INTERFACING THE IBM PC WITH THE STD BUS FOR MULTIPROCESSING

by

Diptish Datta

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

APPROVED:

Dr. Charles E. Nunnally, Chairman

Dr. J. G. Tront

Dr. John McKeeman

August, 1985

Blacksburg, Virginia

INTERFACING THE IBM PC WITH THE STD BUS FOR MULTIPROCESSING

by

Diptish Datta

Dr. Charles E. Nunnally, Chairman

Electrical Engineering

(ABSTRACT)

The advent of the Personal Computer into the technical world has, at an extremely reasonable expense and trouble, made available to us, considerable computational power. But, as it was with computers, the next logical step is to have multiple units running in concert, or, in other words, sharing the load. This leads to the concept of Multiprocessing in order to attain an enhancement in operation speed and superior efficiency. The IBM PC is a versatile and market proven personal computer with a very large volume of software support and the STD Bus is a standard that has been developed to cope with a variable support, i.e. different processors and different I/O capabilities. Together, they combine the user interface - the display and keyboard - of the PC, the processing capabilities of the PC, the I/O capabilities of the STD Bus and the support processing possible on the STD Bus. The resulting system is powerful, easy to use and it has a lot of scope for development.

ACKNOWLEDGEMENTS

I would like to express my gratitude for all the people who helped me complete this thesis.

First of all I would like to thank Dr. C. E. Nunnally for his patience and time. Without his suggestions and support this project would not have materialised at all. All through the persual of my degree in Va. Tech, his help and encouragement was invaluable.

I would like to thank Dr. J. G. Tront because he is responsible to a large degree for my knowledge of hardware design; the VLSI design course offered by him was very enlightening. I have also to thank Dr. J. McKeeman for consenting to be in my committee.

The EE Shop staff in the Electrical Engineering department were always helpful and friendly. and were particularly willing to offer any assistance I needed; I would like to express my appreciation for their help.

My friends, in Blacksburg, have been a constant source of moral support and I would like to thank them. I would specially like to mention and for all the good times we had.

Last, but definitely not the least, I want to express my sincere gratitude for my sister , my mother and my father ; to them I owe my education, whatever I have learnt in my life and the fact that I am here to do my thesis!

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

1.0 INTRODUCTION

Microprocessors are the latest gift to the world of computers. Accelerating technology and development of VLSI have packed an incredible amount of power in small spaces; the expenses also have taken a downward slope. But, man is always looking for cheaper and easier ways of doing his work faster! Like the age old concept of working together to get things done faster and more efficiently, there evolved the idea of making computers work together. Just like working people, some machine or system has some particular features or "talents" that the other system lacks; working in concert, the two systems can combine their resources to give birth to a system that is very much more powerful than either of the two component systems.

The PC being a single CPU machine, is essentially a sequential operation device. The architecture of a single PC is SISD.[1] So,it can execute one instruction or do one operation at a time, and can fetch or send one data unit, to/from memory or I/O device, at a time. The obvious solution to this bottleneck in processing speed is seen in multiprocessing architecture.

---

[1] SISD architecture is "Single Instruction stream Single Data stream", the most elementary architecture.

[2] Multiprocessing is usually MIMD architecture - Multiple Instruction stream Multiple Data stream. Variations like MISD or SIMD are also possible.

The STD bus by ProLog Corporation has been used to develop such a system. The STD bus provides a "slave processor" to the PC - "master" processor. The PC can designate any processing, it wants done, to the processor on the STD bus. The I/O facility of the PC is also enhanced by the STD bus I/O capability. Communication protocols for data and command transfer have been studied. The PC-STD link by RMAC has been used to establish the interface, and multiprocessing features, advantages and scopes have been studied.

This paper demonstrates the flexibility and versatility of the PC-STD system. The different means of exchanging data and information between the two systems, over the link, have been exercised to serve a particular application and to expound the possibility of multiprocessing. But beyond the application developed, this paper demonstrates the immense scope of the system -- as a real time process control unit, as an educational aid to to system development on the STD bus, or as an efficient multiprocessing system.

2.0 MULTIPROCESSING CONCEPTS

We will consider multiprocessing as a processing environment in which more than one processor is employed to attain a certain computational goal. This is also, in general, called Distributed Processing; the definition of distributed processing can be roughly worded as a system in which "the computing functions are dispersed among several physical computing elements; these elements may be co-located or physically separated" [1]. This general definition describes, approximately, the kind of system architecture we are interested in, and in particular, we are interested in locally distributed systems.

2.1 CHARECTERIZATION OF THE MULTIPROCESSOR SYSTEM

The American Standard Vocabulary for Information Processing defines multiprocessing as "a computer employing two or more processing elements under integrated control". The keywords here, are "a computer" and "integrated control"; that the system is described as one computer points to the fact that the system acts on one function - different processors are not executing independent processes; integrated control tells us that there has to be a central master controller or operating system that governs the system functioning.

The general architecture of a MPS (Multiple Processor System) is depicted in Figure 1 on page 4. The interprocessor communication could be a passive communication hardware base or it could be an intelligent processor based

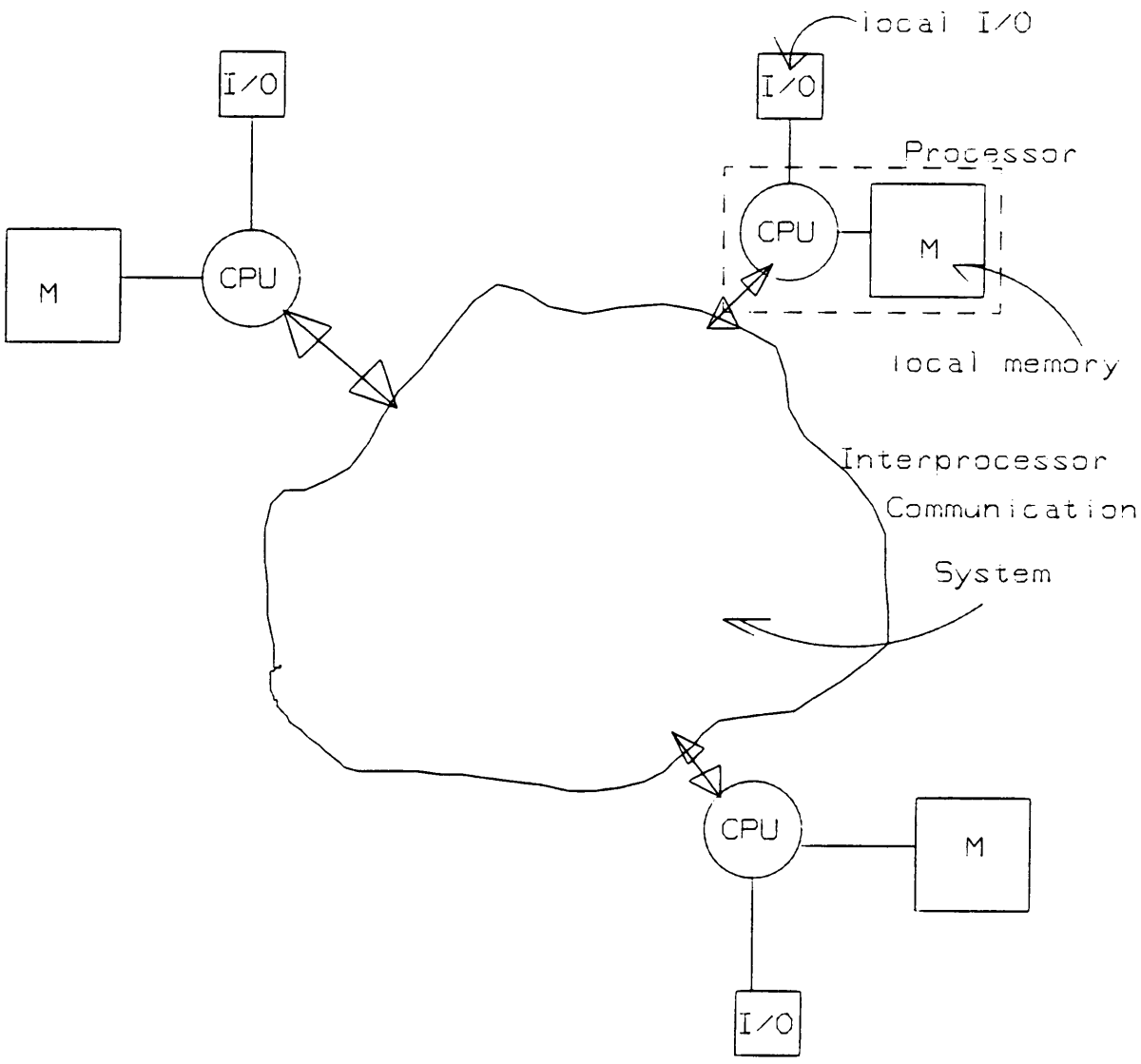

Figure 1. Multiple Processor System

communication control system. The "master" controller function should be running on a particular processor which is responsible for the process deputations and control.

One variation of the MPS that is very close in architecture to the one we are trying to develop, is a system in which there is a common memory block that is shared and accessed, for data or storage, by all the component processors. This architecture gives rise to the problems of arbitration of memory. Memory access contention must be avoided and also, time-efficient memory access has to be designed. Several schemes have been developed to get the best compromise between the low cost of having one memory unit and the problem of sharing this common resource. Since this is not the kind of architecture this project deals with, it will suffice to simply list some of the methods employed, without going into the details of the methods:

- Partitioning memory into pages.
- Using a round-robin or a prioritized bus access protocol.
- Using small local high speed cache memory modules.

So, we will concentrate on the multiprocessing architecture shown in Figure 1 on page 4; it has several processors, with their own local memory blocks, that can execute process segments that a master controller deputes to them. Comparing this system to the common memory model, we can immediately note the advantage of less memory contention. Also, there are well-defined communication channels over which inter-processor communications take place, so that there is little or no chance of a processor

receiving an error data from an errant co-processor; this could happen if the memory is common, in which case there is no way for the receiving processor to check the data - it just reads the data off the memory! System expansion is also very easy.

2.2 MPS ADVANTAGES

2.2.1 Reliability

One field in which the MPS shows remarkable advantages is in increased reliability [2]. A typical hardware failure in a mainframe computer can take from 1/2 hour to 4 hours to fix. "Availability", defined as the average uptime divided by the average uptime plus the average downtime for a computing system, is found to be about 98% for a mainframe computer. In order to reduce the downtime of a mainframe, a backup needs to be provided; but when we consider backups for mainframe computers, we have to have a duplicate or standby for practically all components - CPU and peripherals. This increases redundancy in the system considerably.

In a MPS it is not necessary to duplicate all equipment to attain the desired degree of availability of the system. It is shown [2] that if we consider a MPS of 8 processors, then with the maintaining of a backup database disk and the addition of two spare processors, we can get an extremely high degree of reliability and availability of the system. It is shown that with the addition of two processors to the 8-processor MPS, the expected mean time to system failure is in the order of 13-14 years! And when the system finally fails, the average time taken to fix it will

be in the order of 3-4 hours; and all this for an incremental cost of 20% or so.

2.2.2 Response Time

The enhancement in performance due to faster response time is a very big advantage of the MPS. This advantage arises from two main features:

1. The system is very flexible and can be re-moulded according to the application at hand. The user has the ability to alter task assignments and data flow paths depending on the computing needs, and attempt at getting rid of idle processors. Maximum speed of processing can thus be aimed for.

2. The MPS is dedicated to one job. In a mainframe network environment, the job the user wants done can be unexpectedly swamped by loads from other users and their tasks. The MPS is usually a dedicated system and is designed by the user to get the job done as fast as possible, without interruption by any other tasks.

2.2.3 Physical Security

This is an issue that might be of importance in certain situations like on a ship. The fact that separate processing units and their local memories with the stored data bases can be placed in different locations,

ensures that in case of some catastrophe like fire or sabotage, total loss will not be there in data or equipment.

2.2.4 Program Modularity

This is a side-effect of the basic structure of the system. Program modularity is always desirable because it makes, programming, debugging and testing a lot easier. This can be implemented in any system by the programmer and by the use of good systems management and data structures; the task is just made easier because it is enforced by the modular structure of the MPS.

2.3 DESIGN ISSUES

Two main issues arise when a multiple processor system is to be designed:

- Should the processors be similar - homogeneous or heterogeneous?
- How many processors should be employed?

The first question is ofcourse a debatable one for any system that is to be designed. This is because of the fact that there are advantages and disadvantages to both choices. Homogeneous processors have the advantage of having the same memory modules and components and hence, replacement and standby stock can be minimised. Also, the code that runs

on the different processors are obviously the same - this is a considerable advantage where program development and transfer is the issue.

The choice of heterogeneous components for a MPS might be triggered by the particular application that is being catered to. There are certain features or capabilities that are better in one processor than the other - in some cases a processer might have a certain feature that the other does not have at all. If all the processors do not need to have this capability, then it is definitely a waste to employ all processors with that feature. This is an obvious situation where the designer would choose heterogeneous computers to build the system. The big disadvantage in this choice is the program development. Program module transfer is not possible dynamically. The processors have to be assigned fixed tasks and only data transfer is feasible.

2.3.1 System Description

The Serial Microprocessor Array described by Corsini et.al [3] is an example of the heterogeneous component implementation. (This is the basic structure used in this project.)

The CU (Control Unit) is the main processor or the Master Controller and this is where the main operating system and control is run. The PE's are the Processing Elements; these are satellite processors with their own private memories (see Figure 2 on page 10). The PE's can have their own I/O functions, each one being **unique to the PE** if so desired. A communication link is necessary for inter-processor communication and for commands and data transfer to and from the master computer. This is usually

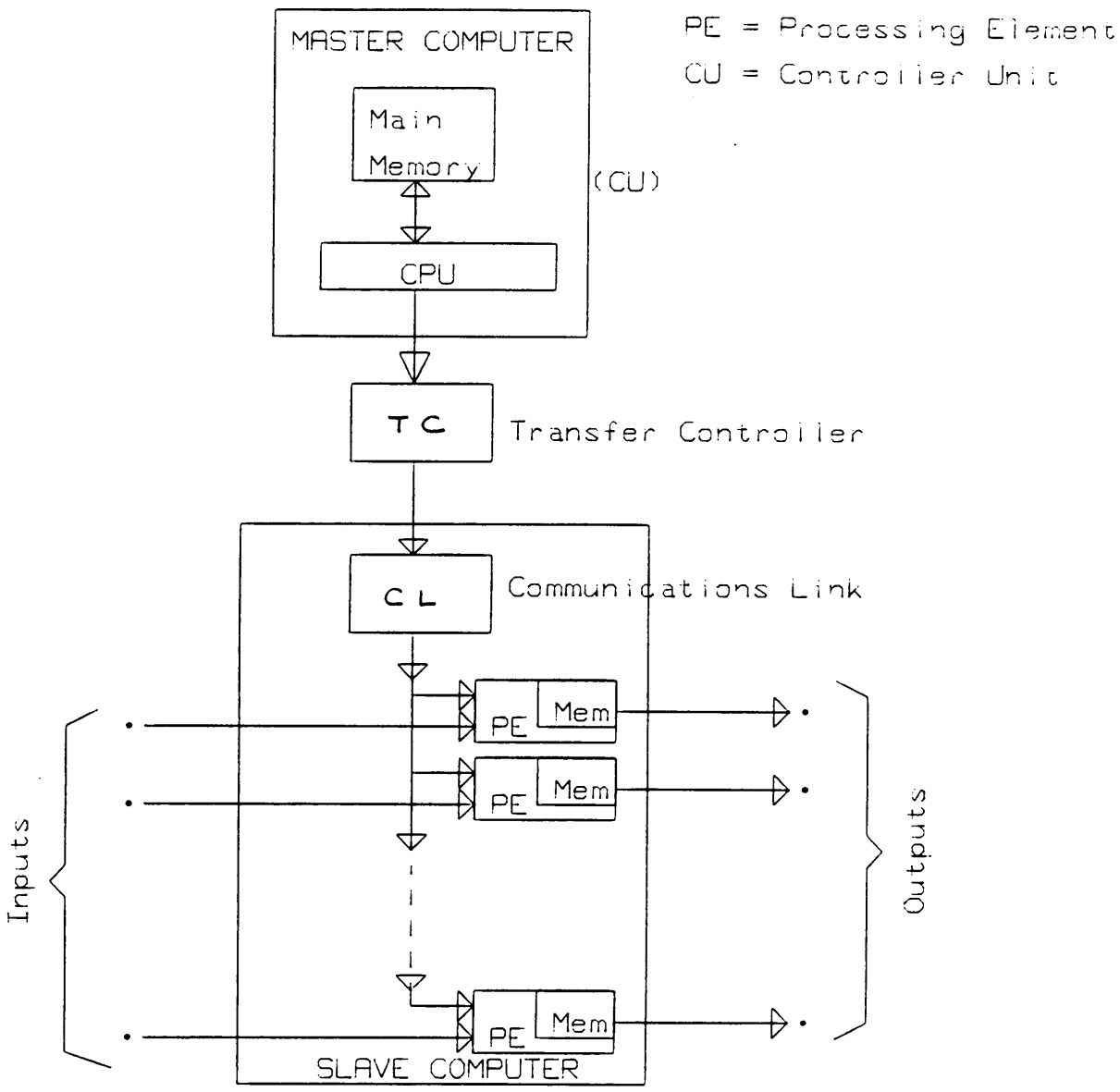

Figure 2. A Hetrogeneous MPS Block Diagram.

a passive hardware communication base. A smart transfer controller like a DMA controller or an Interrupt controller can be included in the data path to take care of the transfer of data without conflict.

3.0 DEVELOPMENT OF THE PC-STD SYSTEM

The aim of the project was to build an inexpensive, easily available system that can demonstrate the advantages of multiprocessing. The STD bus system offers a standardized base for cards that plug into the bus chassis. The card can be chosen from a host of standard cards available, ranging from Processor cards, Memory cards to I/O cards, or they can be custom made to incorporate any function desired - limited only by the physical dimension specifications of the bus. The IBM PC provides the Keyboard and the Display through which the operator can communicate with the system. So, together they form the basis of a reasonably useful system -- in fact, they form the basis of a large number of systems that can be developed from it.

So, what is necessary, to develop such a system, is an interface between the PC and the STD system. The PC-STD link by RMAC was used to establish the link. The overall set up is depicted in Figure 3 on page 13. The components of the system are described in the following sections.

3.1 THE STD BUS

The STD Bus standardizes the physical and electrical aspects of 8-bit modular microprocessor card systems. It supports a family of standard cards available. The different categories of cards available are -

- Peripheral Interface

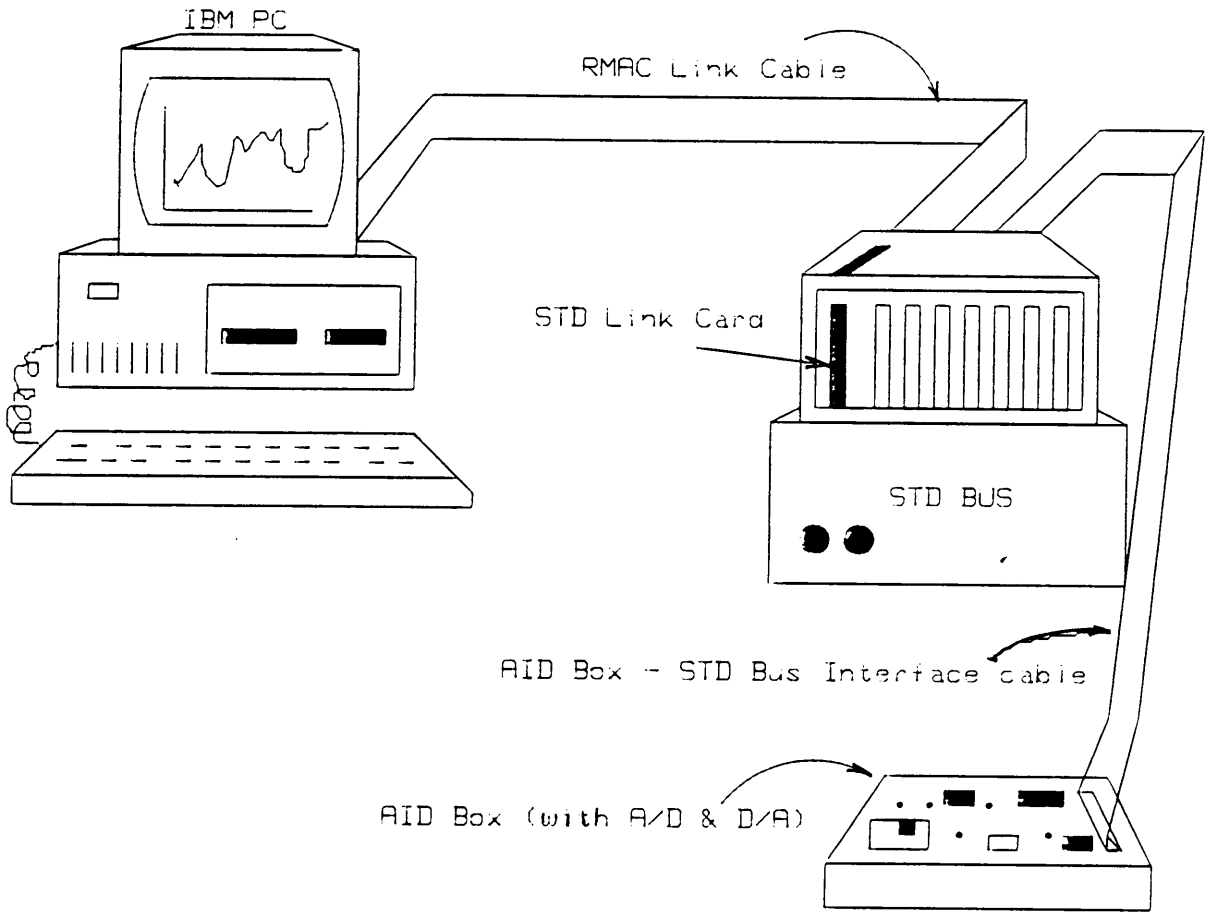

Figure 3. System Set-up: The experimental set up of the system developed is shown.

- Analog I/O
- Industrial I/O
- Digital I/O
- Memory
- Processors
- Utility (extenders, blanks, etc.)

The common 8-bit data bus and 16-bit address bus and all the control signals on the control bus allow for easy control of all the sub-systems. Address decoding logic is incorporated on the cards themselves. This system, capable of processing, storage and extended I/O, presents itself as a useful co-processor unit.

The STD bus, by itself, is not a development station for programs or applications. The lack of high level programming aids or translators makes it impossible to do complicated program development on the STD bus system. But, as a PE (Processing Element), the STD bus system is very well suited (see "System Description" on page 9 for details) because the processing capability and I/O can be used to an advantage by a Controller system - the PC in this case.

3.1.1 STD Components Used

In the experimental set up, used to develop a multiprocessing application, the following cards and subsystems were used on the STD bus -

- AID box for A/D input and D/A output.

- STD Cards -

1. Keypad-Display card (7303)
2. Single Step, Interrupt and Status display card.
3. 8085 Processor card (7801)
4. Static RAM card (7701)
5. AID box adopter card
6. RMAC STD link card

3.2 THE IBM PC

A 8088 microprocessor is the CPU of the IBM PC. It uses an external 8-bit bus and has internal 16-bit capability. To enhance operation speed the 8087 Mat Co-Processor can be installed to run with the 8088 in the max-mode [5]

---

[3] AID = Analog Interface Designer. This box is made by E&L Instruments Incorporated [4].

There are two features of the PC that are germane to the issue of interfacing it with the STD system. The first is the external 8-bit bus used by the PC. This makes interfacing to the 8-bit STD system straightforward and avoids latching and gating problems. Data computed on the STD bus system can be sent to the PC parallelly and the PC can accept and store or process the data directly. When the PC has to send data to the STD bus it has to take care of the 8-bit boundary, or else data bits will be lost. The problem with the PC-STD system that comes to mind here, is the limitation to 8-bit accuracy.

The second feature is the DMA controller of the PC. It is software programmable to accept data directly from external I/O ports without CPU intervention. This offers us the very useful option of high speed data transfer from the STD bus system without processor control.

The PC display can be used to display data in text or graphics modes while the keyboard allows the operator to enter commands. One interesting feature that can be used, is the display of data by poking data into the display memory directly. This enhances speed of display considerably. The PC offers a large volume of software support and scope for program development. High level programming can be done on the PC to realize complicated algorithms, and then it can depute whatever task it deems necessary, to the STD CPU. It can also have the STD CPU do any required I/O operation, process the data if required, and then have the STD system send the data over to the PC memory. This is the basic concept of multiprocessing and has been demonstrated in practice in this paper.

3.3 THE PC-STD LINK

The PC-STD Link developed by RMAC is easy to install, modular, and adoptable to the system [6]. It comprises of two cards -

- The STD link card that conforms to the STD bus specifications and plugs into the STD bus.

- The PC link card that plugs into any of the PC's system expansion slots.

A 40 wire ribbon cable connects the two cards, effectively forming the connection between the two systems.

Unlike the stipulation of a common bus, shared by the PEs and the CUs by some bus arbitration protocol, as explained in "Multiprocessing Concepts" on page 3, this system has two independent bus systems connected by the "link". Communication or data transfer between the two busses is usually via the Mailbox port as depicted in Figure 4 on page 18. The PC as Master controller has the option of taking control of the STD bus and using it for forced I/O.

3.3.1 Link Features

Features of the PC-STD link relevant to multiprocessing requirements are described below.

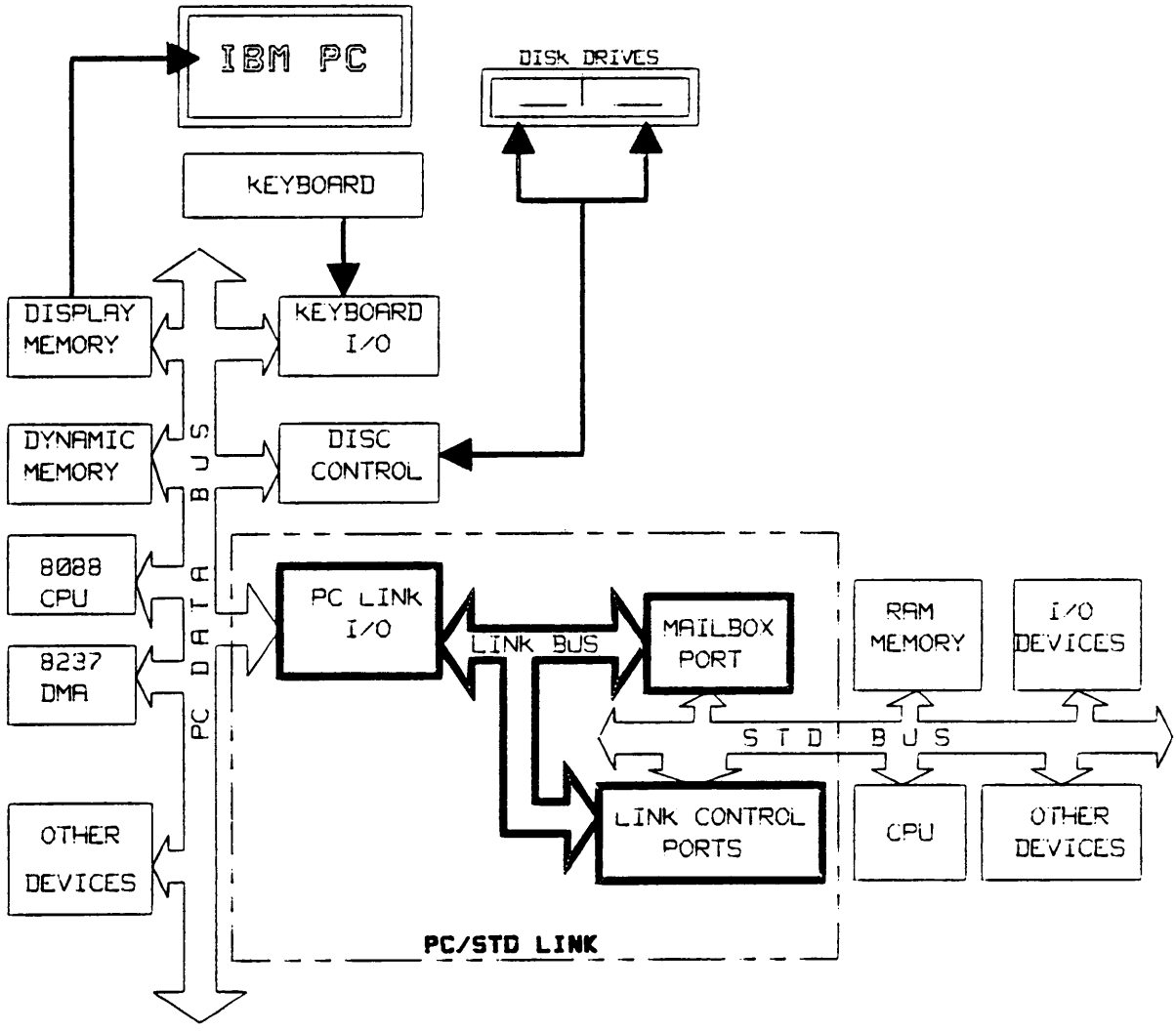

Figure 4. Block Diagram of the PC-STD System.

1. **Link Mailbox Port** - A bidirectional latched port is provided for asynchronous transfer of data between the PC and the STD Bus. There are flag bits, in status ports, that both the PC and the STD Bus system can read; through these flags either processor or system can check the full/empty status of the mailbox. This arrangement facilitates software polled, interrupt driven and DMA operations.

   With the present set up, the mailbox port address for the STD Bus is 0H and for the PC it is 300H. The Control/status port for the STD Bus is I/O port 01H and for the PC it is I/O port 305H.

2. **The Data Port** - This is a non-latched bidirectional buffer between the PC and the STD data busses. Writing or reading of this port by the PC allows for direct access of the STD Bus devices - I/O units or STD Bus memory. The port address, as set up, is 301H.

3. **Address Ports** - The PC can float any desired 16-bit address to the STD address bus through two 8-bit ports. These address ports are output ports and are latched; they are used to address STD bus devices directly during BUSLINK operations. As input ports they allow the PC to monitor the STD address bus. The address for these ports on the PC are 302H and 303H.

4. **Control Port** - This port allows the PC to latch STD Bus control signals from the Bus, and effectively, control the STD Bus operation or device access. AS an example, for STD memory read/write operation, the PC has to establish a BUSLINK or take control of the STD Bus,

issue the appropriate control signals enabling the memory bank, and then commence the necessary action on the memory. As an input port this port allows the PC to monitor the STD Bus status. The address, as set up, is 304H.

5. **Link Service Port** - This port is an important feature in parallel processing because it provides the "gates" on the link for the interrupts and DMA requests from the STD Bus. For unattended transfer of data, from the STD Bus to the PC, the PC has to enable the transfer through this port; otherwise the transfer request from the STD Bus is blocked at the link itself and does not reach the PC. As an input port this port allows the PC to check on the pending interrupts, mailbox status, read/write and reset signals. The port address is 305H.

3.3.2 Capability to Reconfigure Link

The ability to reconfigure the hardware of the link is a feature that needs to be mentioned; refer to Link Manual [6] for details. Jumpers on the PC Link and the STD Link cards allow the user to select PC port addresses, PC interrupts to be used, PC DMA channel to be used, Link reset mode, STD Bus addresses, IOEXP, MEMEX, BUSAK control, Interrupt destinations. This gives the user a wide range of system configurations to choose from, in order to adopt to the particular application he is designing and to adjust to the address spaces available.

The jumpers in the set up used are as follows -

- On PC Link Card

  - **W1** = PC Interrupt IRQ2

  - **W6** = DREQ Channel 1

  - **W7** = DACK Channel 1

  - **W10** = PC Power On resets STD Link

- On STD Link Card

  - **W2** = PBRESET resets STD Link

  - **W3** = PC can control MEMEX

  - **W4** = PC can control IOEXP

  - **W6** = PC can control BUSAK

  - **W7** = INTRQ goes to STD Bus

  - **W10** = Interrupts cause 8085 RST6

3.3.3 Operation Modes

Two distinct modes of operation are possible using the link. Very flexible and versatile applications can be developed by using features of both modes as and when required. The two basic modes are described, in short, below.

3.3.3.1 PC as total Bus Master

In this mode, the PC takes total control of the STD Bus and can use all of the STD Bus devices for I/O. The STD Bus is a passive slave unit used to enhance the I/O capability of the PC.

The operation involves the following steps -

1. Establish BUSLINK with STD Bus. If there is a processor on the STD system then it can grant BUSAK, or else the PC can force a BUSLINK by asserting BUSAK itself.

2. Select the STD device for read/write and float the address on the STD address bus.

3. Read or write data through the data ports in order to accomplish the necessary data exchange.

As an example of this mode of operation, refer to "Appendix B. Controlling the STD Bus from the PC" on page 72 where the Keypad and display

card on the STD Bus is monitored by the PC. The program polls the keypad on the card and echoes the key pressed to the PC screen as well as the card display. There is no sharing of the STD Bus at any time; the PC has total control of the bus to do the I/O.

3.3.3.2 PC-STD Multiprocessing

Use of the STD bus is shared between the STD bus CPU and the PC with some bus transfer protocol to govern the system. Both systems can be working on isolated program modules with only data transfer between the two. The PC can send data or program control information to the STD Bus using BUSLINK, Interrupt, etc. as in the application given in "Appendix D. Main Application - Filter Monitor" on page 95 It can also load data sets into the STD memory using BUSLINK. The PC can receive data from the STD I/O devices directly using BUSLINK or the STD CPU can send data over by Interrupt initiation, by DMA or through PC polled mailbox. One point to be noted, is that all data transfers, except "Forced I/O" using BUSLINK, takes place through the mailbox port. This mode is demonstrated in details in the main application developed. See "Appendix D. Main Application - Filter Monitor" on page 95.

3.4 SCOPE OF THE SYSTEM

Summarizing the features of the two systems - the PC and the STD Bus, we note the following points -

- The PC has user interface features - the keyboard for input and the screen for output or data display; the STD Bus system does not.

- The PC has high level software development capability; the STD Bus system does not.

- The PC has permanent storage capability in its floppy disks and thus, can store data or programs easily; the STD system does not have any such capability.

- The STD Bus system has a wide range of I/O capability and can do real-time data acquisition, under PC control or under STD system processor control; the PC has a very limited capability for I/O.

From these points the PC is clearly seen to be the "Master Controller" and the STD system to be the "Slave system" to the PC. If, for instance, if the operator needs to send some data from his input interface - the PC keyboard - to the STD system, the PC can take the data, and using software running on the PC, load the data on the STD Bus; the STD bus system has no means of "pulling" the data from the PC. Similarly, if the STD Bus has some data that it needs to load into the PC or to display on the PC screen, the STD system cannot take any action except "request" the PC system for service. The PC software can then arrange for the data to be received from the STD Bus and display it, as necessary, to the operator.

Note: The PC-STD Link provides for the PC to take control of the STD Bus - the PC can put the STD CPU on hold and use the Bus and STD devices directly; but there is no way in which the STD system can take over the PC bus.

So, when Multiprocessing has to be implemented on the PC-STD system, the STD system is strictly a "PE" and the PC is the "CU" (see "System Description" on page 9). I/O on the STD Bus can be done either by the PC directly - the PC can take control of the bus by establishing BUSLINK and read or write data directly, or else the PC can depute the STD Bus processor to do the I/O. The later option is very useful when the PC wants a large volume of data; in such a case the PC can be doing something else while the STD bus CPU collects the data set and then the STD system can send the whole set over by some fast transfer protocol like DMA. The PC can also have the STD CPU do some pre-processing of the data if necessary. All this is demonstrated in the main application developed; see "Appendix D. Main Application - Filter Monitor" on page 95.

4.0 SYSTEM APPLICATIONS

Multiprocessing is just one of the applications that one can develop on the PC-STD system. The simplest application one can visualize is the STD Bus as a passive I/O extension facility for the PC. There are several I/O cards available for the STD Bus, for instance the A/D cards to do fast analog to digital conversion, the industrial I/O cards to monitor temperature, etc. This opens a whole range of real time application possibilities for the system. The PC can do dedicated data processing on the data collected from the STD Bus I/O devices. This kind of application is basically an extension of the PC bus to include I/O devices; the PC can read the STD bus for the required data using the addressing capabilities and issuing the right control signals; and then it can process the data as required, display the data to the operator on the screen, print it out to a printer or use the STD bus to output the processed data to some I/O device.

A more complicated and more sophisticated application would be one in which the PC uses the help of a processor on the STD bus to "help it" do the job. There can be different levels or depths to which the PC might want to involve the STD bus; that has to be decided by the system designer to best suit his application. As discussed before, the system is extremely flexible as regards intermixing the basic modes of operation.

4.1 OUTLINES OF POSSIBLE APPLICATIONS

To summarize the scope of the system we can visualize the following kinds of applications; they are explained in Figure 5 on page 28 also.

- The STD could be used as an I/O extension facility where the STD bus I/O cards could be used to read or write data to remote devices while the data processing/display takes place on the PC. The PC acts like a dedicated monitor/controller for some real time process.

- The PC could depute the I/O operation to the STD bus processor and execute some other process simultaneously. This would be useful for an application where some iterative algorithm had to be run on sets of data. In such a case, the PC could operate on the present set of data **while** the STD CPU collects the next set of data and sends it over to the PC by some transfer protocol (e.g. DMA). This introduces the concept and advantages of multi-processing; instead of the PC working serially, waiting for the set of data, processing, and then waiting for the next set, the system saves time by doing both data collection and processing parallelly.

- Going a step further into multiprocessing, the PC could have the STD CPU process some data parallelly with the PC CPU. Such a situation might be advantageous in an application where the PC can use its resources in executing other functions like waiting on the operator while the STD system does all the data processing. In such an appli-

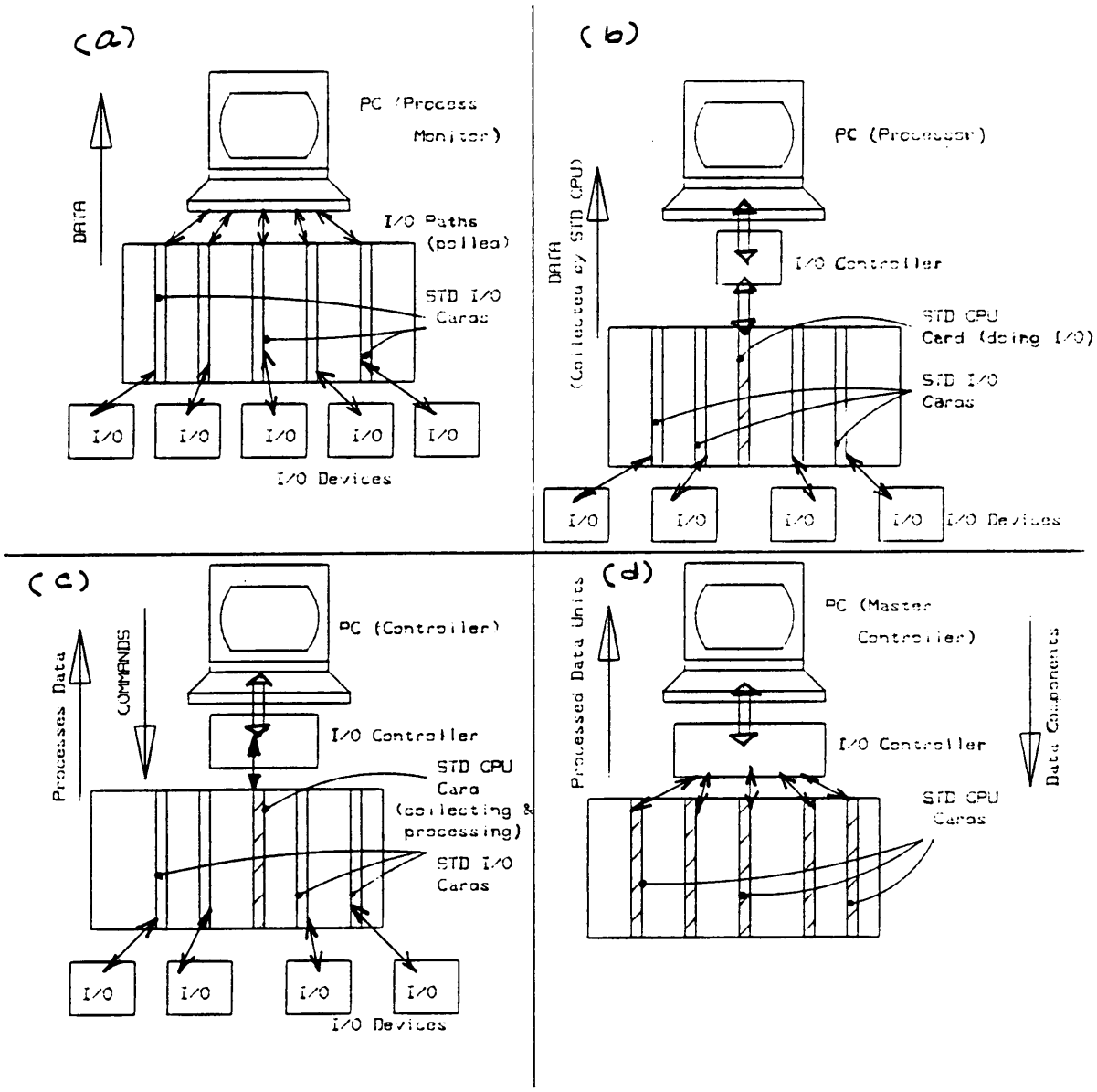

Figure 5. Different Applications of the PC-STD System.

cation the STD software would act on the commands or instructions that the PC sends it through interrupts or DMA or other means, and the PC just waits on the command from the operator and responds to the command by passing the information to the STD bus system.

- In scientific applications like statistical formulations the PC could have satellite STD systems (more than one) compute parts of a process simultaneously and then it could work on the results sent by the different stations. A simple example would be the evaluation of an expression like (A.B)+(C+D)+(E+F)...; in evaluating such an expression the PC could pass the data A & B to one remote processor, C & D to another, and so on. Then the remote stations could simultaneously compute the products and send them back to the PC and the PC would then do the sum to get the result.

So, the performance of the PC and the STD bus system can be combined, using the link, to make a system that can adopt to the particular application in the most efficient way.

4.2 COMMUNICATION PROTOCOLS

The PC-STD system consists of two (or more) systems - a PC and one or more STD systems, running in concert to accomplish a given function faster, more efficiently. The prime consideration in such a set up is the communication techniques - the ways and means in which the units can pass

information/data between themselves. The STD-PC Link offers a reasonably wide choice of data transfer methods as described below.

Data transfer techniques -

4.2.1 Polled Transfer -

This transfer takes place through the Link Mailbox port. The receiving system polls the status bit on the flag port and checks whether the mailbox is full, i.e. if data is ready to be received. It reads the mailbox port when the flag is found to be set. The sending system, on the other hand, waits for the status bit on its flag port to indicate that the mailbox has been cleared and is ready to be written to. It then writes to the mailbox port.

For the STD Link, the STD processor has to check the "SFLAG" - STD Bus System/Interrupt Status port - for the mailbox status. When the STD bus processor is expecting data from the PC over the mailbox, it can poll the PCMAILF bit (bit-4) of the SFLAG port till it is set. It can then read the mailbox port "SMAIL". When the STD bus processor wants to send data to the PC, it polls the STDMAILF* bit (bit-5) of the SFLAG port till it is set. When STDMAILF* is set, it indicates that STD system can write data to mailbox, so then, the STD processor can write to the mailbox.

For the PC Link, the "SERV" - STD BUS SERVICE REQUEST/STATUS - port holds the mailbox status information. If the PC wants to receive data from the STD system via the mailbox it can poll the STDMAILF bit (bit-5) till it is set. STDMAILF indicates that STD data is waiting at the mailbox to be received. The PC can then read the mailbox port "PMAIL" to get the

data. If the PC wants to send the data over the mailbox, it has to check whether the PCMAILF* bit is set. When set, it indicates that the mailbox is available to write data to, and then the PC can write the data to the mailbox port PMAIL.

If some unwanted data in the mailbox is blocking data transfer, i.e. in case the mailbox has some data that the receiving system does not want to receive, and if the sending system has to send new data, then the Link provides the SERV output port on the PC and the SFLAG output port on the STD Link, through which the Sending system can force the mailbox flag clear and proceed with the present transfer.

As an example of this mode of data transfer refer to the procedure CONT_PLOT in "Appendix D. Main Application - Filter Monitor" on page 95 In CONT_PLOT the PC is given the task of plotting the data processed by the STD bus, continuously. So, the PC software does polled input of data and plots the data coming in as soon as it is received.

4.2.2 DMA Transfer -

The mailbox is used for this mode of transfer also. In this mode of data transfer,data can be transferred to or from the PC. This is done by the DMA controller chip installed and available in the PC. The PC link uses jumpers to select one of the DMA channels available. So, the user has to determine which channel is available and configure the link accordingly.

4.2.2.1 The DMA controller

The PC uses the the 8237 High Performance Programmable DMA Controller [5,7]. This chip allows several modes of Direct Memory Access through which external devices could directly transfer information to the system memory. The chip offers a wide variety of control features that helps optimize system organization and enhance data throughput, and being "programmable", the chip makes dynamic reconfiguration through program control feasible.

The DMA controller can transfer data in one of four modes

- **Single Transfer Mode -** In this mode the controller is programmed to make one transfer only. When an external device makes DREQ active the controller makes one transfer, responds to the sender with DACK, and releases the bus to the system processor. This ensures one machine cycle of CPU operation (at least) between transfers. The stipulation enforced on the sending device, for this mode, is that it holds DREQ active till it receives DACK; activating DREQ for a fixed period is not acceptable because the sending device might deactivate DREQ before the CPU cycle is done and hence, before the 8237 can respond to the request.

  In the PC-STD system this involves the STD system checking the mailbox flag until it is set to indicate that the data has been collected.

- **Block Transfer Mode** - In this mode the DMA controller holds the bus and makes transfer continuously until a TC (Terminal Count - activated by the programmed "word count" terminating) signals the end of the transfer. The transfer can also be terminated by a EOP signal (external End Of Process) from the PC CPU. In order to transfer data effectively the sending device needs to have operation speed that matches, the 8237 speed. The DMA controller does not wait for DREQ after the first request is received; it sends DACK and starts reading in data to the system memory from the designated port - till TC. Refer to "DMA Example" on page 34 for more details.

- **Demand Transfer Mode** - This mode of transfer is programmed to make transfers until TC or EOP is encountered or until DREQ goes inactive. So, for a compromise between slow devices holding up the system with Block Transfer mode and Single Transfer being too slow with the PC doing one CPU cycle in between every transfer, this mode of data transfer offers a solution. If the device has data coming continuously, it can hold its DREQ active and thus have the 8237 devote the PC bus solely to transferring data from the device; when the device exhausts the data stream it deactivates DREQ and the bus control goes to the CPU. Bus is reclaimed for fast transfer as soon as the device is ready again just by the device activating DREQ.

- **Cascade Mode** - This mode is used to extend the DMA facility by cascading more than one DMA controller chips. This involves hardware linking that effectively loads one or more DMA channel of the 8237

with another 8237, thus putting 4 channels of DMA through one original channel.

The PC software programs the DMA controller by loading the control registers of the chip. In the present application, only one DMA channel was used and so, no masking is ever required. The PC has to program the Mode register, Word Count register and the Address register. The program reads the Status register for the transfer status.

4.2.2.2 DMA Example

DMA transfer offers a powerful tool in any multiprocessing environment because of the capability to transfer data without program intervention. The DMA controller can be programmed to take over the responsibility of receiving (or sending) data when it is available and putting it in the appropriate location in memory, doing everything in the background while the PC CPU can be working on processing something else.

The program DMAPC.BAS is a program in BASIC that runs on the PC and DMASTD.ASM is an ASM85 program to run on the 8085 processor on the STD Bus; these programs have been developed to study the DMA capability of the PC-STD system. Refer to "Appendix C. DMA Demonstration" on page 84 for program listing and detailed explanation. Since the STD system is used as a satellite system to serve the PC in functions like data collection and processing, the data transfer from the STD system to the PC is of prime importance to us and that is what has been demonstrated in the programs.

The STD system can transfer a set of 16 data bytes upon initiation by the operator. It shows the bytes transferred on the STD keypad display card also. The PC system DMA controller is programmed to receive the data and put it in the PC memory. Just to demonstrate the display-memory characteristic, the destination memory in the PC is its dynamic display memory segment. So, whatever the 8237 puts in the memory, gets echoed to the screen directly.

By changing the "mode register" of the DMA controller, different modes of transfer were tested out. The STD CPU works on the simple transfer protocol of checking the mailbox flag to see if it can be written to and then writing to it the data byte. The results of using the first three modes of transfer are noted below. Cascade transfer is of no relevance here. The 16 bytes transferred are the ASCII codes of the letters of the word "COMPUTER" and the word "*DIPTISH" and the 8 color bits; see "Appendix C. DMA Demonstration" on page 84 for details. The 8085 program alternates between these two sets of data every time the user initiates a transfer by flicking the rocker switch on the STD keypad-display card. The results of the transfer for the word "COMPUTER" are shown.

- **Demand mode with autoinitialize disabled - mode reg = 05H**

  - Result seen on PC screen - **CeOeMePe** - this shows that the STD DREQ is not set off fast enough. The 8237 receives invalid data after every legal data bit. After receiving the first 16 bits, the 8237 TC is active and since autoinitialize is off, the transfers stop.

The 8085 keeps trying to send the rest of its data and hangs up because the 8237 never clears the mailbox.

- **Demand Mode with autoinitialize - mode reg = 15H**

  - Result seen on PC screen - **UeTeEeRe** - the only difference from the previous mode is that the 8237 initializes itself and receives the later half of the 16 data bits sent over.

- **Single Mode with autoinitialize disabled - mode reg = 45H**

  Result seen on PC screen - **COMPUTER** - so, Single Transfer mode transfers the data faithfully. But in this case when the rocker switch on the STD Bus was flicked again, to transfer the next set of bits, the PC DMA controller would not receive and the STD system got hung up in trying to send the rest of the data bits.

- **Single Mode with autoinitialize - mode reg = 55H**

  Result seen on PC screen - **COMPUTER** - transfer is again correct and in addition, when the rocker switch is flicked to tell the STD system to send the next set (*DIPTISH), the 8237 receives the data faithfully and displays the data on the screen.

- **Block Mode with autoinitialize disabled - mode reg = 85H**

- Result seen on PC screen - **CCCCCCCC** - as described in explaining the transfer modes, Block transfer is set off by a DREQ signal from the sending device and then continues reading data till TC. So, the 8237 starts receiving the data when the STD system writes the first letter "C" to the mailbox; it then keeps reading the mailbox 16 times. The STD system is much slower than the 8237 and further delay is introduced by the link cable. As a result, before the STD system software can detect the mailbox empty status and write the next data bit, the 8237 has read the mailbox 16 times. The data latched in the mailbox port is "C" and thus the observed result. The STD system then hangs up because it keeps waiting for the PC to accept the next data.

- **Block Mode with auto initialize - mode reg = 95H**

  - Result seen on PC screen - **RRRRRRRR** - obviously the 8237 reads every leter 16 times, reaches TC, re-initializes itself, and receives the next data bit 16 times. So, the last data bit is seen on the screen.

From the observations above, the Single Transfer Mode is the obvious choice. If data is to come in continuously and the user does not want the STD system to hang up in a loop waiting for the PC to take data off the mailbox, then Autoinitialize should be on. If the PC wants the first set of data and does not want the STD system sending other data over the collected set then autoinitialize should be programmed to be off; in such

a situation, the PC software would have to reset or re-initialize the DMA controller after it has processed the collected data set.

4.2.3 Interrupt Transfer -

This mode of transfer is also done through the mailbox port. The STD link generates interrupts on the INTRQ control line of the STD bus due to one of the following signals -

- PCMAILF - indicating that data has been written to mailbox.

- INTRQC - PC controlled port signal initiated by the PC.

For interrupt controlled data transfer, to the STD system from the PC, the PCMAILF flag interrupt is used. A jumper on the STD link card selects the kind of interrupt to be generated; in our set up we used the jumper W10 to select 8085 RESTART interrupt. So, if interrupts are enabled on the 8085, the PC writing data into the mailbox generates a RESTART interrupt to the 8085. It jumps to the appropriate vector location to service the interrupt.

Note: In the VPI&SU STD Bus set up, the interrupt generated by PCMAILF is **RST6** and the program jumps to 2600H for the interrupt service routine.

This transfer protocol is useful in situations where the PC controller would like to change some functional parameter of the STD system software on the fly, i.e. if the PC wants to put in some information **while** the STD

system is executing its function. There could be several sources of interrupts, in which case the STD system software, on detecting an interrupt, would have to check the status registers to find out which particular device had interrupted. In the application developed, only the PC interrupts the STD system to change the command status when the operator desires so. As a result, the interrupt service routine in the STD memory (see "Appendix D. Main Application - Filter Monitor" on page 95) simply reads in the mailbox and puts it in the right location for the STD software to refer to.

The STD system can also interrupt the PC. For that the PC would have to enable the interrupts at the PC link and set the vector address in its interrupt controller.

4.2.4 Forced Transfer -

This mode of transfer is the only one that can be done without the mailbox port. It is a direct transfer. It is also unique in the sense that transfer from the PC to/from the STD memory or I/O can be initiated by the PC but not by the STD system. The transfer between the PC and the STD system involves the PC taking control of the STD Bus and then doing the read or write operation directly. The STD CPU is put on hold during the transfer.

A good example of this mode of transfer is the TRANS program which is used to down load ASM85 object code to the STD memory; See "Appendix A. TRANS Program" on page 62 for program listing. For this application the PC is used as the station where the 8085 program is developed. The

ASM85 program is assembled on the PC and the object code is generated. This code is loaded into the STD memory and this code is what the 8085 CPU runs. So, when the PC loads the object code into the STD memory, the STD CPU could not be doing anything useful - it is, in fact, being given the code on which it will be working. This means that in transferring the object code the PC does not have to bother about any complicated transfer protocol in which it has to make the transfer transparent to the STD CPU; it can simply go ahead and take control of STD Bus and put the code directly into the memory. This is an ideal situation to used forced transfer.

In the TRANS program the following steps are taken to accomplish the transfer of code -

1. Take control of the Bus - BUSRQ and BUSAK controls are established, thus getting BUSLINK. The STD CPU is "on Hold".

2. Enable STD memory access - the MEMRQ and MEMEX control signals are put on the STD control bus.

3. Latch the address on the address bus - the address at which the present data is to go is found from the ".OBJ" text file[4] and is written to the two address ports, thus latching the address on the STD address bus.

---

[4] The ".OBJ" text file contains data in Hex form. The TRANS program has to interpret the data into the numeric values before using them.

4. Write the data or code-byte to the data bus - the code-byte for that address is found from the ".OBJ" file and is written to the STD bus data port. The data bus gets the data and thus, the data gets written to the memory location addressed by the address bus contents.

In addition to the appendix, also refer to "STD System Development on the PC" for more on TRANS and its use. The transfer of data is done as fast as the PC software can read the data off the disk file and process it to the numerical values from the ASCII characters read off. Here the transfer through is advantageous because it is the simplest and fastest; and because the CPU can be put on hold without penalizing system performance - this in fact starts the system off! Such situations use Forced I/O to an advantage.

Note: The TRANS program also reads back the data written and checks with the data sent to see if they are the same. If they are not then it reports the error to the operator. The problem could be that there is no RAM at the address location or that the STD bus is not powered.

4.3 STD SYSTEM DEVELOPMENT ON THE PC

From our discussions so far on the relative merits and operational necessities of the PC-STD system (see "Scope of the System" on page 23) we note that the STD system can act as a support to the PC, but it has to be "fed" by the PC. In other words, no program development of any considerable size can be done on the STD system; it needs the PC to interpret assembly language programs into 8085 executable machine code and to load

the code into the STD memory whence the 8085 CPU can get and execute the code. This obviously involves two steps - 1)assembling ASM85 programs and 2)loading the executable code into the STD memory.

For assembly of 8085 assembly language programs, I developed my own Assembler. This assembler reads ".ASM" text files from the diskette and assembles the program. It prints the errors, with possible corrections, to the PC screen, and if the user wants, to the printer also. If the program is error free then it goes on to generate the object code in a ".OBJ" file. This file is in the standard INTEL Hex format. The executable version of the assembler is the "ASM85.COM" file.

I had the option of using an 8085 cross assembler developed by 2500 A.D. which has an Assembler, a Linker and a Hex Converter. This is a versatile and powerful software that can link more than one relocatable programs together. It has to be worked in three steps - assembling, linking and converting the .OBJ file to the .HEX file which is in the standard INTEL format.

After the object code in Hex is generated, it has to be loaded into STD memory in order to run it on the 8085 processor. For this I developed the TRANS program. It interactively reads in the name of the file to be transfered. It assumes that the file is in the INTEL Hex format. If it detects any obvious errors in the format, it reports the discrepancy to the user and aborts the transfer. If the file is all right then it picks out the address and code data. It has a procedure to convert the Hex codes to the numeric values. Two characters are passed in and the 8-bit numeric value is computed from them in the function "CONVERT". Refer to "Appendix A. TRANS Program" on page 62 for details. The address and data are used

to put the correct data in the correct positions. The STD memory is read back after writing to it in order to check the fidelity of the data; if any error is noted the program reports it to the user and aborts the transfer.

Using these tools a reasonably complicated program can be developed and loaded into the STD system. The PC can be used to design a multi-processing protocol; the CU (Controlling Unit) of the multiprocessing environment is used to depute the tasks to be done by the PU (processing Unit) through these tools.

4.4 APPLICATION EXAMPLE

After investigating the different aspects of the PC-STD system, an application was developed to demonstrate the main features and how they can be used to an advantage. The application chosen is a Digital Filter Monitor. The system as a whole can simulate a low-pass filter; it can change the filter parameters, thus, effectively changing the filter. It can filter an input analog signal and generate the filtered analog output; it can display the input/output wave-form to the operator; it can test the filter response by generating pure Sine wave samples of any desired Sine wave and showing the input and output plot to the operator. The different modes of operation of the Filter Monitor are depicted in Figure 6 on page 46.

The basic structure of the program is described here in order to focus on the features relevant to multiprocessing and their use. The details of the program, the listing and flow-charts are included in "Appendix D.

Main Application - Filter Monitor" on page 95. The PC software, in the foreground, runs a operator service function; it waits for commands from the operator. The STD Bus system runs the Filter program that has been developed on and loaded from the PC; it receives inputs and processes them to generate filtered outputs - continuously. When the PC receives some command from the operator, it takes the corresponding action by jumping to the appropriate procedure. It can, upon operator command, change the operation mode of the STD system - it can change the input source between the PC generated input sample set and the external analog input; and it can change the output destination between the external analog output and the PC system. The PC conveys this source/destination information to the STD system without interrupting its operation - the transfer is transperent to the user; the source/destination change almost instantaneously. The PC, when receiving the output can, at the same time, service the operator and take the incoming data into a reserved data memory segment; or it can display the incoming data to the PC screen. While displaying the data the PC also does some processing of the data. It checks the amplitude and the frequency of the incoming waves[5] and displays them to the operator. This occurs at the same time as the STD system processing the next set of input data and while the PC waits for the next processed output. On command from the operator, the program can also change the filter constants, thus testing and using a totally new filter. In order to check the response of a different filter the operator does not have

---

[5] Periodic waves are assumed if a steady amplitude-frequency is to be expected.

to stop the program, change the parameters on the STD filter system, and then resume the monitoring/testing operation. All he needs to do, is issue a command to the PC and the PC interactively reads in the new constants, passes them over to the STD system over the link and continues operation - everything is taken care of by the multiprocessing system.

With the basic structure of the system in mind, let us discuss some of the relevant features of the program and some points that need to be noted.

4.4.1 Program features on the PC side

The PC is the Controlling Unit and is the system master. The different multiprocessing features and communication methods are discussed below.

4.4.1.1 Receiving Data in the background

Two modes of DMA transfer are used to receive data from the STD system. The PC is used to wait on the operator and the foreground job it does is simply polling for operator commands. If the operator has set the STD system to send the processed - "filtered" - data to the PC then the STD system simply checks the mailbox port for ready - to - send status and sends the data to the mailbox. The PC has to take the data from there - fast - so that the STD system does not end up waiting for ever to send data. If the PC is waiting for operator commands in this mode, then it has to be arranged for the PC to accept data in the background.

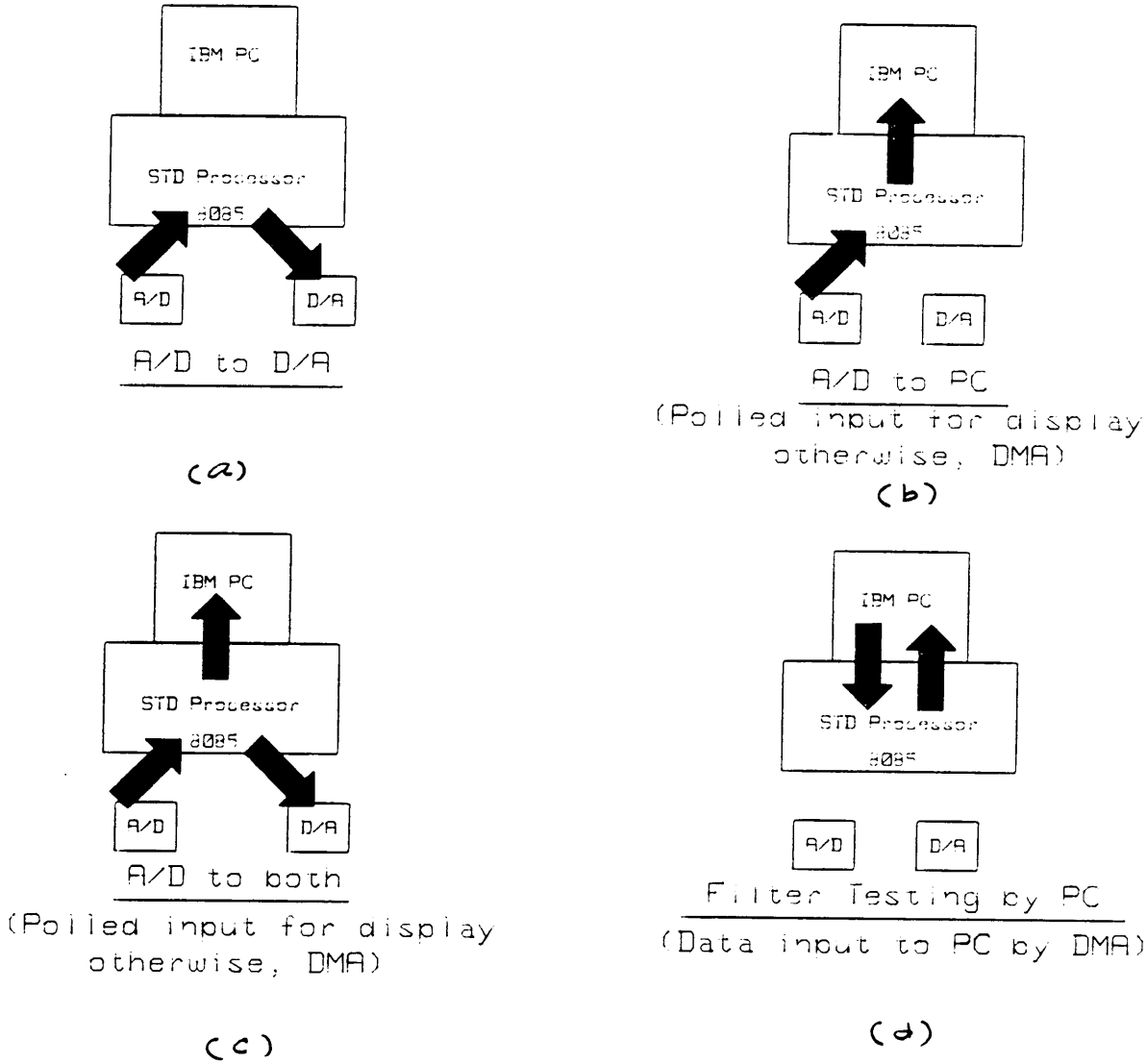

Figure 6. Different Modes of the FILTER MONITOR.: The monitor program offers the modes depicted in the figure. The operator can select the mode through a menu on the PC screen.

This is where the first case of DMA transfer is necessary in this application. The mode used is Single transfer mode with Autoinitialize enabled. The word count is kept at 1000. So, what essentially happens is that the DMA controller in the PC keeps receiving the data and putting it in a fixed block of data memory. After writing 100 data bytes, it simply re-initializes and writes over the same area again. As a result, the STD system is never held up.

The other situation where DMA transfer has been used is when the PC "tests" the filter by loading in PC-processed samples of a specific wave and displaying the processed data. In this case, the operator would, presumably, like to examine the processed output wave in some detail. The program procedure SECT_PLOT (see "Appendix D. Main Application - Filter Monitor" on page 95) can display any chosen segment of the 500 input-output data set on the PC screen. By choosing a small segment the operator can also examine an expanded - magnified or stretched out - form of the wave. So, what we need, is to store the set of 1000 data units in the PC memory for the operator to examine as he wishes. A simple and efficient solution - and possibly the fastest solution - is found in employing the 8237 to do the transfer to memory. The mode used for this is Single transfer with Autoinitialize disabled. After the input data is loaded into the STD memory and the STD system is commanded to start processing the data, the PC programs the 8237 in the above mentioned mode. The software then checks the 8237 status register till the TC flag of the channel used goes up.

Note: After the DMA controller is initialized for the Single transfer mode with no autoinitialize, the TC flag in the status register is set. So,

the **software must perform a dummy read of the status register to clear the flags.** After that the collection of 1000 data units causes the flag to go up - and is correctly detected by the software.

The DMA controller does not re-initialize until the PC software does it, and so the data collected can be examined by the operator. To terminate this mode of the program the operator issues some other command which, if necessary, puts the 8237 back in the autoinitialize mode.

4.4.1.2 Receiving Data Continuously

When the operator commands the STD system to send the processed data to the PC in order that the operator can monitor or examine the input-output profile, the data is sent to the mailbox port for the PC to collect. If the DMA controller is enabled, then the data is picked up and put in the memory right away. But for this application, the data needs to be displayed as soon as it comes in; so, putting the data in the memory is not advantageous in any way.

In this situation polled input is the best. The PC can poll the mailbox port and pull in the data as soon as it detects that data has been written. The PC processes the data - see procedure CONT_PLOT in "Appendix D. Main Application - Filter Monitor" on page 95 for listing; it scales the data in order to find the coordinates for the graphics plot of the data, and it also retains a check on the maximum magnitude of data and the frequency of maxima occurance to find the amplitude and frequency of the incoming waves. So, while the PC processes the incoming data to find the plotting coordinate, the frequency and the amplitude of the wave, the STD system

is performing I/O, reading in the A/D data and processing it to find the filtered output. This is an example of efficient processing through multiprocessing.

4.4.1.3 Sending data through Forced I/O

The program has the capability to change the Filter that is being simulated on the STD bus. In order to do this the PC software accepts the new constants from the operator interactively and then loads them into the STD memory at the fixed locations for the constants. The STD system can then use the constants from these locations to formulate the filtered output.

It is to be noted that for this part of the application, an interruption of the data processing on the STD bus is of no consequence because it involves changing the basic simulation! As a result, we can avoid the complicacy of other transfer protocols by using the Forced I/O technique. The PC simply takes control of the STD bus (see procedure LOAD_DATA in the main application program) by establishing BUSLINK[6] and injects the set of constants, in the format the STD system expects, at the right memory locations. It also sends a command to the STD system telling it that it now has a new filter and so, it has to reinitialize the processing of data. This application situation uses Forced I/O to an advantage.

---

[6] BUSLINK involves sending out BUSRQ signal to the STD Bus control lines and waiting for the 8085 processor on the STD bus to grant BUSAK. This allows for graceful takeover of the bus - without interrupting any 8085 machine cycle.

This method of transferring data is also used when the PC loads the PC-processed data samples of a certain wave to "test" on the STD filter. This application interrupts the continuous processing of real time external data, and so, forced transfer is used to transfer the input set to the STD memory.

4.4.1.4 Commanding the STD system

The PC being the Master controller in the set up, has to command the STD system. Depending upon what the operator wants, the PC has to change the mode of the input and output (see Figure 6 on page 46) of the STD system. When data is being processed, the PC does not want to interrupt operation long enough to hamper continuity. So, Interrupt was used to transfer the command. A small interrupt service routine at location 2600H - vector for RST6 interrupt - reads the data that generates the interrupt (see "Interrupt Transfer -" on page 38 for more details) and puts it in the status byte location. The processing resumes after that with minimum interruption.

It should be noted that the PC-STD link provides for priorityized interrupt structure, and so, multiple interrupting devices could be there. But in our case the situation is simplified, and the need for checking which device is interrupting, is eliminated by the simplicity of the set up.

4.4.1.5 DMA data destination on the PC

This feature of the application deserves special mention because it deals with the PC memory, and random selection of data area could effect the system operation in extremely drastic ways. TURBO Pascal[7] has the special feature (amongst others) of being able to determine the data segment and the offset - in other words, the physical memory address - of any data variable; refer to the TURBO Pascal Manual [8]. A data array is declared in the variable declaration part of the program. Using the feature mentioned above, the program determines the "page" and the offset on the page, for the location assigned to the data array. These values are used to load the DMA controller Page register and the Current Address Registers; refer to "Appendix D. Main Application - Filter Monitor" on page 95 for details. This can be circumvented by assigning a fixed memory location to the data array; but that solution poses the obvious problem of memory area conflict with other variables and also, possible waste of memory area. Allowing the PC to dynamically assign the memory area to the data array, we use memory efficiently and then using the segment and offset variables we program the DMA controller.

---

[7] TURBO Pascal is a high speed Pascal compiler that runs in the MS DOS environment, developed by BORLAND International.

4.4.2 Program Features on the STD side

The STD system is the Slave processor to the PC; it is programmed and run by the PC. Due to the kind of application that it is used for, the STD system processing should be interrupted for the minimum amount of time, for any reason. All this lead to the following interesting features in the STD system software.

4.4.2.1 Status Byte for the STD System

The mode of operation so far as routing the output and receiving the input has to be varied. For this the scheme developed is one using a **Status Word**. This is a reserved byte in the STd memory that has specific bits assigned specific attributes. There are three classes of bit attributes -

- there is a bit to indicate where the input is to come from - the A/D converter or the PC processed data.

- there are bits to indicate where the output is to be routed - to the PC, to the D/A converter, to both or to neither (idle state).

- there is a bit which is a flag indicating when the status has been changed by a command from the PC.

The program running on the STD bus, after every iteration, checks the flag bit of the status word. If it finds that the flag is up, meaning that a new command has been issued, it reinitializes the iteration and starts again - this time operating on the new status set by the PC. The use of a status byte provides a graceful means of keeping track of the mode of operation and of altering modes.

4.4.2.2 Interrupt Service Routine

This posed an interesting problem because the vector for the RST6 interrupt is set at 2600H by the Va.Tech STD system. 2600H is located in the 8085 processor's on-board memory. This memory can be accessed only by the 8085. So, when down-loading the STD system program from the PC, **the interrupt service routine can not be loaded at its vector location** by the PC.

A simple solution to this problem was found by putting the interrupt service routine, in machine code, into the STD common memory area, and then have the 8085 program do a block transfer of the routine to the on-board memory area at 2600H. This wastes programming effort and some initial processing time, but once the 8085 program is run, the routine gets put in position and so, it does not affect the system performance in any way.

4.4.3 Limitations of the application

The first limitation to the system developed, is the A/D and D/A converters used. The AID box[3] was used to do the input and the output. There are two main disadvantages in using this device -

1. The conversion is very slow and hence, it limits the sampling rate of the filter. The maximum sampling rate got is only 125Hz; this low rate is due to the slow processor speed as well as the conversion speed of the AID devices. If a better quality industrial I/O board was used then the system would not only be faster, it would also be more compact physically. The STD program has to wait a considerable time before the A/D converter has a converted value ready for it. After issuing the Start Conversion signal to the converter, the 8085 has to poll the status bit of the converter to check for conversion completed; it was seen that the 8085 has to run about 7 polling loops before the data conversion was done! This, along with the large volume of arithmetic to be done to find the output, increases the processing time for each data quite a lot.

2. The A/D converter takes only positive analog voltages. For this the program had to be modified and also the analog input mode. For the input, the wave was made to ride on a DC positive voltage so that the wave is boosted up to all positive values. The maximum positive voltage the converter can take is +10V. So, a wave swinging from +5V to -5V made too ride on a +5V DC level is the best we can do. Within

the program the input has to be scaled down by subtracting the DC level so that the computations can be done correctly. All this involves extra programming effort as well as increases processing time.

Another disadvantage that comes to mind is the 8-bit boundary of the STD bus devices. The STD bus is a base for 8-bit processors and hence that is a factor that the system designer has to live with - he should take into consideration that fact before starting on the design. For applications that involve a lot of numerical processing in which high resolution or large values have to be dealt with, this is not the right system.

In the PC-STD system developed, the computations were done in decimal arithmetic - so, the results got are in decimal with a resolution of 0.01! But, when sending the data to the PC or to the D/A converter, the decimal part is not used. So, the results had to be rounded off. This involves a loss of accuracy.

4.5 OTHER APPLICATIONS

A feedback speed control system was attempted with the motor that is installed on the AID [3] box. In order to read in the speed of the motor, a Shaft Encoder[8] was used. This unit sits on the motor shaft; it has, inside it, a disk with 500 equally spaced stripes of opaque radial lines along its circumference. An emiter sits on one side of the disk and a receiver on the other side. The receiver produces a high voltage level

---

[8] The Shaft Encoder is manufactured by Hewlett Packard.

when the light from the emitter hits it. So, when the motor is running, the light is interrupted by the stripes on the disk, thus, effectively producing a pulse train from the receiver. The pulse width varies with the speed of the motor and thus, is a measure of the speed.

The 8085 on the STD bus was used to read the speed in; the following steps are involved -

1. Read in the analog voltage level through the digital input port.

2. Check if level is high.

3. If level is high then increment a counter.

4. Repeat above steps till level is low.

The above process gives us the number of high samples, i.e. the pulse width, which in turn, is proportional to the speed of the motor. The problem with the system was that the 8085 processing time was too slow to have a satisfactory sampling rate. The time between the samplings of the port was found to be 10uSec. This gave only 7 high samples for the slowest speed; for all speeds above the half-speed of the motor the number of high samples fluctuates between 1 and 2. So, two major problems are faced - 1)There is very little resolution between speeds - thus effective speed control becomes impractical and 2)The sampling rate being low, the number of samples for a constant speed is not steady. This application,

though feasible for the set up we have, is made impossible by the processing speed limitations.

The other application considered was to have an IEEE-488 [9] bus system on the STD bus.[9] This would allow for a powerful multiprocessing system with extensive I/O monitoring - this can be seen as a useful industrial application.

---

[9] There is a STD Bus compatible card ZT 7488 (IEEE 488 Interface for STD Bus) made by Ziatech Corporation, governed by the INTEL 8291 and 8292, that can implement the IEEE 488 extension to the STD system.

5.0 CONCLUSIONS

The PC-STD bus system has a lot of potential in an industrial or in a lab environment. Its multiprocessing capability can be used to an advantage to enhance system performance or to demonstrate the same for academic purposes. We can summarize the essential features of multiprocessing that was found to be of importance in any system that has to implement multiprocessing.

- Two or more processors to share the processing load.

- Communication link to transfer data and information between them.

- Hardware capability to offer handshaking for data transfer.

- Software capability to develop the required application on the system.

- Software modules to program the data transfer hardware or to do the transfer.

In general, the "hardware for data transfer" can be of two types. One is the simple hardware port to which the devices can write data and the other devices needing this data can read the port to receive it. In this elementary scheme there is always the chance of data access con-

flicts. The sending device, for example, checks the flag to see if the port is free, and then it writes the data to the port. Now, if two units in the multiprocessing system happen to have data to send at the same time, there is a finite probability, however small, that both devices read the status at the same time, find the flag indicating that the port is free to write to, and write to the port at the same time. This will definitely introduce error and loss of data in the system. The PC-STD link by RMAC that was used for the project uses this means and thus, has the mentioned disadvantage. In the set up used, the situation can be more specifically described as when the PC is trying to send a command to the STD bus and the STD bus is trying to send processed data to the PC. Another disadvantage of the system is that data transfer has to have a lot of software support. The software does the actual handshaking for data transfer by checking the flag and writing or reading the data. The hardware simply offers the transfer base.

The other type of hardware is the serial transfer done by the USARTs (the Universal Synchronous Asynchronous Receiver Transmitter). This kind of hardware has to be programmed by the software for the framing of the data - the number of data bits, number of stop bits, kind of parity, etc. and the hardware takes care of packaging the data and sending it or of receiving the data and checking the package to see if the data is correctly transmitted. The hardware has the means to check whether the other side is ready for the transaction and takes care of the transfer independently. This is a much more complicated set up than the previous one, but the trade off lies in its reliability and speed.

Multiprocessing usually involves more than two processors, and is used when the added cost of the link between the systems and the extra software to run the system is balanced by the gain in the speed and efficiency in the process execution. This points to a reasonably complex process for multiprocessing to be used to an advantage. The system developed here is the base of such a system. It demonstrates the capability and the potential of using the inexpensive and flexible components to set up a powerful and efficient processing environment. The obvious extension to the system developed, is to have two or more STD bus systems, each with their own resident processor, connected to the PC by PC-STD links. This would be the ideal multiprocessing scheme with one CU (Control Unit - the PC) and several PUs (Processing Elements - the STD systems). The PC would be used to develop the application and depute the tasks to the different STD systems that are its slaves. The links have a jumper feature that allows the user to alter the addresses of the ports - the status/control ports on the link. So, every STD system could be made to have unique addresses and identities. We can also have local memory for each STD processor or we could have all the processors using a common memory bank with some access priority and protocol. This opens up a wide scope and a wide range of possible applications. Multitasking could also be implemented by the PC controller to have an even more powerful system.

Another possible application is a STD system development station. For program development on the STD bus and to debug the programs easily, the lab set up in Va Tech has to use the mainframe IBM370 computer and a Zenith terminal. The mainframe computer is used for program development using the full screen editors and such other facilities; a software module on

the mainframe can then down load the program through a RS232 link to the STD memory; the terminal linked to the STD bus runs software on the EPROMs on the STD bus to debug or monitor the program. All this can be implemented quite easily with the PC-STD set up. Program development (with assembly) can be done on the PC - this is already feasible with the software developed for this project; the PC can, over the RMAC link down load the object code to the STD memory - this also has been done for the project; and as a further step, software can be developed to run and debug the STD programs - with single stepping, trace, and break-points. This is not a very major problem and, with the tools developed and investigated in this project, can easily materialize as a compact and elegant development system.

The PC-STD system promises to be a easily controllable and yet reasonably powerful system. The vast I/O capability of the STD bus and the user interface of the PC are a useful combination to develop any application. The STD CPU is the basis of a multiprocessing scheme which can enhance any large computational process by doing isolated segments of the computation parallelly. The system is cost effective and immensely flexible so far as system components (on the STD bus) goes. It has the scope of system expansion with minimal problems and the modular nature of the system makes it easy to adapt to. To conclude, the PC-STD system definitely has the basis of a wide range of possibilities, is economical, and simple.

APPENDIX A. TRANS PROGRAM

This is an essential part of the development system that is to be implemented with the PC and the STD Bus. The STD bus does not recognize any other language but Machine Code. The machine code of the program has to be loaded into the memory of the system and then the STD CPU (8085 in our case) can be commanded to execute the instructions from the memory. But machine language programming is not only tedious, it is a total waste of effort and time. Assembly language programs can be developed with relatively more ease; reasonably complicated applications can be developed using assembly languages. Assembly languages are not totally system independent like high level languages - the programmer has to keep track of the registers available, memory locations used, etc. But, the mnemonics are something that makes more sense to the programmer than a set of numbers that is the machine coded version.

The Assembler is used to make the translation from the assembly language to the machine code. The machine code is just a sequence of byte values to be loaded at fixed memory locations. The TRANS program loads the bytes into the memory locations desired. The byte values have to be given to the TRANS program in a particular format; this is the INTEL format.

A.1 THE INTEL OBJ FILE FORMAT

The INTEL format of the Object Code is a standard that is widely used. It has the following rules :

- A row of data bytes starts with a " : ".
- The : is followed by a two character Hex number (a byte) that represents the number of bytes of data that are there in the row.
- The next 4 characters represent the Hex value of the address (2 bytes or a word) from which the data in the row is to be loaded.
- The following 2 characters are not used in most systems - they are kept as "00".
- After this there follows as many two-character sets as are specified by the 2nd and 3rd characters in the row. These are code bytes in Hex.
- Finally, the last two characters in the row gives the check-byte. This is got by a bit wise OR of all the bytes in the row.
- The Object Code file should terminate with a row that has "00" for the 2nd and 3rd characters - 0 bytes of data in this row.

An example of a data line would be -

-:025100004073218 25C306250357

This can be interpreted as meaning - this row has 16 bytes of code 04H, 07H, 32H, ... 25H, 03H. They are to be loaded starting from memory location 2510H. The check byte is 57H.

A.2 THE TRANS PROGRAM FUNCTION

This program does not use the check byte at all. It does make some check on the fidelity of the data transferred to the STD memory though; this is explained later in this section.

The program reads in the name of the file to be transferred. It makes a check on the file type given to it.

Note: The TRANS program accepts object files in INTEL format with the file type .OBJ only. If .HEX files are to be accepted also, a small modification in the INIT procedure of the program can be made to arrange for that.

The program then takes total control of the bus by issuing a BUSRQ and a BUSAK control to the STD Bus. The MEMEX and MEMRQ controls are also issued to enable memory write. Then it proceeds to work on the are read off the file into integer numbers. The program first searches for a ":" in the file; when it finds one, it reads the next two characters, converts them to an integer value and checks to see if it is is greater than 0. If it is 0 then the file is ended; the program terminates. If it is greater than 0 then the program puts the address found in the next 4 characters in two byte variables and starts putting in the code bytes.

1. The address bytes are loaded in the "hiadr" and "loadr" ports.

2. The data byte is written to the data bus port "sbus". This effectively accomplishes the write to the memory location addressed by the address bus contents.

3. The data bus port is read into a variable (chk_data).

4. If chk_data is not the same as the data byte written, error is reported and program terminated.

5. The address is incremented.

6. The above steps are repeated. (The program stops when the number of bytes - 2nd and 3rd characters - reads zero)

So, the reading back of the data offers a means of checking on the data that is written to the STD memory. STD Bus not being powered or there being no memory at the addressed location are two common errors that lead to a discrepancy in the data written and the data read back. The program writes the ":" sign to the PC screen every time it reads it off the .OBJ file at the beginning of every row of characters. This gives the user a stream of ":"s as the TRANS program runs down the file.

A.3 TRANS LISTING

```
{*****************************************************************
*                                                                 *
```

```
*               |  Program     = OBJECT CODE LOADER |     *
*               |  Developed by= Diptish Datta      |     *
*               |  Language    = PASCAL             |     *
*               |  System      = IBM PC             |     *
*               |  Compiler    = Turbo Pascal       |     *
*               |  Date        = June 1985          |     *
*               |-----------------------------------|     *
*                                                          *
*  PROGRAM FUNCTION: The  program transfers object code   *
*      from the PC disk memory to the STD system memory    *
*      It  takes  control  of the STD  Bus;  the  8085     *
*      is put on hold and  the  memory  is  written  to    *
*      the STD memory via the PC-STD Link.                 *
*                                                          *
*  PROGRAM LIMITATIONS: The object code file must comply   *
*      with the INTEL Object  file format.  The  program   *
*      checks the  file  type  of  the  file  to be down   *
*      loaded; it accepts only files with file type .OBJ   *
*      but this can be modified to accept .HEX files.      *
************************************************************}


 program TRANSFER(input,output);
  const
   sbus = $301;
   shad = $303;
```

```
    slad = $302;


  var
    f_obj: text;
    f_name: string[14];
    t1,t2: char;
    bites,ctr: integer;
    hiadr,loadr,sdata: integer;
    chk_data: byte;
    fini: boolean;

{"=============================================================="
 " INIT                                                        "
 " Parameters Passed= none                                     "
 " Called By        = Main Program                             "
 " Function         = reads in the .OBJ file name; takes       "
 "                    control of the STD bus.                  "
 "=============================================================}

  procedure INIT;
   var l:integer;
   begin
    write(' Enter name of file to load: ');
    readln(f_name);writeln;writeln;
    if f_name='' then begin
     writeln(' Required entry omitted.');halt;end;
    l:=length(f_name);
```

```
    if (f_name[1-3]<>'.')or(upcase(f_name[1-2])<>'O')
     or(upcase(f_name[1-1])<>'B')
     or(upcase(f_name[1])<>'J')
     then begin writeln(' Illegal file type.
                    ".OBJ" file expected.');halt;end;
    assign(f_obj,f_name);reset(f_obj);
    port[$304]:=$6A; {**** for DLL PC is total bus
              master; so BUSRQ and BUSAK both given ***}
    fini:=false;
   end;

{"=============================================================="
 " CONVERT                                                      "
 " Parameters Passed= a,b = characters from .OBJ file          "
 " Called By        = Main Program                             "
 " Function         = converts the two character string        "
                      representing a Hex number into an        "
 "                    integer byte.                            "
 "=============================================================="}
  function CONVERT(a,b: char): integer;
   function INT_VAL(c:char): integer;
    begin
     if (not (c in ['0'..'9']))and(not (c in ['A'..'F']))
      then begin writeln(' Error in .OBJ file.
                          Check INTEL format.');halt;end;
     if c in ['0'..'9'] then INT_VAL:=ord(c)-ord('0')
```

```
        else case c of
         'A': INT_VAL:=10;'B': INT_VAL:=11;'C': INT_VAL:=12;
         'D': INT_VAL:=13;'E': INT_VAL:=14;'F': INT_VAL:=15;
        end;
      end;
     begin
      CONVERT:=(INT_VAL(a)*16)+INT_VAL(b);
     end;

{"=============================================================="
 " READ_TILL_COLON                                             "
 " Parameters Passed= f_obj = text file to work on.            "
 " Called By        = Main Program                             "
 " Function         = reads the file sequentially till         "
                      it  encounters a ":"                     "
 "=============================================================="}
   procedure READ_TILL_COLON(var f_obj:text);
    begin
     repeat begin
      if not eof(f_obj) then read(f_obj,t1)
       else begin writeln(' Error in .OBJ file.
                          Check INTEL format.');halt;end;
      end until t1=':';write(':');
    end;

{==============================================================
```

```
|                M A I N      P R O G R A M          |
=================================================}

 begin
  INIT;
  while not fini do begin
   READ_TILL_COLON(f_obj);
   read(f_obj,t1);read(f_obj,t2);
   if (((not(t1 in ['0'..'9']))and
     (not(t1 in ['A'..'F'])))or((not(t2 in ['0'..'9']))
     and(not(t2 in ['A'..'F']))))
      then begin writeln(' Error in .OBJ file.
                          Check INTEL format.');halt;end;
   bites:=CONVERT(t1,t2);
   if bites>0 then begin
    read(f_obj,t1);read(f_obj,t2);hiadr:=CONVERT(t1,t2);
    read(f_obj,t1);read(f_obj,t2);loadr:=CONVERT(t1,t2);
    read(f_obj,t1);read(f_obj,t2);
    for ctr:=1 to bites do begin
     read(f_obj,t1);read(f_obj,t2);sdata:=CONVERT(t1,t2);
     port[shad]:=hiadr;port[slad]:=loadr;
     port[sbus]:=sdata;
     chk_data:=port[sbus];
     if chk_data<>sdata then begin
      writeln(' Memory error (no power or no
                          memory at address).');halt;end;
     loadr:=loadr+1;if loadr=256 then begin
```

```
        loadr:=0; hiadr:=hiadr+1; end;
      end;
    end else fini:=true;
  end;
  port[$304]:=0;
end.
```

APPENDIX B. CONTROLLING THE STD BUS FROM THE PC

The PC can take control of the STD Bus totally and operate on any of the passive devices on it. It uses the control signals and the address and data lines of the STD Bus to access the memory or any I/O device on the Bus.

The Keypad card on the STD Bus is the main user interface available on the STd Bus system. It has a keypad with push-button keys, it has a 8-character display unit, it has a 8-bit LED bank and two rocker switches, and it has a RESET key connected to PBRESET control of the STD Bus. The PC can read the keypad keys or the rocker switches and can write to the display panel or the LED bank, through the PC-STD Link.

B.1 PROGRAM DESCRIPTION

The keypad display card and its application is totally described in the STD manuals [10].

B.1.1 Kepad Card Features

The alphanumeric display can display 8 alphanumeric characters; it uses 16-segment displays for each character. The display characters are latched for each display position and a total of 64 display characters are available. The output port D1H controls the selection of the display position for a particular character and also provides the write control.

Output port D0H is the data port and the character written to it is displayed at the position chosen by port D1.

The Keyboard has 24 keys that are program definable. A 4 by 6 matrix is used to detect these keys. Software has to poll the matrix to detect the key hit. The matrix columns are driven by output port D0 (bits 0-3) and the input port D0 is used to sense the rows (bits 0-5). As the output port bits are activated sequentially, the key that is pressed latches that row bit to the input port. So, knowing which column is being activated, and checking which row is on in the input port, the software can detect which key has been pressed.

The LED port is D0H and the value output to that port gets latched and is displayed on the LED's.

The rocker switches are connected to the port D1 (bits 6-7) and can be read at any time to determine the position of the rocker switches.

B.1.2 Program Steps

The program uses data arrays of character strings to define the different keys. The strings are written to the PC screen when the key is detected. It also defines an array of 192 bytes - this is for the STD display; 8 bytes for the 8 display positions for each of the 24 keys gives a total of 8*24=192 bytes.

The program sequentially sends out an activating signal for the 4 columns and reads in the port D0 to see if any of the bits 0-5 are on. If it finds any bit on, then it references the array with the column that was activated and the row bit that was on, and prints the key to the

screen. it also sends the sequence of 8 characters that represents the key to the keypad display, enabling consequetive positions on the port D1 and issuing the write signals to latch the data. Thus, the key that is pressed on the STd keypad card is displayed on the keypad display panel as well as the PC screen.

An extra feature added, is that of avoiding multiple echoing to the PC screen for the duration that the keypad key is pressed down. A backup byte is used to store the value of the key pressed. If a key is found to be pressed then the software checks if it is the same as the backup; if it is, then the value is not acted upon. When the software does not detect a key pressed, it simply puts an 'x' in the backup byte. So, when a key is pressed again, action is taken on it.

B.2 PROGRAM LISTING

```
{*************************************************************
*      ____________________________________________          *
*      |  Program      = STD BUS KEYPAD MONITOR   |          *
*      |  Developed by= Diptish Datta             |          *
*      |  Language     = PASCAL                   |          *
*      |  System       = IBM PC                   |          *
*      |  Compiler     = Turbo Pascal             |          *
*      |  Date         = June 1985                |          *
*      |------------------------------------------|          *
*                                                            *
*  PROGRAM FUNCTION:  The program polls the STD              *
```

```
*       Bus keypad card and  reads                          *
*       in any key pressed on it. The program then          *
*       displays the key pressed                            *
*       onto the PC screen as well as the Keypad            *
*       card display panel.                                 *
*****************************************************}

program KIPAD(output);
 type
  prt_array = array [1..8] of byte;
  col = integer;
  col_chk = record
              value: byte;
              col_num: col;
            end;
  key_code = record
               chr_strng: string[3];
               index: integer;
             end;
 var
  data_out: array [1..4] of col_chk;
  key_pressed, finished: boolean;
  i,ctr,row_x,col_y,num: integer;
  indata: byte;
  keys: array [1..6,1..4] of key_code;
  prev_key,prst_key: string[3];
```

```
    x_array: array [1..192] of byte;
    prt_strng: prt_array;
    start: integer;

  {"═══════════════════════════════════════════════════════"
   " INIT_KEY_STRING                                       "
   " Parameters Passed= none                               "
   " Called By        = Main Program                       "
   " Function         = initializes the matrix of          "
   "                    character strings that correspond  "
   "                    to the keys on the keypad card.    "
   "═══════════════════════════════════════════════════════"}
    procedure INIT_KEY_STRING;
     var
      j,k,l: integer;
     begin
      keys[1,1].chr_strng:='0';keys[1,2].chr_strng:='1';
      keys[1,3].chr_strng:='2';keys[1,4].chr_strng:='3';
      keys[2,1].chr_strng:='4';keys[2,2].chr_strng:='5';
      keys[2,3].chr_strng:='6';keys[2,4].chr_strng:='7';
      keys[3,1].chr_strng:='8';keys[3,2].chr_strng:='9';
      keys[3,3].chr_strng:='A';keys[3,4].chr_strng:='B';
      keys[4,1].chr_strng:='C';keys[4,2].chr_strng:='D';
      keys[4,3].chr_strng:='E';keys[4,4].chr_strng:='F';
      keys[5,1].chr_strng:='EDT';keys[5,2].chr_strng:='BPT';
      keys[5,3].chr_strng:='XAD';keys[5,4].chr_strng:='STR';
```

```
    keys[6,1].chr_strng:='C';keys[6,2].chr_strng:='CNT';
    keys[6,3].chr_strng:='MOV';keys[6,4].chr_strng:='AJA';
    l:=1;
    for j:=1 to 6 do begin
     for k:= 1 to 4 do begin
      keys[j,k].index:=l;  l:=l+8;
     end;
    end;
   end;

{"=============================================================="
 " INIT_DISPLAY_ARRAY                                          "
 " Parameters Passed= none                                     "
 " Called By        = Main Program                             "
 " Function         = initializes the 8-byte strings that      "
 "                    are to be displayed on the keypad        "
 "                    card display for each key pressed.       "
 "=============================================================="}
  procedure INIT_DISPLAY_ARRAY;
   var k: integer;
   begin
    for k:=1 to 192 do x_array[k]:= $AA;
    for k:=0 to 9 do x_array[5+k*8]:= $B0+k;
    for k:=0 to 5 do x_array[85+k*8]:= $C1+k;
    x_array[131]:=$C5;x_array[132]:=$C4;
    x_array[133]:=$C9;x_array[134]:=$D4;
```

```
    x_array[139]:=$C2;x_array[140]:=$CB;
    x_array[141]:=$D0;x_array[142]:=$D4;
    x_array[147]:=$D8;x_array[148]:=$C1;
    x_array[149]:=$C4;x_array[150]:=$D2;
    x_array[155]:=$D3;x_array[156]:=$D4;
    x_array[157]:=$CF;x_array[158]:=$D2;
    x_array[163]:=$D3;x_array[164]:=$C8;
    x_array[165]:=$C6;x_array[166]:=$D4;
    x_array[171]:=$C3;x_array[172]:=$CF;
    x_array[173]:=$CE;x_array[174]:=$D4;
    x_array[179]:=$CD;x_array[180]:=$CF;
    x_array[181]:=$D6;x_array[182]:=$C5;
    x_array[187]:=$C1;x_array[188]:=$CA;
    x_array[189]:=$C1;x_array[190]:=$BF;
  end;

{"═══════════════════════════════════════════════════"
 " INIT_DATA_ARRAY                                   "
 " Parameters Passed= none                          "
 " Called By        = Main Program                  "
 " Function         = initializes the array of bytes that"
 "                    are to be put out on the port D0 to"
 "                    check the 4 columns sequentially.  "
 "═══════════════════════════════════════════════════"}

  procedure INIT_DATA_VAL;
   begin
```

```
    data_out[1].value:=$FE;data_out[1].col_num:= 1;
    data_out[2].value:=$FD;data_out[2].col_num:= 2;
    data_out[3].value:=$FB;data_out[3].col_num:= 3;
    data_out[4].value:=$F7;data_out[4].col_num:= 4;
   end;

{"═══════════════════════════════════════════════════════"
 " PREPARE_LINK_FOR_IO                                   "
 " Parameters Passed= none                               "
 " Called By        = Main Program                       "
 " Function         = establishes BUSLINK by asserting   "
 "                    BUSRQ and BUSAK and also activates "
 "                    IOEXP and IOREQ controls.          "
 "═══════════════════════════════════════════════════════"}
  procedure PREPARE_LINK_FOR_IO;
   begin
    port[$305]:= $80; port[$304]:= $65;
   end;

{"═══════════════════════════════════════════════════════"
 " CHK_ROW_COL                                           "
 " Parameters Passed= indata = the data read in off port D0"
 " Called By        = READ_AND_ECHO                      "
 " Function         = finds out which row had key pressed. "
 "═══════════════════════════════════════════════════════"}
  procedure CHK_ROW_COL(var indata: byte);
```

```
   begin
    while indata > $3F do indata:= indata - 64;
    case indata of
     0: begin col_y:= data_out[ i].col_num; num:= num+1; end;
     1: row_x:= 1;
     2: row_x:= 2;
     4: row_x:= 3;
     8: row_x:= 4;
     16:row_x:= 5;
     32:row_x:= 6;
    end;
   end;

{"=============================================================="
 " READ_AND_ECHO                                               "
 " Parameters Passed= none                                     "
 " Called By        = Main Program                             "
 " Function         = activates the 4 columns sequentially"
 "                    and checks the input to see if any  "
 "                    key was pressed. It echoes the key  "
 "                    pressed to the PC screen by refering"
 "                    to the matrix made                       "
 "=============================================================="}
  procedure READ_AND_ECHO;
   begin
    key_pressed:= false; port[$302]:= $D0;
```

```
    while not key_pressed do
     begin
      num:= 0;
      for i:=1 to 4 do
       begin
        port[$301]:= data_out[i].value;  indata:= port[$301];
        CHK_ROW_COL(indata);
       end;
      if num=1 then begin
       key_pressed:=true;  prev_key:=prst_key;
       prst_key:=keys[row_x,col_y].chr_strng;
      end
      else prst_key:='x';
     end;
    if prev_key <> prst_key then begin
                        writeln(prst_key);writeln;end;
   end;

{"=============================================================="
 " DISPLAY_KEY                                                  "
 " Parameters Passed= prt_strng = array of ASCII codes for"
 "                                characters to be        "
 "                                displayed on keypad card"
 " Called By        = Main Program                           "
 " Function         = displays the string of 8 characters    "
 "                    on the keypad card on the STD Bus      "
```

```
"═══════════════════════════════════════════════════════"}

 procedure DISPLAY_KEY(prt_strng: prt_array);
  var k: integer;
  begin
   port[$305]:= $80; port[$304]:= $65;
   for k:= 1 to 8 do begin
    port[$302]:= $D0; port[$301]:= prt_strng[k];
    port[$302]:= $D1; port[$301]:= ($08-k);
    port[$301]:= ($10-k); port[$301]:= ($08-k);
   end;
  end;

{═══════════════════════════════════════════════════════
|              M A I N     P R O G R A M              |
═══════════════════════════════════════════════════════}

 begin
  INIT_DATA_VAL; PREPARE_LINK_FOR_IO;
  INIT_KEY_STRING; INIT_DISPLAY_ARRAY;
  finished:= false; prst_key:='x';
  writeln('This program will echo the key hit on
                                          the STD bus.');
  writeln('Hitting ''0'' on the keypad terminates
                                 program.');writeln;writeln;
  while not finished do
   begin
    READ_AND_ECHO;
```

```
  if prev_key <> prst_key then begin
   start:= keys[row_x,col_y].index;
   for ctr:=start to (start+7) do
                  prt_strng[ctr-start+1]:= x_array[ctr];
   DISPLAY_KEY(prt_strng);
  end;
  if keys[row_x,col_y].chr_strng='0' then finished:= true;
 end;
end.
```

APPENDIX C. DMA DEMONSTRATION

In multiprocessing environments, transfer of information through DMA is of particular interest because this mode of transfer takes place totally in the background. In the foreground, s system could be executing some function while the DMA controller in the system has been programmed to expect certain values. The DMA controller knows the number of data units expected and the address in memory at which the data is to be kept; once programmed with these values, the controller can receive incoming data independent of system control and store it as directed.

C.1 PROGRAM DESCRIPTION

The STD system runs a program that simply sends off a string of eight characters to the mailbox for the PC. This action is initiated by the operator by flipping the left rocker switch on the STD keypad card. Two sets of characters are stored in the STD memory and they are alternated between, by the STD system; this is done in order to see the different data set coming in.

The PC routine simply programs the DMA controller to receive the data and goes into a loop where it checks for a key to be hit by the operator; the striking of a key terminates the program - pulls the PC routine out of the loop. The loop is, essentially, the foreground function here.

In order to see the data being DMA'ed in, the memory portion chosen for the data destination is the PC screen display area.

C.1.1 The Color Graphics Screen

The color graphics screen on the PC uses a memory block starting from B800H:0000H. There are 80 columns in the screen; so, a row can take 80 characters. Each character is represented by a byte having the ASCII code of the character and a byte with the color code for the background color - total 2 bytes. So, there are 160 bytes of memory for each row - each of the character positions taking up two consequetive bytes.

C.1.2 STD Program Function

Although there are only 8 characters to be transferred to the PC, we have to transfer 16 bytes because the PC needs the color bytes too. The way the alternate sets of data were sent is by keeping a counter byte which increments every time the flipping of the switch is detected. Then the lsb of the byte is checked - if it is 0 then one set is sent and if it is 1 then the other set is sent. This should give us an alternating set of data; there is a special point to be noted here - see "Keypad Card Switch Bouncing" on page 93 for explanation.

The Keypad card on the STD system is used to display the data being sent out also. The steps needed to do this is explained in "Appendix B. Controlling the STD Bus from the PC" on page 72. We need to note one point in this connection here. The code for the characters A-Z that displays them on the keypad card is the ASCII code of the characters added to 128 (80H). So, after the data byte is sent off to the mailbox, the program adds 128 to the byte and then sends it to the display card. Also, the

counter byte mentioned above is sent to the keypad card LED bank so that the operator can keep a check on that too.

The program runs through the following steps :

1. Wait for the switch to be flipped up
2. Increment counter register
3. Check lsb of counter register and choose data set accordingly - the start address of the chosen data set is put in the HL register pair.
4. Send data to mailbox
5. Add 128 to data
6. Send data to keypad display
7. Send color byte (65 = red) to mailbox
8. Increment address registers (HL)
9. Repeat steps 4 to 8, 8 times
10. Display counter register at keypad card LED bank
11. Wait for switch to be flipped down
12. Repeat all steps continuously

The STD system keeps sending data to the PC and shows the data set that was sent, on the keypad display.

C.1.3 STD Program Listing

```
2500 A.D. 8085 CROSS ASSEMBLER   -   VERSION 3.01a
-------------------------------------------------
```

```
                INPUT  FILENAME:   DMASTD.ASM

                OUTPUT FILENAME:   DMASTD.OBJ



    1                                   This program runs on the
8085.  It needs the program DMAPC.BAS running on the
    2                                   PC.  What it does is, it
transfers, thru DMA, the data "*DIPTISH" and
    3                                   "COMPUTER", alternately
with switching the keypad switch.  DMAPC sets up the
    4                                   DMA controller to receive
the set of 8 data bits with the adjacent 8 color
    5                                   bits (total 16 bits) into
the display memory.
    6                                   So, effectively, yu can see
the data being DMA-ed directly on the PC screen!
    7
    8   8000                              ORG 8000H
    9   8000   2A 44 49 50               DB '*DIPTISH'
        8004   54 49 53 48
   10   8200                              ORG 8200H
   11   8200   43 4F 4D 50               DB 'COMPUTER'
        8204   55 54 45 52
   12         00 00                 SWCHDN  EQU  00H
   13         80 00                 SWCHUP  EQU  80H
```

```
14          00 00               ZERO    EQU  00H
15          65 00               RED     EQU  65H
16          D0 00               LED     EQU  0D0H
17          D0 00               SWCH    EQU  0D0H
18          D1 00               DSPLY   EQU  0D1H
19          FF 00               LEDON   EQU  0FFH
20          08 00               DATA    EQU  8
21          01 00               DMASTAT EQU  1
22          20 00               DMATST  EQU  20H
23          00 00               DMAPRT  EQU  00H
24
25   8500                             ORG 8500H
26   8500   1E 00                     MVI E,ZERO; LSB - REG-E
DECIDES WHICH SET OF DATA TO SEND
27
28   8502   3E 00              STR:  MVI A,ZERO
29   8504   D3 D0                    OUT LED
30   8506   DB D0                    IN  SWCH
31   8508   FE 80                    CPI SWCHUP
32   850A   C2 02 85                JNZ STR      ; LOOP
TILL SWITCH UP
33                                               ;
34   850D   1C                        INR E
35   850E   3E FF                     MVI A,LEDON
36   8510   D3 D0                    OUT LED      ; LEDON
IMPLIES START OF TRANSFER
```

```
   37   8512   7B                    MOV A,E
   38   8513   E6 01                 ANI 1
   39   8515   CA 1E 85              JZ  EVEN      ; CHOOSE
   40   8518   21 00 80              LXI H,8000H   ; DATA SET
   41   851B   C3 21 85              JMP BOO       ; USING
   42   851E   21 00 82        EVEN: LXI H,8200H   ; LSB (E)
   43                                             ;
   44   8521   06 08           BOO:  MVI B,DATA    ;# OF DATA
TO BE TRANSMITTED.
   45   8523   DB 01           LOOP: IN  DMASTAT
   46   8525   E6 20                 ANI DMATST
   47   8527   CA 23 85              JZ  LOOP      ; LOOP
WHILE MAILBOX NOT READY
   48                                             ;
   49   852A   7E                    MOV A,M
   50   852B   D3 00                 OUT DMAPRT    ; TO PC
   51                                             ;
   52   852D   C6 80                 ADI 128       ; ASCII
FOR KEYPAD DISPLAY
   53   852F   D3 D0                 OUT LED       ; SENT TO
   54   8531   78                    MOV A,B       ; POSITION
   55   8532   3D                    DCR A         ; SELECTED
   56   8533   D3 D1                 OUT DSPLY     ; POSITION
OUTPUT, BUT WITH MASK ON.
   57   8535   C6 08                 ADI 8         ; MASK OFF
   58   8537   D3 D1                 OUT DSPLY     ;DISPLAYED
```

```
   59   8539   D6 08                    SUI 8        ; "SEALED"
   60   853B   D3 D1                    OUT DSPLY  ;BY MASKING.
   61                                              ;
   62   853D   23                       INX H
   63   853E   DB 01              LOOP1
  DMASTAT
   64   8540   E6 20                    ANI DMATST
   65   8542   CA 3E 85                 JZ  LOOP1
   66   8545   3E 65                    MVI A,RED    ;COLOR BIT
   67   8547   D3 00                    OUT DMAPRT   ;SENT OUT.
   68                                              ;
   69   8549   05                       DCR B
   70   854A   C2 23 85                 JNZ LOOP     ; CONTINUE
TILL SET COMPLETED.
   71                                              ;
   72   854D   7B                 ASS:  MOV A,E
   73   854E   D3 D0                    OUT LED
   74   8550   DB D0                    IN  SWCH
   75   8552   FE 00                    CPI ZERO
   76   8554   C2 4D 85                 JNZ ASS      ; WAIT IN
LOOP TILL SWITCH IS FLIPPED DOWN.
   77   8557   C3 02 85                 JMP STR      ; REPEAT.
   78                                              ;
   79   855A                            END

  LINES ASSEMBLED:     79          ASSEMBLY ERRORS:     0
```

C.1.4 PC Program Function

The PC chooses the 11th row, 40th column to start the display. So, the 8 characters are displayed in columns 40 to 47. Each row uses 160 bytes of memory; 10 rows use 10*160 bytes. 39 column positions use 78 bytes; so, the 40th column position starts with the 79th byte. Therefore, the starting location in memory to reach the 11th row and 40th column is (78+(10*160)) = 1678 = 068EH in the segment B800H = B800H:068EH. This gives the physical address of B868EH.

The DMA controller current address register was loaded with the address 868EH and the segment register had B. This explains how to decide on the destination of the DMA data and program the 8237 with it. After programming the DMA controller address, the mode is set to single transfer mode with autoinitialize. When the Channel-1 mask is removed, it is ready for transfer. There is an irritating hatching seen on the screen due to the refresh action of the 8237 which also works on the display memory area that we are using here. As the switch is flicked up on the STD keypad card, the data set (*DIPTISH or COMPUTER) is sent over and can be seen on the keypad display and on the PC screen. The PC program is written in BASIC.

C.1.5 PC Program Listing

```
1 'This program receives data from the STD system over
the link through THE DMA
2 'controller. While the program loops (in statement
no. 170) waiting for the user to
3 'terminate program by hitting a key the DMA
controller injects data coming in
4 'at specified display memory area. So, the data
is echoed immediately to the
5 'screen at the 11th row, starting from the 40th col.
6 '
10 DEF SEG=&HB800          'defines the PC segment to color
screen buffer
20 DMALO = &H8E
30 DMAHI = &H86            '868EH is the for row 11, col 40.
40 OUT &HA,5               'set DMA channel-1 mask
50 OUT &HC,0               '
60 OUT &H2,DMALO           'the address of the 11th row,
70 OUT &H2,DMAHI           '40th col in Curr Addr Reg.
80 OUT &H83,&HB            'DMA controller page register
90 OUT &H3,15              'the Current Word Count Register
100 OUT &H3,0              '   has 15 (16 bytes to be taken)
110 OUT &HB,&H55           'DMA controller Mode
Register (55H = Single txfer, autoinit)
120 OUT &HA,1              'enable DMA Channel-1
```

```
130 OUT &H305,&HA0        'STD bus srvice port - sysrun &
enable STD-mail.
140 OUT &H307,&H8  : CLS  'PC link DREQ enable.
150 FOR MA=0 TO 7
155 POKE (80-1+2*MA)+10*160,65
160 NEXT                  'the color bytes of the 8
locations chosen are made red
170 A$=INKEY$: IF A$="" THEN 170  ' loop till key is
pressed - DMAed data is seen
180 OUT &H307,0           'disable DMA
190 OUT &HA,5             'disable Channel-1
200 OUT &H83,0            'reset controller page register.
210 STOP
```

C.2 KEYPAD CARD SWITCH BOUNCING

This is a feature of the switch on the STD keypad card. The switch bounces and has several transitions between 1 and 0 before it settles to the value it is switched to. Debouncing software that introduce a small delay can easily be written to get rid of the intermediate values. This was not done in the STD program used here and that helps demonstrate just how fast the DMA controller can receive data. The LED bank on the keypad card shows the counter and it is seen that the counter counts up 2 or 3 times for every flick of the switch. On the PC screen and on the keypad display, the transitions are not seen; all that can be seen is the tran-

sition from one data set to another. But actually, there are 2 or 3 changes of data that are passed in to the PC!

APPENDIX D. MAIN APPLICATION - FILTER MONITOR

This application was designed to use the various communication methods and show the feasibility of multiprocessing on the PC-STD system. The different modes of data transfer have been used to an advantage in the different situations; an attempt has been made to show the kind of application or situation in which a particular transfer protocol would be best used.

D.1 THE DIGITAL FILTER

Several ways of implementing a digital filter are encountered in the Digital Controls texts. The one chosen for this application has the advantage of simplicity and flexibility. Simplicity is in the form of the equation used, making implementation easy; flexibility is in the filter to be represented - the coefficients can be altered to give us a first order filter, a second order filter or simply a gain. The filter equation is taken from the Controls book by Houpis and Lamont [11]. The equation is the following -

C(K) = A2*R(K-2) + A1*R(K-1) + A0*R(K) + B2*C(K-2) + B1*C(K-1)

where the R(K)'s are the past and current inputs and the C(K)'s are the past outputs of the filter.

D.1.1 Z-transform of the filter

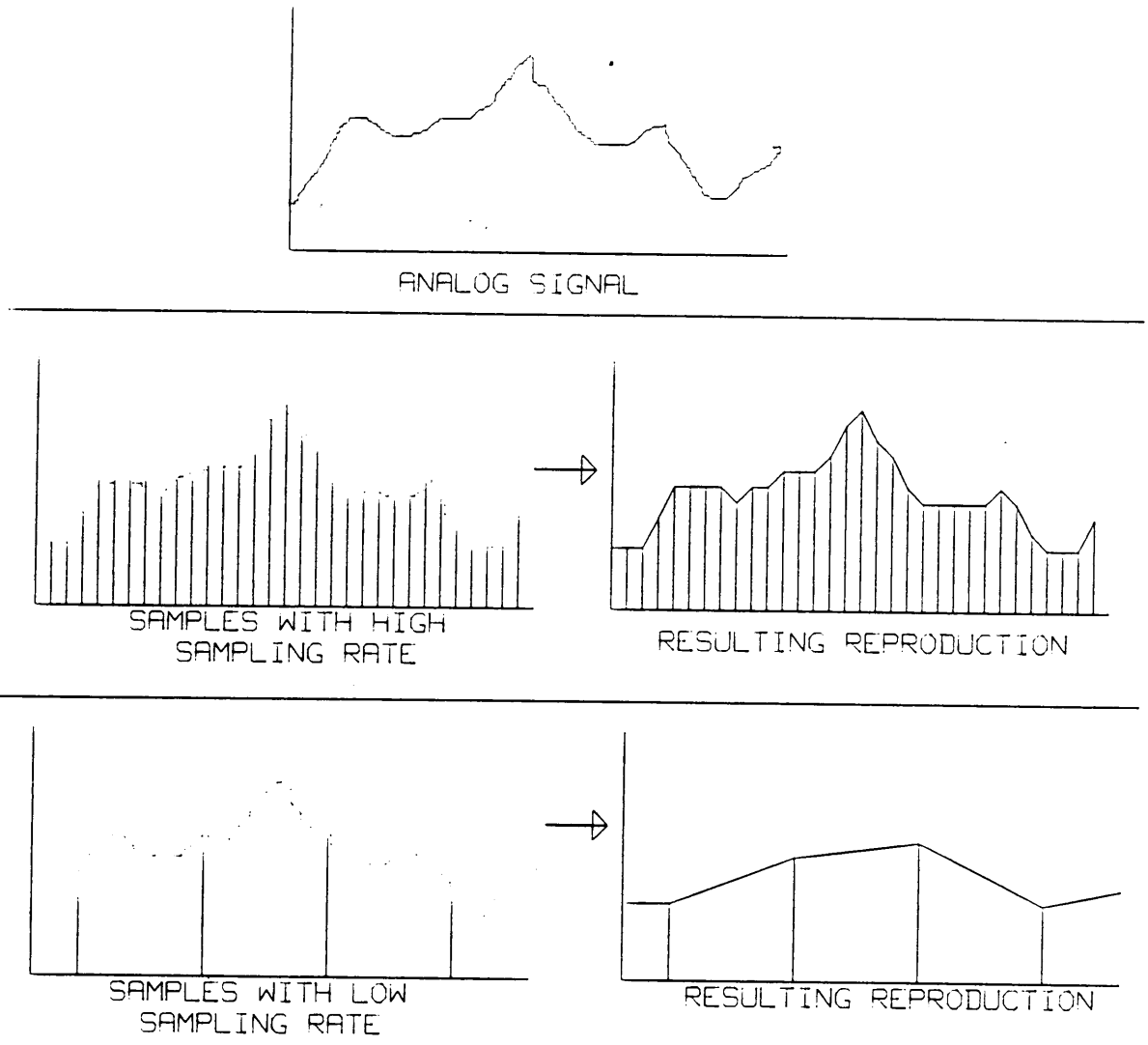

Figure 7. Sampling an analog signal: An analog signal sampled at a regular interval gives a series of digital levels that represent the original wave. Higher the frequency of sampling, the better the representation.

For digital systems the basic factor is that the processing is done on discrete digital values. A digital system is never absolutely continuous. An analog or continuous variation can be "digitized" by sampling the analog signal. The closer the samples, the better is the representation of the original wave by the digital sequence; see Figure 7 on page 96. So, the sampling of the analog wave has to be done at a reasonably high rate in order to get a satisfactory representation of the original wave.

Z-transforms are used to design and analyze sampled-data control systems. The Z-transform of the filter equation given above is the following -

$$C(z) = A2.R(z).z^{-2} + A1.R(z).z^{-1} + A0.R(z) + B2.C(z).z^{-2} + B1.C(z).z^{-1}$$

This can be interpreted in words as - "The current output value is given by the sum of 1)A2 * the input sample before the last one 2)A1 * the last input sample 3)A0 * the present input sample 4)B2 * the output value before the last one and 5)B1 * the previous output value." Going to the next step, we can deduce the transfer function that this equation represents as -

$$\frac{C(z)}{R(z)} = \frac{A2*z^{-2} + A1*z^{-1} + A0}{1 - B1*z^{-1} - B2*z^{-2}}$$

In order to confirm that this does represent a filter we can use the fact that $z = e^{+sT}$ to get the equation in s-domain.[10] The transfer function in the s-domain is :

$$\frac{C(s)}{R(s)} = \frac{A2*e^{-2sT} + A1*e^{-sT} + A0}{1 - B1*e^{-sT} - B2*e^{-2sT}}$$

Let us assume that A0 = P; A1 = A2 = 0; B1 = K and B2 = 0. Then we have

$$\frac{C(s)}{R(s)} = \frac{P}{1 - K*e^{-sT}}$$

It is clear that as the frequency of the input R(s) increases, the denominator of the transfer function increases and thus, the output C(s) decreases. This is a first order low-pass filter.

In order to see how the filter works, we can use the Z-transform of the R(z) function - which is a Sine wave in our case - and we can perform long division to find the infinite series of coefficients for C(z). The sampling frequency in our application was 125Hz; so, the time interval T between samples is 1/125. Response to a wave with amplitude 40 and frequency 0.2 was determined. The coefficients are the consecutive output levels expected from the filter. The process is shown in Figure 8 on page 99.

---

[10] s = Laplace Transform.

	A0	A1	A2	B1	B2
FILTER CONSTANTS	0.20	0.00	0.00	0.90	0.00

	FREQUENCY	AMPLITUDE
INPUT SIGNAL	5 Hz	40

$$C(K) = \frac{0.2z}{z - 0.9} \quad \frac{A.z\,Sin\omega T}{z^2 - 2z\,Cos\omega T + 1}$$

[$\omega = 2 \times 3.14 \times 5$
T = sampling interval
$= 1/125$]

$$= (0.2 \times 40 \times 0.251) \frac{z}{z - 0.9} \quad \frac{z}{z^2 - 1.937z + 1}$$

BY LONG DIVISION - WITHOUT THE CONSTANT TERM :

$$\frac{z^2}{z^3 - 2.837z^2 + 2.743z - 0.9} = z^{-1} + 2.837z^{-2} + 5.306z^{-3} + 8.171z^{-4}$$

$$+ 11.18z^{-5} + 14.08z^{-6} + 16.63z^{-7} + 18.63z^{-8} + 19.89z^{-9} + 20.31z^{-10}$$

$$+ 19.83z^{-11} + 18.43z^{-12} + 16.19z^{-13} + 13.22z^{-14} + 9.68z^{-15} + 5.77z^{-16}$$

$$+ 1.72z^{-17} - 2.24z^{-18} - 5.88z^{-19} \ldots$$

Figure 8. Hand Calculation for the Filter

The filter can thus be changed by looking at the backwards differential equation and putting in the coefficients accordingly. If the sampling rate is high enough, the output should depict the performance of the analog filter with same characteristics.

D.1.2 Program Implementation Tools

The poles of the transfer function in the Z-domain have to be within the unit circle in the Z-plane - i.e., the real and imaginary parts of the poles have to be less than 1 in magnitude. This directly tells us that decimal arithmetic has to be done for the processing. There are several algorithms for decimal arithmetic that can be used. They usually use exponential representation of the numbers and the time taken is also quite long. The range of the numbers that can be represented is obviously very large in such cases. In this application the **range of the numbers was limited to -255.99 to +255.99.**

The functions necessary to carry out the filter iterations are only "+", "-" and "*" - addition, subtraction and multiplication. The way in which this was implemented is explained in the following sections.

D.1.2.1 Number Representation

A number was represented in the program by three bytes :

- integer byte
- decimal byte

- sign flag byte

The range of the integer part of the number is obviously determined by the 8-bit boundary of the byte. So, the integer part can be upto 255. The decimal part is limited to 99 because it is the decimal part and a resolution of 0.01 was deemed enough for our purpose. Besides, the result gets rounded off to an integer value anyway, so any additional accuracy would not be useful. The sign flag has only two possible states - 0 indicating "+" and 80H indicating "-".

D.1.2.2 Multiplication of two numbers

In this application the only multiplication that has to be done is that of an integer number - the input sample from the A/D is an integer, with a decimal value - the filter constants are decimal numbers and have an integer and a decimal part. The routine MULT (see listing) does this multiplication. The input value, R(K) or C(K), is a signed integer of one byte - the most significant bit being the sign bit and two's complement being used to represent the number. The filter constant it is to be multiplied with is in the format given in "Number Representation" on page 100.

The flowchart of the multiplication procedure is given in Figure 9 on page 102. The algorithm is quite obvious from the listing and the flowchart. The basic multiply routine (MUPLY) and divide routine (DIV) are used; they are elementary routines that work on integers and use the shift commands to accomplish fast multiplication and division. The basic

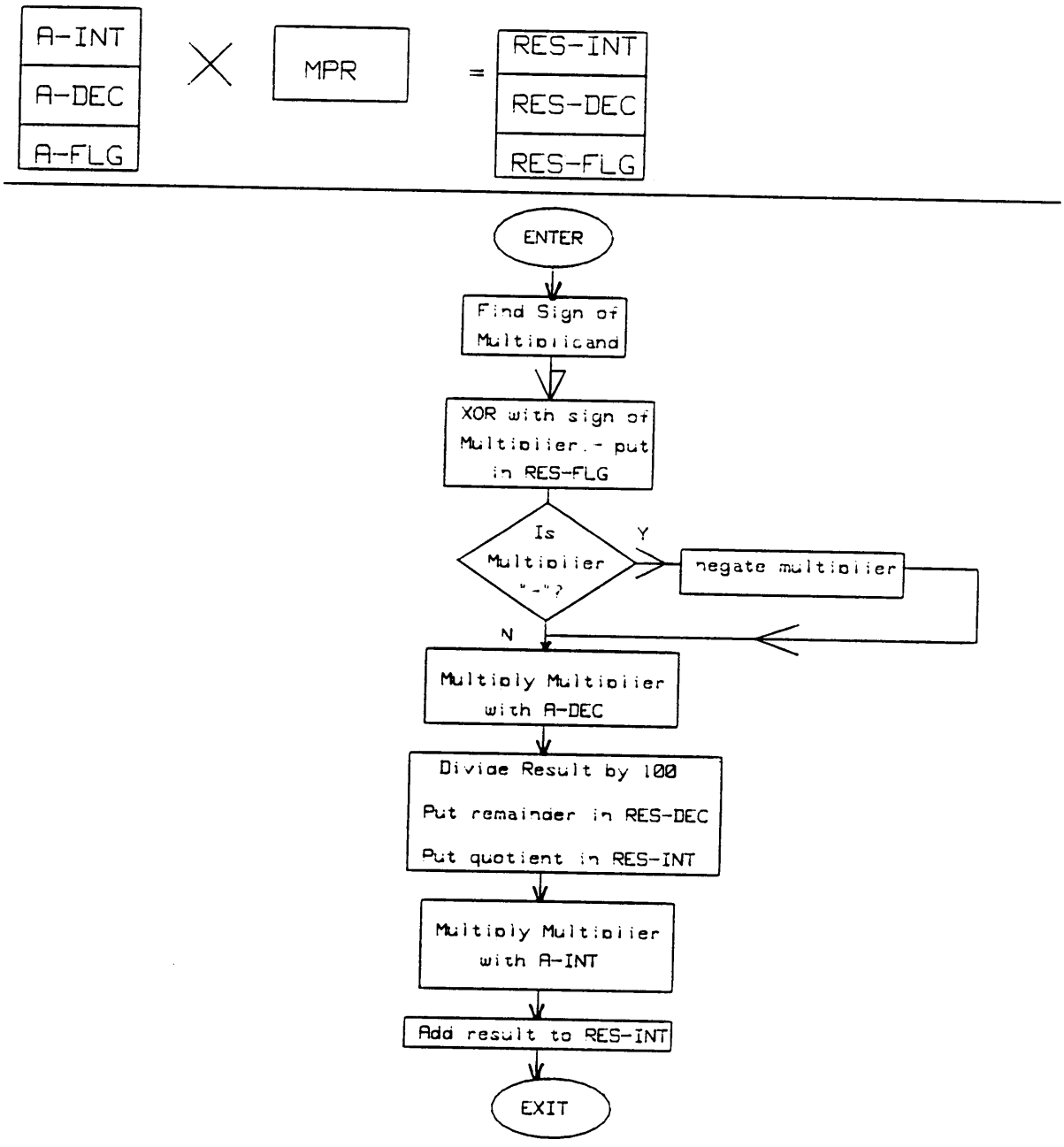

Figure 9. Flowchart for Multiplication

principle employed here is as follows : The sign of the result is decided by the exclusive OR of the signs of the two multiplicands. The magnitude of the integer multiplicand is multiplied with the decimal part of the other multiplicand first. The result of this multiplication is divided by 100 to get the decimal part of the result and the "carry" to the integer part. Then the integer multiplicand is multiplied with the integer part of the other input and the result is added to the "carry" to get the final integer part of the result.
Refer to Figure 10 on page 104 for a diagram of the logic explained.

D.1.2.3 Summing the numbers

The ADDER routine takes care of addition. It adds two decimal numbers that are in the format given in "Number Representation" on page 100 and stores the result in the same format. It is used to add the result of each multiplication operation to the total (and thus forms the output value after 5 runs). There are two operations that might can be resorted to in order to sum the numbers - one when both numbers are the same sign, and the other when they are different sign.

Same Sign: If the numbers are same sign - either both positive or both negative, then the operation is addition of the numbers; the sign of the result is same as the sign of the values being added. For this, the decimal parts are first added; if the result is greater than 99 then 100 is subtracted repeatedly from it till the result is less than or equal to

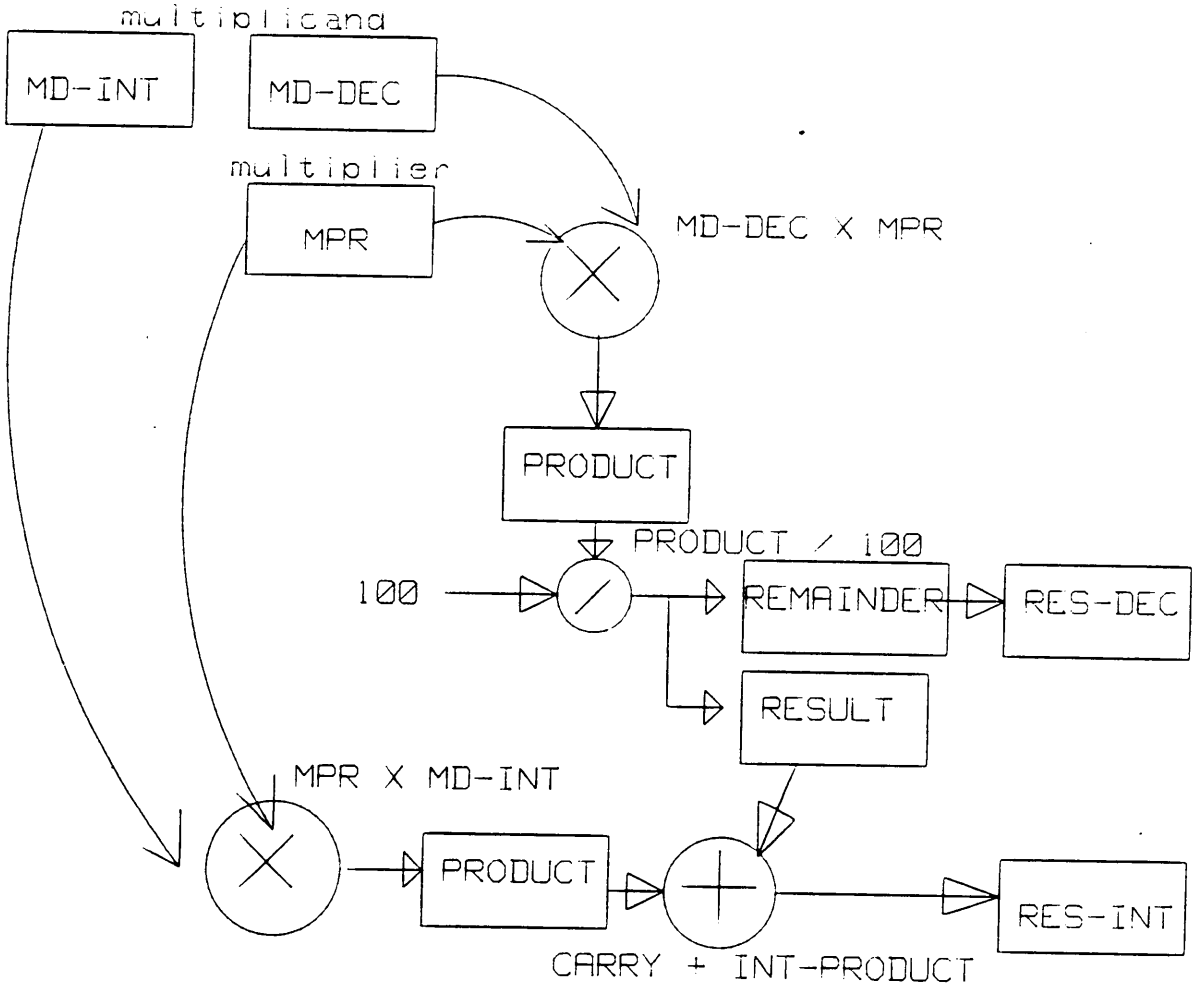

Figure 10. Multiplication Logic.

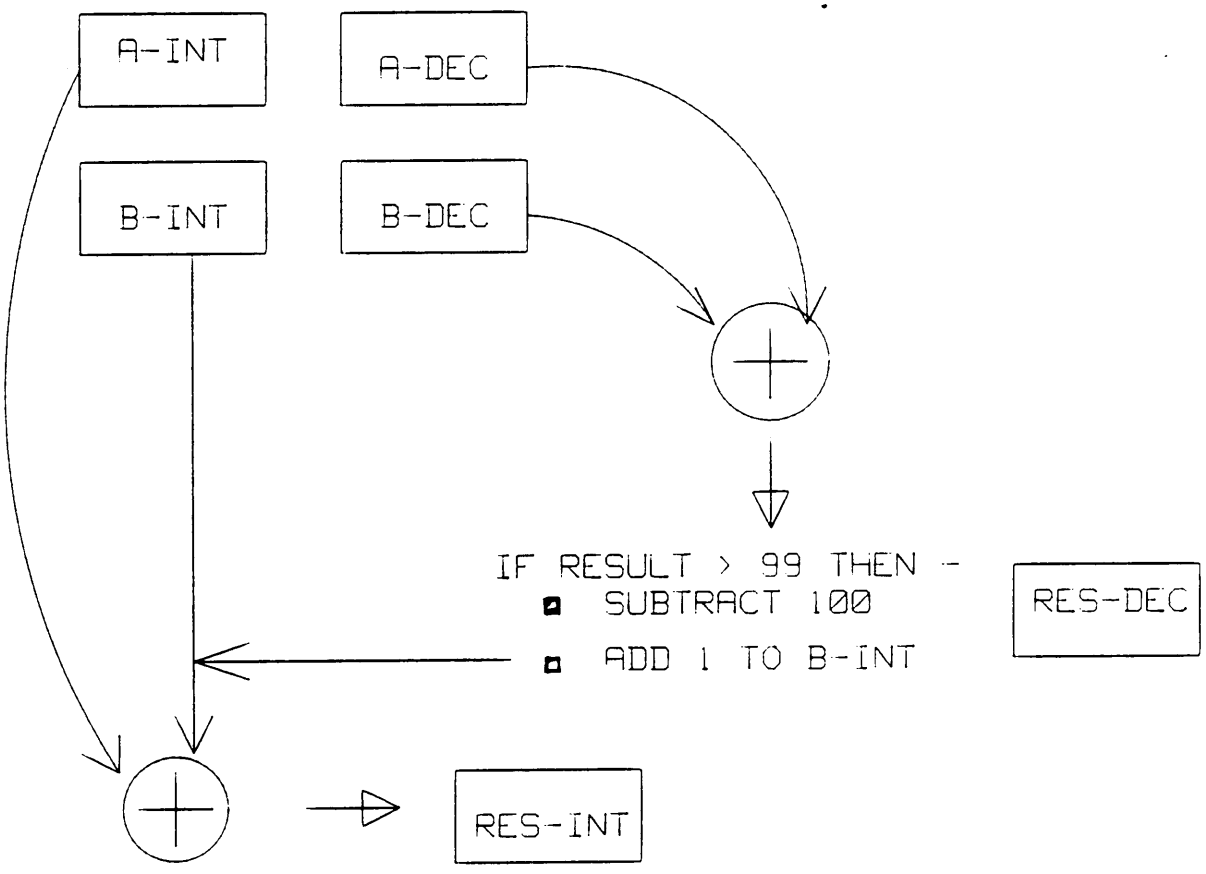

Figure 11. Summing two numbers with same sign.

99. The number of times 100 had to be subtracted, is added with the integer parts of the numbers to get the integer part of the result.
Refer to Figure 11 on page 105 for the flowchart of this operation.

Different Signs: This poses considerably more complications and situations.
The sign of the result is first decided by the following operations :

1. Check which of the numbers is the one with negative sign; assuming A and B to be the numbers, suppose A is negative.
2. Check which of the numbers is greater; there can be two cases - CASE 1 : A > B and the other CASE 2 : A < B.
3. Set the sign of the result - if the number with the negative sign is greater than the other then the result is negative, and vice versa; for CASE 1 sign of result is "-" and for CASE 2 sign is "+".

Subtraction is then done so that the result is positive; in the cases taken for example, the program would do (A - B) for CASE 1 and (B - A) for CASE 2. Refer to Figure 12 on page 107 for flowchart. The decimal bytes are first subtracted; if the result is negative then 100 is added to the result and 1 is added to the integer byte of the number that is being subtracted. This is nothing but the basic "carry" principle in subtractions. Then the integer part of the result is found by subtracting the integer parts (the result **has** to be greater than or equal to zero).

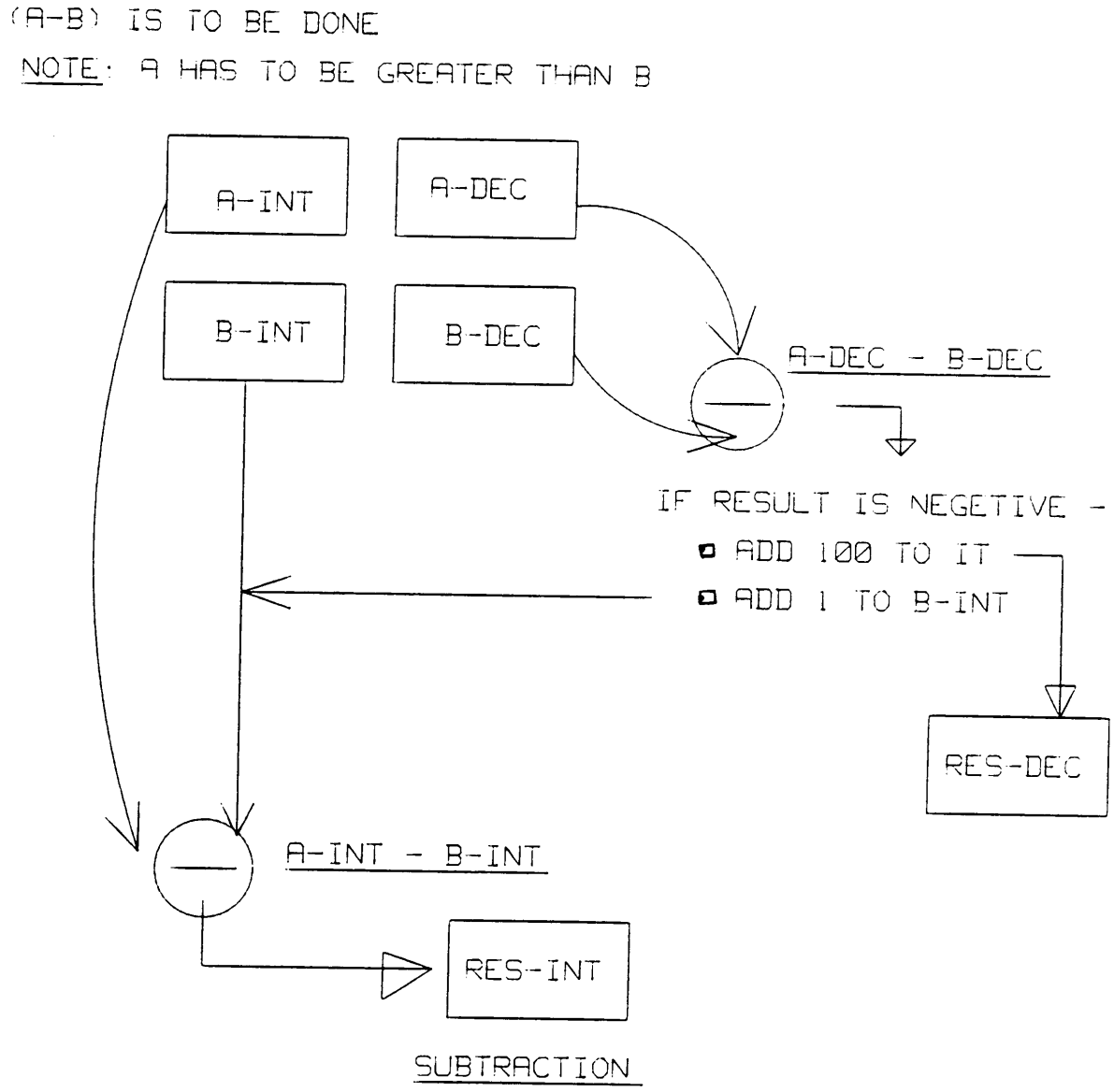

Figure 12. Summing Two Numbers With Different Signs.

D.1.3 Iteration Logic

As explained in "Z-transform of the filter" on page 95, every output level has to be computed by using the present input level - R(K), past inputs - R(K-1) and R(K-2), and past output levels - C(K-1) and C(K-2). The past values to be used for computing the present output, are initialized to zero in the beginning. Then, as the input values are sampled at every time step, the output is computed and the input and output values are stored in the locations reserved for them - to be used in the next iteration.

Refer to Figure 13 on page 109 for the steps involved in the iteration. The STD system keeps computing the iteration on data that it receives either from the A/D converter or from the PC generated data array. It makes decisions on the status byte that is loaded and controlled by the system master - the PC. It checks the new_info bit of the status byte after every iteration; if it finds that the PC has interrupted the system to change status, then it reinitializes the iteration bases and starts processing again. The routing of the output and the source of the input values are now changed as the PC has loaded into the status byte. The bit assignment for the status byte is also given in Figure 13 on page 109.

D.1.4 Digital Filter Results

The Digital Filter program computed the values very accurately - atleast as accurately as rounding off to byte values allows. The output

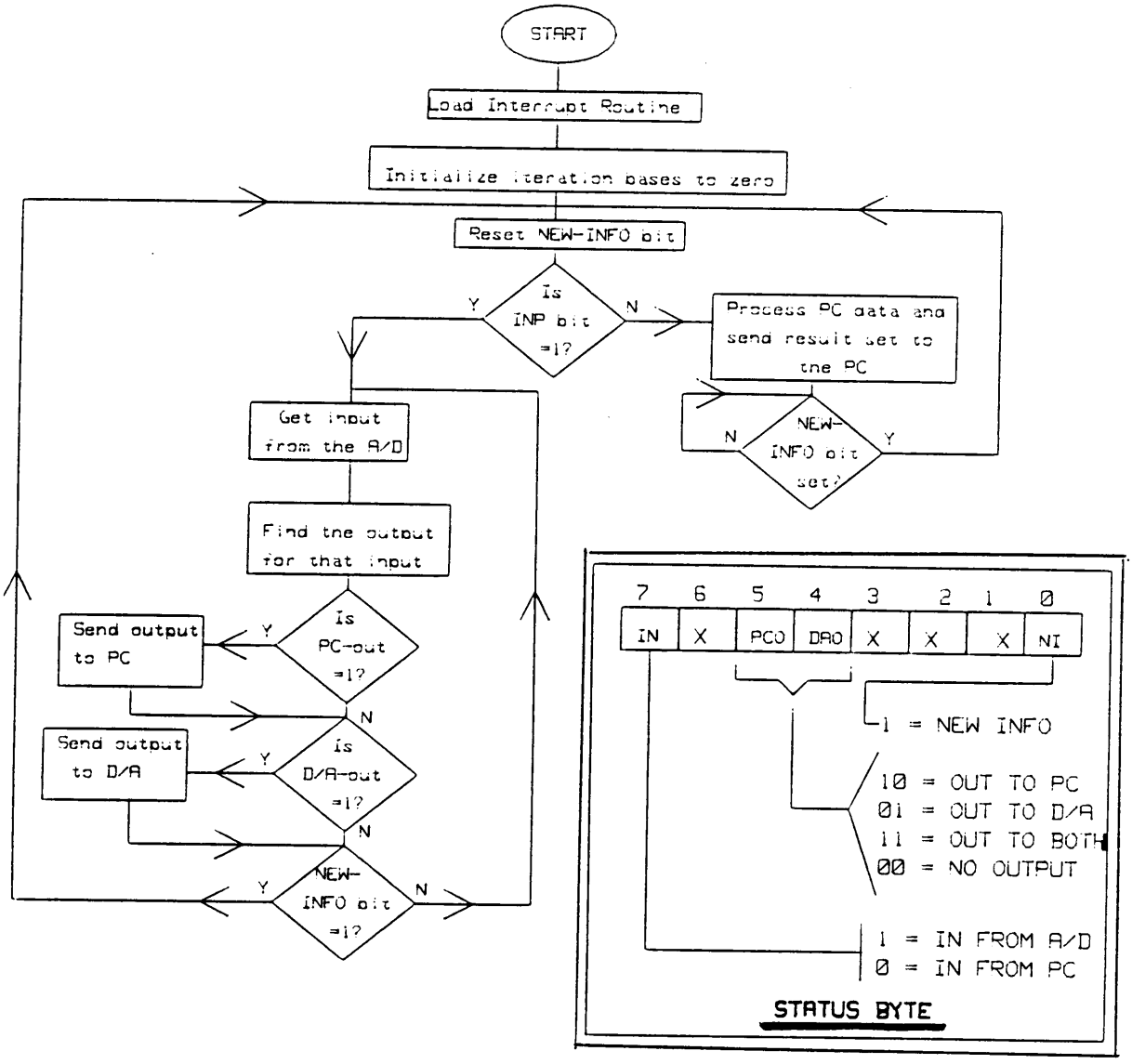

Figure 13. The Overall Program Flow for the STD system.: The general flow of the program running on the STD CPU is shown. Also, the Status byte is explained.

values computed, when routed to the PC, can easily be printed out (instead of plotting them) and can be examined for correctness. We can compare the results of the program with the values computed by hand using the iteration and also using the long division method of finding the consecutive output - C(K) - values as demonstrated in Figure 8 on page 99. The same filter constants were used and the same input wave (frequency and amplitude same) was fed to it; the Z-txform of the filter used is :

$$C(z) \qquad 0.2\ z$$

$$R(z) \qquad z - 0.9$$

The input wave is a Sine wave with amplitude 40 and frequency 5Hz. The results are tabulated below.

Sample No. (K)	Input R(K)	Prev Output C(K-1)	Output C(K)	Output From Z txform	Output From Dig. Filter
0	0	0	0	0	0
1	10	0	2.0	2.0	2
2	19	2.0	5.6	5.6	6
3	27	5.6	10.4	10.6	11
4	34	10.4	16.2	16.2	16
5	38	16.2	22.2	22.2	22
6	40	22.2	28.0	28.0	28
7	39	28.0	33.0	33.2	33
8	36	33.0	36.9	37.2	37
9	31	36.9	39.4	39.8	39
10	24	39.4	40.3	40.6	40
11	15	40.3	39.2	39.6	39
12	5	39.2	36.3	36.8	36
13	-1	36.3	32.5	32.4	32
14	-15	32.5	26.3	26.4	26
15	-23	26.3	19.0	19.2	19
16	-30	19.0	11.1	11.54	11
17	-36	11.1	2.79	3.44	3
18	-38	2.79	-5.09	-4.48	-5
19	-40	-5.09	-12.5	-11.76	-12

D.2 PROGRAM LISTING (FILTR.ASM)

```
      2500 A.D. 8085 CROSS ASSEMBLER   -   VERSION 3.01a
      ---------------------------------------------------

                INPUT  FILENAME:   FILTR.ASM
                OUTPUT FILENAME:   FILTR.OBJ



    1
    2
    3   8000                                    ORG   8000H
    4           D0 00           LED     EQU  0D0H ; LED PORT
ON THE STD KEYPAD CARD
    5           D1 00           DISPOS  EQU  0D1H ; PORT TO
SET POSITION FOR DISPLAY ON STD CARD
    6           F9 00           ADSTRT  EQU  0F9H ; OUT
PORT TO START A/D CONVERSION
    7           F9 00           ADSTAT  EQU  0F9H ; STAT PORT
FOR A/D
    8           01 00           ADREDY  EQU  01H  ; READY BIT
ON STATUS PORT
    9           F8 00           ADDATA  EQU  0F8H ; A/D DATA
   10           F8 00           DAPORT  EQU  0F8H ; D/A PORT
```

```
   11          01 00          SFLAG   EQU  01H  ; STAT FLAG
   12          20 00          S2PST   EQU  20H  ; BIT-5 = 1
INDICATES STD CPU CAN WRITE MAIL DATA
   13          00 00          SMAIL   EQU  00H  ; STD MAIL
   14          DD 00          ENDMK   EQU  0DDH
   15
   16   8000   C3 0C 85                   JMP   MAIN
   17
   18   8003                  PCDATA: DS 500    ; ABSOLUTE
ADDRESS FROM 8003H TO 81F6H
   19
   20   81F7   81             STATUS: DB 81H    ; ABSOLUTE
ADDR 81F7H
   21
   22   81F8   00             A0INT:  DB 0  ;  ADDR 81F8H
   23   81F9   14             A0DEC:  DB 20 ;
   24   81FA   00             A0FLG:  DB 0  ;
   25   81FB   00             A1INT:  DB 0  ; |
   26   81FC   00             A1DEC:  DB 0  ; |
   27   81FD   00             A1FLG:  DB 0  ; |   FILTER
   28   81FE   00             A2INT:  DB 0  ; |   CONSTS
   29   81FF   00             A2DEC:  DB 0  ;  \__(DEFAULT
   30   8200   00             A2FLG:  DB 0  ;  /   AT STRT
   31   8201   00             B1INT:  DB 0  ; |    A0=0.2
   32   8202   5A             B1DEC:  DB 90 ; |    B1=0.9)
   33   8203   00             B1FLG:  DB 0  ; |
```

```
34   8204   00              B2INT:   DB 0  ; |
35   8205   00              B2DEC:   DB 0  ;
36   8206   00              B2FLG:   DB 0  ;
37
38   8207   00              RK0:     DB 0          ; R(K)
39   8208   00              RK1:     DB 0          ; R(K-1)
40   8209   00              RK2:     DB 0          ; R(K-2)
41   820A   00              CK1:     DB 0          ; C(K-1)
42   820B   00              CK2:     DB 0          ; C(K-2)
43
44   820C                   INTRES:  DS 1          ;
45   820D                   DECRES:  DS 1          ;RESULT
46   820E                   FLGRES:  DS 1         ;(OF MULT)
47
48   820F                   TOTINT:  DS 1          ;
49   8210                   TOTDEC:  DS 1          ;TOTAL(OF
50   8211                   TOTFLG:  DS 1          ;MULT
51                                                  RESULTS)
52   8212   C3 CF CD CD     COM:     DB 0C3H,0CFH,0CDH,0CDH,
0C1H,0CEH,0C4H,0BFH ; "COMMAND?"
       8216   C1 CE C4 BF
53   821A   D0 C3 A0 B2     P2P:     DB 0D0H,0C3H,0A0H,0B2H,
0A0H,0D0H,0C3H,0A0H ; "PC 2 PC "
       821E   A0 D0 C3 A0
54   8222   C1 C4 A0 B2     A2D:     DB 0C1H,0C4H,0A0H,0B2H,
0A0H,0C4H,0C1H,0A0H ; "AD 2 DA "
```

```
        8226   A0 C4 C1 A0
   55   822A   C1 C4 A0 B2      A2P:    DB 0C1H,0C4H,0A0H,0B2H,
0A0H,0D0H,0C3H,0A0H ; "AD 2 PC "
        822E   A0 D0 C3 A0
   56   8232   C1 C4 A0 B2      A2BTH:  DB 0C1H,0C4H,0A0H,0B2H,
0A0H,0C2H,0D4H,0C8H ; "AD 2 BTH"
        8236   A0 C2 D4 C8
   57   823A   C1 C4 A0 B2      A2X:    DB 0C1H,0C4H,0A0H,0B2H,
0A0H,0D8H,0D8H,0D8H ; "AD 2 XXX"
        823E   A0 D8 D8 D8
   58
   59
   60                                          ;ORG 2600H --
INTERRUPT SERVICE ROUTINE
   61   8242   F3          INTR: DB 0F3H       ; DI
   62   8243   F5             DB 0F5H          ; PUSH  PSW
   63   8244   C5             DB 0C5H          ; PUSH  B
   64   8245   D5             DB 0D5H          ; PUSH  D
   65   8246   E5             DB 0E5H          ; PUSH  H
   66   8247   DB 01          DB 0DBH,01H      ;IN1
  01H
   67   8249   E6 10          DB 0E6H,10H      ; ANI   10H
   68   824B   CA 05 26       DB 0CAH,05H,26H  ; JZ    IN1
   69   824E   DB 00          DB 0DBH,00H      ; IN    0
   70   8250   32 F7 81       DB 032H,0F7H,81H ; STA  STATUS
   71   8253   E1             DB 0E1H          ; POP   H
```

```
   72   8254   D1            DB 0D1H          ; POP   D
   73   8255   C1            DB 0C1H          ; POP   B
   74   8256   F1            DB 0F1H          ; POP   PSW
   75   8257   FB            DB 0FBH          ; EI
   76   8258   C9            DB 0C9H          ; RET
   77   8259   DD            DB 0DDH          ; 0DDH is
end marker.
   78
   79                              ------------------------
   80                               LODINT LOADS THE
INTERRUPT SERVICE ROUTINE AT 2600H
   81                               THIS IS NECESSARY
BECAUSE PC-STD LINK CAN'T AVAIL 8085 ONBOARD MEMORY, AND
   82                               THE INTERRUPT VECTOR
FOR THE MAILBOX INTERRUPT IS IN THAT BLOCK (AT 2600H).
   83   825A   01 42 82        LODINT:        LXI   B,INTR
   84   825D   11 00 26                       LXI   D,2600H
   85   8260   0A              LD1:           LDAX  B
   86   8261   FE DD                          CPI   ENDMK
   87   8263   CA 6C 82                       JZ    LD2
   88   8266   12                             STAX  D
   89   8267   03                             INX   B
   90   8268   13                             INX   D
   91   8269   C3 60 82                       JMP   LD1
   92   826C   FB              LD2:           EI  ; ENABLE
8085 INTERRUPTS
```

```
  93   826D   3E 10                          MVI   A,10H
  94   826F   D3 01                          OUT   01H  ;
ENABLE MAILBOX INTERRUPT ON LINK CARD
  95   8271   C9                             RET
  96                          ------------------------
-------------------------------------
  97                                MUPLY ENTRY: D =
MULTIPLICAND AND C = MULTIPLIER
  98                                     EXIT: BC = RESULT
  99   8272   06 00            MUPLY:        MVI   B,0 ;
INITIALIZE MSB OF RESULT
 100   8274   1E 09                          MVI   E,9 ;BIT
COUNTER
 101   8276   79               MUP0:         MOV   A,C
 102   8277   1F                             RAR
 103   8278   4F                             MOV   C,A
 104   8279   1D                             DCR   E
 105   827A   C8                             RZ
 106   827B   78                             MOV   A,B
 107   827C   D2 80 82                       JNC   MUP1
 108   827F   82                             ADD   D
 109   8280   1F               MUP1:         RAR
 110   8281   47                             MOV   B,A
 111   8282   C3 76 82                       JMP   MUP0
 112
 113                          ------------------------
```

```
------------------------------------
  114                              DIV ENTRY: BC= DIVIDEND
;DE= DIVISOR
  115                                   EXIT: BC= QUOTIENT
;DE= REMAINDER
  116   8285   7A              DIV:          MOV   A,D
;NEGATE DIVISOR
  117   8286   2F                            CMA
  118   8287   57                            MOV   D,A
  119   8288   7B                            MOV   A,E
  120   8289   2F                            CMA
  121   828A   5F                            MOV   E,A
  122   828B   13                            INX   D
  123   828C   21 00 00                      LXI   H,0     ;
INITIAL VALUE FOR REMAINDER
  124   828F   3E 11                         MVI   A,17    ;
LOOP COUNTER
  125   8291   E5              DV0:          PUSH  H ; SAVE
REMAINDER
  126   8292   19                            DAD   D ;SUBTR
DIVISOR (ADD NEGATIVE)
  127   8293   D2 97 82                      JNC   DV1; UNDE-
RFLOW, RESTORE HL
  128   8296   E3                            XTHL
  129   8297   E1              DV1:          POP   H
  130   8298   F5                            PUSH  PSW ;SAVE
```

```
LOOP CTR
  131   8299   79              MOV   A,C
  132   829A   17              RAL
  133   829B   4F              MOV   C,A
  134   829C   78              MOV   A,B
  135   829D   17              RAL
  136   829E   47              MOV   B,A
  137   829F   7D              MOV   A,L
  138   82A0   17              RAL
  139   82A1   6F              MOV   L,A
  140   82A2   7C              MOV   A,H
  141   82A3   17              RAL
  142   82A4   67              MOV   H,A
  143   82A5   F1              POP   PSW
  144   82A6   3D              DCR   A
  145   82A7   C2 91 82        JNZ   DV0
  146                          ;
  147   82AA   B7              ORA   A
  148   82AB   7C              MOV   A,H
  149   82AC   1F              RAR
  150   82AD   57              MOV   D,A
  151   82AE   7D              MOV   A,L
  152   82AF   1F              RAR
  153   82B0   5F              MOV   E,A
  154   82B1   C9              RET
  155
```

```
  156                           ----------------------
-------------------------------------
  157                                 ERROR FLAGS RESULT
OVERFLOW (RES>255) AT SOME POINT IN PROGRAM
  158   82B2   3E FF           ERROR:          MVI   A,0FFH
  159   82B4   D3 D0                           OUT   LED
  160   82B6   C9                              RET
  161
  162                           ----------------------
----------------------------------------
  163                                 MULT ENTRY: B = A/B INT ;
C = A/B DEC; D = R/C - 127 ; E = A/B FLG
  164                                      EXIT: INTRES = INT PART
DECRES = DEC PART ; FLGRES = SIGN OF RESULT
  165   82B7   C5              MULT:           PUSH  B ;SAVE
INTEGER PART OF A/B
  166   82B8   7A                              MOV   A,D
  167   82B9   E6 80                           ANI   80H ;SIGN
  168   82BB   F5                              PUSH  PSW ;SAVE.
  169   82BC   AB                              XRA   E ;SIGN OF
RESULT (XOR OF THE TWO SIGNS)
  170   82BD   21 0E 82                        LXI   H,FLGRES
  171   82C0   77                              MOV   M,A ;STORE
  172   82C1   F1                              POP   PSW ;REST-
ORE SIGN OF R/C IN A
  173   82C2   CA C9 82                        JZ    MULT1
```

```
  174   82C5   7A                              MOV   A,D
  175   82C6   2F                              CMA
  176   82C7   3C                              INR   A
  177   82C8   57                              MOV   D,A ; -R/C
IF IT WAS NEGETIVE
  178                              NOW, SIGN OF RESULT IS SET.
D HAS UNSIGNED R/C AND C HAS DEC PART OF A/B
  179
  180   82C9   CD 72 82        MULT1:         CALL  MUPLY
  181
  182                               NOW BC SHOULD HAVE RESULT
AND D STILL HAS R/C VALUE.
  183                                 ----------------------
--------------
  184   82CC   D5                              PUSH  D
  185   82CD   16 00                           MVI   D,0
  186   82CF   1E 64                           MVI   E,100
  187                            NOW DE HAS 100 AND BC HAS
RESULT OF MUPLY
  188   82D1   CD 85 82                        CALL  DIV
  189                             NOW BC HAS "PART" OF THE
INT RESULT AND DE HAS THE REMAINDER --- THE DEC PART
  190   82D4   78                              MOV   A,B
  191   82D5   FE 00                           CPI   0
  192   82D7   C4 B2 82                        CNZ   ERROR
  193   82DA   21 0C 82                        LXI   H,INTRES
```

```
  194   82DD   71                          MOV   M,C
  195   82DE   23                          INX   H ; DECRES
  196   82DF   73                          MOV   M,E
  197
  198   82E0   D1                          POP   D ;RESTORE
  199   82E1   C1                          POP   B ;R/C VAL
AND INT PART PART OF A/B
  200   82E2   48                          MOV   C,B
  201   82E3   CD 72 82                    CALL  MUPLY ;MUL
-TIPLIES INT PART OF A/B WITH R/C
  202   82E6   78                          MOV   A,B
  203   82E7   FE 00                       CPI   0
  204   82E9   C4 B2 82                    CNZ   ERROR
  205   82EC   21 0C 82                    LXI   H,INTRES
  206   82EF   7E                          MOV   A,M
  207   82F0   81                          ADD   C
  208   82F1   DC B2 82                    CC    ERROR
  209   82F4   77                          MOV   M,A ; THIS
IS THE FINAL INTEGER PART OF RESULT
  210   82F5   C9                          RET
  211
  212                        ----------------------------
-------------------------------------
  213                             ADDER ADDS THE "RES" VALUE
TO THE "TOT" VALUE AND PUTS RESULT IN "TOT"
  214                             IT TAKES CARE OF THE SIGNS
```

```
OF THE TWO DECIMAL NUMBERS AND ADDS OR
  215                                SUBTRACTS AS NECESSARY.
  216   82F6   21 0E 82        ADDER:          LXI   H,FLGRES
  217   82F9   7E                              MOV   A,M
  218   82FA   21 11 82                        LXI   H,TOTFLG
  219   82FD   AE                              XRA   M ; IF
Z-FLAG IS SET THEN BOTH ARE SAME SIGN
  220   82FE   C2 22 83                        JNZ   DIFF
  221                              SAME SIGN --- JUST ADD THE
TWO. SIGN OF RESULT SAME AS BEFORE.
  222   8301   21 0D 82                        LXI   H,DECRES
  223   8304   7E                              MOV   A,M
  224   8305   21 10 82                        LXI   H,TOTDEC
  225   8308   86                              ADD   M ; MAX
IS 99 + 99 --- NO OVERFLOW PROBLEM
  226   8309   FE 63                           CPI   99
  227   830B   CA 11 83                        JZ    OK
  228   830E   D4 DC 83                        CNC   ADJST ;
IF RESULT OF ADDITION IS > 99 THEN NEED TO ADJUST
  229   8311   21 10 82        OK:             LXI   H,TOTDEC
  230   8314   77                              MOV   M,A ;
STORE DEC PART IN TOTDEC
  231
  232   8315   21 0C 82                        LXI   H,INTRES;
INTEGER PART
  233   8318   7E                              MOV   A,M
```

```
  234   8319   21 0F 82                    LXI   H,TOTINT
  235   831C   86                          ADD   M   ; ADD
INTEGER PARTS
  236   831D   DC B2 82                    CC    ERROR
  237   8320   77                          MOV   M,A ;STORE
INTEGER PART OF RESULT
  238   8321   C9                          RET
  239
  240                          DIFFERENT SIGNS ---
  241   8322   7E             DIFF:        MOV   A,M ; HL
HAS TOTFLF - GET IT
  242   8323.  FE 80                       CPI   80H   ;
CHECK TOT SIGN
  243   8325   CA 7B 83                    JZ    TOTNEG
  244                          IF RES IS NEGETIVE: (TOT -
RES) IS TO BE DONE ......
  245
  246   8328   21 0C 82       RESNEG:      LXI   H,INTRES
  247   832B   7E                          MOV   A,M
  248   832C   21 0F 82                    LXI   H,TOTINT
  249   832F   BE                          CMP   M       ;
Z=1=>EQUAL. C=1 => RES < TOT.
  250   8330   CA 5E 83                    JZ    DIFF1   ;
EQUAL. SO, CHECK DEC PARTS
  251   8333   DA 4A 83                    JC    DIFF2
  252   8336   21 0C 82       DIFF5:       LXI   H,INTRES;
```

```
NO CARRY
  253   8339   46                          MOV   B,M      ;
  254   833A   23                          INX   H        ;
SO, RES>TOT
  255   833B   4E                          MOV   C,M      ;
  256   833C   21 0F 82                     LXI   H,TOTINT;
FOR "+" RES
  257   833F   56                          MOV   D,M      ;
  258   8340   23                          INX   H        ;
NEED TO DO (RES - TOT)
  259   8341   5E                          MOV   E,M      ;
  260   8342   21 11 82                     LXI   H,TOTFLG;
SIGN OF RESULT = "-"
  261   8345   36 80                       MVI   M,80H    ;
(TOT - RES) = -(RES - TOT)
  262   8347   C3 C2 83                     JMP   SUBTR
  263
  264   834A   21 0F 82        DIFF2:      LXI   H,TOTINT;
  265   834D   46                          MOV   B,M      ;
RES < TOT
  266   834E   23                          INX   H        ;
  267   834F   4E                          MOV   C,M      ;
FOR "+" RES
  268   8350   21 0C 82                     LXI   H,INTRES;
  269   8353   56                          MOV   D,M      ;
NEED TO DO (TOT - RES)
```

```
  270   8354   23                          INX   H       ;
  271   8355   5E                          MOV   E,M     ;
SIGN OF RESULT = "+"
  272   8356   21 11 82                    LXI   H,TOTFLG;
  273   8359   36 00                       MVI   M,0H    ;
(TOT - RES) = +(TOT - RES)
  274   835B   C3 C2 83                    JMP   SUBTR   ;
  275
  276   835E   21 0D 82        DIFF1:      LXI   H,DECRES;
COMPARING DEC PARTS ......
  277   8361   7E                          MOV   A,M
  278   8362   21 10 82                    LXI   H,TOTDEC
  279   8365   BE                          CMP   M
  280   8366   CA 6F 83                    JZ    DIFF4;EQL.
  281   8369   DA 4A 83                    JC    DIFF2   ;
C=1 --- TOT - RES
  282   836C   C3 36 83                    JMP   DIFF5   ;
C=0 --- RES - TOT
  283
  284   836F   21 0F 82        DIFF4:      LXI   H,TOTINT;
  285   8372   36 00                       MVI   M,0
  286   8374   23                          INX   H
  287   8375   36 00                       MVI   M,0
  288   8377   23                          INX   H
  289   8378   36 00                       MVI   M,0
  290   837A   C9                          RET
```

```
  291
  292                                IF TOT IS NEGETIVE: (RES
-TOT) IS TO BE DONE .....
  293
  294   837B   21 0C 82        TOTNEG:        LXI   H,INTRES
  295   837E   7E                             MOV   A,M
  296   837F   21 0F 82                       LXI   H,TOTINT
  297   8382   BE                             CMP   M        ;
Z=1 => EQUAL. C=1 => RES < TOT.
  298   8383   CA B1 83                       JZ    DIFF7;EQL,
-- CHECK DEC PARTS
  299   8386   DA 9D 83                       JC    DIFF8
  300
  301   8389   21 0C 82        DIFF9:         LXI   H,INTRES;
SIGN = "+". NEED TO DO (RES - TOT)
  302   838C   46                             MOV   B,M
  303   838D   23                             INX   H
  304   838E   4E                             MOV   C,M
  305   838F   21 0F 82                       LXI   H,TOTINT
  306   8392   56                             MOV   D,M
  307   8393   23                             INX   H
  308   8394   5E                             MOV   E,M
  309   8395   21 11 82                       LXI   H,TOTFLG
  310   8398   36 00                          MVI   M,0H
  311   839A   C3 C2 83                       JMP   SUBTR
  312
```

```
 313   839D   21 0F 82        DIFF8:        LXI   H,TOTINT;
SIGN = "-". NEED TO DO (TOT - RES)
 314   83A0   46                            MOV   B,M
 315   83A1   23                            INX   H
 316   83A2   4E                            MOV   C,M
 317   83A3   21 0C 82                      LXI   H,INTRES
 318   83A6   56                            MOV   D,M
 319   83A7   23                            INX   H
 320   83A8   5E                            MOV   E,M
 321   83A9   21 11 82                      LXI   H,TOTFLG
 322   83AC   36 80                         MVI   M,80H
 323   83AE   C3 C2 83                      JMP   SUBTR
 324
 325   83B1   21 0D 82        DIFF7:        LXI   H,DECRES
 326   83B4   7E                            MOV   A,M
 327   83B5   21 10 82                      LXI   H,TOTDEC
 328   83B8   BE                            CMP   M
 329   83B9   CA 6F 83                      JZ    DIFF4   ;
 330   83BC   DA 9D 83                      JC    DIFF8
 331   83BF   C3 89 83                      JMP   DIFF9
 332
 333                          SUBTR SUBTRACTS (DE) FROM
(BC). IT PUTS RESULT IN LOCATION TOT.
 334   83C2   79              SUBTR:        MOV   A,C
 335   83C3   93                            SUB   E
 336   83C4   4F                            MOV   C,A
```

```
  337   83C5   78                              MOV   A,B
  338   83C6   9A                              SBB   D
  339   83C7   DC B2 82                        CC    ERROR  ;
RESULT HAS TO BE POSITIVE !
  340   83CA   47                              MOV   B,A
  341   83CB   79                              MOV   A,C
  342   83CC   E6 80                           ANI   80H
  343   83CE   CA D5 83                        JZ    SUB1
  344   83D1   79                              MOV   A,C  ; IF
DEC PART IS NEGETIVE, ADD 100 TO IT.
  345   83D2   C6 64                           ADI   100
  346   83D4   4F                              MOV   C,A
  347   83D5   21 0F 82          SUB1:        LXI   H,TOTINT
  348   83D8   70                              MOV   M,B
  349   83D9   23                              INX   H
  350   83DA   71                              MOV   M,C
  351   83DB   C9                              RET
  352
  353                          ---------------------------
----------------------------------------
  354                            ADJST TAKES CARE OF THE
CARRY IN DECIMAL ADDITION
  355                                    ENTRY: A HAS THE
ADDITION RESULT OF THE DEC PARTS -- TO BE ADJUSTED.
  356                                    EXIT: A - ADJUSTED.
  357   83DC   D6 64             ADJST:        SUI   100
```

```
  358   83DE   21 0F 82                    LXI   H,TOTINT
  359   83E1   34                          INR   M  ; ADD
THE CARRY TO THE INT PART
  360   83E2   C9                          RET
  361
  362                        ------------------------
----------------------------------------
  363                              FINDCK FINDS THE CURRENT O/P
C(K)=A2*R(K-2) + A1*R(K-1) + A0*R(K) + B2*C(K-2) + B1*C(K-1)
  364                                IT ADJUSTS THE DECIMAL
RESULT (TOT) AND PUTS IT IN CK1.
  365   83E3   AF             FINDCK:     XRA   A
  366   83E4   32 0F 82                    STA   TOTINT
  367   83E7   32 10 82                    STA   TOTDEC
  368   83EA   32 11 82                    STA   TOTFLG ;
INITIALIZE TOT VALUES TO 0
  369   83ED   21 F8 81                    LXI   H,A0INT
  370   83F0   46                          MOV   B,M
  371   83F1   23                          INX   H
  372   83F2   4E                          MOV   C,M
  373   83F3   23                          INX   H
  374   83F4   5E                          MOV   E,M
  375   83F5   21 07 82                    LXI   H,RK0
  376   83F8   56                          MOV   D,M
  377   83F9   00                          NOP
  378   83FA   CD B7 82                    CALL  MULT
```

```
379   83FD   00          NOP
380   83FE   CD F6 82     CALL  ADDER
381   8401   00          NOP
382   8402   21 FB 81     LXI   H,A1INT
383   8405   46          MOV   B,M
384   8406   23          INX   H
385   8407   4E          MOV   C,M
386   8408   23          INX   H
387   8409   5E          MOV   E,M
388   840A   21 08 82     LXI   H,RK1
389   840D   56          MOV   D,M
390   840E   00          NOP
391   840F   CD B7 82     CALL  MULT
392   8412   00          NOP
393   8413   CD F6 82     CALL  ADDER
394   8416   00          NOP
395   8417   21 FE 81     LXI   H,A2INT
396   841A   46          MOV   B,M
397   841B   23          INX   H
398   841C   4E          MOV   C,M
399   841D   23          INX   H
400   841E   5E          MOV   E,M
401   841F   21 09 82     LXI   H,RK2
402   8422   56          MOV   D,M
403   8423   00          NOP
404   8424   CD B7 82     CALL  MULT
```

```
405   8427   00          NOP
406   8428   CD F6 82    CALL  ADDER
407   842B   00          NOP
408   842C   21 01 82    LXI   H,B1INT
409   842F   46          MOV   B,M
410   8430   23          INX   H
411   8431   4E          MOV   C,M
412   8432   23          INX   H
413   8433   5E          MOV   E,M
414   8434   21 0A 82    LXI   H,CK1
415   8437   56          MOV   D,M
416   8438   00          NOP
417   8439   CD B7 82    CALL  MULT
418   843C   00          NOP
419   843D   CD F6 82    CALL  ADDER
420   8440   00          NOP
421   8441   21 04 82    LXI   H,B2INT
422   8444   46          MOV   B,M
423   8445   23          INX   H
424   8446   4E          MOV   C,M
425   8447   23          INX   H
426   8448   5E          MOV   E,M
427   8449   21 0B 82    LXI   H,CK2
428   844C   56          MOV   D,M
429   844D   00          NOP
430   844E   CD B7 82    CALL  MULT
```

```
  431   8451   00                          NOP
  432   8452   CD F6 82                    CALL  ADDER
  433   8455   00                          NOP
  434
  435   8456   21 0F 82                    LXI   H,TOTINT
  436   8459   7E                          MOV   A,M
  437   845A   E6 80                       ANI   80H
  438   845C   CA 66 84                    JZ    FIN1
  439   845F   3E FF                       MVI   A,0FFH
  440   8461   D3 D0                       OUT   LED
  441   8463   C3 63 84        FIN2:       JMP   FIN2  ;
HANGS UP HERE IF PROCESSED DATA GOES BEYOND LIMIT
  442
  443   8466   01 08 82        FIN1:       LXI   B,RK1
  444   8469   11 09 82                    LXI   D,RK2
  445   846C   0A                          LDAX  B
  446   846D   12                          STAX  D       ;
RK1 --> RK2
  447   846E   0B                          DCX   B
  448   846F   1B                          DCX   D
  449   8470   0A                          LDAX  B
  450   8471   12                          STAX  D       ;
RK0 --> RK1
  451   8472   01 0A 82                    LXI   B,CK1
  452   8475   11 0B 82                    LXI   D,CK2
  453   8478   0A                          LDAX  B
```

```
  454   8479   12                              STAX  D       ;
CK1 --> CK2
  455   847A   21 0F 82                        LXI   H,TOTINT
  456   847D   46                              MOV   B,M     ;
INT PART OF RES --> B
  457   847E   23                              INX   H
  458   847F   7E                              MOV   A,M     ;
DEC PART-->A
  459   8480   FE 32                           CPI   50
  460   8482   DA 86 84                        JC    FIN3
  461   8485   04                              INR   B       ;
IF DEC PART > 50 THEN ADD 1 TO INT PART
  462   8486   23              FIN3:           INX   H
  463   8487   7E                              MOV   A,M; SIGN
  464   8488   FE 80                           CPI   80H
  465   848A   C2 90 84                        JNZ   FIN4
  466   848D   AF                              XRA   A
  467   848E   90                              SUB   B       ;
NEGATE INT PART IF SIGN IS "-"
  468   848F   47                              MOV   B,A
  469   8490   21 0A 82        FIN4:           LXI   H,CK1
  470   8493   70                              MOV   M,B     ;
ADJUSTED LAST OUTPUT --> CK1
  471   8494   C9                              RET
  472
  473                          ---------------------------
```

```
---------------------------------------

  474                                OUT2PC SENDS THE OUTPUT
AND THE INPUT VALUE TO PC.
  475   8495   DB 01          OUT2PC:       IN    SFLAG
  476   8497   E6 20                        ANI   S2PST
  477   8499   CA 95 84                     JZ    OUT2PC  ;
WAIT FOR MAILBOX TO BE READY
  478   849C   3A 0A 82                     LDA   CK1     ;
GET C(K) VALUE
  479   849F   C6 80                        ADI   128     ;
ADJUST IT --- X-AXIS ZERO ON PC IS AT 128.
  480   84A1   D3 00                        OUT   SMAIL   ;
SEND OUT DATA
  481   84A3   DB 01          OUT1:         IN    SFLAG
  482   84A5   E6 20                        ANI   S2PST
  483   84A7   CA A3 84                     JZ    OUT1    ;
WAIT FOR MAILBOX TO BE READY
  484   84AA   3A 07 82                     LDA   RK0    ;
GET R(K) VALUE
  485   84AD   C6 80                        ADI   128
  486   84AF   D3 00                        OUT   SMAIL ;
SEND IN DATA
  487   84B1   C9                           RET
  488
  489                          ------------------------
---------------------------------------
```

```
  490                           OUT2DA TAKES THE VALUE AT
CK1 AND PUTS IT OUT TO THE D/A CONVERTER
  491   84B2   3A 0A 82      OUT2DA:      LDA   CK1
  492   84B5   C6 80                      ADI   128     ;
GET VALUE AND ADJUST
  493   84B7   D3 F8                      OUT   DAPORT  ;
  494   84B9   C9                         RET
  495
  496                           -------------------------
-------------------------------------
  497                           ADINP RECEIVES INPUT VALUES
FROM THE A/D CONVERTER AND PUTS IT IN RK0
  498   84BA   D3 F9           ADINP:       OUT   ADSTRT
  499   84BC   DB F9           ADIN1:       IN    ADSTAT
  500   84BE   E6 01                       ANI   ADREDY
  501   84C0   CA BC 84                    JZ    ADIN1    ;
WAIT TILL CONVERSION COMPLETE
  502   84C3   DB F8             IN    ADDATA   ; GET DATA
  503   84C5   D6 80             SUI   128      ; ADJUST DATA
  504   84C7   32 07 82          STA   RK0      ; STORE IT
  505   84CA   C9                         RET
  506
  507                           -------------------------
-------------------------------------
  508                           PCIPC0 TAKES DATA (RK0
VALUES) FROM THE "PCDATA" ARRAY AND SENDS CK0 TO PC
```

```
 509                            THIS MODULE ASSUMES THAT
PC SENDS DATA IN THE RIGHT FORM --> -128 TO +127
 510                                            ''''''''''''''''''''''''
''''''''''''''''''''''''''''
 511   84CB   21 1A 82      PCIPCO:       LXI   H,P2P
 512   84CE   CD F5 84                    CALL  SHOSTAT
 513   84D1   01 03 80                    LXI   B,PCDATA
 514   84D4   0B                          DCX   B
 515   84D5   03             PIPO1:       INX   B
 516   84D6   79                          MOV   A,C
 517   84D7   FE F7            CPI   0F7H    ; IF ADDRESS
 518   84D9   C2 E3 84         JNZ   PIPO2   ;IS AT 81F7H
 519   84DC   78               MOV   A,B     ; THEN IT IS
 520   84DD   FE 81            CPI   81H     ; THE END OF
 521   84DF   C2 E3 84         JNZ   PIPO2   ;THE  ARRAY.
 522   84E2   C9                          RET
 523   84E3   C5             PIPO2:       PUSH  B ; STORE
DATA ADDRESS
 524   84E4   11 07 82                    LXI   D,RK0
 525   84E7   0A                          LDAX  B
 526   84E8   12                          STAX  D   ; GET
DATA TO RK0
 527   84E9   00                          NOP
 528   84EA   CD E3 83                    CALL  FINDCK  ;
CALCULATE C(K)
 529   84ED   00                          NOP
```

```
  530   84EE   CD 95 84                    CALL  OUT2PC ;
DATA IN CK1 TO PC MAILBOX
  531   84F1   C1                          POP   B
  532   84F2   C3 D5 84                    JMP   PIPO1
  533
  534                        ------------------------
----------------------------------------
  535                              SHOSTAT DISPLAYS THE CURRENT
STATUS ON THE KEYPAD MONITOR DISPLAY.
  536                               SHOSTAT ENTRY: (HL) = START
ADDR OF 8 DATA BITS MAKING THE MESSAGE.
  537   84F5   0E 08          SHOSTAT:      MVI   C,8 ; CTR
  538   84F7   7E            SHO1:         MOV   A,M ; GET
FIRST CHARACTER
  539   84F8   D3 D0                       OUT   LED ; PUT
IT ON LED PORT -- SAME AS DISPLAY DATA PORT
  540   84FA   79                          MOV   A,C
  541   84FB   3D                          DCR   A
  542   84FC   D3 D1                       OUT   DISPOS ;
SELECT DISPLAY POSITION -- NO WRITE ENABLE
  543   84FE   C6 08                       ADI   8      ;
  544   8500   D3 D1                       OUT   DISPOS ;
  545   8502   D6 08                       SUI   8
  546   8504   D3 D1                       OUT   DISPOS ;
  547   8506   23                          INX   H
  548   8507   0D                          DCR   C
```

```
  549   8508   C2 F7 84                      JNZ   SHO1   ;
WRITE 8 CHARACTERS
  550   850B   C9                            RET
  551
  552                          ******************************
******************************************
  553   850C   CD 5A 82       MAIN:          CALL  LODINT ;
LOAD INTERRUPT SERVICE ROUTINE AT 2600H
  554
  555   850F   AF             MN3:           XRA   A
  556   8510   32 08 82                      STA   RK1
  557   8513   32 09 82                      STA   RK2
  558   8516   32 0A 82                      STA   CK1
  559   8519   32 0B 82                      STA   CK2 ;INI
TIALIZE ALL R/C VALUES TO 0.
  560
  561   851C   3A F7 81        LDA   STATUS  ; GET STATUS
  562   851F   E6 FE           ANI   0FEH    ;RESET  THE
  563   8521   32 F7 81        STA   STATUS  ;"NEWINFO" BIT
  564   8524   E6 80           ANI   80H     ; CHECK
"INPUT" BIT
  565   8526   C2 3F 85                      JNZ   MN1
  566                          PC-INPUT ......
  567   8529   CD CB 84                      CALL  PCIPCO  ;
WORK ON THE SET OF DATA IN PCDATA ARRAY
  568   852C   21 12 82                      LXI   H,COM
```

```
  569   852F   CD F5 84                      CALL  SHOSTAT
  570   8532   3A F7 81        MN2:          LDA   STATUS
  571   8535   D3 D0                         OUT   LED  ;
SHOW STATUS BITS ON KEYPAD LED'S
  572   8537   E6 01                         ANI   01H
  573   8539   CA 32 85                      JZ    MN2  ;
WAIT HERE TILL NEW COMMAND SENT FROM PC
  574   853C   C3 0F 85                      JMP   MN3
  575
  576                          A/D INPUT ......
  577   853F   3A F7 81        MN1:          LDA   STATUS
  578   8542   E6 30                         ANI   30H  ;
MASK OUT ALL BUT THE OUTPUT BITS OF STATUS
  579   8544   FE 00                         CPI   0
  580   8546   C2 52 85                      JNZ   MN5
  581   8549   21 3A 82                      LXI   H,A2X  ;
-> NO OUTPUT
  582   854C   CD F5 84                      CALL  SHOSTAT
  583   854F   C3 74 85                      JMP   MN4
  584   8552   FE 10           MN5:          CPI   10H
  585   8554   C2 60 85                      JNZ   MN6
  586   8557   21 22 82                      LXI   H,A2D   ;
->  TO D/A
  587   855A   CD F5 84                      CALL  SHOSTAT
  588   855D   C3 74 85                      JMP   MN4
  589   8560   FE 20           MN6:          CPI   20H
```

```
  590   8562   C2 6E 85                    JNZ   MN7
  591   8565   21 2A 82                    LXI   H,A2P   ;
-> OUT TO PC
  592   8568   CD F5 84                    CALL  SHOSTAT
  593   856B   C3 74 85                    JMP   MN4
  594   856E   21 32 82        MN7:        LXI   H,A2BTH ;
-> OUT TO BOTH (PC AND D/A)
  595   8571   CD F5 84                    CALL  SHOSTAT
  596
  597   8574   CD BA 84        MN4:        CALL  ADINP
  598   8577   CD E3 83                    CALL  FINDCK
  599   857A   3A F7 81                    LDA   STATUS
  600   857D   E6 20                       ANI   20H     ;
CHECK IF OUTPUT TO BE SENT TO PC
  601   857F   C4 95 84                    CNZ   OUT2PC  ;
SEND TO PC IF BIT-5 IS SET
  602   8582   3A F7 81                    LDA   STATUS
  603   8585   E6 10                       ANI   10H     ;
CHECK IF OUTPUT IS TO BE SENT TO D/A
  604   8587   C4 B2 84                    CNZ   OUT2DA  ;
SEND TO D/A CONVERTER IF BIT-4 IS SET
  605   858A   3A F7 81                    LDA   STATUS
  606   858D   D3 D0                       OUT   LED     ;
SHOW STATUS ON KEYPAD LED'S
  607   858F   E6 01                       ANI   01H
  608   8591   C2 0F 85                    JNZ   MN3
```

```
609    8594    C3 74 85                    JMP    MN4
610
611    8597                                END

LINES ASSEMBLED:    611          ASSEMBLY ERRORS:     0
```

D.3 FILTER MONITOR ON THE PC

The Monitor program is designed to run a menu driven operator service function. The menu is displayed on the screen on running the program and the program then waits for the operator to issue a command. If the STD system had been commanded previously to send output to the PC, then the PC can keep receiving the data, sent by the STD system, in the background; DMA is used to make this possible. This ensures that the STD system processing is not held up by the PC. The program jumps to the specific service routine to service the command issued by the operator and returns to the command mode or the menu screen when the service is over. The main functions that the operator can choose between are :

- Terminate program

- Change Input and Output paths -

  - Input from A/D and Output to D/A
  - Input from A/D and Output to PC

- Input from A/D and Output to both D/A and PC

- Test the filter by loading the STD system with PC generated samples.

- Change the filter by changing the constants.

The different functions are described in the following sections.

D.3.1 Monitor Functions

The command byte for every mode is given; refer toFigure 13 on page 109 for bit assignments. In order to quit the application, the PC programs the DMA controller in the autoinitialize mode and issues a command to the STD system that tells it to take inputs from the A/D converter and send the outputs to neither the PC nor the D/A. The status byte sent as command is 81H :

1	0	0	0	0	0	0	1

This allows for a graceful termination of the program because the STD system effectively becomes passive; it does process the data read from the A/D converter and keeps running until it is reset, but the output is not sent anywhere.

The other functions involve two steps -

1. Sending the appropriate command.

2. Taking action on incoming data - if any.

D.3.1.1 A/D to D/A

For this mode of operation, all the program has to do is to send the status necessary to the STD system. The PC is not involved in the operation of the STD system filter at all; the STD system receives data from the converter, processes it and sends the data out to the D/A. Thus, the PC program returns to servicing the user - waiting for any new command, after sending the command byte out. The byte sent out is 91H :

1	0	0	1	0	0	0	1

D.3.1.2 A/D to PC

In this mode the operator, presumably, wants to examine the data that the STD system is processing. For this the program sends out the status that will route the output to the PC. It then programs the DMA controller to be in the autoinitialize mode so that the STD system can keep sending data. The program asks the operator if he wants to view the incoming data and if he does, then it jumps to a Plot routine. The command sent out to the STD system is A1H :

1	0	1	0	0	0	0	1

In order to plot the data, the PC has to stop the DMA controller from receiving the data and putting them in memory; the program should take the data as they come in and plot them to the screen. For this, the first thing to do is to disable the DMA controller. Then the data sent by the STD system will wait at the mailbox until the PC software receives it. The data coming in is scaled by the PC and plotted; see-- Heading id 'PLOT' unknown -- for details. This goes on continuously till the operator wishes to discontinue the mode. Hitting any key pulls the PC out of this mode; the PC then sets the 8237 into autoinit mode again and returns to the menu screen.

D.3.1.3 A/D to both PC and D/A

This mode only involves a different command - actually, the OR of the previous two command bytes because the function is the sum of the two previous functions. The PC plots the incoming data, if the operator so desires, just the same as described above. The Status byte sent is B1H :

1	0	1	1	0	0	0	1

D.3.1.4 Testing the Filter

This function involves a slightly different mode of operation because here the real time data from the A/D is not used and the output is not sent to the D/A either. The PC processes the data samples. It interactively reads in the desired frequency and amplitude of the Sine wave and

generates the data samples in the LOAD_DATA procedure. Simple Sine arithmetic function is used; the increments in the variable is computed using the sampling frequency of the STD filter (125Hz) and the frequency of the wave given by the operator. The data is loaded into the STD memory at a fixed, predecided location using Forced transfer. The STD Bus is then released and a new status is also forced in. This status tells it to jump to a program segment which works on the data array loaded by the PC; the consecutive iterations are done on this set of data and the outputs are sent to the mailbox for the PC to receive. The status sent is 01H :

0	0	0	0	0	0	0	0

The 8237 is programmed in the autoinitialize disabled mode for this application. So, the 8237 receives the set of 500 inputs and 500 outputs from the STD system as fast as the 8085 can process the data. The status register in the 8237 has its TC flag for Channel 1 set after the 1000 bytes of data have been received (the Current Word Count Register was loaded with 999) and this is detected by the software which, all this time, was polling the flag. When this happens the program asks the operator whether he wants to examine the data processed. If he does, then the SECT_PLOT procedure is used to view different segments of the data set collected. This offers the versatility of magnifying small stretches of data if so desired. The operator can thus scan the set of 500 data inputs and outputs to check how the system works.

D.3.1.5 Plotting the Data

The color graphics screen is used for this function. This offers a 320x200 dots color graphics screen. The DRAW procedure built into the TURBO Pascal compiler[11] is used to plot the data. The data range is from 0 to 255. It is scaled to fit the y-axis range of 160 (the simple operation of 160*data/255 does the scaling) and the y-co-ordinate is thus found. For the x-axis, the length of the axis is 300 points. So, 300 consecutive points can be scaled and plotted with the x and y co-ordinates just explained.

For**SECT_PLOT** we need to plot segments of data in the x-axis range of 300. This involves scaling in the x-axis also. If, for instance, the operator wants to view samples 200 to 349 - that is 150 data units, then we simply need to plot the data units every alternate x-axis point! The y-axis scaling is obviously the same.

For**CONT_PLOT** there is the new problem of data units after the 300th data. To take care of the "Wrap-Around" feature necessary, buffer arrays are created for this routine. What it does is, it stores the set of 300 data units being plotted, in the buffer reserved. Then when the 301th data and onwards are to be plotted the program first draws a blank over the buffer data at that x-coordinate and then draws the present data to the screen and stores the data in the buffer for the next wrap around. This process is quite easy to see in the procedure CONT_PLOT.

---

[11] It is made for the MSDOS (and CPM) environment and includes non standard Pascal features that exploits the PC facilities; DRAW is one such feature.

D.3.1.6 Changing the Filter

This is done by using Forced Transfer. The program interactively reads in the constants that the operator wants to load, it asks the user whether he is satisfied with the entries, and if so, then the program goes ahead to load the data into the STD memory. The data area for the constants are fixed; see "Program Listing (FILTR.ASM)" on page 112After the PC injects the new constants, it also initializes the iteration bases (previous inputs and outputs) to zero - after all, the system is a totally new one; a new filter is to be checked. It sets the new-info bit in the status byte so that the STD system starts the process anew. The PC program does not do any kind of checking on the constants put in. If the values are such that the filter is unstable, then the output will reflect that.

D.3.2 General Comments on The Program

The general flow of the program is depicted inFigure 14 on page 149. The program monitors the data frequency and the amplitude continuously when it is in the Plot mode. This does not slow down the system because the processing is done during the time in which the STD system is doing its iterations to prepare the data to be sent to the PC. So, the PC uses the time between data arrival from the STD system, by finding the amplitude and frequency. The way this is done is, it stores the input value in a buffer variable if it is greater than the last input. It keeps doing this till the input is less than the last input to the PC. This last value is the maximum for this cycle - the amplitude. The frequency is found by

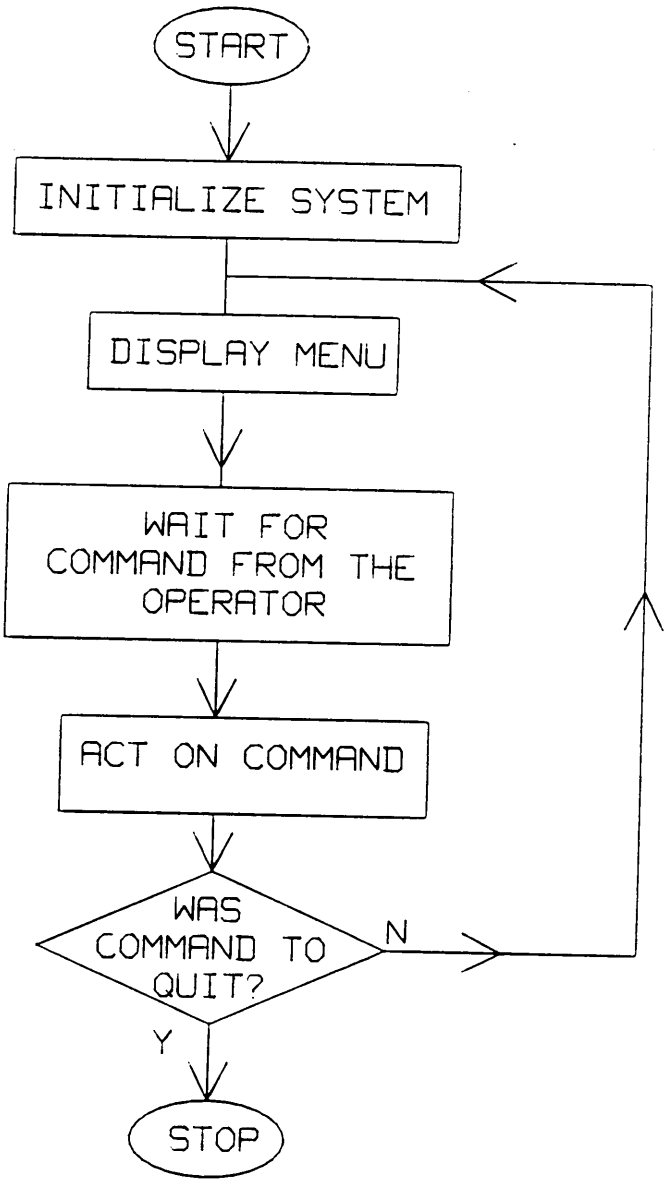

Figure 14. General Flow of Monitor Program

finding the number of samples that are there between each peak - using the sampling interval of the STD system (1/125) the program then computes the frequency. All this assumes periodic waves.

D.4 PROGRAM LISTING (MONITR.PAS)

```
{*****************************************************************
 *                                                               *
 *          |  Program      = STD BUS FILTER MONITOR |           *
 *          |  Developed by= Diptish Datta           |           *
 *          |  Language     = PASCAL                 |           *
 *          |  System       = IBM PC                 |           *
 *          |  Compiler     = Turbo Pascal           |           *
 *          |  Date         = June 1985              |           *
 *          |----------------------------------------|           *
 *                                                               *
 *  PROGRAM FUNCTION: The program works                          *
 *      in concert with the "FILTR" proram                       *
 *       running on the 8085 on the STD Bus.                     *
 *       It is menu driven and acts as a                         *
 *       operator interface to the Digital                       *
 *       Filter implemented on the STD bus                       *
 *       system. It can tell the STD  system                     *
 *       to take  inputs  from  the  A/D                         *
 *       converter or to accept operator controlled              *
 *       data-set from the PC. It                                *
```

```
*       can arrange for the processed output          *
*       to be sent to  the  D/A on the                *
*       STD Bus or to the PC or to both.              *
*       It can display the input &  output            *
*       on the PC screen. The Program also            *
*       allows the  operator  to  change              *
*       the Filter.                                   *
*                                                     *
*  PROGRAM LIMITATIONS: Due to the low data           *
*       processing speed of the 8085,                 *
*       the time taken to plot the waves on           *
*       the PC and the slow A/D process               *
*       on the STD Bus, the sampling rate is          *
*       only about 125Hz. This  limits                *
*       the maximum input frequency, for a            *
*       reasonable  sampling, to  12.5Hz              *
*       only.                                         *
*                          The program does not       *
*       check the  filter  constants                  *
*       entered by the operator. If the               *
*       filter implemented has poles out of           *
*       the unit circle then it is unstable;          *
*       the operator has to check  the                *
*       conditions of the filter he wants to check.   *
*****************************************************}
```

```
program FILTER_MONITOR (input,output);
 const
  smail = $300;
  sbus  = $301;
  slad  = $302;
  shad  = $303;
  sctl  = $304;
  serv  = $305;
 var
  data: array [1..1000] of byte;
  auto_init: boolean;
  k: integer;
  ans: char;
  page: byte; {DMA controller page register}
  new_status,curr_status: byte;
  quit: boolean;
  pc_freq: integer;

{"=============================================================="
 " INIT_DMA_CNTRLR                                              "
 " Parameters Passed= auto - whether to put DMA Controller"
 "                           in autoinit mode or not.          "
 " Called by        = INIT_SYSTEM, PCIN_PCOUT                   "
 " Function         = Initializes DMA Controller.               "
 "=============================================================="}
  procedure INIT_DMA_CNTRLR(auto: boolean);
```

```
    var
     temp: integer;
    begin
     port[$0A]:= 5; port[$0C]:= 0; {disable Ch-1}
     temp:= seg(data);
     page:= (temp and $F000) shr 12; {m.s. nibble of
                                segment register of data}
     port[$83]:= page;
     temp:= (temp shl 4) + ofs(data); {absolute addr of
                          data is given by temp (on page)}
     port[$02]:= temp and $00FF;
     port[$02]:= temp shr 8; {Ch-1 curr addr}
     port[$03]:= $E7; port[$03]:= 3; {Ch-1 curr
                                        word count = 999}
     port[$302]:= 0;
     if auto then port[$0B]:= $55 {single transfer,addr
                                  inc,autoinit,write,Ch-1}
      else port[$0B]:= $45;       {same, but w/o autoinit}
     port[$0A]:= 1; {enable Ch-1}
     port[serv]:=$A0; port[$307]:= 8; {sysrun, stdmailen,
                                          drqen - to link}
    end;

{"=============================================================="
 "  ANS_IS_YES                                                  "
 "  Parameters Passed= none                                     "
```

```
"  Called By        = SECT_PLOT, CHANGE)FILTER,LOAD_DATA,"
"                     ADIN_PCOUT, ADIN_BOTHOUT,PCIN_PCOUT"
"  Function         = waits for "Y" or "N" response from "
"                     operator and returns                "
"                     a boolean value true if ans is yes."
"═══════════════════════════════════════════════════════"}

  function ANS_IS_YES: boolean;
   var answered: boolean; c: char;
   begin
    answered:= false;
    while not answered do begin
     repeat begin end until KeyPressed; read(kbd,c);
     if upcase(c) in ['Y','N'] then answered:= true;
    end;
    if upcase(c)='Y' then ANS_IS_YES:=true
       else ANS_IS_YES:=false;
   end;


{"═══════════════════════════════════════════════════════"
" AXES                                                   "
" Parameters Passed= none                                "
" Called By        = SECT_PLOT, CONT_PLOT                "
" Function         = invokes the color graphics mode     "
"                     and draws the axes                  "
"═══════════════════════════════════════════════════════"}

  procedure AXES;
```

```
    begin
     GraphColorMode; Palette(3); GraphBackGround(0);
     Draw(10,180,310,180,3); Draw(10,20,10,180,3); {axes}
     Draw(230,3,250,3,1); GoToXY(33,1);write('OUTPUT');
     Draw(230,12,250,12,2); GoToXY(33,2);write('INPUT');
     GoToXY(2,24); write('IN_AMP =');
     GoToXY(15,24); write('OUT_AMP =');
     GoToXY(30,24);write('FREQ =');
    end;

{"═══════════════════════════════════════════════════════"
 " WAIT_FOR_TC                                           "
 " Parameters Passed= none                              "
 " Called By        = PCIN_PCOUT                        "
 " Function         = waits for the STD Bus system      "
 "                    to send the set of 500            "
 "                    inputs and 500 outputs by DMA.    "
 "═══════════════════════════════════════════════════════"}
   procedure WAIT_FOR_TC;
    var
     check: byte;
     ch: char;
    begin
     writeln; write(' Waiting for 500 units of data .... ');
     check:= port[$08]; {clear port before starting check}
     repeat
```

```
      check:= port[$08] and 2
     until (check <> 0)or(KeyPressed);
     if KeyPressed then begin
      read(kbd,ch);
      writeln(^G,' DATA COLLECTION INTERRUPTED BY USER.');
     end
     else begin
      port[$0A]:= 5; port[$83]:= 0; {stop DMA -- so, after
          this procedure is called INIT_DMA has to be done}
      writeln('O.K. !');writeln;
     end;
    end;

{"═══════════════════════════════════════════════════"
 " SECT_PLOT                                         "
 " Parameters Passed= none                          "
 " Called By        = PCIN_PCOUT                    "
 " Function         = plots any desired segment of the "
 "                    500 input-output set          "
 "                    of data collected. The maximum "
 "                    window allowed is 300.        "
 "═══════════════════════════════════════════════════"}
   procedure SECT_PLOT;
    var
     t_str,t_fin: integer;
     no_of_data,inrvl,range,t_ctr: integer;
```

```
 pt_A,pt_B,pt_C,pt_D: byte;
 tim_1,tim_2: integer;
 dd: real;
 done: boolean;
 max_B,max_D: byte;
begin
 repeat
  TextMode; done:= false;
  writeln(' You can examine any segment
    (max size 300) of the set of 500 data units.');
  writeln;
  while not done do begin
   write('       Enter start point: '); readln(t_str);
   if (t_str>0)and(t_str<500) then done:=true; end;
  writeln; done:= false;
  while not done do begin
   write('       Enter finish point: '); readln(t_fin);
   if (t_fin>1) and (t_fin<501)and(t_fin>t_str)
     and((t_fin-t_str)<300) then done:=true; end;
  AXES;
  GoToXY(36,24); write(pc_freq:2,'Hz');
  no_of_data:= t_fin-t_str;
  inrvl:= round(300/no_of_data); range:= $FF;
  t_ctr:= 0;
  max_B:= 0; max_D:= 0;
  for k:= t_str to t_fin do begin
```

```
        pt_A:= data[2*k-1];pt_B:= data[2*k+1];
        pt_C:=data[2*k];pt_D:=data[2*k+2];
        if (pt_B > max_B)and(pt_B > pt_A) then max_B:= pt_B;
        if (pt_D > max_D)and(pt_D > pt_C) then max_D:= pt_D;
        if (pt_D < max_D)and(max_D > 128) then begin
           GoToXY(10,24);write((max_D-128):3);max_D:=0;end;
        if (pt_B < max_B)and(max_B > 128) then begin
           GoToXY(24,24);write((max_B-128):3);max_B:=0;end;
        dd:= (160/range)*pt_A; pt_A:= 180 - round(dd);
        dd:= (160/range)*pt_B; pt_B:= 180 - round(dd);
        dd:= (160/range)*pt_C; pt_C:= 180 - round(dd);
        dd:= (160/range)*pt_D; pt_D:= 180 - round(dd);
        tim_1:= 10+(t_ctr*inrvl); tim_2:= tim_1+inrvl;
        t_ctr:= t_ctr+1;
        Draw(tim_1,pt_A,tim_2,pt_B,1);
        Draw(tim_1,pt_C,tim_2,pt_D,2);
       end;

       GoToXY(2,1);write('Another Segment ? (Y/N)');
      until not ANS_IS_YES;
     end;

{"=============================================================="
 " MAIL                                                        "
 " Parameters Passed= none                                     "
 " Called By        = CONT_PLOT                                "
```

```
 " Function          = waits for input to the mailbox from "
 "                     the STD Bus system.                  "
 "                     It can be forced out of a wait       "
 "                     loop by hiiting any key.             "
 "═══════════════════════════════════════════════════════"}

   function MAIL: byte;
    var serv_stat,stdmailf: byte;
    begin
     if ans <> ' ' then begin
      stdmailf:= $20;
      repeat
       serv_stat:= port[serv];
       serv_stat:= serv_stat and stdmailf;
      until (serv_stat<>0)or(KeyPressed);
      if KeyPressed then read(kbd,ans);
      MAIL:= port[smail];
     end;
    end;

{"═══════════════════════════════════════════════════════"
 " CONT_PLOT                                             "
 " Parameters Passed= none                              "
 " Called By        = ADIN_PCOUT, ADIN_BOTHOUT, CALL_PLOT "
 " Function         = takes the data sent from the STD   "
 "                     Bus and plots it to              "
 "                     the PC screen. It assumes that   "
```

```
"                    the STD system sends the                "
"                    data in the right order, i.e.           "
"                    output, input - alternately.             "
"═══════════════════════════════════════════════════════════"}

  procedure CONT_PLOT;
   var
    bak_A,bak_B,bak_C,bak_D: array [1..300] of byte;
    pt_A,pt_B,pt_C,pt_D: byte;
    tim_1,tim_2: integer;
    t_ctr: integer;
    max_B,max_D: byte;
    pre_B,pre_D: byte;
    sample_no: integer;
    frequency: integer;
   begin
    port[$0A]:= 5; port[$83]:= 0; { disable DMA }
    max_B:= 0; max_D:= 0;
    for k:=1 to 300 do begin bak_A[k]:=100; bak_B[k]:=100;
      bak_C[k]:=100; bak_D[k]:=100; end;
    AXES; GoToXY(2,1); write('HIT SPACE BAR TO QUIT');
    pt_A:=MAIL; pt_C:=MAIL;
    pt_B:=MAIL; pt_D:=MAIL; t_ctr:= 0;
    pre_B:= pt_B; pre_D:= pt_D;
    sample_no:=1;
    pt_A:= 180 - round((160/$FF)*pt_A);
    pt_B:= 180 - round((160/$FF)*pt_B);
```

```
pt_C:= 180 - round((160/$FF)*pt_C);
pt_D:= 180 - round((160/$FF)*pt_D);
repeat
 tim_1:= 10 + t_ctr; tim_2:= tim_1 + 1;
 t_ctr:= t_ctr+1;
 Draw(tim_1,bak_A[t_ctr],tim_2,bak_B[t_ctr],0);
                              {erase previous A-B line}
 Draw(tim_1,bak_C[t_ctr],tim_2,bak_D[t_ctr],0);
                              {erase previous C-D line}
 Draw(tim_1,pt_A,tim_2,pt_B,1);
 Draw(tim_1,pt_C,tim_2,pt_D,2);
 bak_A[t_ctr]:=pt_A; bak_B[t_ctr]:=pt_B;
 bak_C[t_ctr]:=pt_C; bak_D[t_ctr]:=pt_D;
 if (pt_A=180) and (pt_B=180) then begin
     bak_A[t_ctr]:=179; bak_B[t_ctr]:=179; end;
 if (pt_C=180) and (pt_D=180) then begin
     bak_C[t_ctr]:=179; bak_D[t_ctr]:=179; end;
 if t_ctr=300 then t_ctr:=0;
 pt_A:=pt_B; pt_C:=pt_D;
 pt_B:= MAIL; pt_D:= MAIL;
 sample_no:= sample_no+1;
 if (pt_B > max_B)and(pt_B > pre_B) then max_B:=pt_B;
 if (pt_D > max_D)and(pt_D > pre_D) then max_D:= pt_D;
 if (pt_D < max_D)and(max_D > 128) then begin
  GoToXY(10,24);write((max_D-128):3);max_D:=0; end;
 if (pt_B < max_B)and(max_B > 128) then begin
```

```
      GoToXY(24,24);write((max_B-128):3);max_B:=0;
      frequency:= round((500/4)/sample_no); GoToXY(36,24);
      write(frequency:2,'Hz');sample_no:=1;end;
    pre_B:= pt_B; pre_D:= pt_D;
    pt_B:= 180 - round((160/$FF)*pt_B);
    pt_D:= 180 - round((160/$FF)*pt_D);
  until ans = ' ';
  ans:='x';
  port[$0A]:= 1; port[$83]:= page;{ restart DMA }
 end;

{"=============================================================="
 " PUSH_IN                                                      "
 " Parameters Passed= hi, lo - address bytes                    "
 "                    dat    - data byte                       "
 " Called By        = CHANGE_FILTER, LOAD_DATA                  "
 " Function         = puts the address on the STD               "
 "                    address bus and the data                  "
 "                    on the STD data bus. If BUSLINK has       "
 "                    been made, i.e. the                       "
 "                    PC has control of the STD bus, and if"
 "                    the IOEXP or the                          "
 "                    MEMEX control signal is given, then       "
 "                    the data gets                             "
 "                    written to the appropreate STD device"
 "=============================================================="}
```

```
  procedure PUSH_IN(hi,lo,dat: byte);
   begin
    port[shad]:= hi; port[slad]:= lo; port[sbus]:= dat;
   end;

{"═══════════════════════════════════════════════════"
 " CHANGE_FILTER                                     "
 " Parameters Passed= none                           "
 " Called By        = PRINT_MENU                     "
 " Function         = interactively reads in the filter "
 "                    constants and then             "
 "                    loads them into the STD memory at "
 "                    the right place.               "
 "═══════════════════════════════════════════════════"}
  procedure CHANGE_FILTER;
   var
    cons: array [1..5] of real;
    inr,dec,sgn: byte;
    chk: byte;
    hiadr,loadr: integer;
    temp: real;
   begin
    repeat
     TextMode;
     writeln('The digital filter computes the output
              with the following equation: ');WRITELN;
```

```
writeln('   C(K) = A0.R(K) + A1.R(K-1) + A2.R(K-2)
                          + B1.C(K-1) + B2.C(K-2)');
writeln;writeln;
writeln('   Enter the constants:');
write  ('      A0 = '); readln(cons[1]);
write  ('      A1 = '); readln(cons[2]);
write  ('      A2 = '); readln(cons[3]);
write  ('      B1 = '); readln(cons[4]);
write  ('      B2 = '); readln(cons[5]);
writeln;
port[sctl]:=$2A; {bus request sent}
repeat  chk:= port[sctl] and $40 until chk=$40;
                        {wait till busak from 8085}
loadr:= $F8;
hiadr:= $81; { A0 starts at 81F8h in STD memory}
for k:= 1 to 5 do begin
 if cons[k]<0 then begin sgn:= $80;
  cons[k]:= Abs(cons[k]); end else sgn:=0;
 inr:= Trunc(Int(cons[k]));
 dec:= Round(100.0*(Frac(cons[k])));
 PUSH_IN(hiadr,loadr,inr);
 loadr:= loadr+1; if loadr=256 then begin
    loadr:= 0; hiadr:= hiadr+1; end;
 PUSH_IN(hiadr,loadr,dec);
 loadr:= loadr+1; if loadr=256 then begin
    loadr:= 0; hiadr:= hiadr+1; end;
```

```
        PUSH_IN(hiadr,loadr,sgn);
        loadr:= loadr+1; if loadr=256 then begin
           loadr:= 0; hiadr:= hiadr+1; end;
      end;
      loadr:= $F7; hiadr:= $81; {status addr = 81F7H}
      curr_status:= curr_status or $01;
      PUSH_IN(hiadr,loadr,curr_status);
      port[sctl]:= 0; {restart 8085}
      writeln;
      writeln;write(' ARE ALL ENTRIES CORRECT ?(Y/N) ');
     until ANS_IS_YES;
    end;

{"═══════════════════════════════════════════════════"
 " LOAD_DATA                                         "
 " Parameters Passed= none                           "
 " Called By        = PCIN_PCOUT                     "
 " Function         = interactively reads in the required "
 "                    Sine wave frequency            "
 "                    and amplitude. It then generates and"
 "                    loads into STD  memory, a set of  "
 "                    500 samples of input wave.     "
 "═══════════════════════════════════════════════════"}
   procedure LOAD_DATA;
    const
     inrvl = 0.008;
```

```
var
 done: boolean;
 ampl,freq: real;
 hiadr,loadr: integer;
 data: byte;
 phase,time: real;
 chk: byte;
begin
 TextMode; done:= false;
 write(' New set of data to be loaded
                        into STD memory ?(Y/N) ');
 if ANS_IS_YES then begin
  while not done do begin
   writeln; writeln;
   writeln(' Sine wave samples are taken
       at 125 Hz. 500 samples are taken.');writeln;
   write  ('   Enter Sine wave frequency: ');
   readln(freq);
   write  ('                    amplitude: ');
   readln(ampl);
   writeln; done:=true;
   if ampl>127.0 then begin writeln(' Maximum
            amplitude that can be accepted is 127.');
    done:= false;end
    else begin
     if freq<0.125 then begin
```

```
        write(' 500 samples will cover less
             than half cycle. OK ?(Y/N) ');
       if ANS_IS_YES then done:=true
        else done:= false; writeln; end;
     if freq>12.5 then begin write(' Sampling
      rate is too low for this frequency. OK ?(Y/N) ');
      if ANS_IS_YES then done:= true
       else done:= false; writeln; end;
   end;
end;
pc_freq:= round(freq);
port[sctl]:=$2A; {bus request sent}
repeat  chk:= port[sctl] and $40
  until chk=$40; {wait till busak from 8085}
GoToXY(2,25); write('LOADING 500 SAMPLES .....');
time:=0; hiadr:= $80; loadr:= $03;
for k:=1 to 500 do begin
 GoToXY(28,25); write((500-k):3);
 phase:= time * freq * 2 * Pi;
 data:= round(ampl * Sin(phase));time:= time+inrvl;
 PUSH_IN(hiadr,loadr,data);
 loadr:= loadr+1; if loadr=256 then begin loadr:=0;
 hiadr:= hiadr+1;end;
end;
port[sctl]:=0; {restart 8085}
```

```
    end;
   end;

{"=============================================================="
 " SEND_COMMAND                                                 "
 " Parameters Passed= cmd - the status byte                     "
 " Called By        = INIT_SYSTEM, PCIN_PCOUT, ADIN_DAOUT,      "
 "                    ADIN_PCOUT, ADIN_BOTHOUT                  "
 " Function         = sends the command byte to the STD         "
 "                    link mailbox. The                         "
 "                    8085 interrupt service routine            "
 "                    receives the new status.                  "
 "                    The procedure waits for the 8085 to       "
 "                    accept the command.                       "
 "                    If 8085 does not accept the data the      "
 "                    operator can force program to quit        "
 "                    the wait loop by hitting any key.         "
 "=============================================================="}
  procedure SEND_COMMAND(cmd: byte);
   var
    check,pcmailen: byte;
    ch: char;
   begin
    if cmd <> curr_status then begin
     pcmailen:= $10;
     repeat
```

```
      check:= port[serv] and pcmailen;
    until (check<>0)or(KeyPressed);
    if KeyPressed then begin
      read(kbd,ch); TextMode;
      writeln; writeln(^G,' COULD NOT SEND COMMAND ...
                          REMOTE SYSTEM NOT ACCEPTING.');
      for k:= 1 to 30000 do begin end;
      for k:= 1 to 30000 do begin end;
    end
    else begin port[smail]:= cmd; curr_status:= cmd; end;
  end;
end;

{"=============================================================="
 " INIT_SYSTEM                                                  "
 " Parameters Passed= none                                      "
 " Called By        = PRINT_MENU, Main Program                  "
 " Function         = initializes the DMA Controller to         "
 "                    autoinitialize mode                       "
 "                    so that if 8085 sends data, it is         "
 "                    automatically taken                       "
 "                    in and stored in the reserved data        "
 "                    array. It also puts                       "
 "                    the 8085 in a passive mode i.e. not       "
 "                    sending output anywhere.                  "
 "=============================================================="}
```

```
  procedure INIT_SYSTEM;
   begin
    auto_init:= true;
    INIT_DMA_CNTRLR(auto_init); {normally whatever
     data the 8085 sends gets stored cyclically in the data}
    ans:='x';
    curr_status:= $00; new_status:= $81;
    SEND_COMMAND(new_status); {8085 set to work on
                A/D data; output not sent anywhere}
   end;

{"═══════════════════════════════════════════════════"
 " PCIN_PCOUT                                        "
 " Parameters Passed= none                           "
 " Called By        = PRINT_MENU                     "
 " Function         = allows the operator to load desired"
 "                    frequency Sine wave, waits for 8085"
 "                    to process the data and displays  "
 "                    result, if so desired.            "
 "═══════════════════════════════════════════════════"}
  procedure PCIN_PCOUT;
   begin
    TextMode;
    LOAD_DATA; TextMode;
    new_status:= curr_status and $DF; {stop PC output}
    SEND_COMMAND(new_status);
```

```
    auto_init:= false; INIT_DMA_CNTRLR(auto_init);
    new_status:= $21; {input from PC, output to PC}
    SEND_COMMAND(new_status);
    write(' REMEMOTE PROCESSING OF DATA IN PROGRESS.');
    WAIT_FOR_TC;
    writeln;writeln;writeln;
    write(' Do you want to examine
                    the data collected ?(Y/N) ');
    if ANS_IS_YES then SECT_PLOT;
    auto_init:= true; INIT_DMA_CNTRLR(auto_init);
                {PC in continuous collection mode again}
  end;

{"=============================================================="
 " ADIN_DAOUT                                                   "
 " Parameters Passed= none                                      "
 " Called By        = PRINT_MENU                                "
 " Function         = sends the appropreate command to the"
 "                    STD Bus system.                           "
 "=============================================================="}
  procedure ADIN_DAOUT;
   begin
    new_status:= $91; SEND_COMMAND(new_status);
   end;
```

```
{"═══════════════════════════════════════════════════════"
 " ADIN_PCOUT                                            "
 " Parameters Passed= none                               "
 " Called By        = PRINT_MENU                         "
 " Function         = sends the appropreate command to the "
 "                    STD Bus system. it goes into display "
 "                    mode if operator wants to see result "
 "═══════════════════════════════════════════════════════"}
   procedure ADIN_PCOUT;
    begin
     new_status:= $A1; SEND_COMMAND(new_status);
     TextMode;
     writeln; write(' Do you want the incoming
                      data to be displayed ? (Y/N) ');
     if ANS_IS_YES then CONT_PLOT;
    end;

{"═══════════════════════════════════════════════════════"
 " ADIN_BOTHOUT                                           "
 " Parameters Passed= none                               "
 " Called By        = PRINT_MENU                         "
 " Function         = sends the appropreate command to the "
 "                    STD Bus system. it goes into display "
 "                    mode if operator wants to see result "
 "═══════════════════════════════════════════════════════"}
   procedure ADIN_BOTHOUT;
```

```
    begin
     new_status:= $B1; SEND_COMMAND(new_status);
     TextMode;
     writeln; write(' Do you want the incoming
                          data to be displayed ? (Y/N) ');
     if ANS_IS_YES then CONT_PLOT;
    end;

{"=============================================================="
 " CALL_PLOT                                                    "
 " Parameters Passed= none                                      "
 " Called By        = PRINT_MENU                                "
 " Function         = calls the Continuous Plot procedure"
 "                    to plot the incoming data.                "
 "=============================================================="}
   procedure CALL_PLOT;
    var c: char;
    begin
     TextMode; write(' Data coming in, if any, is
                 plotted ... hit any key to continue.');
     repeat begin end until keypressed; read(kbd,c);
     CONT_PLOT;
    end;

{"=============================================================="
 " PRINT_MENU                                                   "
```

```
" Parameters Passed= quit - boolean function, returned   "
"                          true to the main program if  "
"                          operator wants to quit.      "
" Called By        = Main Program                        "
" Function         = prints the menu of operations on the"
"                    screen, waits for                   "
"                    command and takes appropreate action"
"═══════════════════════════════════════════════════════"}

  procedure PRINT_MENU(var quit: boolean);
   var
    corekt: boolean;
    ch: char;
   begin
    TextMode(C80); TextColor(LightCyan);
    GoToXY(31,4); write('OPERATIONS  MENU');
    GoToXY(31,5); write('*****************');
    TextColor(Yellow);
    GoToXY(17,9); write('Q');
    GoToXY(17,11);write('A');
    GoToXY(17,13);write('B');
    GoToXY(17,15);write('C');
    GoToXY(17,17);write('D');
    GoToXY(17,19);write('E');
    GoToXY(17,21);write('F');
    TextColor(LightBlue);
    GoToXY(19,9); write(': Quit');
```

```
GoToXY(19,11);write(': Test Filter');
GoToXY(19,13);write(': Samples from A/D;
                                output to D/A');
GoToXY(19,15);write(': Samples from A/D;
                                output to PC');
GoToXY(19,17);write(': Samples from A/D;
                            output to both D/A and PC');
GoToXY(19,19);write(': Display incoming data');
GoToXY(19,21);write(': Change Filter');
TextColor(LightRed+blink);
GoToXY(32,24);write('ENTER COMMAND');
corekt:= false;
while not corekt do begin
 repeat begin end until KeyPressed;
 read(kbd,ch); if upcase(ch)
  in ['Q','A','B','C','D','E','F'] then corekt:= true;
end;
case upcase(ch) of
 'Q': begin quit:= true; TextMode; INIT_SYSTEM; end;
 'A': PCIN_PCOUT;
 'B': ADIN_DAOUT;
 'C': ADIN_PCOUT;
 'D': ADIN_BOTHOUT;
 'E': CALL_PLOT;
 'F': CHANGE_FILTER;
```

```
    end;
   end;

{""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

        -----------  MAIN PROGRAM  ------------

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""}

  begin
   INIT_SYSTEM;
   quit:= false;
   repeat PRINT_MENU(quit) until quit;
  end.
```

REFERENCES

1. Liebowitz, B.H., and Carson, J.H., "Tutorial: Distributed Processing", 2nd Ed., IEEE EHO 127-1, IEEE Computer Society, Long Beach, Calif., 1980.

2. Liebowitz, B.H. and Carson, J.H., "Multiple Processor Systems for Real-Time Applications", Prentice Hall, Inc.

3. Corsini, P., Frosini, G., Grandoni, F., Galati, G., LaManna, M., "The Serial Microprocessor Array (SMA): Microprogramming and Application Examples", IEEE Transactions, Vol.6., pp230

4. AID Manual, E&L Instruments, Inc.

5. IBM PC Technical Reference Manual, IBM Corporation, 1983.

6. User/Technical Manual, PC/STD Link (IBM PC - STD Bus Interface), RMAC, April 1984.

7. Short, Kenneth L., "Microprocessors and Programmed Logic", Prentice Hall.

8. TURBO Pascal Reference Manual, Borland International.

9. Stone, Harold S., "Microcomputer Interfacing", Addison Wesley.

10. Technical Manual - Series 7000, STD Bus, Pro-Log Corporation, 1979.

11. Houpis, Constantine H., Lamont, Gary B., "Digital Control Systems - Theory, Hardware, Software", McGraw Hill Corporation.

BIBLIOGRAPHY

1. Arden B.W, and Berenbaum, A.D, "A Multimicroprocessor Computer System Architecture"; IEEE Proceedings 5th Symp., Operating System Review Vol.9, Nov 1975, pp114-121

2. D'Azzo, John L. and Houpis, Constantine H., "Linear Control Systems - Analysis and Design", 2nd Ed. McGraw Hill.

3. Enslow Jr. Philip H., "Multiprocessors and Parallal Processing", John Wiley & Sons, Inc., 1974

4. Enslow Jr. Philip H., "Multiprocessor Organization - A Survey", Computing Survey, Vol.9, No.1, Mar 1977, pp103-129

5. Fuller, S.H. and Green,A.I., "Multi Microprocessors: An Overview and Working Example", "IEEE Proceedings, Vol.66, No.2, Feb 1978, pp216-228

6. Jones, Anita K., Schwarz Peter, "Experience Using Multiprocessor Systems - A Status Report", Computing Surveys, Vol.12, No.2, June 1980, pp121-164

7. Liebowitz, Burt H., "Multiple Processor Mini Computer Systems", Computer Design, Oct 1978 (pp87-95) and Nov 1978 (pp121-131)

8. MCS 80-85 Family User's Manual, INTEL Corporation

9. Nutt, Gary J., "Microprocessor Implementation of a Parallal Processor", Computer Architecture News, Vol.5, pp147-150

10. Parker, Y., "Multi Microprocessor Systems", Academic Press, 1983

11. Programming Manual - 8080/8085 Assembly Language, INTEL Corporation

12. Tinari Jr., Rocco and Rao, Sathyanarayan S., "Microcomputer based Digital Signal Processing Laboratory Experiments", Computers in Education Division (COED) of ASEE, Vol.V, No.2, Apr-June 1985

13. Yue, W.Y. and Halverson, R.P., "Making the Most of Multiprocessing for Microcomputers", Computer Design, Feb 1982, pp101-105

The vita has been removed from the scanned document