device - Data Space
1	—-	Specific Device - CSR Space
2	—-	Broadcast - Data Space
0	—-	Broadcast - CSR Space
4,5	—-	Reserved - Specific Device
6,7	—-	Reserved - Broadcast

MS code for Data Cycles		
MS		**Interpretation**
0	—-	Random Data
1	—-	Block Transfer - Handshake
2	—-	Secondary address
3	—-	Pipelined Transfer - (non-Handshake)
4-6	—-	Reserved - (Handshake)
7	—-	Reserved - Pipeline

SS codes		
SS		**Interpretation**
0	—-	Command accepted, no problem
1	—-	Module is currently digitising an event
2	—-	empty or full
3	—-	—
4	—-	Not used
5	—-	Not used
6	—-	R/W from a non-existent register
7	—-	Secondary write to a non-existent register address

Table 4.8: **FASTBUS MS and SS codes**

is executed in a write data cycle. The routine is called with the secondary address in the 32-bit register, RD=0 , MS=2 (see MS code table 4.8) and the PDREG register initialised with the value 28. The PDREG register is used for error diagnostics. The routine asserts DS (Data Synch) and waits for the Data Acknowledge signal DK set by the slave.

The operation is terminated by the host removing all its signals (including DS) from the bus.

3- Block Read Routine

A flow chart of the block-read routine is shown in figure 4.8. The routine first sets RD to initiate a read data operation and then ensures that DK is reset. The PDREG register is loaded with the appropriate diagnostic code and the appropriate MS code is asserted. Here MS=1 means select block-read (see MS code table 4.8). The routine then asserts DS and waits for the acknowledge DK. If DK is not received in time, the routine exits with a timeout error. After DK is received, the data transfer occurs.

When all the module's data are successfully transferred, the slave responds with SS=2 (see SS codes in table 4.8). To facilitate debugging of the SMI code by the host, the PDREG register is assigned different values depending on progress through the address/read sequence.

The above three sequences are called by the CODE1 main routine listed in the following, and outlined in the flow chart of figure 4.9.

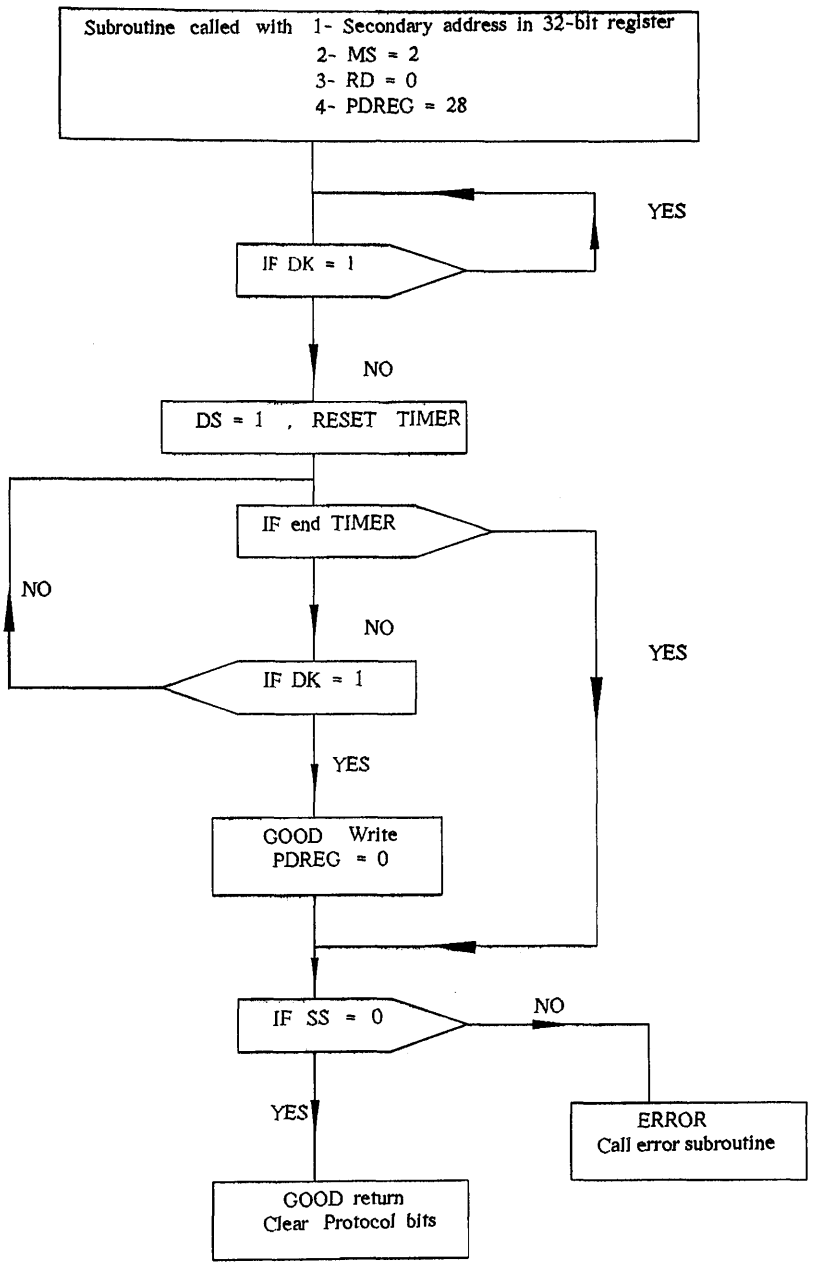


Figure 4.7: **Secondary Address Routine**

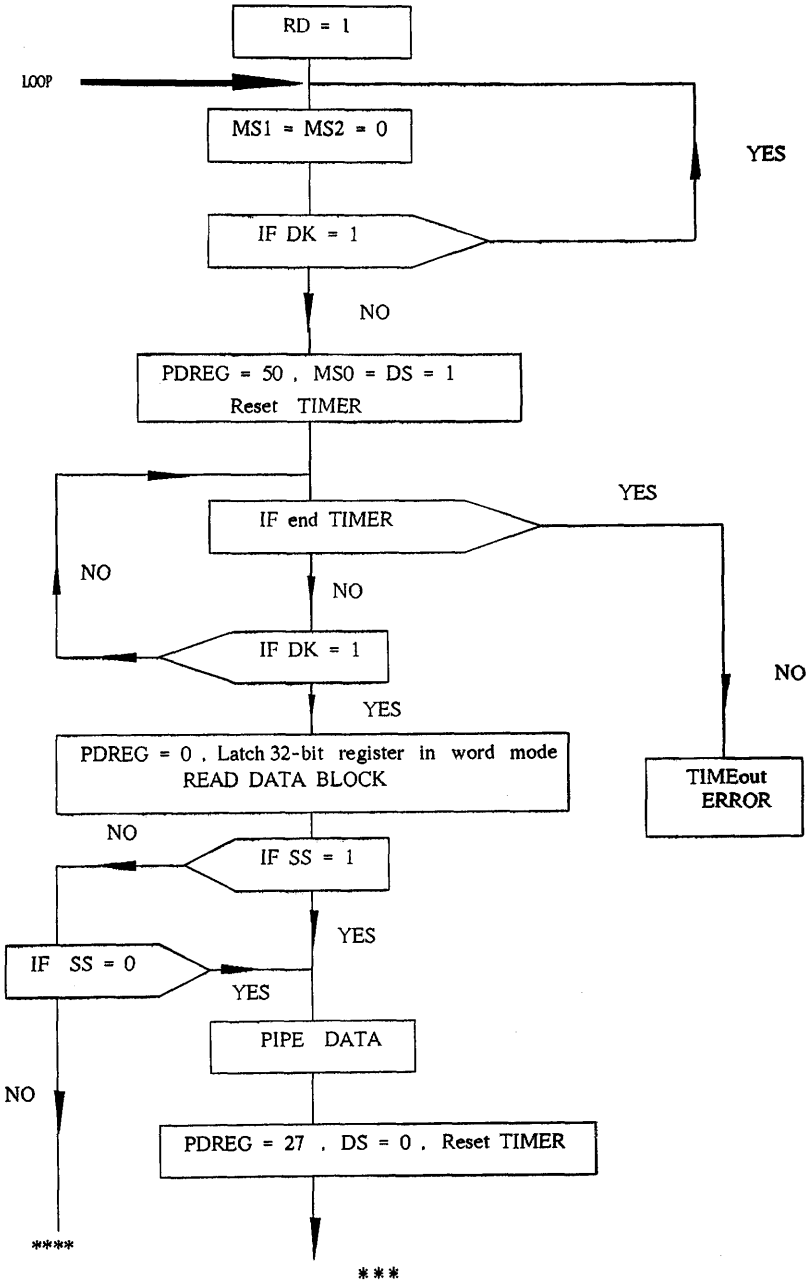


Figure 4.8: **Block Read Routine**

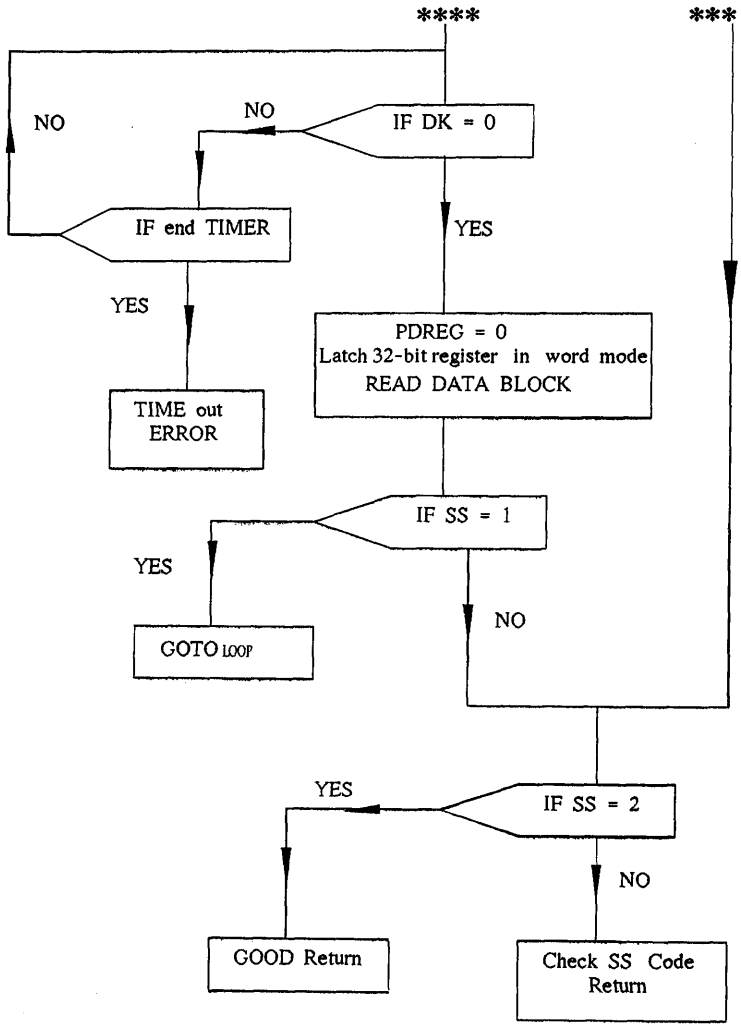


Figure 4.8 : **Block Read routine continued**

comm:	CODE1 main routine								comments
comm:									
line: 85	d	00	4	fb	00	0	0	00	call fb : init Fastbus Prot.
line: 86	8	00	4	00	00	0	2	3f	
line: 87	8	00	0	00	08	1	2	3f	get station in TCNT
line: 88	d	00	4	eb	00	0	0	00	call eb : pri. adr.
line: 89	a	00	4	00	00	0	0	00	pause
line: 8a	8	00	4	00	08	0	0	00	TCNT = 0
line: 8b	8	00	0	00	01	0	0	00	32-bit = sec. adr.
line: 8c	8	00	4	28	02	0	2	1f	
line: 8d	d	00	4	d9	00	0	6	02	call d9–Fb write
line: 8e	d	00	4	bc	00	0	2	10	call bc : Block Read
line: 8f	c	00	4	00	00	0	0	00	go to idle loop

To read a FASTBUS module with CODE1, the host issues an EXEC to address "85". Whereas standard LeCroy code needs three consecutive EXEC calls for each block-read, CODE1 needs only one.

Before the host starts CODE1, it puts primary and secondary address parameters into output register R2. Byte 0 contains the primary address and byte 1 contains the secondary address. This requires less host intervention than the standard LeCroy code where the address parameters are passed in separate operations.

The execution of CODE1 starts by performing the FASTBUS protocol initialisation. The FASTBUS initialisation sub-routine located at the address "fb" is taken from the standard LeCroy code. It clears the TCNT register, 32-bit register and the protocol bits.

Before calling the primary address sub-routine located at address "eb",

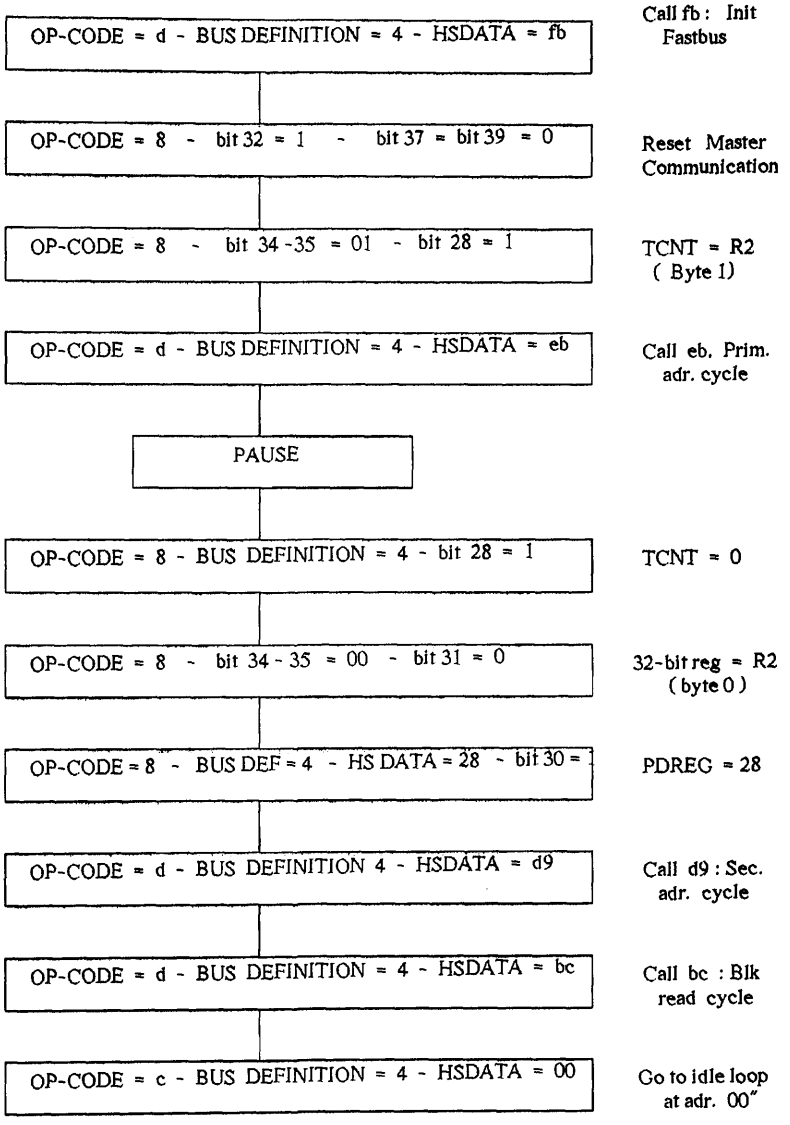


Figure 4.9: **CODE1 Main Routine**

the primary address is latched into the TCNT register from byte 1 of output register R2 (inst "86-87"). Byte mode is enabled by setting bit 32 and byte 1 is selected with bits 34-35 = 01.

A pause instruction has been added to allow signals to settle down at the end of the operation, and the TCNT register is cleared for future use.

The secondary address parameter contained in byte 0 of output register R2 is latched into the 32-bit register. The 32-bit register is selected by setting bit 31, and byte 0 is selected with bit 34-35 = 00. The write sub-routine located at address "d9" is then called to perform the secondary addressing cycle.

Once the primary and secondary addressing cycles have been successfully excuted, the block-read sub-routine located at address "bc" is called. This transfers digitised data into the 1821 SMI data memory. The data transferred are either from one TDC/ADC, if the module is configured for normal block readout, or from a group of TDC/ADC modules if configured for MULTIBlock readout.

At the end of a block readout, CODE1 goes back to the idle loop.

4.3.3 Host-CODE1 Interaction function

The host-CODE1 interaction function, "R-block", is written in C and is shown in the flow chart of figure 4.10. It is called for each block-read required.

"R-block" first performs I/O register configuration. It enables the auto increment mode of the DMAR by setting output register R6 bits 12 and 13, [R6=3000(hex)], and strobes a DMAR load by setting bit 4 of register R7. Output register R3 is loaded with configuration data 840c(hex). This selects the pedestal subtract pipeline as the source of the DMB, with the data memory as the destination. Once configuration is completed, "R-block" calls the EXEC function to execute the readout operation.

At the end of a block-read, "R-block" transfers the data from the data memory to a VME buffer BUFF1, from where a decode function sorts it into a standard format and stores it in another VME buffer BUFF2.

4.3.4 Rear-panel code: CODE2

As with CODE1, CODE2 is structured as a main routine which calls separate subroutines to perform different tasks, but it does more than CODE1. Whereas CODE1 needs host intervention to pass parameters, start the SMI code and transfer data from the SMI data memory, CODE2 needs only one EXEC command before data taking starts. In fact, CODE2 contains the primary and secondary addresses to pass to the appropriate sub-routines, whereas in CODE1 they are passed by the host. Address instructions "40" to "7c" (60 addresses) are reserved for this purpose. As each module needs two address instructions, one for the primary address and the other for the secondary address, a full crate

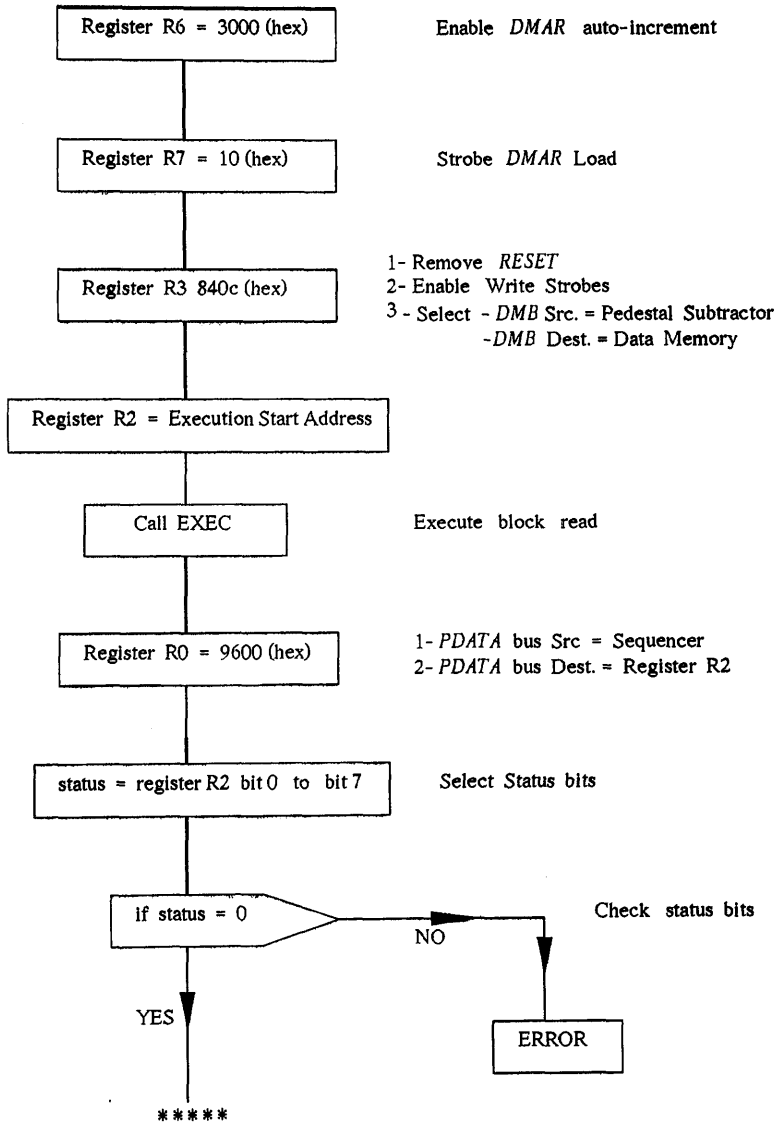


Figure 4.10: R-block Function

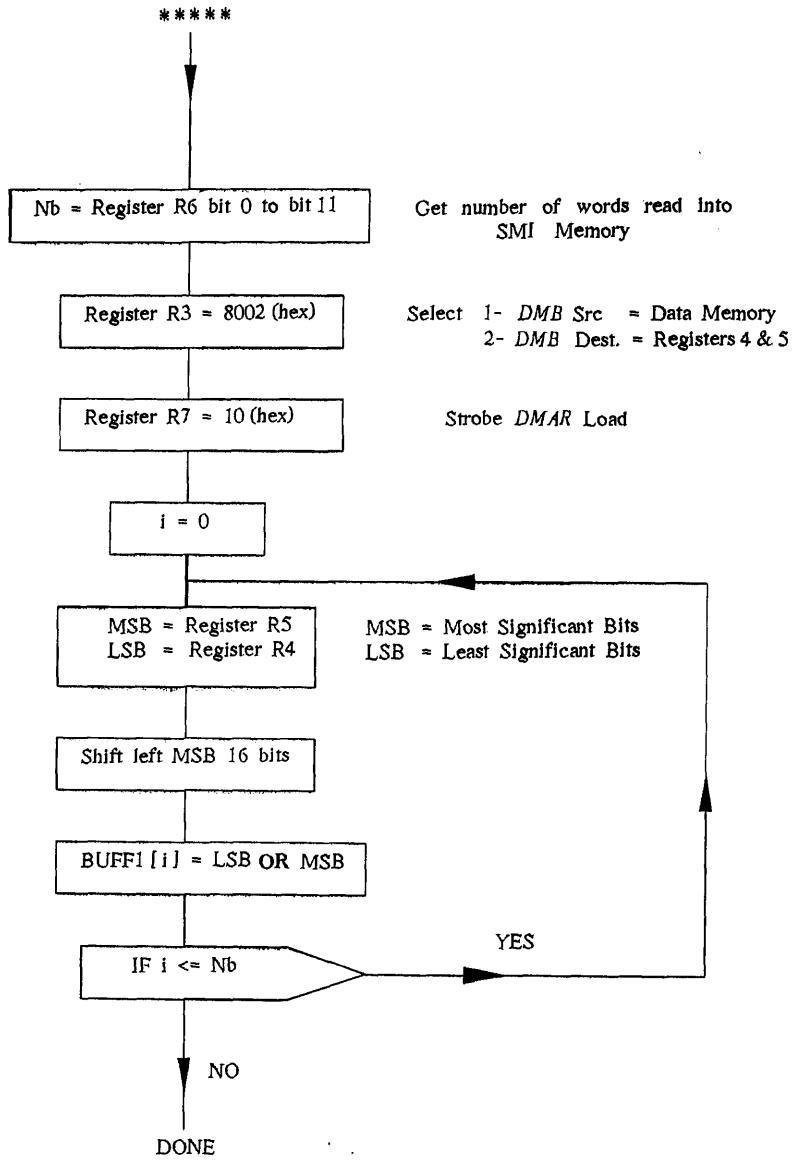


Figure 4.10 : **R-block function continued**

of 24 modules can be read out by CODE2.

Instructions "41" to "7c" are not included in the CODE2 data file when it is created under the text editor, and from line "40" to line "7f", it looks as follows :

```
comm:   start of module slot and sub-adr
line: 40  4  00  4  00  00  0  0  00   return to calling routine
comm:
comm:          end of module table
line: 7e  8  00  4  00  02  0  0  42   INP1=0,good return
line: 7f  c  00  4  30  00  0  3  08   jump to 30. read scan loop
```

Instead, at the FASTBUS initialisation stage the acquisition system generates the address lines from information read in from module specification parameter files. A flow chart of the C function "SMI-code-gen", which generates SMI code address lines is given in figure 4.11. The function uses 2 vectors, "prim[0-7]" and "sec[0-7]", corresponding respectively to the primary and secondary address lines. A third vector "end[0-7]" is used to generate the last instruction, which performs a branch (op-code = "JMP" = c) to the instruction "7e". The HSDATA field of the instruction word, corresponding to byte 2 of the vectors, contains the primary or secondary address. As the "prim" and "sec" instructions are called from the main routine, the address instruction op-code is "RETN" = 4, which performs a return to the calling routine.

The "SMI-code-gen" function returns one of two status flags as follows :

0 = OK, good return

W = W SMI write error during operation

An example of generated code is given in the following. Suppose we have 3 ADC's located at the stations 10, 15, 19, which are read via data register DSR2.

The generated instruction "41" to "47" will look like this :

```
comm:   start of module slot and sub-adr
line: 40  4  00  4  00  00  0  0  00
comm:
line: 41  4  00  4  10  08  0  2  3f   TCNT = 10 :fastbus slot 10
line: 42  4  00  4  02  01  0  0  00   32-bit = 02 : data reg. 2
line: 43  4  00  4  15  08  0  2  3f   TCNT = 15 : slot 15
line: 44  4  00  4  02  01  0  0  00   32-bit = 02 : data reg. 2
line: 45  4  00  4  19  08  0  2  3f   TCNT = 19 : slot 19
line: 46  4  00  4  02  01  0  0  00   32-bit = 02 : data reg. 2
line: 47  c  00  4  7e  00  0  0  00   br to 7e:end fst read
comm:
comm:          end of module table
line: 7e  8  00  4  00  02  0  0  42   INP1=0,good return
line: 7f  c  00  4  30  00  0  7  08   set RDOC, jump to 30.
```

As with CODE1, CODE2 enters into the idle loop located at the address "00" when it is loaded into the sequencer control store memory. When the sequencer execution is called by the host, CODE2, instead of performing a read operation as CODE1 does, branches to a real-time loop located at the address "30". In fact the data readout in CODE2 is initiated by the front panel signal *INP1* of the 1821 SMI. So when the host starts the execution of CODE2, it is not involved any more, as at the end of readout the sequencer returns to the real-time loop.

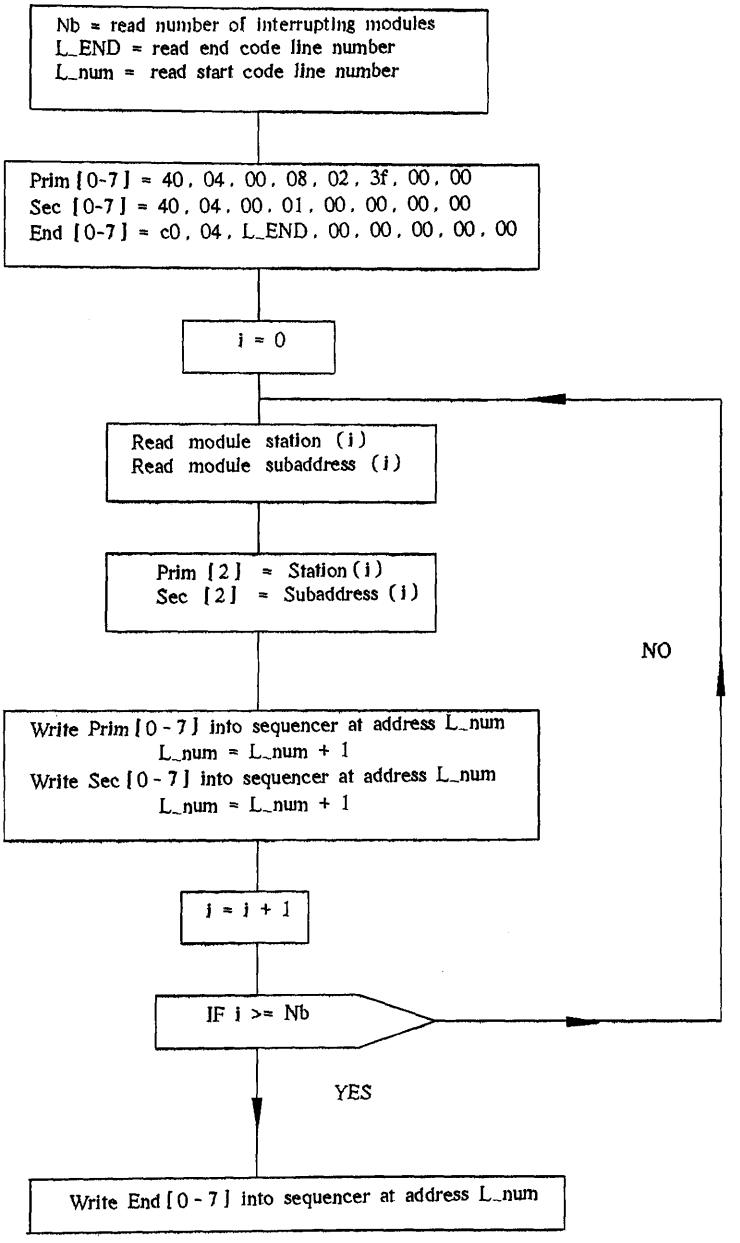


Figure 4.11: SMI-code-gen Function

The SMI code for the real-time loop outlined in the flow chart of figure 4.12 is as follows :

comm:	CODE2 real-time loop								comments
line: 30	8	8d	4	00	02	0	7	20	tst HOST bit
line: 31	a	8d	4	35	00	0	0	00	br to 35 if HOST
line: 32	8	f0	4	00	00	0	0	00	tst not INP1
line: 33	a	f0	4	30	00	0	0	00	loop back if not INP1
line: 34	c	00	4	84	00	0	0	42	clr INP1. br to 84: read
line: 35	c	00	4	00	08	0	0	00	br to idle loop

CODE2 checks if the host demands attention through the HOST bit (bit 5 of output register R0). This bit is set by the host when the user requests a stop to the data readout and the SMI returns to the idle loop. The host must call sequencer execution to activate CODE2 again. The branch to the main FASTBUS readout routine depends on the INP1 signal state. When set, it is cleared and a branch is made to readout.

The SMI code of the CODE2 main routine shown in the flow chart of figure 4.13 is as follows :

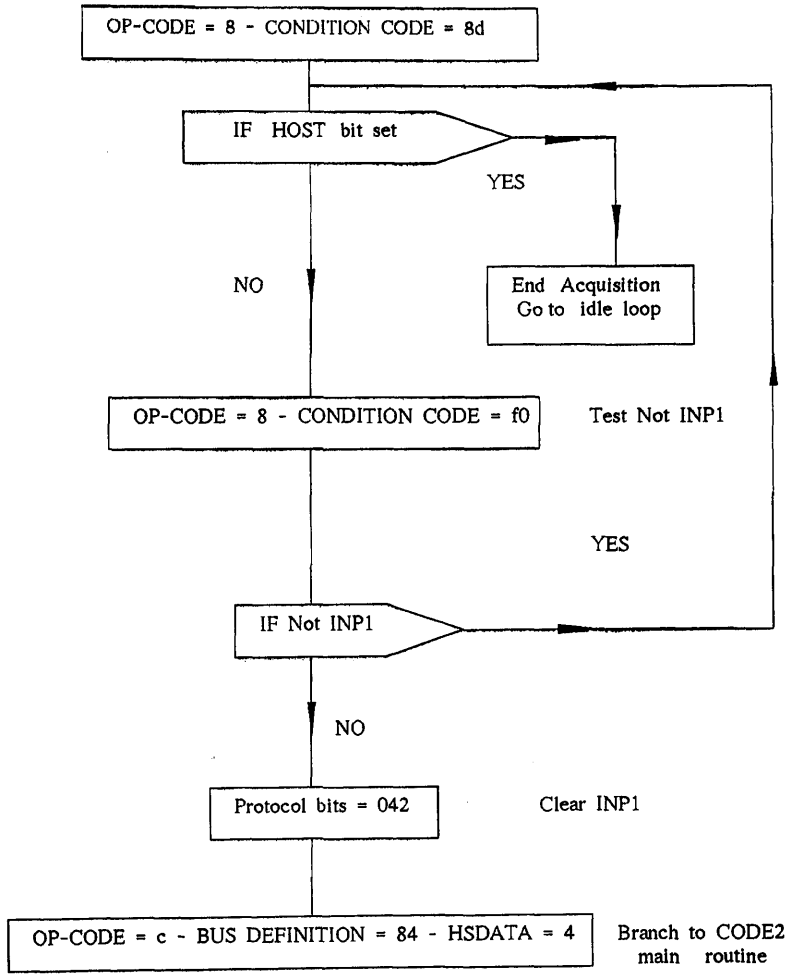


Figure 4.12: CODE2 Real-Time Loop

comm:	CODE2 main routine								comments
line: 84	8	00	4	40	02	3	5	80	NREG = 40
line: 85	d	00	4	fb	00	0	0	00	call fb
line: 86	8	00	4	00	12	3	5	80	incr NREG
comm:									
line: 87	d	00	c	00	42	3	5	80	get prim. adr
line: 88	d	00	4	d1	00	0	0	00	call d1 : exec. prim. adr
comm:									
line: 89	a	00	4	00	00	0	0	00	pause
line: 8a	8	00	4	00	12	3	5	80	incr NREG
comm:									
line: 8b	d	00	c	00	42	3	5	80	get sec. adr
line: 8c	8	00	4	28	02	0	2	1f	
line: 8d	d	00	4	d9	00	0	6	02	call d9 : exec. sec. adr
comm:									
line: 8e	d	00	4	bc	00	0	2	10	call blk read bc
line: 8f	c	00	4	85	00	0	0	00	br to 85 for next module

First, the module table start address ("40") is put into the NREG register. After the FASTBUS protocol initialisation, the NREG register is incremented to point to the first primary address instruction.

By performing a call to the address contained in NREG, CODE2 gets the primary and secondary address parameters. NREG is incremented after each call operation. As with CODE1, once primary and secondary address cycles are performed, CODE2 executes the block-read routine located at the address "bc". At the end of the block-read, unlike CODE1 which returns to the idle loop, CODE2 loops again to read the next module.

The last instruction in the module address table performs a branch to the instruction "7e". To allow the next readout operation, INP1 is cleared (instruction "7e") and the final operation is to set RDOC. This SMI front panel

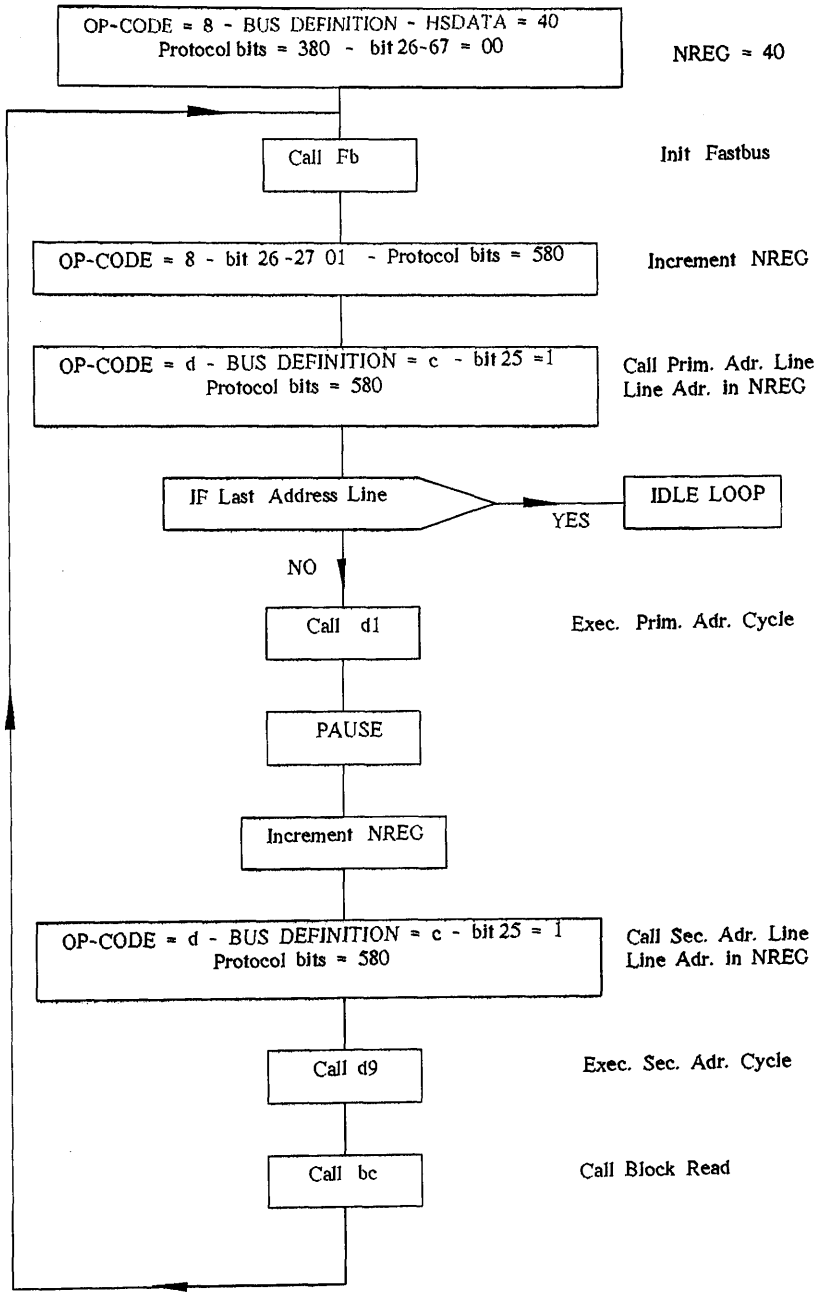


Figure 4.13: CODE2 Main Routine

output can be connected to the INP1 input of a subsequent SMI, so that several SMI's can be "daisy chained" for readout purposes.

The data are transferred via the rear panel auxiliary SMI port and a 32-bit ECL bus to a high speed VME-based memory HSM8170, where it is read by the host and decoded into BUFF2, as in the CODE1 sequence of operations. At start-up, the acquisition system initialises the HSM8170 registers and RAM. The flow chart of the HSM8170 initialisation function "HSM-init" is given in figure 4.14. First, to avoid any disruption during the HSM8170 configuration, the ECL bus port to RAM is disabled (control register bit 12 = 0). The address pointer register is initialised to point to the RAM address 00000(hex) (Address pointer register bits 0 - 18 = 0), and the RAM size is set to fffff(hex) (W-count register bit 0 - 18 = 1). The ECL bus port is then enabled and the RAM is initialised with zero values.

4.3.5 Host-CODE2 Interaction function

The host-CODE2 interaction requires only two operations, one to start CODE2 and the other to stop CODE2. For this purpose, two functions have been added within the control function CONTROL. These are : "start-smi" and "stop-smi". A flow chart of the "start-smi" function is shown in figure 4.15. First, output register R3 is loaded with the configuration 840d(hex). This removes the RESET signal (bit 15 = 1) and enables memory write strobes from the sequencer pipeline

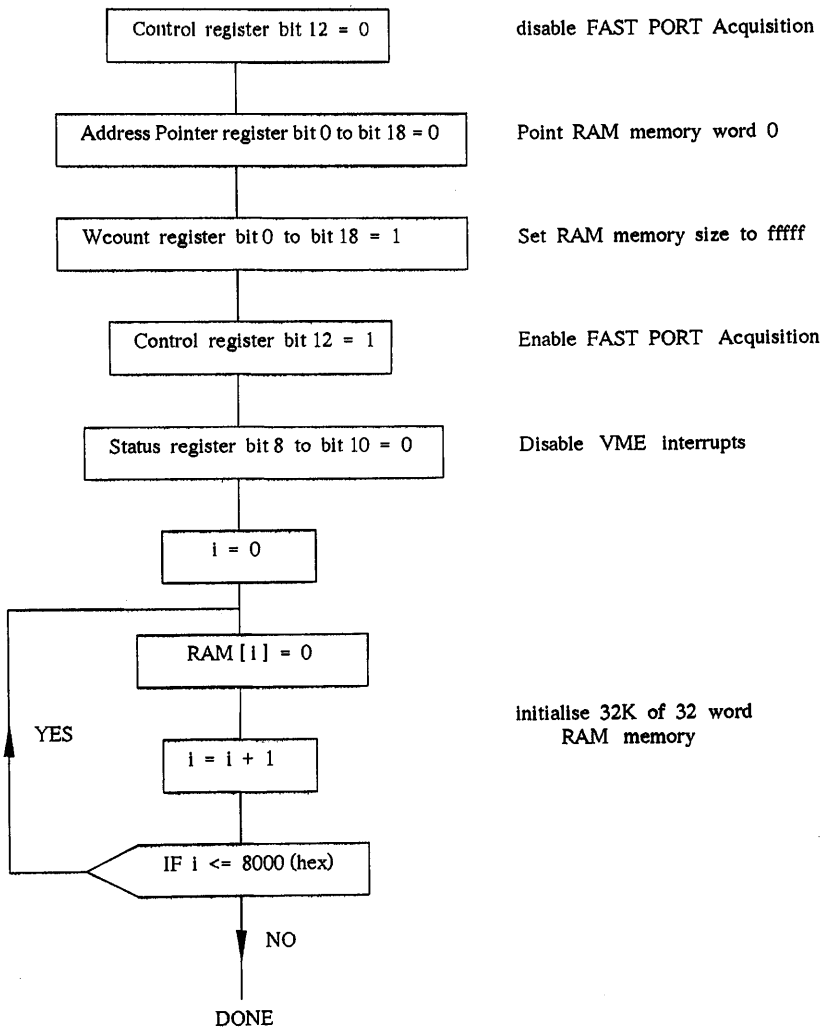


Figure 4.14: HSM-init Function

(bit 10 = 1). Loading the configuration bits 0 to 3 with the value d(hex) selects the pedestal subtract subsystem output as the source of the DMB, with the AUX connector as its destination. The "stop-smi" function sets HOST bit (bit 5 of register R0) and disables the interrupts.

The EXEC function is then called. This puts CODE2 into the real-time loop. Two tests are implemented within the real-time loop. The first tests the HOST bit (bit 5 of I/O register R0) and the second one tests the front panel signal INP1.

The HOST bit is cleared by loading output register R2 with the configuration 9600 (hex). Also, this selects the sequencer as the PDATA bus source, with output register R2 as its destination. Input register R2 is then read to get the execution status.

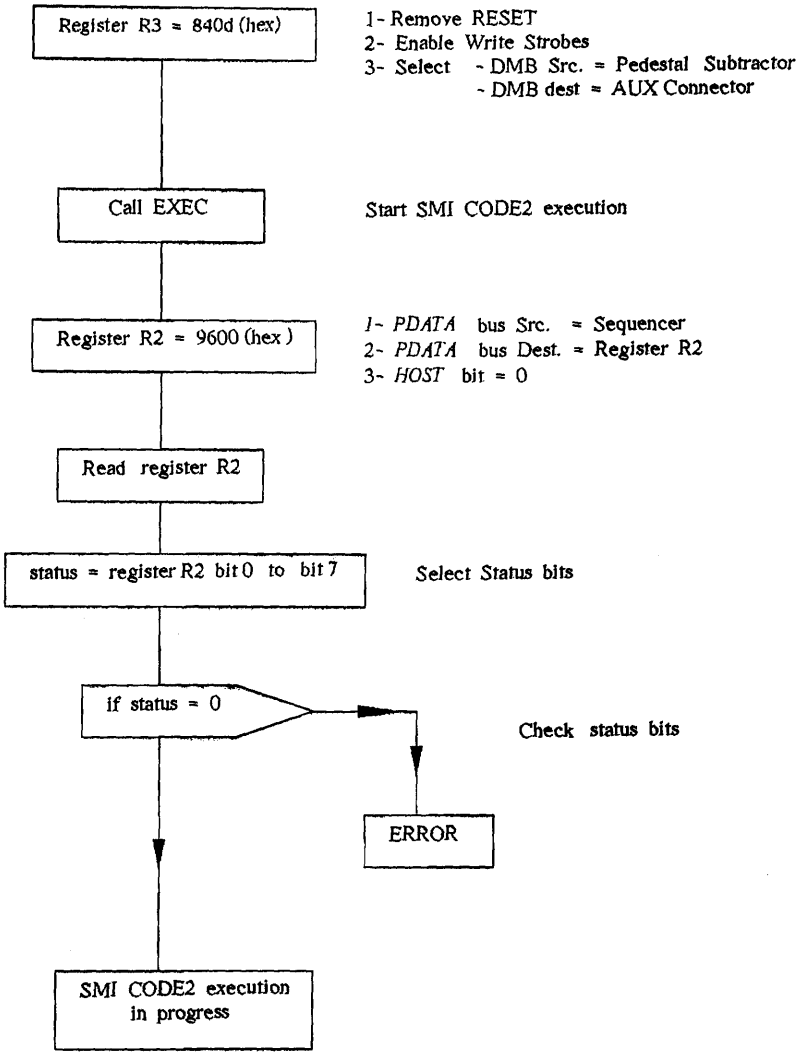


Figure 4.15: **start-smi** Function

Chapter 5

CONCLUSION

Some measurements of the performance achieved with the old and new FASTBUS to VME-bus interface configurations are given in this section, to allow the evaluation of the new hardware/software system. A test experiment involving the readout of two FASTBUS crates was set up. In the first crate two modules were read, a Phillips 10c2 (32 channels) and a STR136 (4 channels). However, due to data compression hardware in the 10c2, a total of 6 channels were generally read. In the second crate three STR200 modules with 32 scalers each, a total of 96 channels, were read.

The VME software was modified so that at critical times during the event processing, an output register, connected to a NIM logic output on the CBD8210 CAMAC branch driver, was toggled on and off so that time might be measured on an oscilloscope. This toggling operation took about 1μs and had a small effect on the overall dead times. Signals associated with the SMI readout were accessible without special software modifications.

5.1 Data Acquisition Dead Time using CODE1

The software CODE1 (section 4.3.2) makes use of the SMI to CAMAC to VME-bus link for data transfer. As shown in figure 5.1, three different times, T_i, T_a and T_r have been measured in the test experiment described above. T_r constitutes the total dead time engendered by one event and, T_i represents the delay in the OS9 system responding to an external interrupt. At T_0 the IRQ signal is input,

and after a time T_i, the execution of the IRQ service routine starts. It sends a wake-up signal to the "acqu" readout process, which is activated after a time T_a. Thus the data readout is started 132μs after T_0. The data readout from "acqu" is finished after 6 ms, which represents the total dead time T_r engendered by one event.

In this case, after the data are digitised, they are stored in the SMI data memory. The task "acqu" performs both the data transfer to a VME buffer via CAMAC and the data formatting, which takes around 5.9 ms, very much greater than the time for CODE2 described in section 5.2. The cause of this difference is discussed in section 5.3.

5.2 Data Acquisition Dead Time using CODE2

The software CODE2 (section 4.3.4) does not involve CAMAC. The data are transferred from FASTBUS directly into the high speed VME memory HSM8170. The data are then decoded from the HSM memory into a CPU RAM buffer.

Unlike CODE1, with CODE2 two sets of timing were measured : VME CPU timing and FASTBUS SMI timing. Initially "acqu" is dormant, waiting for the IRQ event to activate it, and the SMI is in a polling loop, waiting for the INP1 signal (section 4.3.4) to start the FASTBUS readout.

At T_0, both the IRQ signal and the ADC trigger signal are input. During the

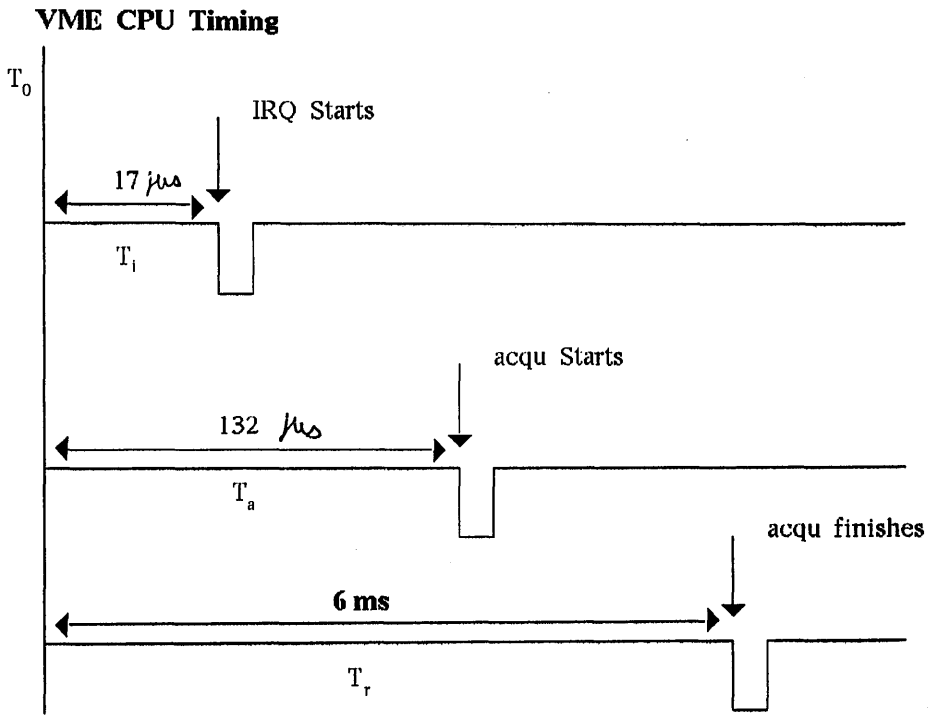


Figure 5.1: **Dead Time Using CODE1**

time which the VME software takes to activate "acqu", the FASTBUS readout operation is in progress. Three different times T_1, T_2 and T_3 have been measured from FASTBUS. The data digitising is finished after T_1. At this time the INP1 signal is sent to the SMI. This makes CODE2 exit from the polling loop so that data readout from all modules configured in the crate is performed. The data are written into the HSM8170 memory. For each word-write cycle into the HSM8170 memory a WSI signal is sent which enables the data write operation. Around $2\mu s$ separates two consecutive WSI signals, and around $2\mu s$ separates the readout of two consecutive modules. The first crate is read after time T_2. CODE2 then sends the $RDOC_1$ signal, which is fed into the next SMI. This starts the readout of the second crate. CODE2 loaded in the second SMI performs the same operations. When it finishes it sends $RDOC_2$. In the present test only two crates were used, so that the $RDOC_2$ signal was not used.

T_i, T_a and T_r have also been measured in the same way as for CODE1. As shown in figure 5.2, the data transfer from FASTBUS to the VME fast memory HSM8170 is achieved before "acqu" starts, and takes around $60\mu s$. Thus when "acqu" starts, the data are already stored in the HSM8170 memory. The task "acqu" performs only the data formatting which takes 0.6 ms. Thus the total dead time T_r for an event is around 0.7 ms.

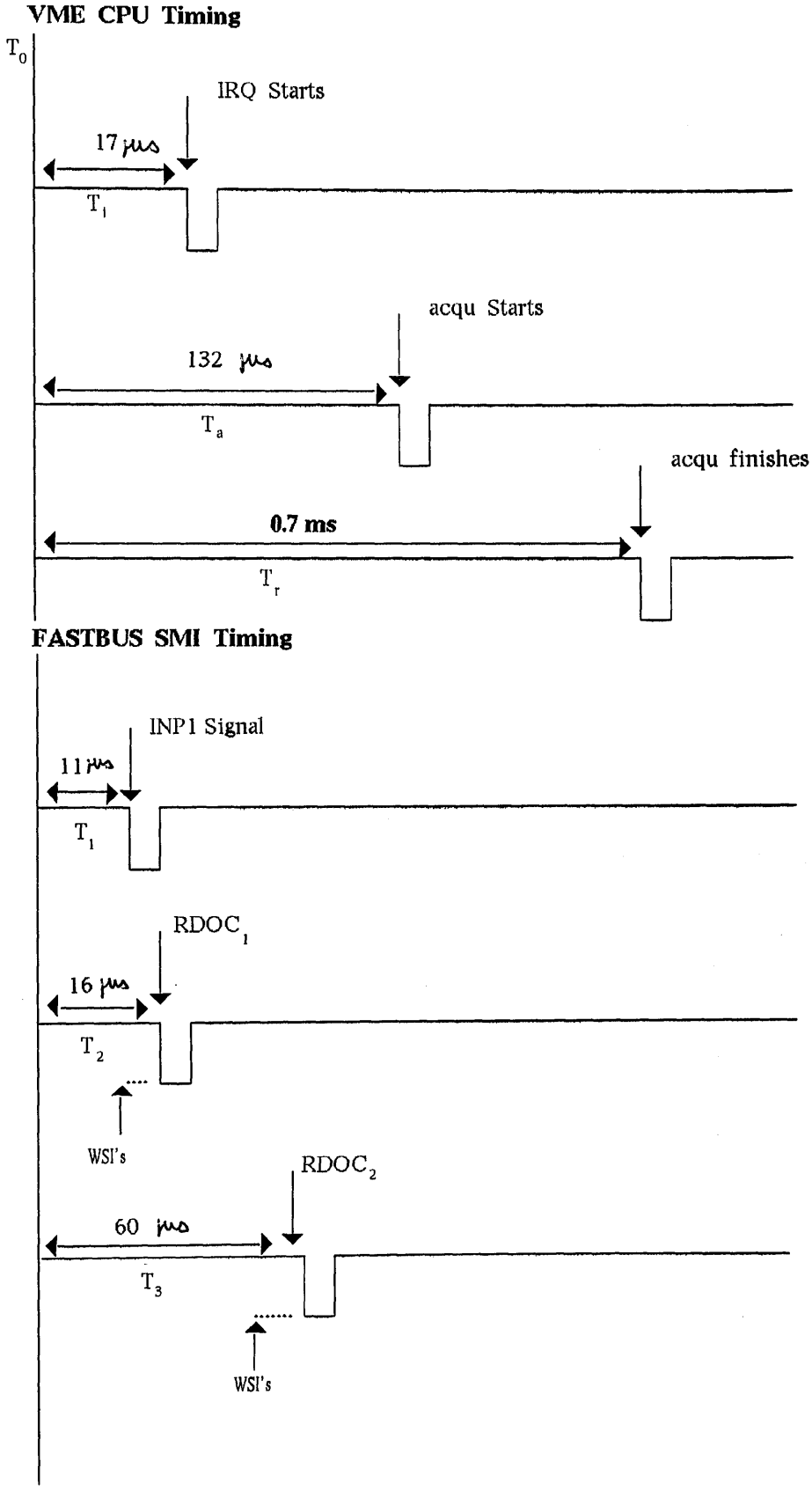


Figure 5.2: **Dead Time Using CODE2**

5.3 Interpretation

The improvement described above has been successful in reducing the dead time for a test readout from 6 ms to 0.7 ms, which will allow the acquisition rate to be greater by approximately a factor of 8.

The use of CODE2 in the SMI reduced the workload for the task "acqu" running on the VME computer. Whereas with SMI CODE1, "acqu" performs both data transfer and data formatting, with CODE2 it performs only data formatting.

As the data formatting function is performed in the same way for both readout methods, and from figure 5.2 took approximatively 0.6 ms, the bulk of the $\sim$ 6 ms dead time associated with the CODE1 method comes from the data transfer. This is due to the many read/write CAMAC operations performed to execute a single transfer between FASTBUS and VME. For each block-read, at least 48 read/write CAMAC operations are required to start SMI code CODE1 transferring the data into the SMI data memory, and 3 CAMAC read operations are performed for each 16-bit word transferred from the SMI memory to VME. Each CAMAC operation takes several microseconds, so that for transfer of large amounts of data the CAMAC interface becomes unacceptably slow.

5.4 Future Improvement

At present, the readout speed achieved with the new SMI to VME-bus data transfer software is more than adequate to meet the requirements of current experiments. However, if experiments become larger, requiring even more channels to be read, further modifications may be required in the data acquisition system in order to keep the dead time as small as possible.

As shown in figure 5.2, the SMI block-readout operation is in progress while the system sends the event to start "acqu". Thus, if the number of channels to read become larger, "acqu" might start before the end of the block read routine ($T_3 > T_a$). This could be resolved by delaying the IRQ signal but in order to keep the dead time as small as possible, the SMI block-read routine could be modified by using a LeCroy hardware block-read, which would effectively halve the FASTBUS transfer time. Investigations of this block-read mode with non-LeCroy modules will constitute the next phase of this project.

Appendix A

SMI Code Download Function : LOAD()

The C code of the LOAD function is given in figure A.1, and is shown schematically in the flow chart of figure A.2. The parameter passed to the function, when called, is the SMI RAM number (0 to 7). The LOAD function is initiated during the data acquisition initialisation. Register R0 is loaded with 9980(hex) which selects output register R1 as the source of the PADDR bus, with the sequencer as the destination, and output register R2 as the PDATA bus source with the sequencer as the destination. With this configuration, the host can address any byte in the sequencer control store memory and write any data byte value into it.

Once this configuration is set, the output register R1 is loaded with the byte

0	Complete without error
-1	Disk file open error
-2	Format error on input line
-3	Bad line sequence
-4	SMI write failure(s) detected

Table A.1: **LOAD Function Flags**

address of the sequencer control store memory and output register R2 is loaded with the instruction word, in the LSB. The instruction is then strobed into memory by setting bit 8 of output register R7.

Five different flags, as shown in table A.1, can be returned by LOAD. These are set at the end of the load operation.

```
/*********************************************************************
        Routine to download microcode from Host to Smi Sequencer
        Code assumed to reside on disk file /dd/ACQU/FB/DATA/smi_X.d
        where X=0,1,2,3,4,5,6,7 is the smi ROM/RAM to load
        This is passed as a parameter.
        Returns.....0  Completed without error
        -1  Disk file open error
        -2  Format error on input line
        -3  Bad line sequence (should increase monotonically)
        -4  SMI write failure(s) detected
*********************************************************************/

int LOAD(RAM)
unsigned short RAM;             /* ROM/RAM to load */
{
        char line[133];         /* char array to hold input line */
        char comm[20];          /* char array for label "comm:" or "line:" */
        char s_ram[2];          /* to convert RAM short to char */
        char SMI_file[30];      /* string = name of disk file */
        unsigned short l_num;   /* line number read from input line */
        short l_curr;           /* current line number */
        FILE *SMIcode_fp;       /* ->position in disk file */
        unsigned short W;       /* SMI write error count */
        unsigned short i;       /* code index */

        unsigned short code[8]; /* 8 code bytes per line        */
        unsigned short word[8]; /* 8 code bytes per line        */

/*      Get the right file      */
        sprintf(s_ram,"%1d",RAM);       /* convert short to character */

        strcpy(SMI_file,"./DATA/smi_X.d");
        SMI_file[11] = s_ram[0];

/*      Attempt to open the disk file   */
        if((SMIcode_fp = fopen(SMI_file,"r")) == NULL){
                fprintf(stderr," Failed to open %s for SMI microcode\n",
                SMI_file);
                return(-1);
        }

/*      Read file line by line and download to SMI until EOF or err
        Assume 132 chars max in line    */

        code[6] = code[7] = 0;
        W = (*W_SMI)(0,0x9080);     /* set up download direct to sequencer */
        l_curr = -1;                /* init line number */
        while(fgets(line,132,SMIcode_fp) != NULL){
                sscanf(line,"%s",comm);
                if(strcmp(comm,"line:")         /
                        if(sscanf(line,"%s%hx,%x%x,%hx%hx%hx%hx%hx%hx%hx",
                        comm,&l_num,&word[0],&word[1],&word[2],&word[3],
                        &word[4],&word[5],&word[6],&word[7]) != 10){
                                fprintf(stderr," Format error after line %x\n",
                                l_curr);
                                return(-2);
                        }
                        else if(l_curr >= l_num){
                                fprintf(stderr," Bad line sequence %x %x\n",
                                l_curr,l_num);
                                return(-3);
                        }
                        else{                   * looks OK download line */


                           code[0] = (word[0]<<4) + (word[1]>>4);

                          word[1] = word[1] & 0x000f;
                          code[1] = word[2] + (word[1]<<4);
                          code[2] = word[3];
                          code[3] = word[4];
                          code[4] = word[6] + (word[5]<<4);
                          code[5] = word[7];


                               W += W_line(l_num,code,0x100);
                               l_curr = l_num; /* update current line */
                        }
                }
        }
        fprintf(stderr," End of direct Sequencer %d download at line %x\n",
        RAM, l_curr);
        if(W != 0){
                fprintf(stderr," %d write errors during load\n",W);
                return(-4);
        }
        return(0);
}
```

Figure A.1: **LOAD Function C Code**

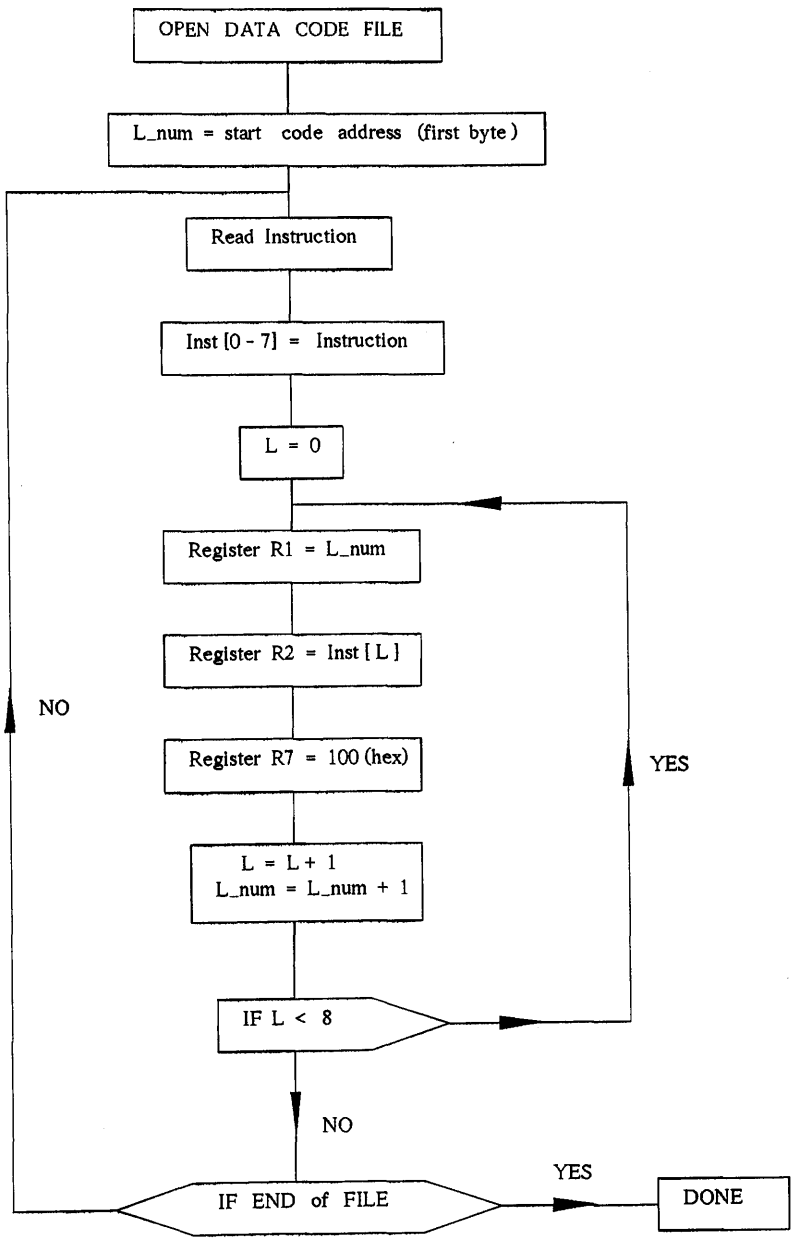


Figure A.2: **LOAD Function**

Appendix B

Function to Trigger SMI Execution : EXEC()

The C code for the EXEC function is given in figure B.1, and shown schematically in the flow chart of figure B.2. The parameter passed to the EXEC function, when called, is the start code address "addr".

Output register R0 is configured to select output register R1 as the source of the PADDR bus, with the sequencer as the destination, and output register R2 as the PDATA bus source with the sequencer as the destination. Once this configuration is done, the host clears bit 7 of output register R0, to put the sequencer in execution mode, and writes the program start address to output register R2. The execution is started when bit 0 of output register R7

```
EXEC (addr)
begin

X = (*R-SMI)(0 , &st);              /* set up SMI registers */
st = st & 0x001f | 0x9900;

X += (*W-SMI)(0 , st | 0x0080);
X += (*W-SMI)(1 , 0);
X += (*W-SMI)(0 , st);
X += (*W-SMI)(1 , addr*8);          /* load start address */
X += (*W-SMI)(7 , 0x0001);          /* ignite sequencer */

i = 0;

bcl:                                /* loop to check end of execution */
X += (*R-SMI)(7 , &st);
i++;
N = ( st & 0x0010);
if (N # 0) & (i<=1000)
Goto bcl;

if (i > 1000);
pr " exec. incompleted "
return(1);
end;

return(X);
end;
```

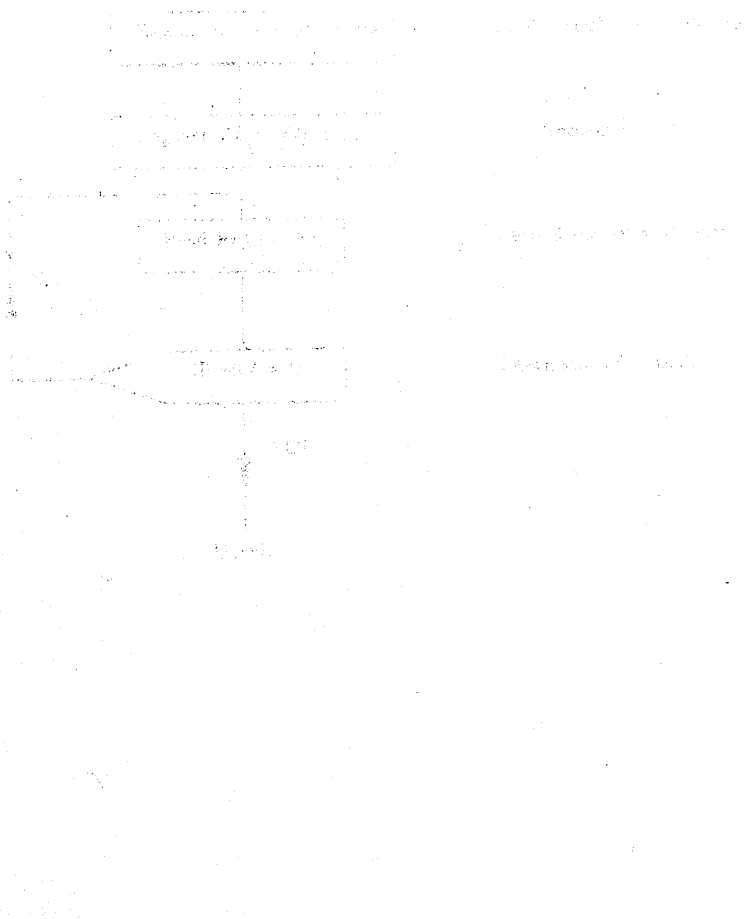
Figure B.1: **EXEC Function C Code**

0	Complete without error
1	SMI macro not completed
n	n = No of errors on (*W-SMI), (*R-SMI)

Table B.1: **EXEC Function Flags**

is set. This generates the sequencer control memory "GO" strobe.

Three different flags, shown in table B.1, can be returned by the EXEC function.



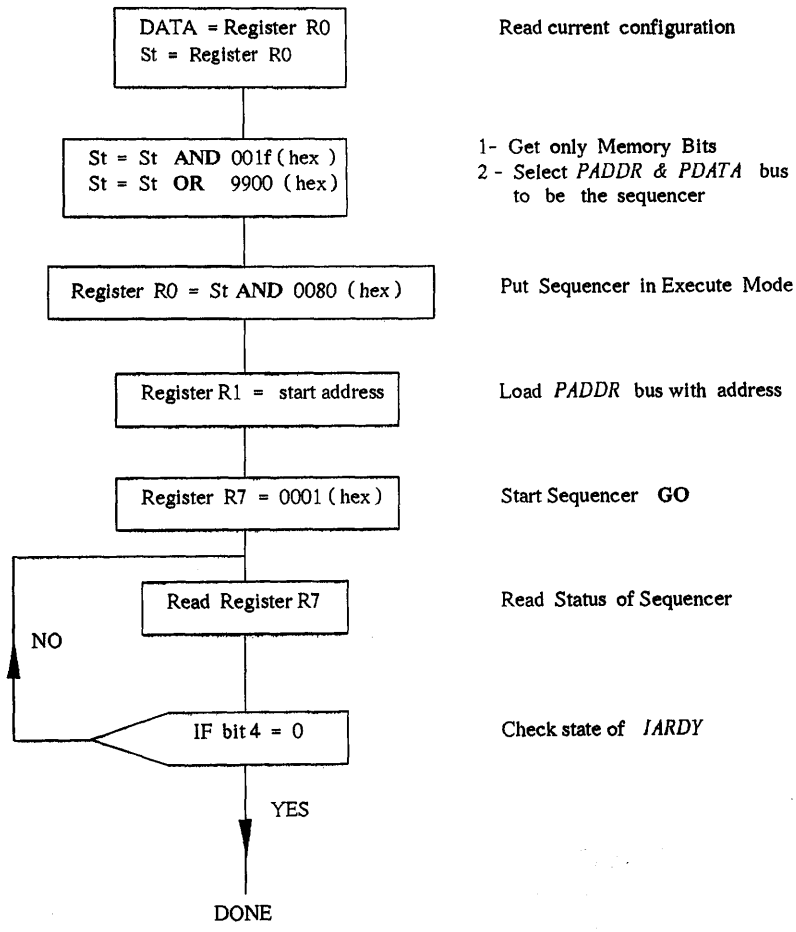


Figure B.2: **EXEC Function**

Appendix C

FASTBUS parameter file

```
*       KELVIN LABORATORY, UNIVERSITY OF GLASGOW
*       Master fastbus parameter file
*
*       1st line for SMI setup
*       SMI     RAM     No.Slots        Panel
        1       1       3               rear
*       Following 12 lines for modules in crate 1
*       Module          Slot    Read    Test    Config. File
*
        PHIL_10c6       25      b       n       ./data/phil_10c.d
        PHIL_10c2       24      b       n       ./data/phil_10c.d
        STRUCK_136      10      b       n       NULL
*
*       Next Crate
*       SMI     RAM     No.Slots        Panel
        2       1       3               rear
*       Following  1 lines for modules in crate 10
*       Module          Slot    Read    Test    Config.file
        STRUCK_200      12      b       n       ./data/str_200_clr.d
        STRUCK_200      13      b       n       ./data/str_200_clr.d
        STRUCK_200      14      b       n       ./data/str_200_clr.d
```

References

[1] Richard Fernow, "Introduction to Experimental Particle Physics", Cambridge University Press, 1986.

[2] W.R.Leo, "Techniques for Nuclear and Particle Physics Experiments", Springer-Verlag.

[3] Eltec Electronik Mainz, EUROCOM 6, Hardware Manual, 68030 CPU board.

[4] Elect-68K-System, Hardware Manual, EUROCOM 5.

[5] "VMV bus one slot VIC8250 ", CES User's Manual, Ver. 2.0, July 1990.

[6] "CAMAC Branch Driver CBD 8210", CES User's Manual".

[7] "Interconnects for the FASTBUS SMI Model 1821", LeCroy AN-28A, Feb. 1985.

[8] "Model 2891A CAMAC FASTBUS Interface", LeCroy Operator's Manual, Revised March 1989.

[9] High Speed Memory with ECLine Interface HSM8170, CES User's Manual, July 1988.

[10] "Manual 1821/ECL", LeCroy Operator's Manual, July 1985.

[11] "Passing Data to VME via ECLine", LeCroy AN-46.

[12] "What is CAMAC", CERN-NP CAMAC Note 45-00, Feb 73.

[13] B.Zacharov, "CAMAC Systems : A pedestrian's guide", Daresbury Nucl. Phy. Lab, 1972.

[14] EUR 4100, Esone Committee, Italy, 1975.

[15] R.S.Larsen, IEEE, NS-29 (74-78), No 1, Feb 82.

[16] H.Verweij, IEEE, NS-31 (211-213), No 1, Feb 84.

[17] "An Introduction to FASTBUS", LeCroy AN-26.

[18] D.Burckhart, "An Introduction to FASTBUS", CERN, Data Handling Division, DD/84/8, July 1984.

[19] "FASTBUS Software Workshop", Data Handling Division, CERN 85-15, 4 Nov. 85, .

[20] "Model 10c2 FASTBUS QDC", Phillips Specification Manual.

[21] "Model 10c6 FASTBUS TDC", Phillips Specification Manual.

[22] "STR136/DIFF FASTBUS ECL I/O Latch", STRUCK Technical Manual.

[23] "STR200 FASTBUS Scalers", STRUCK Technical Manual.

[24] L. Costrell, IEEE NS-30, No. 4, Aug 83.

[25] L.Paffratn et al, IEEE, NS-29 (90-93), No 1, Feb 84.

[26] "OS9/68000 Source Level Debugger User Manual", Microware Systems Corporation, 1987.

[27] "OS9 Operating System Manuals", Ver. 2.2, Microware Systems Corporation.

[28] Peter Dibble, "An advanced programmers guide to OS-9/68000", Walden Miller, 1988.

[29] "MC68020 User's Manual", Motorola Inc, 1984-1985

[30] B.W. Kernighan and D.M. Ritchie, "C programming language", Prentice-Hall,INC, London, 1978.

[31] "OS9 Language Manuals", Ver. 2.2, Microware Systems Corporation.

[32] "1821's User's Manual", LeCroy, Revised March 1987.

[33] "Using the Model 1821 Segment Manager/Interface", AN-28C.

[34] W.Farr et al, IEEE, NS-31(217-224), No 1, Feb 84.

[35] "Interactive FASTBUS Software Toolkit (LIFT)", Lecroy Operator's Manual, Ver. 2.60-2, April 1988